Framework-Based Programming

ASP.NET Core, Laravel and Spring Boot, from First Route to Deployed App

Irfan Subakti

Profitina
profitina.com

Framework-Based Programming

ASP.NET Core, Laravel and Spring Boot, from First Route to Deployed App

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 ..... **xxix**
Who this book is for xxix
One app, three frameworks xxix
What is in it xxx
How the code was checked xxx
How to read it xxxi
About the author xxxi
Acknowledgments xxxii

CHAPTER 1 From Web Programming to Frameworks **1**
1.1 Bu Wati's Price List 1
1.2 The Idea in Plain Words 2
1.2.1 What a web application does for every request 2
1.2.2 Library or framework: who calls whom? 2
1.2.3 Model, View, Controller 3
1.2.4 What a framework gives you, and what it costs 4
1.3 Why ASP.NET Core, Laravel and Spring Boot 5
1.4 Setting Up the Workbench 7
1.4.1 Visual Studio Code 7
1.4.2 The .NET 10 SDK 7
1.4.3 PHP 8.5 and Composer 8
1.4.4 JDK 25 and Maven 8
1.4.5 Databases: SQLite and MySQL 9
1.4.6 Checking everything 9
1.5 Building It Without a Framework 10
1.5.1 In C#, with `HttpListener` 10
1.5.2 In PHP, with the built-in server 13
1.5.3 In Java, with `com.sun.net.httpserver` 15
1.6 The Same Price List With a Framework 18
1.6.1 ASP.NET Core minimal APIs 18
1.6.2 Running it 20
1.6.3 A first real project in each framework 20
1.7 Running the Four Programs Side by Side 21
1.8 Making It Better: What We Still Did by Hand 23
1.9 Using It for Real 23
1.9.1 Your own shop, clinic or farm 23
1.9.2 Choosing a framework for a real project 23
1.9.3 Profitina in practice 23
1.10 Common Mistakes, and How to Avoid Them 24
1.11 Chapter Summary 24
1.12 Exercises 25
1.13 Solutions to the Exercises 25
References 26

CHAPTER 2 Foundations Shared by Modern Frameworks: MVC, Routing, ORM, DI, Templates **28**
2.1 Where We Left Off 28
2.2 The Idea in Plain Words 29
2.3 Building a Tiny Framework: MiniFrame 30
2.3.1 Routing 30

2.3.2 Dependency injection ... 32
2.3.3 Object-relational mapping ... 34
2.3.4 Templates ... 37
2.3.5 The application on top ... 39
2.4 Running It ... 43
2.5 The Same Feature in the Three Real Frameworks ... 44
2.5.1 Routing ... 44
2.5.2 Dependency injection ... 45
2.5.3 Object-relational mapping ... 45
2.5.4 Templates ... 47
2.5.5 One feature, three frameworks ... 50
2.6 Making It Better: What MiniFrame Leaves Out ... 51
2.7 Using It for Real ... 52
2.7.1 Reading any framework through the four foundations ... 52
2.7.2 Profitina in practice ... 52
2.7.3 Planning a new feature with the four questions ... 52
2.8 Common Mistakes, and How to Avoid Them ... 53
2.9 Chapter Summary ... 53
2.10 Exercises ... 54
2.11 Solutions to the Exercises ... 54
References ... 55

CHAPTER 3 ASP.NET Core MVC I: Project, Pipeline, Routing, Controllers and Razor Views ... 57

3.1 Bu Wati's Website ... 57
3.2 The Idea in Plain Words ... 58
3.2.1 The host and the pipeline ... 58
3.2.2 One request through MVC ... 58
3.3 Creating the Project ... 59
3.3.1 The MVC template ... 59
3.3.2 What we change in the template ... 60
3.4 `Program.cs`: Services, Then Pipeline ... 61
3.5 The Model: Data and Rules ... 63
3.5.1 Services, outlets and money ... 63
3.5.2 The price rules ... 64
3.5.3 View models and the form model ... 65
3.5.4 The catalog: data access behind an interface ... 66
3.6 Controllers and Routing ... 68
3.6.1 Conventional routing: `HomeController` and `OutletsController` ... 68
3.6.2 Attribute routing: `ServicesController` ... 70
3.6.3 What actions return ... 73
3.6.4 Model binding: the quote form ... 73
3.7 Razor Views ... 73
3.7.1 Settings for all views: `_ViewImports` and `_ViewStart` ... 74
3.7.2 The layout ... 74
3.7.3 Pages, a partial view and a section ... 75
3.7.4 Tag helpers and the quote form ... 78
3.8 Running It ... 79
3.8.1 In a terminal or in VS Code ... 79
3.8.2 What the browser shows ... 80
3.8.3 Every address, checked ... 84
3.8.4 The log ... 85
3.9 Making It Better ... 86
3.9.1 The fat-controller version, and why we did not write it ... 86
3.9.2 Friendly errors that keep their status ... 86

3.9.3 What is still missing ... 86
3.10 Using It for Real ... 86
3.10.1 Your own business on LaundryMvc ... 86
3.10.2 A preview of deployment ... 87
3.10.3 Profitina in practice ... 87
3.11 Common Mistakes, and How to Avoid Them ... 87
3.12 Chapter Summary ... 88
3.13 Exercises ... 88
3.14 Solutions to the Exercises ... 89
References ... 90

CHAPTER 4 Evaluation and Discussion I: The Profitina Laundry Showcase Site, Solved in Full ... 92

4.1 Where We Left Off ... 92
4.2 How to Approach a Framework Problem ... 93
4.3 Problem 4.1: The Showcase Site ... 94
4.3.1 The problem ... 94
4.3.2 Understanding it by hand ... 94
4.3.3 Designing it on paper ... 95
4.3.4 The solution ... 95
4.3.5 Running it ... 105
4.3.6 A tempting wrong solution ... 107
4.3.7 Check yourself ... 107
4.4 Problem 4.2: Where Is My Laundry? ... 108
4.4.1 The problem ... 108
4.4.2 Understanding it by hand ... 108
4.4.3 Designing it on paper ... 108
4.4.4 The solution ... 109
4.4.5 Running it ... 110
4.4.6 A tempting wrong solution ... 112
4.4.7 Check yourself ... 113
4.5 Problem 4.3: The Route Table by Hand ... 113
4.5.1 The problem ... 113
4.5.2 Understanding it by hand ... 114
4.5.3 Designing it on paper ... 114
4.5.4 The solution: let the application show its routes ... 114
4.5.5 Running it ... 114
4.5.6 A tempting wrong solution ... 115
4.5.7 Check yourself ... 116
4.6 Problem 4.4: A Response-Time Header ... 116
4.6.1 The problem ... 116
4.6.2 Understanding it by hand ... 116
4.6.3 Designing it on paper ... 116
4.6.4 The solution ... 117
4.6.5 Running it ... 117
4.6.6 A tempting wrong solution ... 117
4.6.7 Check yourself ... 118
4.7 Problem 4.5: Maintenance Mode That Keeps Its Style ... 118
4.7.1 The problem ... 118
4.7.2 Understanding it by hand ... 119
4.7.3 Designing it on paper ... 119
4.7.4 The solution ... 119
4.7.5 Running it ... 119
4.7.6 A tempting wrong solution ... 120
4.7.7 Check yourself ... 121

4.8 Using It for Real 121
4.8.1 A showcase for your own business 121
4.8.2 Tracking pages and personal data 121
4.8.3 Profitina in practice 121
4.9 Common Mistakes, and How to Avoid Them 122
4.10 Chapter Summary 122
4.11 Exercises 123
4.12 Solutions to the Exercises 123
References 124

CHAPTER 5 ASP.NET Core MVC II: Entity Framework Core and Real CRUD 126
5.1 Where We Left Off: A Shop Without a Memory 126
5.2 The Idea in Plain Words 127
5.2.1 Tables on one side, objects on the other 127
5.2.2 Two moments: design time and run time 127
5.2.3 LINQ is the same language everywhere 128
5.2.4 Why SQLite 128
5.3 Designing the Data 128
5.4 LINQ First, Without a Database 130
5.5 Installing Entity Framework Core 133
5.6 The Model Classes 134
5.6.1 Service: from record to entity 134
5.6.2 Customer, Order and Payment 134
5.7 The DbContext 136
5.8 Seeding the Database and Keeping Chapter 3's Promise 138
5.8.1 Seed data 138
5.8.2 The catalog on the database 140
5.8.3 Registering everything in Program.cs 140
5.9 Migrations: From Classes to Tables 142
5.10 Real CRUD for Customers 142
5.10.1 Getting the DbContext 146
5.10.2 Read: queries, AsNoTracking and Include 146
5.10.3 Create: binding, checking, saving 147
5.10.4 Update: change a tracked entity 147
5.10.5 Delete: confirm, then refuse or remove 147
5.10.6 The views 147
5.11 Orders, Payments and the Price List 150
5.11.1 The order counter 150
5.11.2 The owner's price list and the counter page 155
5.12 Running It 157
5.13 Making It Better 158
5.14 Using It for Real 158
5.15 Profitina in Practice 159
5.16 Common Mistakes, and How to Avoid Them 159
5.17 Chapter Summary 160
5.18 Exercises 160
5.19 Solutions to the Exercises 161
References 162

CHAPTER 6 ASP.NET Core MVC III: Validation, Identity and Deployment 163
6.1 Where We Left Off: An Open Door 163
6.2 The Idea in Plain Words 163
6.3 Validation with Data Annotations 165
6.3.1 Attributes on the model 165
6.3.2 A validation attribute of our own 167

6.3.3 Running the rules without a web server 168
6.3.4 What MVC does with the rules 170
6.4 Form Classes Instead of Entities 170
6.4.1 Why a separate class 170
6.4.2 A rule about two fields 171
6.4.3 The controller with form classes 172
6.5 Checks in the Browser, Too 176
6.6 Identity: Who Is at the Counter? 177
6.6.1 What Identity provides 177
6.6.2 Program.cs with Identity, authorization and deployment settings 179
6.6.3 What is stored instead of a password 182
6.6.4 Roles and the first accounts 182
6.7 Signing In and Keeping Doors Closed 184
6.7.1 The sign-in page 184
6.7.2 Who may do what 186
6.7.3 Policies: rules with names 187
6.8 Deployment: From Laptop to Server 187
6.8.1 Environments and settings 187
6.8.2 Publishing 188
6.8.3 Migrations on the server 188
6.8.4 A Linux server with Nginx and systemd 188
6.8.5 Windows with IIS 190
6.8.6 The small things that make it last 190
6.8.7 A release checklist 190
6.9 Running It 191
6.10 Making It Better 192
6.11 Using It for Real 192
6.12 Profitina in Practice 192
6.13 Common Mistakes, and How to Avoid Them 193
6.14 Chapter Summary 193
6.15 Exercises 193
6.16 Solutions to the Exercises 194
References 195

CHAPTER 7 Laravel I: Artisan, Routing, Blade and Eloquent 197
7.1 Where We Left Off: The Same Shop, Another Language 197
7.2 The Idea in Plain Words 197
7.3 Installing PHP, Composer and Laravel 199
7.4 Artisan, the Command Line That Writes Code 200
7.5 Migrations: The Tables Written in PHP 200
7.6 Eloquent Models 204
7.6.1 Enums for the status and the payment method 204
7.6.2 The models 206
7.6.3 Eloquent and EF Core side by side 210
7.7 Seeding from the Chapter 3 Files 211
7.8 Routing 213
7.9 Controllers 215
7.9.1 Customers: validation and CRUD 215
7.9.2 Orders, payments and prices 218
7.9.3 Seeing the SQL 222
7.10 Blade Templates 222
7.10.1 The layout 222
7.10.2 Escaping, forms and old input 223
7.10.3 The order page 225
7.10.4 Razor and Blade side by side 227

7.11 Running It 228
7.12 Making It Better 229
7.13 Using It for Real 229
7.14 Profitina in Practice 230
7.15 Common Mistakes, and How to Avoid Them 230
7.16 Chapter Summary 231
7.17 Exercises 231
7.18 Solutions to the Exercises 232
References 233

CHAPTER 8 Evaluation and Discussion II: The Same Feature Set in ASP.NET Core and Laravel 234

8.1 Where We Left Off 234
8.2 How to Approach a Framework Problem 234
8.2.1 Six steps 234
8.2.2 Where the code lives, and what was run 235
8.3 Problem 8.1: One Price Rule, Two Frameworks 236
8.3.1 The problem 236
8.3.2 Understanding it by hand 236
8.3.3 Designing it on paper 236
8.3.4 The ASP.NET Core solution 237
8.3.5 The Laravel solution 239
8.3.6 A tempting wrong solution 240
8.4 Problem 8.2: The Order Workflow in One Place 241
8.4.1 The problem 241
8.4.2 Understanding it by hand 241
8.4.3 Designing it on paper 241
8.4.4 The ASP.NET Core solution 242
8.4.5 The Laravel solution 242
8.4.6 A tempting wrong solution 243
8.5 Problem 8.3: The Order List, Without N+1 and Without Waste 243
8.5.1 The problem 243
8.5.2 Understanding it by hand 243
8.5.3 Designing it on paper 244
8.5.4 The ASP.NET Core solution 244
8.5.5 The Laravel solution 250
8.5.6 A tempting wrong solution 253
8.6 Problem 8.4: A Customer Search That Finds Sari 253
8.6.1 The problem 253
8.6.2 Understanding it by hand 253
8.6.3 Designing it on paper 254
8.6.4 The ASP.NET Core solution 254
8.6.5 The Laravel solution 257
8.6.6 A tempting wrong solution 260
8.7 Problem 8.5: No Overpayment 260
8.7.1 The problem 260
8.7.2 Understanding it by hand 260
8.7.3 Designing it on paper 260
8.7.4 The ASP.NET Core solution 260
8.7.5 The Laravel solution 260
8.7.6 A tempting wrong solution 261
8.8 Problem 8.6: The Daily Revenue Report 261
8.8.1 The problem 261
8.8.2 Understanding it by hand 261
8.8.3 Designing it on paper 262

8.8.4 The ASP.NET Core solution 262
8.8.5 The Laravel solution 264
8.8.6 A tempting wrong solution 266
8.9 Problem 8.7: A Rack for Every Finished Bag 266
8.9.1 The problem 266
8.9.2 Understanding it by hand 267
8.9.3 Designing it on paper 267
8.9.4 The ASP.NET Core solution 267
8.9.5 The Laravel solution 269
8.9.6 A tempting wrong solution 269
8.10 The Problems Side by Side 269
8.11 Profitina in Practice 270
8.12 Common Mistakes, and How to Avoid Them 270
8.13 Chapter Summary 271
8.14 Exercises 271
8.15 Solutions to the Exercises 272
References 273

CHAPTER 9 Laravel II: Migrations, Seeders, Middleware and Form Requests 274
9.1 Bu Wati's Third Outlet 274
9.2 The Idea in Plain Words 275
9.2.1 Migrations: a logbook of changes to the database 275
9.2.2 Factories and seeders: a believable month, made up on purpose 275
9.2.3 Middleware: the checkpoints every request passes 276
9.2.4 Form requests: the counter checklist, in one place 276
9.3 Where Chapter 8 Left the Project 277
9.4 Migrations in Depth 278
9.4.1 What the database remembers 278
9.4.2 Changing a table that already holds orders 279
9.4.3 The commands you need 280
9.4.4 Four rules for a database that other people share 281
9.5 Factories and Seeders 281
9.5.1 Models that can be invented 281
9.5.2 Writing the factories 283
9.5.3 Writing the seeders 285
9.5.4 Running them 288
9.6 Middleware 288
9.6.1 The onion, without Laravel 288
9.6.2 A "before" middleware for a rule of the shop 290
9.6.3 An "after" middleware 291
9.6.4 Registering the middleware 291
9.6.5 Attaching middleware to routes 292
9.7 Form Requests 294
9.7.1 The order checklist 294
9.7.2 A rule that needs the order 295
9.7.3 Controllers that trust their input 296
9.8 Running It 300
9.9 Making It Better 300
9.10 Using It for Real 301
9.11 Common Mistakes, and How to Avoid Them 301
9.12 Chapter Summary 302
9.13 Exercises 302
9.14 Solutions to the Exercises 303
References 306

CHAPTER 10 Laravel III: Authentication, API Resources and Testing 307
10.1 A Price List Anyone Could Change 307
10.2 The Idea in Plain Words 308
10.2.1 Who are you, and what may you do? 308
10.2.2 Sessions for browsers, tokens for programs 308
10.2.3 Resources: deciding what an answer contains 309
10.2.4 Tests: a tireless second pair of hands 309
10.3 Users and Roles 309
10.4 Signing In to the Staff Pages 313
10.4.1 The sign-in controller 313
10.4.2 Guarding the routes 314
10.4.3 A gate for the price list 316
10.5 An API for Programs 318
10.5.1 Installing Sanctum 318
10.5.2 Issuing and revoking tokens 318
10.5.3 The API routes 319
10.5.4 One way to create an order 320
10.6 API Resources 326
10.7 Testing 328
10.7.1 A unit test for the price rule 328
10.7.2 Feature tests for signing in 329
10.7.3 Feature tests for the API 331
10.7.4 Running the tests 334
10.8 Running It 334
10.9 Making It Better 335
10.10 Using It for Real 335
10.11 Common Mistakes, and How to Avoid Them 336
10.12 Chapter Summary 336
10.13 Exercises 337
10.14 Solutions to the Exercises 337
References 339

CHAPTER 11 Spring Boot I: Initializr, REST Controllers and Spring Data JPA 340
11.1 Raka's Question 340
11.2 The Idea in Plain Words 341
11.2.1 A container that builds your objects 341
11.2.2 Boot: sensible defaults, switched on by what you depend on 341
11.2.3 Entities, repositories and the SQL you do not write 341
11.2.4 Controllers that speak JSON 342
11.3 Creating the Project 343
11.3.1 Spring Initializr 343
11.3.2 The build file 343
11.3.3 The layout and the entry point 345
11.3.4 Settings 345
11.4 The Domain: Entities 346
11.4.1 The four tables 346
11.4.2 Customer 346
11.4.3 Service, and a name that is already taken 348
11.4.4 Order and Payment 349
11.4.5 Statuses stored as the other frameworks store them 353
11.4.6 The rules, in plain Java 355
11.5 Repositories 356
11.6 REST Controllers 358
11.6.1 Records for what goes out 358
11.6.2 Records for what comes in 359

11.6.3 The controllers 360
11.7 Loading the Price List 364
11.8 Running It 366
11.9 Testing the Rules 366
11.10 Making It Better 367
11.11 Using It for Real 368
11.12 Common Mistakes, and How to Avoid Them 368
11.13 Chapter Summary 369
11.14 Exercises 369
11.15 Solutions to the Exercises 370
References 371

CHAPTER 12 Evaluation and Discussion III: One REST API in Laravel and Spring Boot 373

12.1 Where We Left Off 373
12.2 How to Approach an API Problem 373
12.3 The Contract 374
12.4 Problem 12.1: The Price List for Programs 376
12.4.1 The problem 376
12.4.2 Understanding it by hand 376
12.4.3 Comparing the two 376
12.4.4 The solution in both frameworks 376
12.4.5 A tempting wrong solution 378
12.5 Problem 12.2: One Way to Write a Time 379
12.5.1 The problem 379
12.5.2 Understanding it by hand 379
12.5.3 Comparing the two 379
12.5.4 The solution in both frameworks 379
12.5.5 A tempting wrong solution 381
12.6 Problem 12.3: One Shape for Every Error 382
12.6.1 The problem 382
12.6.2 Understanding it by hand 382
12.6.3 Comparing the two 382
12.6.4 The solution in Spring 382
12.6.5 A tempting wrong solution 385
12.7 Problem 12.4: Every Error at Once 385
12.7.1 The problem 385
12.7.2 Understanding it by hand 385
12.7.3 Comparing the two 386
12.7.4 The solution in Spring 386
12.7.5 A tempting wrong solution 391
12.8 Problem 12.5: Moving an Order, and Closed Orders 391
12.8.1 The problem 391
12.8.2 Understanding it by hand 391
12.8.3 Comparing the two 391
12.8.4 The solution in Spring 392
12.8.5 A tempting wrong solution 392
12.9 Problem 12.6: Payments 392
12.9.1 The problem 392
12.9.2 Understanding it by hand 392
12.9.3 Comparing the two 392
12.9.4 The solution in Spring 392
12.9.5 A tempting wrong solution 393
12.10 Problem 12.7: The Daily Report 393
12.10.1 The problem 393
12.10.2 Understanding it by hand 394

12.10.3 Comparing the two 394
12.10.4 The solution in both frameworks 394
12.10.5 A tempting wrong solution 398
12.11 Checking the Contract 398
12.11.1 The checker 398
12.11.2 A real run 402
12.11.3 Tests inside each framework 402
12.12 Using It for Real 405
12.13 Common Mistakes, and How to Avoid Them 406
12.14 Chapter Summary 406
12.15 Exercises 407
12.16 Solutions to the Exercises 407
References 408

CHAPTER 13 Spring Boot II: Layered Architecture, Validation and Full CRUD with Thymeleaf 409

13.1 Where We Left Off 409
13.2 The Idea in Plain Words 410
13.2.1 One class that does everything 410
13.2.2 Three layers, one direction 410
13.2.3 What each layer may know 411
13.2.4 Transactions: all or nothing 411
13.3 Where the Code Goes 412
13.4 The Service Layer 412
13.4.1 Three exceptions 412
13.4.2 Customers 414
13.4.3 Orders 418
13.4.4 The daily report 424
13.5 Thin API Controllers 425
13.6 Validating Form Input 432
13.6.1 Jakarta Bean Validation 432
13.6.2 Form records 433
13.6.3 Rule messages on the right field 434
13.7 Pages with Thymeleaf 435
13.7.1 The customer pages 435
13.7.2 Post/Redirect/Get 438
13.7.3 Templates 439
13.7.4 Orders and payments 441
13.8 One Place for Errors 448
13.9 SQLite, Configuration and Running It 449
13.10 Testing the Service Layer 452
13.11 Making It Better 455
13.12 Using It for Real 456
13.13 Profitina in Practice 456
13.14 Common Mistakes, and How to Avoid Them 456
13.15 Chapter Summary 457
13.16 Exercises 457
13.17 Solutions to the Exercises 458
References 459

CHAPTER 14 Spring Boot III: Spring Security, REST API Design and Automated Testing 461

14.1 Where We Left Off 461
14.2 The Idea in Plain Words 461
14.3 Accounts and Passwords 462

14.3.1 Never store a password 462
14.3.2 Staff and roles 463
14.3.3 Telling Spring Security about our staff 465
14.4 Tokens for the API 466
14.4.1 The idea of Sanctum's tokens 466
14.4.2 The token table and the token service 467
14.4.3 The filter 471
14.4.4 Signing in and signing out 473
14.5 Two Filter Chains 474
14.5.1 The configuration 474
14.5.2 Following requests by hand 477
14.5.3 The login page 477
14.5.4 CSRF: why the pages need it and the API does not 479
14.5.5 A second line of defense in the service 479
14.6 Designing the REST API 482
14.6.1 Resources, URLs and methods 483
14.6.2 Safe and idempotent methods 483
14.6.3 Status codes and error bodies 484
14.6.4 Envelopes and pages 484
14.6.5 What the API still lacks: versions 485
14.7 Automated Tests on Four Levels 485
14.7.1 The pyramid 485
14.7.2 The web layer alone 488
14.7.3 The database queries 490
14.7.4 The whole application 491
14.8 Making It Better 493
14.9 Using It for Real 493
14.10 Profitina in Practice 494
14.11 Common Mistakes, and How to Avoid Them 494
14.12 Chapter Summary 495
14.13 Exercises 495
14.14 Solutions to the Exercises 495
References 496

CHAPTER 15 Comparing the Three Frameworks: Performance, Productivity, Ecosystem, Choosing 498
15.1 Where We Left Off 498
15.2 The Idea in Plain Words 498
15.3 One Application, Three Vocabularies 499
15.3.1 The same concepts 499
15.3.2 The life of one request 501
15.3.3 Testing compared 501
15.3.4 The same endpoint three times 502
15.4 Performance 504
15.4.1 What a public benchmark measures 504
15.4.2 Measuring it yourself 505
15.4.3 Checking the numbers with Little's law 506
15.4.4 Start-up time and memory 506
15.5 Productivity 507
15.6 Ecosystem and Community 508
15.6.1 What developers use 508
15.6.2 Packages, hosting and help 509
15.7 Support Calendars 509
15.8 Security by Default 510
15.9 Choosing 511

15.10 Using It for Real ... 511
15.11 Profitina in Practice ... 512
15.12 Common Mistakes, and How to Avoid Them ... 512
15.13 Chapter Summary ... 513
15.14 Exercises ... 513
15.15 Solutions to the Exercises ... 514
References ... 514

CHAPTER 16 Evaluation and Discussion IV ... 516
16.1 Where We Left Off ... 516
16.2 How to Approach the Capstone ... 516
16.3 Problem 16.1: The Contract ... 518
16.3.1 The problem ... 518
16.3.2 Understanding it by hand ... 518
16.3.3 The contract ... 518
16.3.4 The checker ... 519
16.3.5 Reading a failing run ... 524
16.3.6 A tempting wrong solution ... 524
16.4 Problem 16.2: The Rules in Three Languages ... 524
16.4.1 The problem ... 524
16.4.2 Understanding it by hand ... 524
16.4.3 The solutions ... 525
16.4.4 A tempting wrong solution ... 525
16.5 Problem 16.3: The ASP.NET Core Capstone ... 526
16.5.1 The problem ... 526
16.5.2 Designing it on paper ... 526
16.5.3 The domain and the storage ... 527
16.5.4 The service ... 529
16.5.5 The API, its errors and its doors ... 534
16.5.6 Running it ... 545
16.5.7 The counter board ... 547
16.5.8 A tempting wrong solution ... 549
16.6 Problem 16.4: The Laravel Capstone ... 549
16.6.1 The problem ... 549
16.6.2 Designing it on paper ... 549
16.6.3 The solution ... 549
16.6.4 Running it ... 556
16.6.5 A tempting wrong solution ... 556
16.7 Problem 16.5: The Spring Boot Capstone ... 556
16.7.1 The problem ... 556
16.7.2 The solution ... 556
16.7.3 Running it ... 558
16.8 Comparing the Three Solutions ... 559
16.9 Bu Wati's Decision ... 560
16.10 Using It for Real ... 560
16.11 Profitina in Practice ... 561
16.12 Common Mistakes, and How to Avoid Them ... 561
16.13 Chapter Summary ... 562
16.14 Exercises ... 562
16.15 Solutions to the Exercises ... 563
References ... 564

List of Figures

Figure 1.1 A library is called by your program; a framework calls your program. With a framework, the only code you write is the part that is about your business. 3

Figure 1.2 One request through Model–View–Controller: the router chooses the controller action, the controller asks the model and passes the result to a view, and the view's HTML is the response. 4

Figure 1.3 Web frameworks and technologies used by professional developers in the Stack Overflow Developer Survey 2025 (19,460 respondents to this question); the three frameworks of this book are in teal. 6

Figure 2.1 The four foundations along one request: routing chooses the action, dependency injection builds the controller, the ORM turns objects into SQL and rows into objects, and the templates turn objects into HTML. 29

Figure 2.2 Matching `GET /services/under/15000` against the route table: the segment counts rule out the first two routes, and the third fits because its parameter `{max:int}` accepts digits. 32

Figure 2.3 The registrations made at start-up, and the graph the container builds when a controller is requested. Only the transient controller is new on every request. 34

Figure 2.4 Above, a row of the table `service` becomes a `Service` object because column names equal property names. Below, the condition `s => s.Price <= 15000` is read as an expression tree and printed as SQL. 37

Figure 2.5 From model to page: the page template is filled with the model, the result is inserted into the layout, and the whole page is sent. 39

Figure 3.1 The request pipeline of LaundryMvc: every request passes through the logging, status-code and routing middleware to an endpoint, and the response comes back through the same layers. 58

Figure 3.2 One request, `GET /services/S0004`, through LaundryMvc: from bytes to `HttpContext`, through middleware and routing, to a controller built by the container, a view and the layout. 59

Figure 3.3 The home page of LaundryMvc: the layout's header and menu, the view's text with counts from the view model, and three cards drawn by the partial view `_ServiceCard`. 80

Figure 3.4 The list of services at `/services`, rendered by `Views/Services/Index.cshtml`. 81

Figure 3.5 A price quote: model binding filled `QuoteForm` from the address, `PriceCalculator` applied the minimum weight and the pickup fee, and the view shows the result with its explanations. 82

Figure 3.6 The quote form after `Quantity=abc`: the conversion error from model binding appears in the validation summary, and the field is marked. 83

Figure 3.7 The friendly 404 page for `/services/under/abc`, produced by the status-code middleware and `HomeController.Error`. 83

Figure 4.1 The six steps used for every problem in this chapter; the dashed arrow is the loop between running the program and fixing it. 93

Figure 4.2 The site map of LaundryShowcase: every address is an attribute route of one controller under the prefix `/laundry`; the tracking route with a constraint belongs to Problem 4.2. 95

Figure 4.3 The showcase home page: the tagline comes from `site.json`, the counts from the data. 105

Figure 4.4 The company page: every number is computed by `LaundryData.Facts()` from the order data. 106

Figure 4.5 The collaboration page: offers and asks for investors, partners and job candidates, from `site.json`. 106

Figure 4.6 The tracking page for order B0072: ready on rack A01, with service, outlet, dates and amount, and nothing about the customer. 111

Figure 4.7 The maintenance page, answered with status 503 and styled by the site's own style sheet, which maintenance mode lets through. 120

Figure 5.1 How EF Core connects the classes to the database: migrations at design time, LINQ translated into SQL and the change tracker at run time. 128

Figure 5.2 The Profitina Laundry database of this chapter: every order belongs to one customer and one service, and an order can have many payments. 129
Figure 6.1 The path of a form submission through the Chapter 6 application: authentication and authorization before the controller, model binding and validation before the action, and the four ways the request can end. 165
Figure 6.2 The laundry application on a Linux server: Nginx in front with the certificate, Kestrel behind it on the loopback address, systemd keeping it running, and the data folder with the database, the cookie keys and a nightly backup. 189
Figure 7.1 One request through the Laravel application: the front controller, the web middleware group, the router, the controller, Eloquent and Blade. 198
Figure 8.1 The seven problems of this chapter and the layer each one belongs to. 235
Figure 8.2 SQL statements needed to show one page of 20 orders, by the way the list is loaded; the counts follow from how each call is documented to load data. 244
Figure 9.1 The path of one request, a payment on order 7, through middleware, route model binding and a form request; the red boxes are the places where the request stops before the controller. 277
Figure 9.2 The migration history of the project: each run of `migrate` forms a batch, the `migrations` table records it, and `migrate:rollback` undoes only the last batch. 279
Figure 10.1 Two ways to prove who is asking: a browser signs in once and carries a session cookie; a program exchanges the password for a token and sends it with every request; both end at the same authorization check. 308
Figure 10.2 What `OrderResource` does with an order: fields passed through, converted or computed, relations nested when loaded, and internal columns left out. 327
Figure 11.1 One `POST /api/orders` through a Spring Boot application: embedded Tomcat, the DispatcherServlet, Jackson, Bean Validation, the controller, a Spring Data repository, Hibernate and the database. 342
Figure 11.2 The four tables of Profitina Laundry and the entity classes that map them; arrows go from a foreign key to the primary key it refers to. 346
Figure 12.1 One contract, three implementations and one checker: the clients depend only on the contract, and `check_api.php` verifies every server against it. 374
Figure 12.2 The order's workflow as the contract states it: one step at a time, departure only when the balance is 0, and a departed order closed to every change. 391
Figure 13.1 The layers of the Spring Boot laundry application: page and API controllers speak HTTP, services do the shop's work inside transactions, repositories speak SQL; the rules themselves are plain Java from Chapter 11. 410
Figure 13.2 Post/Redirect/Get: an invalid form is answered with the same page and its messages; a valid one with a redirect, so that refreshing the page never takes the same order twice. 438
Figure 14.1 The two filter chains: requests under `/api/` go to the stateless chain with bearer tokens and JSON refusals; all others go to the chain with a login form, a session cookie and CSRF protection. 476
Figure 14.2 The four levels of tests in the Chapter 14 project, from plain unit tests at the bottom to tests of the whole application at the top. 485
Figure 15.1 TechEmpower Round 23 (February 2025), Fortunes test: responses per second of the best ASP.NET Core, Spring and Laravel implementations, on a logarithmic scale. 504
Figure 15.2 Our own measurement of the ASP.NET Core quote endpoint on a 2-vCPU virtual machine (September 25, 2026): throughput levels off after 8 workers while the p99 latency keeps growing. 506
Figure 15.3 Stack Overflow Developer Survey 2025: share of respondents who used each framework in the past year, and share of its users who want to keep using it ("admired"). 508
Figure 15.4 The support windows of the versions used in this book; the dashed line is September 25, 2026. 509
Figure 16.1 One contract, three implementations: the checker `check_api.php` sends the same 25 requests to each server and compares the answers with the contract. 517
Figure 16.2 The capstone's tables: services from the price list (`S0001` to `S0012`), customers, orders with pickup, payments, staff and API tokens. 526
Figure 16.3 The counter board of the ASP.NET Core capstone as it ran on September 25, 2026: two orders in the queue, one ready to collect. 548

List of Tables

Table 1.1 The jobs a web framework takes over, and the chapters of this book where each one is built for Profitina Laundry. 5
Table 1.2 The three frameworks of this book side by side, with the versions current in September 2026. 6
Table 1.3 The VS Code extensions used in this book. Install only the ones for the framework you are working on if your computer is slow. 7
Table 1.4 Installing the three toolchains. Open a new terminal after each installation, so that the new commands are found. 8
Table 2.1 The files of MiniFrame in `Chapter02/miniframe`. The four foundations take 226 lines together. 30
Table 2.2 The four foundations in MiniFrame and in the three frameworks of this book. 51
Table 3.1 The files created by `dotnet new mvc`, and what each one is for. 59
Table 3.2 How the conventional route `{controller=Home}/{action=Index}/{id?}` maps addresses to actions. Matching ignores upper and lower case. 68
Table 3.3 The attribute routes of `ServicesController`. A template starting with `/` ignores the controller's prefix. 72
Table 3.4 The status codes returned by the running LaundryMvc for every address of this chapter, from `verify.sh`. 84
Table 4.1 The five problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most. 93
Table 4.2 The address table of Problem 4.1, including three addresses that test the edges. 94
Table 4.3 Three orders from `Bill.csv`, one per status, worked out by hand. The amount of B0072 checks: $7 \times 8000 = 56000$. 108
Table 4.4 The address table of Problem 4.2. 108
Table 4.5 The route table predicted by hand for twelve requests. Two different reasons hide behind the 404s: no route, or an action that returned `NotFound()`. 114
Table 4.6 The five tempting wrong solutions of this chapter, what each one does, and what catches it. 122
Table 5.1 How the C# model and the SQLite tables correspond. 130
Table 5.2 The actions of `CustomersController`, their addresses, and the SQL they cause. 143
Table 6.1 The validation attributes used in this chapter. 166
Table 6.2 Who may use which pages after this chapter. 186
Table 6.3 The release checklist for the laundry server. 191
Table 7.1 Where things live in a Laravel 13 project, and their counterparts in ASP.NET Core. 199
Table 7.2 The Artisan commands used in this chapter. 200
Table 7.3 Everyday operations in EF Core and in Eloquent. 211
Table 7.4 The seven routes of `Route::resource('customers', …)`. 215
Table 7.5 Razor and Blade constructs for the same tasks. 228
Table 8.1 The seven problems, their solutions in both frameworks, and what each teaches. 270
Table 9.1 The files that this chapter adds to or changes in the Chapter 8 project. 278
Table 9.2 The migration commands of Artisan and when to use each. 281
Table 9.3 Built-in route middleware of Laravel that this book uses, with their short names. 292
Table 10.1 The refusals of this chapter's application: status code, meaning and the part that answers. 317
Table 11.1 The same jobs in Laravel and in Spring Boot. 343
Table 12.1 The endpoints of the Profitina Laundry API contract; a question mark marks an optional field. 375

Table 12.2 The refusals of the contract, and the body each one carries. 376
Table 13.1 What each layer may know, and what it must not know. 411
Table 13.2 The packages of the Chapter 13 project. 412
Table 13.3 The Bean Validation constraints used by the forms. 433
Table 13.4 The URLs of the customer pages. 438
Table 14.1 Six requests followed through the two chains. 477
Table 14.2 The Profitina Laundry API after Chapter 14. 483
Table 14.3 The status codes of the API and when each is used. 484
Table 14.4 The four levels of tests, with the 24 runs of this chapter. 486
Table 15.1 What to compare, and what kind of evidence each question needs. 499
Table 15.2 The same concepts under three names. 500
Table 15.3 Lines needed for the same two features in the Chapter 16 capstone, counted on September 25, 2026. 507
Table 15.4 Release and support dates, collected from endoflife.date on September 25, 2026. 509
Table 15.5 Security defaults compared. 510
Table 15.6 Four situations and the framework the procedure suggests. 511
Table 16.1 The Profitina Laundry API contract after Chapter 16. 519
Table 16.2 Nine prices worked by hand. 525
Table 16.3 The three capstones compared. 559

List of Listings

Listing 1.1 `nofw-csharp/Program.cs`: the price list with no framework. The comments (1) to (6) mark the jobs a framework would do. 10
Listing 1.2 `nofw-php/index.php`: the same price list in PHP without a framework. The script is at once router, controller, model and view. 13
Listing 1.3 `nofw-java/RawLaundry.java`: the price list in Java without a framework, on the JDK's built-in HTTP server. 15
Listing 1.4 `aspnet/HelloLaundry.csproj`. The `.Web` SDK makes this an ASP.NET Core project; `net10.0` targets .NET 10. 18
Listing 1.5 `aspnet/Program.cs`: the price list with ASP.NET Core minimal APIs. Routing, parameter extraction, status codes, JSON and the server are the framework's. 18
Listing 2.1 `Router.cs`: a route table, template matching and parameter extraction. 30
Listing 2.2 `Container.cs`: registrations, two lifetimes and constructor injection through reflection. 32
Listing 2.3 `Mapper.cs`: rows become objects by matching names, and a C# condition becomes the text of an SQL `WHERE` clause. 34
Listing 2.4 `Template.cs`: holes, loops and a layout, with escaping by default. 37
Listing 2.5 `templates/layout.html`: the frame every page shares; the page's own HTML goes where `{{{ body }}}` is. 38
Listing 2.6 `templates/services.html`: a loop over the rows of the model, with every value escaped. 39
Listing 2.7 `Laundry.cs`: the model, the repository and the controller of the price list. No routing, parsing, SQL or HTML escaping appears here. 39
Listing 2.8 `Program.cs`: registrations, routes, the offline demo and a minimal HTTP server. 41
Listing 2.9 `routes/web.php` in Laravel: three routes, each pointing to a controller method; `whereNumber` is the constraint. 44
Listing 2.10 `app/Models/Service.php`: an Eloquent model. The properties state only what differs from Laravel's conventions. 46
Listing 2.11 `Service.java`: a JPA entity mapped to the table `service`. 46

Listing 2.12 `ServiceRepository.java`: a Spring Data repository. The query is derived from the method's name. 46
Listing 2.13 `resources/views/layouts/app.blade.php`: the Blade layout; `@yield` marks the slots. 47
Listing 2.14 `resources/views/services/index.blade.php`: the list page, filling the layout's two slots. 47
Listing 2.15 `app/Http/Controllers/ServiceController.php`: the Laravel controller. `findOrFail` answers 404 by itself when the code does not exist. 48
Listing 2.16 `templates/layout.html`: two Thymeleaf fragments, the head (with a parameter) and the header. 48
Listing 2.17 `templates/services/index.html`: the list page in Thymeleaf; `th:each` is the loop, `th:text` the escaped hole. 49
Listing 2.18 `ServiceController.java`: the Spring MVC controller, with its repository injected through the constructor. 49
Listing 3.1 `Properties/launchSettings.json`: `dotnet run` starts the app at `http://localhost:4030` in the Development environment. 60
Listing 3.2 `appsettings.json`: log levels and allowed hosts. 60
Listing 3.3 `Program.cs`: the services are registered on the builder, then the pipeline is built on the application. 61
Listing 3.4 `Models/Service.cs`: a service with its code, description and price, and three properties derived from the description. 63
Listing 3.5 `Models/Outlet.cs`: an outlet of Profitina Laundry. 64
Listing 3.6 `Models/Money.cs`: rupiah amounts in the Indonesian style, formatted in one place. 64
Listing 3.7 `Models/PriceCalculator.cs`: the shop's pricing rules, with an explanation for every rule applied. 64
Listing 3.8 `Models/HomeViewModel.cs`: exactly what the home page shows. 65
Listing 3.9 `Models/QuoteForm.cs`: the input of the price-quote form, filled by model binding. 66
Listing 3.10 `Data/ILaundryCatalog.cs`: what controllers may ask about services and outlets. 66
Listing 3.11 `Data/CsvCatalog.cs`: the catalog implemented on the two CSV files, read once at start-up. 66
Listing 3.12 `Controllers/HomeController.cs`: the home page, the about page, and the page shown for error status codes. 68
Listing 3.13 `Controllers/OutletsController.cs`: the outlets, reached through the conventional route. 69
Listing 3.14 `Controllers/ServicesController.cs`: attribute routes, a constraint, a form action and a JSON action. 70
Listing 3.15 `Views/_ViewImports.cshtml`: namespaces and tag helpers for every view. 74
Listing 3.16 `Views/_ViewStart.cshtml`: every view uses the layout `_Layout`. 74
Listing 3.17 `Views/Shared/_Layout.cshtml`: the shared frame, with a menu that marks the current page. 74
Listing 3.18 `Views/Home/Index.cshtml`: the home page, with a partial view for each card. 75
Listing 3.19 `Views/Shared/_ServiceCard.cshtml`: one card, written once and used for every popular service. 76
Listing 3.20 `Views/Home/About.cshtml`: a page that adds the opening hours to the layout's footer through a section. 76
Listing 3.21 `Views/Services/Index.cshtml`: the table of services, shared by the actions `Index` and `Under`. 76
Listing 3.22 `Views/Services/Details.cshtml`: one service, with a link to a quote for it. 77
Listing 3.23 `Views/Outlets/Index.cshtml`: the list of outlets. 77
Listing 3.24 `Views/Outlets/Details.cshtml`: one outlet. 77
Listing 3.25 `Views/Shared/Error.cshtml`: the friendly page for 404 and for other error statuses. 78
Listing 3.26 `Views/Services/Quote.cshtml`: the quote form, built with tag helpers, and the result table. 78
Listing 4.1 `content/site.json`: every word of the showcase that is not computed from data. 95

Listing 4.2 `Models/SiteContent.cs`: the shape of `site.json` as C# records. 96
Listing 4.3 `Models/Records.cs`: services, outlets, orders without customers, and the facts computed from them. 97
Listing 4.4 `Data/LaundryData.cs`: the shop's data, loaded once, and the facts the pages show. 97
Listing 4.5 `Program.cs` of LaundryShowcase: content and data registered as singletons, a middleware pipeline, a redirect, the controllers, and a development-only route list. 99
Listing 4.6 `Controllers/ShowcaseController.cs`: one attribute-routed controller for the whole showcase. 101
Listing 4.7 `Views/_ViewImports.cshtml`: `@inject` puts the site's content into every view. 102
Listing 4.8 `Views/Shared/_Layout.cshtml`: the showcase's frame and menu. 102
Listing 4.9 `Views/Showcase/Home.cshtml`: the tagline and counts computed from the data. 103
Listing 4.10 `Views/Showcase/Profile.cshtml`: the story and the three promises, from `site.json`. 103
Listing 4.11 `Views/Showcase/Company.cshtml`: the mission, the numbers from the data, and the team. 104
Listing 4.12 `Views/Showcase/Services.cshtml`: the price list. 104
Listing 4.13 `Views/Showcase/Collaboration.cshtml`: what the business offers and asks of each audience. 105
Listing 4.14 `Tempting/AbsoluteRoute/Program.cs`: one leading slash moves the profile out of `/laundry` 107
Listing 4.15 `Views/Showcase/TrackForm.cshtml`: a `GET` form with one field. 109
Listing 4.16 `Views/Showcase/Track.cshtml`: the status in words, then the facts of the order, with no personal data. 109
Listing 4.17 `Views/Shared/Error.cshtml`: one sentence for a missing page or order, with the address that was asked for. 110
Listing 4.18 `Tempting/RegexBraces/Program.cs`: a regular expression with single braces inside a route template. 112
Listing 4.19 `Tempting/Ambiguous/Program.cs`: two attribute routes that match exactly the same addresses. 115
Listing 4.20 `Middleware/ResponseTimeMiddleware.cs`: the header is set in an `OnStarting` callback, just before the headers are sent. 117
Listing 4.21 `Tempting/LateHeader/Program.cs`: the header is set after the rest of the pipeline has already started the response. 117
Listing 4.22 `appsettings.json` of LaundryShowcase: the maintenance switch, off by default. 119
Listing 4.23 `Tempting/MaintenanceFirst/Program.cs`: the maintenance check before `UseRouting`, where no endpoint has been chosen yet. 120
Listing 5.1 `LinqDemo/Program.cs`: the chapter's queries on in-memory lists, with the web app's own model classes. 130
Listing 5.2 `LaundryMvc.csproj` after adding the two EF Core packages. 133
Listing 5.3 `appsettings.json`: the connection string and the SQL log. 133
Listing 5.4 `Models/Service.cs`: the Chapter 3 record turned into an entity class with setters and an `IsActive` flag. 134
Listing 5.5 `Models/Customer.cs`: a customer and the navigation property to the customer's orders. 134
Listing 5.6 `Models/Order.cs`: an order with its foreign keys, status, timestamps and payments; `Paid` and `Balance` implement the balance formula. 135
Listing 5.7 `Models/Payment.cs`: one payment towards one order. 135
Listing 5.8 `Data/LaundryDbContext.cs`: the five tables and the rules the conventions cannot guess 136
Listing 5.9 `Data/SeedData.cs`: services and outlets from the Chapter 3 CSV files, then sample customers, orders and payments. 138
Listing 5.10 `Data/EfCatalog.cs`: the Chapter 3 catalog interface answered by EF Core 140
Listing 5.11 `Program.cs`: the Chapter 3 pipeline with the DbContext registered and the catalog swapped. 140

Listing 5.12 `Controllers/CustomersController.cs`: complete CRUD for customers with EF Core. 143
Listing 5.13 `Views/Customers/Index.cshtml`: the customer list with a search box. 148
Listing 5.14 `Views/Customers/_Form.cshtml`: the fields shared by the Create and Edit pages. 148
Listing 5.15 `Views/Customers/Details.cshtml`: one customer and the customer's orders, newest first. 148
Listing 5.16 `Views/Customers/Create.cshtml`: the new-customer page around the shared form. 149
Listing 5.17 `Controllers/OrdersController.cs`: the order list, order pages, new orders priced by Chapter 3's calculator, the workflow and payments. 150
Listing 5.18 `Views/Orders/Details.cshtml`: one order, its workflow button, its payments and the payment form. 153
Listing 5.19 `Views/Orders/Create.cshtml`: the counter form for a new order. 155
Listing 5.20 `Controllers/PricesController.cs`: the owner's price list, with automatic service codes. 155
Listing 5.21 `Controllers/CounterController.cs`: four numbers for the counter, each one SQL statement. 157
Listing 5.22 Exercise 5.3: the three best customers, as lines to add to `LinqDemo/Program.cs`. 161
Listing 5.23 Exercise 5.5: an action to add to `OrdersController` that removes one payment of one order. 161
Listing 6.1 `Models/Customer.cs` with validation attributes: a required name of 2 to 80 characters and a required Indonesian mobile number. 165
Listing 6.2 `Models/Service.cs` with rules for the description and the price. 166
Listing 6.3 `Models/IndonesianMobileAttribute.cs`: a validation attribute for Indonesian mobile numbers, with a helper that stores every number in one spelling. 167
Listing 6.4 `ValidationDemo/Program.cs`: the application's own validation rules and Identity's password hasher, run in a console. 168
Listing 6.5 `Models/OrderForm.cs`: exactly what the new-order form may send, with its rules, including one that involves two fields. 171
Listing 6.6 `Models/PaymentForm.cs`: the payment form and its rules. 172
Listing 6.7 `Controllers/OrdersController.cs` with `OrderForm` and `PaymentForm`. 172
Listing 6.8 `Views/Orders/Create.cshtml` bound to `OrderForm`. 176
Listing 6.9 `Data/LaundryDbContext.cs` deriving from `IdentityDbContext`, so that users and roles live in the same database as the orders. 177
Listing 6.10 `Program.cs`: Identity, the fallback authorization policy, and the settings that deployment needs. 179
Listing 6.11 `Models/AppRoles.cs`: the two roles of the shop. 182
Listing 6.12 `Data/IdentitySeeder.cs`: migrations, roles and the first accounts at start-up. 183
Listing 6.13 `appsettings.Development.json`: demo accounts for the developer's computer only. 184
Listing 6.14 `Controllers/AccountController.cs`: sign in, sign out and the refusal page. 184
Listing 6.15 `Views/Account/Login.cshtml`: the staff sign-in form. 185
Listing 6.16 `Views/Account/AccessDenied.cshtml`: what a signed-in user sees when a page is not for their role. 186
Listing 6.17 `appsettings.Production.json`: the database and the cookie keys in a fixed folder, and quieter logging. 187
Listing 6.18 `deploy/laundrymvc.service`: the systemd unit that keeps the app running. 189
Listing 6.19 `deploy/nginx-laundrymvc.conf`: Nginx forwarding to Kestrel with the forwarded headers. 189
Listing 6.20 Exercise 6.2: `Models/NotInFutureAttribute.cs`, a validation attribute of our own for dates. 194
Listing 6.21 Exercise 6.3: a form class with one field and the action that uses it, to add to `OrdersController`. 195
Listing 7.1 `2026_09_01_000001_create_customers_table.php`: the customers table. 200
Listing 7.2 `2026_09_01_000002_create_services_table.php`: services and outlets, with their codes as primary keys. 201

Listing 7.3 `2026_09_01_000003_create_orders_table.php`: orders with their foreign keys, status and timestamps. 202
Listing 7.4 `2026_09_01_000004_create_payments_table.php`: payments, deleted together with their order. 203
Listing 7.5 `app/Enums/OrderStatus.php`: the three statuses of an order and the workflow between them. 204
Listing 7.6 `app/Enums/PaymentMethod.php`: cash, QRIS and transfer. 204
Listing 7.7 `app/Support/Money.php`: rupiah formatted the Indonesian way. 205
Listing 7.8 `tools/enum_demo.php`: the enums and the money helper, run with plain PHP. 205
Listing 7.9 `app/Models/Customer.php`: a customer and the relationship to the customer's orders. 206
Listing 7.10 `app/Models/Service.php`: a string key, casts, and three computed attributes. 207
Listing 7.11 `app/Models/Order.php`: casts, relationships, and the balance of an order. 208
Listing 7.12 `app/Models/Payment.php`: a payment and its order. 209
Listing 7.13 `app/Models/Outlet.php`: the three outlets of Chapter 3. 210
Listing 7.14 `database/seeders/DatabaseSeeder.php`: services and outlets from the Chapter 3 CSV files, then sample customers, orders and payments. 211
Listing 7.15 `routes/web.php`: the public pages, the staff pages, and two extra actions on orders. 213
Listing 7.16 `app/Http/Controllers/CustomerController.php`: complete CRUD for customers with Eloquent. 215
Listing 7.17 `app/Http/Controllers/OrderController.php`: the order list, order pages, new orders priced by the rules of Chapter 3, and the workflow. 218
Listing 7.18 `app/Http/Controllers/PaymentController.php`: recording a payment for an order. 220
Listing 7.19 `app/Http/Controllers/PriceController.php`: the owner's price list. 220
Listing 7.20 `app/Http/Controllers/CounterController.php`: four numbers for the counter. 222
Listing 7.21 `resources/views/layouts/app.blade.php`: the layout shared by every page. 223
Listing 7.22 `resources/views/customers/_form.blade.php`: the fields shared by the create and edit pages. 224
Listing 7.23 `resources/views/customers/edit.blade.php`: the edit form with its CSRF token and method field. 224
Listing 7.24 `resources/views/customers/show.blade.php`: one customer, the orders, and the delete button. 224
Listing 7.25 `resources/views/orders/show.blade.php`: one order with its workflow button, payments and payment form. 225
Listing 7.26 Exercise 7.3: a price-quote route and controller method (add `use Illuminate\Http\Request;` to the controller) 232
Listing 7.27 Exercise 7.5: the `up` and `down` methods of the migration that adds a rack column. 232
Listing 8.1 `Models/PriceCalculator.cs`: the price rules of Chapter 3 in C#, with `decimal` arithmetic. 237
Listing 8.2 `CoreCheck/Program.cs`: the price rules and the workflow on fixed cases, in C#. 237
Listing 8.3 `app/Domain/PriceCalculator.php`: the same rules in PHP, with exact integer arithmetic. 239
Listing 8.4 `tools/core_check.php`: the same cases as `CoreCheck`, in plain PHP. 239
Listing 8.5 `tools/wrong_price.php`: the price computed with floats, three ways. 240
Listing 8.6 `Domain/OrderWorkflow.cs`: the workflow and the new rule, as a plain class. 242
Listing 8.7 `app/Domain/OrderWorkflow.php`: the workflow and the new rule in PHP. 242
Listing 8.8 `Models/OrderRow.cs`: exactly the columns of the order list. 244
Listing 8.9 `Domain/PagedList.cs`: one page of an ordered query and what the pager needs. 244
Listing 8.10 `Controllers/OrdersController.cs`: the order list as a projection with pages, the workflow of Problem 8.2, and the payment check of Problem 8.5. 245
Listing 8.11 `Views/Orders/Index.cshtml`: the order list from `OrderRow` records, with the pager. 249
Listing 8.12 `app/Http/Controllers/OrderController.php`: the order list with pages and sums, prices from Problem 8.1, and the workflow of Problem 8.2. 250

Listing 8.13 `Controllers/CustomersController.cs`: a case-insensitive search with pages. 254
Listing 8.14 `Views/Shared/_Pager.cshtml`: one pager for every list, keeping the query string. 256
Listing 8.15 `app/Http/Controllers/CustomerController.php`: the same search with pages in Laravel. 257
Listing 8.16 `resources/views/_pager.blade.php`: the pager in Blade. 259
Listing 8.17 `app/Http/Controllers/PaymentController.php`: a payment limited to the balance. 260
Listing 8.18 `Models/DailyRevenueRow.cs`: one line of the report. 262
Listing 8.19 `Controllers/ReportsController.cs`: money received per day and method, grouped in the database. 262
Listing 8.20 `Views/Reports/Daily.cshtml`: the daily revenue report. 263
Listing 8.21 `app/Http/Controllers/ReportController.php`: the same report with the query builder and collections. 264
Listing 8.22 `resources/views/reports/daily.blade.php`: the report in Blade. 265
Listing 8.23 `Models/OrderForm.cs` with the rack and the two-decimal rule. 267
Listing 8.24 `2026_09_20_000001_add_rack_to_orders_table.php`: the rack column, added and removed. 269
Listing 9.1 `2026_09_27_000001_add_outlet_id_to_orders_table.php`: a new column with a foreign key, and the old orders filled in. 279
Listing 9.2 `app/Models/Order.php`: the Chapter 8 model with `HasFactory` and the new relationship to the outlet. 281
Listing 9.3 `database/factories/CustomerFactory.php`: one invented customer. 283
Listing 9.4 `database/factories/OrderFactory.php`: one invented order, priced by the real rule, with states for finished and departed orders. 283
Listing 9.5 `database/seeders/PriceListSeeder.php`: the real price list and outlets, safe to run again. 285
Listing 9.6 `database/seeders/BusyMonthSeeder.php`: forty invented customers and a month of orders that move through the workflow as real ones do. 286
Listing 9.7 `database/seeders/DatabaseSeeder.php`: real data always, made-up data only on a developer's or a test computer. 287
Listing 9.8 `tools/pipeline_demo.php`: a middleware pipeline in plain PHP, the same idea that Laravel's pipeline is built on. 288
Listing 9.9 `app/Http/Middleware/EnsureOrderIsOpen.php`: nothing may change an order that has left the shop. 290
Listing 9.10 `app/Http/Middleware/AddServerTiming.php`: an "after" middleware that measures the rest of the pipeline. 291
Listing 9.11 `bootstrap/app.php`: `AddServerTiming` joins the `web` group, and `EnsureOrderIsOpen` gets the short name `order.open`. 291
Listing 9.12 `routes/web.php`: the routes of Chapter 8, with the routes that change an order grouped behind `order.open`. 292
Listing 9.13 `app/Http/Requests/StoreOrderRequest.php`: the checklist for a new order, with tidying, rules, messages and field names. 294
Listing 9.14 `app/Http/Requests/StorePaymentRequest.php`: simple rules first, then a check against what is still owed. 295
Listing 9.15 `app/Http/Controllers/PaymentController.php`: by the time `store` runs, the request has been checked. 296
Listing 9.16 `app/Http/Controllers/OrderController.php`: the Chapter 8 controller with `StoreOrderRequest` in `store` and the outlets in `create`. 297
Listing 9.17 `2026_10_01_000001_add_promised_at_to_orders_table.php`: a new column filled row by row, in chunks. 303
Listing 9.18 `app/Http/Middleware/LogSlowRequests.php`: a route middleware with a parameter. 304
Listing 10.1 `app/Enums/Role.php`: the two roles of the shop. 309

Listing 10.2 `2026_10_04_000001_add_role_to_users_table.php`: every user gets a role, cashier unless told otherwise. 310
Listing 10.3 `app/Models/User.php`: a user with a role, a hashed password and API tokens. 310
Listing 10.4 `database/factories/UserFactory.php`: cashiers by default, and an `owner()` state. 311
Listing 10.5 `database/seeders/DatabaseSeeder.php`: two demo accounts, on developers' computers only. 312
Listing 10.6 `app/Http/Controllers/Auth/LoginController.php`: show the form, sign in, sign out. 313
Listing 10.7 `routes/web.php`: public pages, guest-only sign-in, staff pages behind `auth`, and the price list for the owner. 314
Listing 10.8 `bootstrap/app.php`: the API routes, timing for both groups, and the sign-in page for guests. 316
Listing 10.9 `app/Providers/AppServiceProvider.php`: one gate, `manage-prices`, for the owner only. 316
Listing 10.10 `app/Http/Controllers/Api/TokenController.php`: a password in, a token out; and a way to give it back. 318
Listing 10.11 `routes/api.php`: tokens, customers, orders and payments, behind `auth:sanctum`. 319
Listing 10.12 `app/Actions/CreateOrder.php`: the one place where checked input becomes a priced order. 320
Listing 10.13 `app/Http/Controllers/Api/OrderController.php`: list, create, show and move orders, answering with resources. 321
Listing 10.14 `app/Http/Requests/UpdateOrderStatusRequest.php`: the wanted status must be the one the workflow allows next. 323
Listing 10.15 `app/Http/Controllers/Api/PaymentController.php`: record a payment, answer with the updated order. 325
Listing 10.16 `app/Http/Requests/StoreCustomerRequest.php`: a new customer, with the phone reduced to digits and checked for duplicates. 325
Listing 10.17 `app/Http/Resources/OrderResource.php`: the JSON shape of an order. 327
Listing 10.18 `app/Http/Resources/ServiceResource.php`: a line of the price list, as a phone needs it. 328
Listing 10.19 `tests/Unit/PriceCalculatorTest.php`: six price cases, one test method. 328
Listing 10.20 `tests/Feature/AuthTest.php`: guests, sign-in, wrong passwords, the owner's price list and API tokens. 329
Listing 10.21 `tests/Feature/OrderApiTest.php`: prices, validation, the workflow, closed orders, payments and filters, through the API. 331
Listing 10.22 `tests/Feature/PriceListAccessTest.php`: the price list is the owner's, on every route. 337
Listing 11.1 `pom.xml`: Spring Boot 4.1.1 as the parent, Java 25, and five dependencies. 343
Listing 11.2 `LaundryApplication.java`: the whole application starts here. 345
Listing 11.3 `application.properties`: an in-memory database, snake_case JSON and problem details for errors. 345
Listing 11.4 `Customer.java`: a customer, with a unique phone number. 346
Listing 11.5 `Service.java`: a line of the price list, keyed by its own code, with the name and unit worked out from the description. 348
Listing 11.6 `Order.java`: an order, priced when it is created, moved along by the shop's workflow. 349
Listing 11.7 `Payment.java`: money received for an order; an order may be paid in parts. 352
Listing 11.8 `OrderStatus.java`: three statuses, each with the word that the database and the JSON use. 354
Listing 11.9 `OrderStatusConverter.java`: stores `queued`, not `QUEUED`, in every status column. 354
Listing 11.10 `LaundryRules.java`: the price rule and the workflow, with the same whole-number arithmetic as the PHP version. 355
Listing 11.11 `CustomerRepository.java`: every standard operation for customers, plus two methods derived from their names. 356

Listing 11.12 `OrderRepository.java`: pages of orders, filtered by status, and one order with everything it needs. 357
Listing 11.13 `OrderView.java`: the JSON of one order, built from the entity. 358
Listing 11.14 `PageView.java`: a page as `data` and `meta`, counting pages from 1 as Laravel does. 358
Listing 11.15 `CreateOrderRequest.java`: the fields of a new order and their rules, including one rule about two fields. 359
Listing 11.16 `ServiceController.java`: the active price list, and one service by its code. 360
Listing 11.17 `OrderController.java`: list, show, create and move orders, each inside a transaction. 360
Listing 11.18 `DemoData.java`: the real price list from `Service.csv`, and the three customers of Chapters 5 and 7 with their orders. 364
Listing 11.19 `LaundryRulesTest.java`: the price rule and the workflow, tested without Spring. 366
Listing 11.20 `CustomerOrdersController.java`: a customer's orders, newest first, or 404. 370
Listing 12.1 `Api/ServiceController.php` (Laravel): the active price list as a resource collection. 376
Listing 12.2 `routes/api.php` (Laravel): the routes of Chapter 10, the price list and the daily report. 377
Listing 12.3 `DataView.java` (Spring): the contract's envelope in one line. 378
Listing 12.4 `ServiceController.java` (Spring): the same list, now in the envelope. 378
Listing 12.5 `OrderResource.php` (Laravel): every time in the contract's format. 379
Listing 12.6 `OrderView.java` (Spring): times with a fixed pattern, and the payments only when asked for. 380
Listing 12.7 `ApiValidationException.java` (Spring): a broken rule, with the messages per field. 382
Listing 12.8 `ApiExceptionHandler.java` (Spring): validation errors, broken rules, status exceptions and unreadable bodies, each in the contract's shape. 383
Listing 12.9 `CreateOrderRequest.java` (Spring): one-field rules only. 386
Listing 12.10 `OrderController.java` (Spring): every error at once, the workflow as 422, closed orders as 409, and payments. 386
Listing 12.11 `CreatePaymentRequest.java` (Spring): amount and method. 392
Listing 12.12 `PaymentView.java` (Spring): one payment, with its time in the contract's format. 393
Listing 12.13 `Api/ReportController.php` (Laravel): counts per status, the day's totals and payments, every status present. 394
Listing 12.14 `OrderRepository.java` (Spring): the queries of Chapter 11 plus the two the report needs. 395
Listing 12.15 `ReportController.java` (Spring): the same report, with a `LinkedHashMap` that lists every status in order. 396
Listing 12.16 `contract/check_api.php`: sixteen checks of the contract, against any server. 398
Listing 12.17 `tests/Feature/ContractTest.php` (Laravel): the contract's new points, with a fixed clock. 402
Listing 12.18 `OrderApiTest.java` (Spring): the price rule, every error at once, the whole life of an order and the report, through MockMvc. 404
Listing 13.1 `RuleException.java`: input that breaks a rule, with its messages per field. 412
Listing 13.2 `NotFoundException.java`: a record that does not exist. 413
Listing 13.3 `ConflictException.java`: a request that the record's present state does not allow. 413
Listing 13.4 `Customer.java`: Chapter 11's customer with one new method, `change` 414
Listing 13.5 `CustomerRepository.java`: queries derived from method names. 415
Listing 13.6 `CustomerService.java`: search, create, update and delete customers, inside transactions. 415
Listing 13.7 `LaundryApplication.java`: the entry point, and the shop's clock in Western Indonesian Time. 418
Listing 13.8 `OrderRepository.java`: pages, one order with its details, and the report queries. 418
Listing 13.9 `OrderService.java`: taking orders, moving them along and taking money, with every rule in one place. 419
Listing 13.10 `ReportService.java`: the figures of one day. 424
Listing 13.11 `OrderApiController.java`: read JSON, call `OrderService`, write JSON. 425

Listing 13.12 `CustomerApiController.java`: customers in JSON. 428
Listing 13.13 `ServiceApiController.java`: the price list in JSON. 428
Listing 13.14 `ReportApiController.java`: the daily report in JSON. 429
Listing 13.15 `ApiExceptionHandler.java`: every API error in the contract's shape. 430
Listing 13.16 `CustomerForm.java`: the customer form, with the same limits as the API's request. 433
Listing 13.17 `OrderForm.java`: the new-order form, which knows how to become a `NewOrder`. 434
Listing 13.18 `PaymentForm.java`: the payment form. 434
Listing 13.19 `Forms.java`: the service's rule messages become errors on form fields. 435
Listing 13.20 `CustomerController.java`: the full CRUD for customers as pages. 435
Listing 13.21 `fragments.html`: the page head, the navigation and the flash messages. 439
Listing 13.22 `customers/list.html`: the searchable customer list. 439
Listing 13.23 `customers/form.html`: one form for adding and editing, with messages next to each field. 440
Listing 13.24 `OrderController.java`: the queue, a new order, one order, status steps and payments. 441
Listing 13.25 `HomeController.java`: the front page is the queue. 445
Listing 13.26 `orders/list.html`: one page of orders, filtered by status. 445
Listing 13.27 `orders/form.html`: taking in a bag of laundry. 446
Listing 13.28 `orders/detail.html`: an order with its total, payments and balance, and the button for the next step. 446
Listing 13.29 `WebErrorAdvice.java`: every `NotFoundException` on a page becomes a 404 page. 448
Listing 13.30 `error/404.html`: the page for a record that does not exist. 448
Listing 13.31 `pom.xml`: Chapter 12's build plus Thymeleaf and SQLite; H2 stays for the tests. 449
Listing 13.32 `application.properties`: SQLite in a file, the schema kept up to date, snake_case JSON. 450
Listing 13.33 `src/test/resources/application.properties`: a fresh H2 database for every test run. 450
Listing 13.34 `OrderServiceTest.java`: the service layer with mocked repositories and a fixed clock. 452
Listing 14.1 `Role.java`: two roles are all the shop needs. 463
Listing 14.2 `Staff.java`: a person who may log in, with a password hash and a role. 463
Listing 14.3 `StaffRepository.java`: find a staff member by e-mail. 464
Listing 14.4 `StaffDetailsService.java`: from an e-mail to Spring Security's view of a user. 465
Listing 14.5 `DemoStaffSeeder.java`: the owner and one cashier, with hashed passwords. 465
Listing 14.6 `ApiToken.java`: one device's key to the API. 467
Listing 14.7 `ApiTokenRepository.java`: the standard methods are enough. 468
Listing 14.8 `TokenService.java`: issue, find and revoke tokens, keeping only hashes. 468
Listing 14.9 `BearerTokenFilter.java`: from a bearer token to an authenticated staff member. 471
Listing 14.10 `TokenApiController.java`: `POST /api/tokens` gives a token, `DELETE /api/tokens/current` takes it back. 473
Listing 14.11 `SecurityConfig.java`: one chain for the API, one for the pages. 474
Listing 14.12 `LoginController.java`: only the `GET`; Spring Security handles the `POST`. 478
Listing 14.13 `login.html`: e-mail and password, with messages for a failed login and a logout. 478
Listing 14.14 `fragments.html`: the navigation now ends with a logout form. 478
Listing 14.15 `CustomerService.java`: `delete` is for the owner, whoever calls it. 479
Listing 14.16 `CustomerApiController.java`: the new `DELETE /api/customers/{id}` 481
Listing 14.17 `pom.xml`: security, and one test starter per slice. 486
Listing 14.18 `CustomerApiControllerTest.java`: the web layer and the security rules, with the service mocked. 488
Listing 14.19 `OrderRepositoryTest.java`: the report queries and `existsByCustomerId` against H2. 490
Listing 14.20 `TokenFlowTest.java`: sign in, use the token, sign out, and see the token refused. 491

Listing 15.1 `PerfLab/Program.cs`: the quote endpoint as an ASP.NET Core minimal API. 502
Listing 15.2 `routes/api.php`: the quote endpoint as a Laravel route. 502
Listing 15.3 `QuoteController.java`: the quote endpoint as a Spring Boot controller. 503
Listing 15.4 `LoadGen/Program.cs`: a load generator reporting throughput and latency percentiles. 505
Listing 16.1 `check_api.php`: 25 checks of the contract, for any of the three servers. 519
Listing 16.2 `LaundryCapstone.csproj`: the `UseEf` switch decides whether EF Core is compiled in. 526
Listing 16.3 `LaundryRules.cs`: price and workflow in integer arithmetic, as in Chapters 8 and 11. 527
Listing 16.4 `ILaundryStore.cs`: everything the application needs from storage. 529
Listing 16.5 `LaundryService.cs`: customers, orders, status steps, payments and the daily figures, with every rule in one place. 529
Listing 16.6 `Exceptions.cs`: a broken rule, a missing record, a conflicting state. 534
Listing 16.7 `Dtos.cs`: request records with validation attributes, and the views in the contract's shape. 534
Listing 16.8 `OrdersController.cs`: read JSON, call `LaundryService`, write JSON. 537
Listing 16.9 `ApiErrors.cs`: every error in the contract's shape. 538
Listing 16.10 `CustomersController.cs`: the phone kept as digits, and `DELETE` for the owner only. 540
Listing 16.11 `BearerTokenHandler.cs`: from a bearer token to a signed-in staff member, with the contract's 401 and 403. 541
Listing 16.12 `TokenService.cs`: Sanctum-style tokens, hashed with SHA-256 and compared in constant time. 542
Listing 16.13 `Program.cs`: storage, JSON, errors, security and the start-up data, in order. 544
Listing 16.14 `BoardController.cs`: the queue and the finished orders, without a login. 547
Listing 16.15 `Views/Board/Index.cshtml`: order numbers and first names, refreshed every 30 seconds. 547
Listing 16.16 `routes/api.php`: the new `DELETE` route, for the owner only. 549
Listing 16.17 `AppServiceProvider.php`: the gates of Chapter 10 and the new `delete-customers`. 550
Listing 16.18 `CustomerController.php`: `destroy` refuses a customer with orders with 409. 551
Listing 16.19 `routes/web.php`: the board is a public route. 552
Listing 16.20 `BoardController.php`: the queue and the finished orders. 553
Listing 16.21 `board.blade.php`: the same board as Razor's, in Blade. 554
Listing 16.22 `CapstoneTest.php`: a cashier's delete, the owner's two deletes, and the board's privacy. 555
Listing 16.23 `BoardController.java`: the board reads its orders through `OrderService`. 556
Listing 16.24 `board.html`: the same board in Thymeleaf. 557

List of Equations

(5.1) 129
(8.1) 236
(8.2) 236
(12.1) 375
(12.2) 393
(13.1) 424
(13.2) 424
(14.1) 463
(14.2) 466
(15.1) 502
(15.2) 506

(16.1).....518
(16.2).....518

Preface

Every web application answers the same question thousands of times a day: a browser asks for an address, and the program must find the right piece of code, collect the right data, turn it into a page or an answer and send it back — correctly, safely, and quickly. You can write all of that yourself, and the first chapter of this book does, once, so that you see how much of it there is. A framework is the decision not to write it again: somebody has already built the routing, the database access, the forms, the sign-in and the tests, and you fill in only what is particular to your application. Learning to work *with* such a framework — to know where each piece of your code belongs, what the framework does for you, and what it still leaves to you — is what framework-based programming is, and it is the whole subject of this book. Keep that sentence in mind and nothing in these pages will feel arbitrary.

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

Who this book is for

You are assumed to know basic web programming: what HTML and a form are, what happens when a browser sends a request and a server answers it, and how to write a small program with variables, loops, functions and classes in at least one language. You are *not* assumed to know C#, PHP or Java, any framework, or how to set one up. Every term is introduced in plain words before it is used, every framework is installed step by step on Windows, macOS and Linux, and every idea is shown working before it is explained in general.

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

One app, three frameworks

A framework is a way of organizing a program, not a feature of one language, and this book treats it that way. It teaches three of the most widely used server-side frameworks, each in its own language: **ASP.NET Core** with C#, **Laravel** with PHP, and **Spring Boot** with Java. And it builds the same application in all three: **Profitina Laundry**, a laundry business in Surabaya with its customers, its price list, its orders and its payments. The same four tables, the same rules about prices and order status, and in the end the same JSON API, checked by the same test script, appear three times, in three vocabularies. Seeing one application written three ways is the quickest route to understanding the ideas that every framework shares — Model–View–Controller, routing, object-relational mapping, dependency injection and templates — rather than the habits of one tool, so that whichever framework your next project or job uses, you will already know what lies underneath it.

What is in it

Chapters 1 to 3 build the ground: why frameworks exist, shown by writing the same small price list four times — without a framework in C#, PHP and Java, and then with ASP.NET Core — with a full set-up of VS Code, the .NET SDK, PHP and Composer, a JDK and Maven, and a database on Windows, macOS and Linux; the foundations shared by modern frameworks — MVC, routing, ORM, dependency injection and templates — built by hand in a tiny framework of our own, MiniFrame, and then found again in the three real ones; and a first real ASP.NET Core MVC application: the project, `Program.cs` with its services and its request pipeline, the model, controllers and routing, and Razor views. Chapters 5 to 7 give the application a memory and a second language: Entity Framework Core, LINQ, migrations and real CRUD for customers, orders, payments and the price list; validation, form classes, ASP.NET Core Identity for sign-in, and deployment from a laptop to a server; and the same shop in Laravel, with Artisan, migrations written in PHP, Eloquent models, routing, controllers and Blade templates. Chapters 9 to 11 take the application further and to a third framework: Laravel migrations in depth, factories and seeders, middleware and form requests; users and roles, sign-in, an API for programs with API resources, and testing; and Spring Boot, with Initializr, entities, Spring Data JPA repositories and REST controllers. Chapters 13 to 15 finish the work: a layered Spring Boot application with a service layer, validation, Thymeleaf pages and one place for errors; Spring Security with tokens for the API, REST API design, and automated tests on four levels; and a comparison of the three frameworks — performance, productivity, ecosystem and community, support calendars and security by default — ending in a way to choose among them.

Chapters 4, 8, 12 and 16 are **Evaluation and Discussion** chapters. Each closes one quarter of the book by solving complete problems in full — every problem restated, designed, programmed, tested, run where it could be run, and explained — with the tempting wrong solution shown and taken apart. They range from a showcase site with order tracking, a route table written by hand, a response-time header and a maintenance mode, through one price rule and one order workflow solved in both ASP.NET Core and Laravel, lists without the N+1 problem, a search that finds its customer and payments that cannot overpay, to one REST API contract — one way to write a time, one shape for every error, orders that move and close, a daily report — met by Laravel and Spring Boot alike, and finally the capstone: Profitina Laundry complete in all three frameworks, compared side by side, and a decision.

Throughout, one small business grows chapter by chapter: **Profitina Laundry**, a laundry that began with one washing machine, two dryers and a notebook, and that grows to three outlets across Surabaya. Its old website was twenty files that nobody dared to touch; its wish was modest — *one change should be made in one place* — and the book grants it one framework at a time: a public site with its services and a price quote, a counter application with a real database, staff who must sign in, an API for the drivers' phone app, and at last the same shop in three frameworks, with the people who run it choosing between them. Each chapter also has a *Using It for Real* section that shows how the same ideas carry over to other small businesses and everyday problems, and several have a *Profitina in Practice* section that shows them at work in a real product.

How the code was checked

Every program in this book is collected, chapter by chapter, in the accompanying `Code` folder, each chapter with its `aspnet`, `laravel` or `spring` projects (only the frameworks the chapter uses), and a

README that explains how to build and run each one in a terminal and in VS Code on Windows, macOS and Linux. Every listing in the book is the file in that folder, character for character.

We should be honest about how far that code was run. The machine on which this book was written had the .NET 10 SDK, PHP and a JDK, but it could not download packages from NuGet, Packagist or Maven Central. So everything that needs only .NET itself — the programs of Chapters 1 to 4, MiniFrame, the small demonstrations of later chapters, and the capstone's ASP.NET Core application in its in-memory build — was built and run there, and its output and screenshots in the book are real. The Entity Framework Core, Laravel and Spring Boot applications could not be built there. Their code was written against the current official documentation, every PHP file was checked with `php -l`, every Java file was compiled with `javac` against stand-ins for the library types, the shared business rules were run in plain PHP and Java, and the API contract checker of Chapters 12 and 16 was run against a reference server. Where the book shows the output of these applications, it is marked as expected output, and no screenshot of them has been invented; our own diagrams take their place. Each chapter's README says exactly what was run and what was not, and names the first command you should run yourself — `dotnet build`, `composer install` with `php artisan test`, or `mvn test` — because on your machine, with the packages downloaded, that command is the real check.

How to read it

With an editor open and the framework installed. Type the examples in rather than copying them; the typing is where the errors happen, and the errors are where the learning happens. Before you run a request, predict what the framework will do with it — which route matches, which controller runs, what the database is asked, which view answers — and then run it and compare. Break each example deliberately once it works, and read what the error page or the log says. Read the three frameworks side by side as you reach them: when something looks new in Laravel or Spring Boot, ask where you met it in ASP.NET Core. 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 all live links: in the electronic edition, click "Listing 5.3" or "[4]" anywhere in the text and you are taken straight to it. The contents and the lists at the front of the book work the same way.

About the author

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

Acknowledgments

To the students who worked through the earlier forms of this material and asked the questions that showed where an explanation was still thin, and to the colleagues at ITS with whom the course has been taught and refined over the years. The book owes its standard, too, to the official documentation of ASP.NET Core, Entity Framework Core, Laravel, Spring Boot and Spring Security, and to the communities that maintain them, which are cited wherever they are used.

Surabaya, September 2026

Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

From Web Programming to Frameworks

Every web application answers the same question thousands of times a day: a browser asks for an address, and the program must find the right piece of code, collect the right data, turn it into a page and send it back without a mistake. You can write all of that yourself, and a first course in web programming rightly makes you do it once. This chapter shows, with one small program written four times, what changes when a *framework* does most of that work for you. By the end of the chapter you will be able to explain what a framework is and how it differs from a library; describe the Model–View–Controller pattern that ASP.NET Core, Laravel and Spring Boot share; say why this book uses exactly these three; install the whole workbench (VS Code, the .NET 10 SDK, PHP 8.5 with Composer, JDK 25 with Maven, SQLite and MySQL) on Windows, macOS or Linux; build and run the price list of Profitina Laundry by hand in C#, PHP and Java; build it again with ASP.NET Core in a fraction of the code; and list precisely which jobs moved from your code into the framework.

1.1 Bu Wati's Price List

Profitina Laundry opened its first counter in Rungkut, Surabaya, in January 2019. Bu Wati Wijaya, the owner, started with one washing machine, two dryers, a scale that faces the customer and a notebook for the orders. Today there are three outlets (Rungkut, Keputih and, since January 2026, Wiyung), fourteen people, twelve services from "wash, dry and fold" at Rp 8.000 per kilogram to "shoes deep clean" at Rp 45.000 per pair, and close to three thousand orders a year.

Some years ago a student who helped at the counter built the shop a small website: a home page, a page with the price list, a page per service. It was exactly the kind of site a first web programming course teaches you to build, and it worked. Every page was its own PHP file. Each file opened the database, ran a query, and printed HTML with `echo`. The header with the shop's name and the menu was copied into every file.

Then the shop grew, and the site began to hurt. When the Wiyung outlet opened, the new address had to be typed into five files; one was forgotten, and for two weeks the "contact" page sent customers to the wrong street. When the price of ironing went up, it changed on the price list but not on the service page, which had its own query. A customer typed a strange address into the browser, something like `/service.php?id=<script>…`, and the page printed it back into the HTML unescaped; nothing bad happened that time, but the student who looked at it went pale. And when Bu Wati asked for a simple thing, "can customers see whether their laundry is ready?", nobody dared to touch the code, because nobody could say which of the twenty files would break.

Bu Wati's wish, when she finally explained it, was modest: *one change should be made in one place*. A new address should be typed once. A price should live in one place. Every page should look the same without anyone copying the header. Every value printed into a page should be made safe automatically, not only when someone remembers. And a new feature should have an obvious home.

That wish is the whole reason frameworks exist. This book follows Profitina Laundry through three of them. The shop and its data stay the same from the first chapter to the last: customers, outlets, services with their prices, orders (the database calls them *bills*), payments and the status of each order. What changes is the framework, so that you can see what each one does for you and what it costs. This chapter starts with the smallest possible piece, the price list, and builds it four times.

1.2 The Idea in Plain Words

1.2.1 What a web application does for every request

Strip away the details and every request to every web application goes through the same eight steps. Picture a customer who opens `http://…/services/S0004` on her phone to see the price of dry-cleaning a shirt.

1. **Wait** for a connection and read the request: the method (`GET`), the path (`/services/S0004`), the headers.
2. **Route**: decide which piece of code is responsible for this path and method.
3. **Extract**: cut the variable parts out of the path (here the code `S0004`) and convert them to the right type.
4. **Fetch**: ask the data store for the service with that code.
5. **Decide**: apply the rules of the business (is there such a service? is the customer allowed to see it?).
6. **Render**: turn the data into HTML (or JSON for another program), escaping every value so that text from outside can never become code.
7. **Respond**: choose the status code (`200 OK`, `404 Not Found`, …), set the headers, send the bytes.
8. **Survive**: log what happened, and if anything went wrong, send a polite error page instead of crashing.

Only steps 4 and 5 are about *laundry*. Steps 1, 2, 3, 6, 7 and 8 are the same for a laundry, a clinic, a school or a bank. A program that does all eight by hand spends most of its lines on things that have nothing to do with its business, and those are exactly the lines where most bugs live: a wrong `if` in the routing, an id cut out of the path one character too early, a value printed without escaping.

1.2.2 Library or framework: who calls whom?

Programmers have always shared code. A **library** is a collection of ready-made routines that *your* program calls when it needs them: a function that parses a date, a class that reads a CSV file, a routine that encodes HTML. Your program is in charge; it decides the order of things and calls the library now and then.

A **framework** turns this around. It is an almost complete program with holes in it, and *you* fill the holes. The framework waits for requests, routes them, extracts the parameters, calls your code at the right moment, takes what your code returns, renders it and sends it. Ralph Johnson and Brian Foote described this in 1988, long before the web: a framework is a reusable design whose control flow belongs to the framework, while the application supplies the parts that differ [1]. The same idea is often summarized as the *Hollywood principle*: “don’t call us, we’ll call you” [2]. Its textbook name is **inversion of control**, because the control flow is inverted: the framework calls your code, instead of your code calling the library [3].

Figure 1.1 puts the two side by side. On the left, one program does all nine small jobs of a request itself and calls a library now and then; the amber jobs are the ones where the bugs of Bu Wati's old site came from. On the right, the framework does the plumbing and calls three small pieces of your code: a controller, a model and a view.

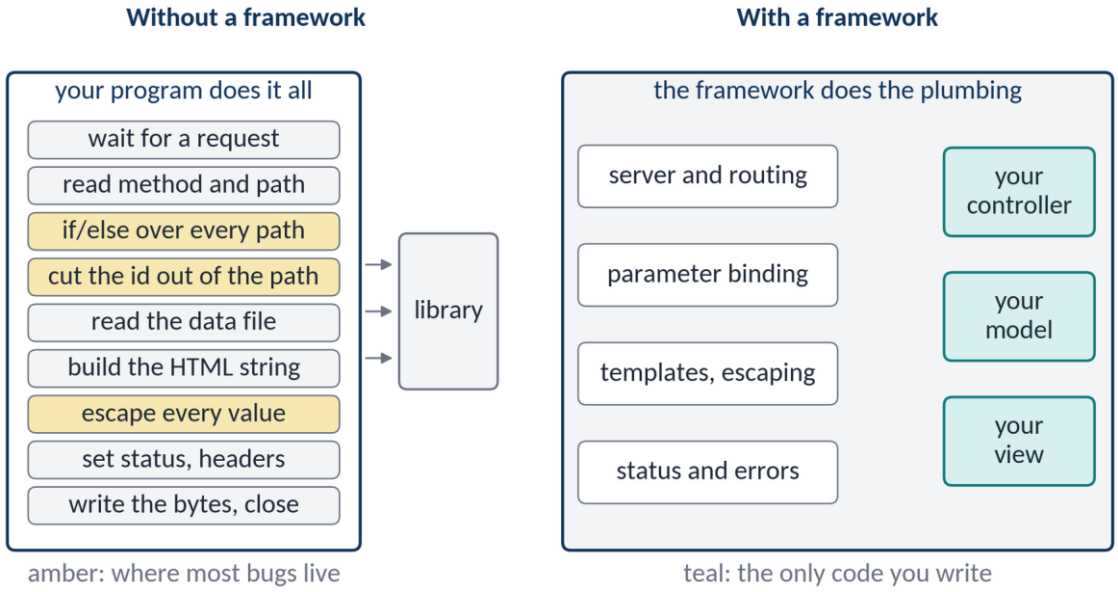


Figure 1.1 A library is called by your program; a framework calls your program. With a framework, the only code you write is the part that is about your business.

> **DEFINITION.** A *library* is code that your program calls. A *framework* is code that calls your program: it owns the main loop and the order of events, and it calls the pieces you supply (handlers, controllers, views) at the moments it decides. This reversal is called *inversion of control*. A *web framework* is a framework whose main loop is the request–response cycle of HTTP.

One consequence surprises beginners: in a framework project, you often cannot find `main`. In ASP.NET Core it is a short `Program.cs` that builds the application and calls `app.Run()`, after which the framework takes over; in Laravel it is `public/index.php`, which you never edit; in Spring Boot it is one line, `SpringApplication.run(...)`. Your code is spread over controllers and views that nothing in your own files seems to call. The framework calls them, following its *conventions*: a class named `ServicesController` in the folder `Controllers`, a view named `Details.cshtml` in the folder `Views/Services`. Learning a framework is, to a large extent, learning its conventions.

1.2.3 Model, View, Controller

Almost every server-side web framework organizes the holes it leaves for you in the same way, called **Model–View–Controller**, or MVC. The pattern was invented at Xerox PARC for the Smalltalk-80 user interface and first described in print by Krasner and Pope in 1988 [4]; web frameworks adapted it to the request–response cycle.

> **DEFINITION.** In a web application organized as *Model–View–Controller*, the **model** holds the data and the business rules (a service and its price, the rule "at least 3 kg"); the **view** turns data into what the user sees, usually an HTML template; and the **controller** receives one request, asks the model for what is needed, chooses a view and hands it the data. A **router**, part of the framework, decides which controller method (an *action*) handles each request.

Figure 1.2 follows the customer's request for `/services/S0004` through the five steps of MVC. The browser sends the request (1). The router finds that the method `Details` of the services

controller is responsible and calls it with `"S0004"` (2). The controller asks the model for the service with that code, and the model fetches it from the database (3). The controller hands the `Service` object to a view (4), and the view's HTML goes back to the browser with the status `200 OK` (5).

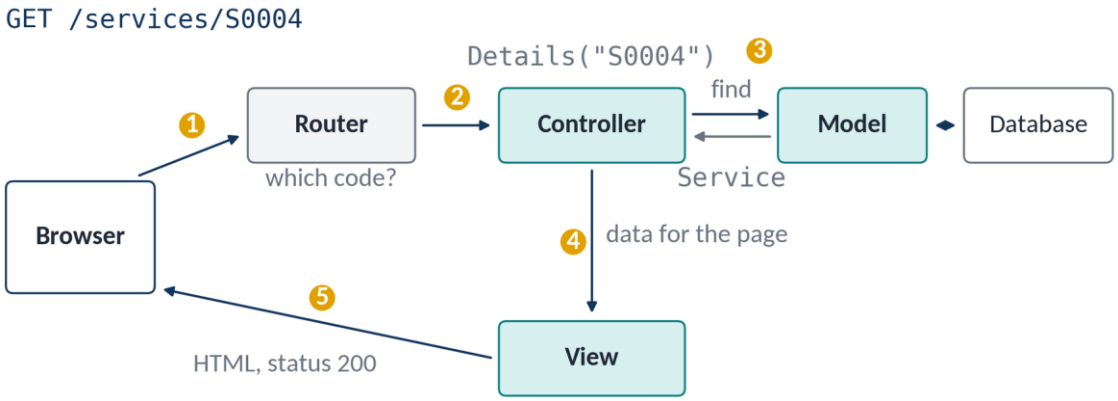


Figure 1.2 One request through Model–View–Controller: the router chooses the controller action, the controller asks the model and passes the result to a view, and the view's HTML is the response.

Why split a page into three parts? Because the three parts change for different reasons and at different speeds. The price of ironing is a model matter and should change in one place, whatever pages show it. The color of the header is a view matter; changing it must never risk a price. Which page answers which address is a controller matter. When each kind of change has one home, Bu Wati's wish, "one change in one place", comes true almost by itself.

> **FAT CONTROLLER, THIN MODEL.** The most common structural mistake in a first framework project is to put the business rules into the controller, because that is where the request arrives. A controller that computes prices, checks minimum weights and builds SQL strings cannot be reused by another page, by a mobile app's JSON endpoint or by a test. Keep controllers thin: receive the input, call one or two model methods, choose a view. Chapter 3 shows the difference with the price-quote rule of Profitina Laundry.

1.2.4 What a framework gives you, and what it costs

Table 1.1 lists the jobs that the three frameworks of this book take over, next to the chapter where you meet each one in practice. None of it is magic: every row is ordinary code that someone wrote once, very carefully, and that millions of applications now share.

Table 1.1 The jobs a web framework takes over, and the chapters of this book where each one is built for Profitina Laundry.

Job	What the framework does for you	In this book
Serving HTTP	A tested, fast web server that handles many requests at once	Ch. 1, 3
Routing	Maps a method and a URL pattern to one action, extracts parameters	Ch. 2, 3, 7, 11
Model binding	Converts query strings, form fields and JSON into typed objects	Ch. 3, 9, 13
Templates	Layouts, partial views and automatic HTML escaping	Ch. 2, 3, 7, 13
Data access	An ORM that maps tables to classes and writes the SQL	Ch. 5, 7, 11
Migrations	Versioned changes to the database schema, replayable anywhere	Ch. 5, 9, 11
Validation	Declarative rules on the model, error messages next to fields	Ch. 6, 9, 13
Security	Login, access rules, password hashing, CSRF tokens	Ch. 6, 10, 14
Configuration	Settings per environment from files and environment variables	Ch. 3, 4, 6
Testing	Test hosts and helpers that run requests without a browser	Ch. 10, 14

The costs are real too, and it is better to name them at the start. A framework must be *learned*: its conventions, its vocabulary, its way of doing things. It makes some things so easy that you stop understanding them, which is why this chapter and the next build the machinery by hand once. It brings *dependencies*, packages that must be downloaded, kept up to date and trusted. And it moves: ASP.NET Core ships a new major version every November, Laravel every spring, Spring Boot twice a year, and every new version asks for a little work. Section 1.3 gives the current versions and how long each is supported.

1.3 Why ASP.NET Core, Laravel and Spring Boot

There are dozens of web frameworks. This book teaches three, each in its own language, because together they cover most of the server-side jobs a graduate will meet, and because seeing the same ideas three times is the fastest way to tell the idea from the syntax.

ASP.NET Core is Microsoft's open-source, cross-platform web framework for C#. It runs on Windows, macOS and Linux, carries its own web server (Kestrel) and is the recommended way to build any new web application on .NET. Its long-term support (LTS) release is **.NET 10**, released in November 2025 and supported for three years; the September 8, 2026 update brought the runtime to 10.0.12 [5].

Laravel is the most widely used framework of PHP, the language still behind a large part of the web. Laravel 13 was released on March 17, 2026; it requires PHP 8.3 or newer and supports PHP 8.3, 8.4 and 8.5, with bug fixes until the third quarter of 2027 and security fixes until March 2028 [6]. PHP 8.5 itself was released on November 20, 2025; its latest update at the time of writing is 8.5.11 of September 24, 2026 [7].

Spring Boot is the standard way to build server applications in Java. It sits on the Spring Framework and adds auto-configuration, an embedded server and ready-made "starters". The current line is Spring Boot 4.1 (4.1.1 of August 20, 2026), which runs on Java 17 up to 26; the 4.0 line is supported until the end of 2026 [8]. This book uses Java 25, the long-term-support release of September 2025, whose latest update is 25.0.4.1 of August 18, 2026 [9].

How much are the three used? The largest open survey of developers, the Stack Overflow Developer Survey, asked in 2025 which web frameworks and technologies people had done extensive development work with in the past year. **Figure 1.3** shows the answers of the 19,460

professional developers who answered that question [10]. ASP.NET Core was used by 21.3%, Spring Boot by 15.6% and Laravel by 9.3%; the older ASP.NET (the .NET Framework predecessor of ASP.NET Core) was still in use by 15.0%. Node.js and React are at the top because they are used on the front end by nearly everyone, including many who also use one of our three on the server.

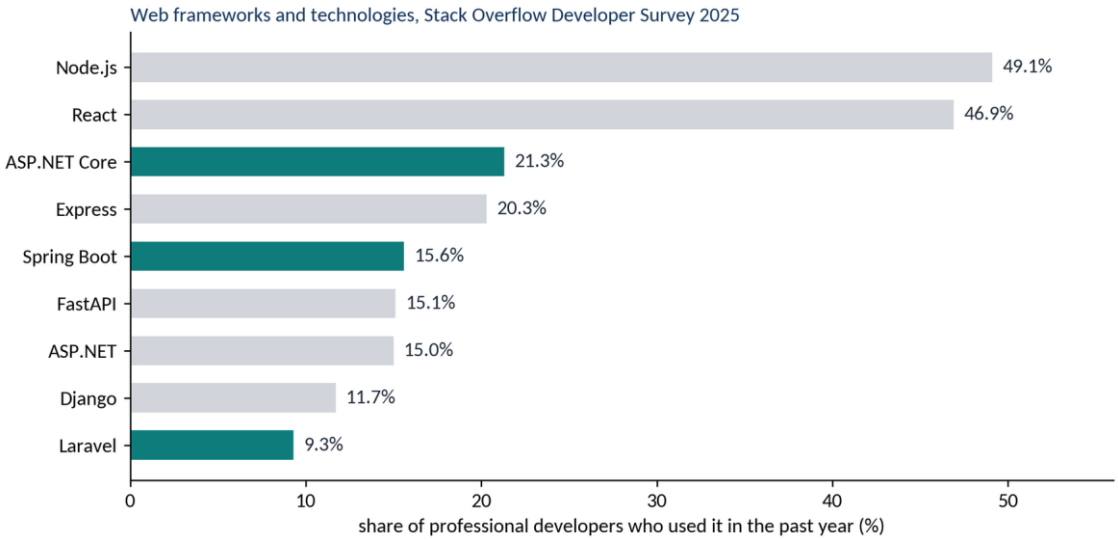


Figure 1.3 Web frameworks and technologies used by professional developers in the Stack Overflow Developer Survey 2025 (19,460 respondents to this question); the three frameworks of this book are in teal.

Treat such figures as a snapshot, not a ranking to memorize: surveys measure who answered, not the whole world, and they change every year. What matters for a beginner is that each of the three is mature, actively developed, well documented, and asked for by employers, and that each represents one of the three big families of server-side development: .NET, PHP and the Java Virtual Machine.

Table 1.2 compares the three at a glance. Every row will become familiar in the coming chapters.

Table 1.2 The three frameworks of this book side by side, with the versions current in September 2026.

	ASP.NET Core	Laravel	Spring Boot
Language	C# 14	PHP 8.5	Java 25
Current version (Sept. 2026)	10.0 LTS (runtime 10.0.12)	13.x	4.1.1
Created by	Microsoft, open source	Taylor Otwell, open source	VMware, now Broadcom, open source
Project tool	`dotnet` CLI	Composer, `laravel`, `php artisan`	Maven or Gradle, Spring Initializr
Packages from	NuGet	Packagist	Maven Central
Web server	Kestrel, built in	PHP's server, or Nginx/Apache with PHP-FPM	Tomcat, embedded
Routing	Attributes or central patterns	Central files `routes/web.php`	Annotations on methods
Templates	Razor (`.cshtml`)	Blade (`.blade.php`)	Thymeleaf (`.html`)
ORM	Entity Framework Core	Eloquent	Spring Data JPA (Hibernate)
Chapters	3–6	7–10	11–14

1.4 Setting Up the Workbench

This section installs everything the book needs. Do it once, calmly, and check each tool before moving on; most problems in the first weeks with a framework are installation problems in disguise. The versions are those current in September 2026; if a newer patch exists when you read this, take it.

1.4.1 Visual Studio Code

All three frameworks are used with the same editor in this book: **Visual Studio Code** (VS Code), free and available for Windows, macOS and Linux from https://code.visualstudio.com. On Windows run the installer and tick "Add to PATH"; on macOS drag the application to *Applications*, open it, press **Cmd+Shift+P** and run *Shell Command: Install 'code' command in PATH*; on Ubuntu or Debian install the `.deb` package, and on Fedora the `.rpm`. Afterwards the command `code .` opens the current folder.

VS Code learns each language through *extensions*. Open the Extensions view (**Ctrl+Shift+X**, on macOS **Cmd+Shift+X**) and install those of **Table 1.3**, or type the `code --install-extension` commands in a terminal.

Table 1.3 The VS Code extensions used in this book. Install only the ones for the framework you are working on if your computer is slow.

Extension (identifier)	For	What it adds
C# Dev Kit (`ms-dotnettools.csdevkit`)	ASP.NET Core	C# language support, Solution Explorer, run and debug [11]
PHP Intelephense (`bmewburn.vscode-intelephense-client`)	Laravel	PHP completion, go to definition, errors as you type
Laravel (`laravel.vscode-laravel`)	Laravel	Blade support, completion of routes, views and config [12]
Extension Pack for Java (`vscjava.vscode-java-pack`)	Spring Boot	Java language support, Maven, run and debug [13]
Spring Boot Extension Pack (`vmware.vscode-boot-dev-pack`)	Spring Boot	Spring Initializr in VS Code, Spring dashboard [13]
SQLite Viewer (`qwtel.sqlite-viewer`)	all	Opens a `.db` file as tables

1.4.2 The .NET 10 SDK

The **SDK** (software development kit) contains the C# compiler, the `dotnet` command, the project templates and both *runtimes*: the .NET runtime and the ASP.NET Core runtime, the latter also called the *shared framework*. Install the SDK, not only a runtime. **Table 1.4** has one line per system; the official download page lists the installers [14].

Table 1.4 Installing the three toolchains. Open a new terminal after each installation, so that the new commands are found.

System	.NET 10 SDK	PHP 8.5 + Composer	JDK 25 + Maven
Windows 10/11	`winget install Microsoft.DotNet.SDK.10`	php.new installer in PowerShell (below)	`winget install EclipseAdoptium.Temurin.25.JDK`, Maven zip from maven.apache.org
macOS	`.pkg` from dotnet.microsoft.com, or `brew install --cask dotnet-sdk`	php.new installer (below), or `brew install php composer`	`brew install openjdk@25 maven`
Ubuntu 24.04 and later	`sudo apt install dotnet-sdk-10.0`	php.new installer (below)	`sudo apt install openjdk-25-jdk maven`
Fedora	`sudo dnf install dotnet-sdk-10.0`	php.new installer (below)	`sudo dnf install java-25-openjdk-devel maven`

If your Linux release does not offer a `openjdk-25-jdk` package yet, install Eclipse Temurin 25 from https://adoptium.net, which publishes packages for Debian, Ubuntu, Fedora and others; if it does not offer `dotnet-sdk-10.0`, Microsoft's `dotnet-install.sh` script on the download page installs the SDK into your home folder [14].

1.4.3 PHP 8.5 and Composer

Laravel needs PHP, its package manager **Composer** and, for convenience, the **Laravel installer**. The Laravel 13 documentation recommends one command per system that installs all three from https://php.new [15]:

```
# macOS
/bin/bash -c "$(curl -fsSL https://php.new/install/mac/8.5)"
# Linux
/bin/bash -c "$(curl -fsSL https://php.new/install/linux/8.5)"
# Windows PowerShell, run as administrator
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object
System.Net.WebClient).DownloadString('https://php.new/install/windows/8.5'))
```

If PHP and Composer are already installed (for example from your web programming course), add only the installer with `composer global require laravel/installer` [15]. Composer can also be installed on its own from https://getcomposer.org [16]. On Windows, the Laravel team's free desktop application *Herd* installs PHP, Composer, the installer and a local web server in one step; it is a good choice if the command line still feels strange.

1.4.4 JDK 25 and Maven

Spring Boot needs a Java Development Kit and a build tool. This book uses **JDK 25** (any distribution: Eclipse Temurin, Oracle, Microsoft, Amazon Corretto; they are built from the same OpenJDK source) and **Apache Maven** 3.9, whose current release is 3.9.16 [17]. Maven is optional in practice, because every project generated by **Spring Initializr** (https://start.spring.io) contains a small script, `mvnw` (`mvnw.cmd` on Windows), that downloads the right Maven by itself [18]. After installing, make sure `java -version` names version 25; if an older Java answers, set `JAVA_HOME` to the new JDK's folder and put its `bin` folder first on the `PATH`.

1.4.5 Databases: SQLite and MySQL

Two databases are used in this book. **SQLite** is a whole SQL database in one file, with no server to install; it is the default database of a new Laravel 13 project [15], and ASP.NET Core and Spring Boot talk to it through a driver package. The current SQLite release line is 3.53, first released on April 9, 2026 [19]; you need no separate installation, because each framework's driver carries it, but the command-line tool `sqlite3` (from https://sqlite.org/download.html, or `brew install sqlite`, `sudo apt install sqlite3`) and the *SQLite Viewer* extension are handy for looking inside a database.

MySQL is the server database most hosting providers offer, and the one Chapters 6, 10 and 14 deploy to. Its current long-term-support release is **MySQL 9.7 LTS**, the first LTS line since 8.4 [20]; the community server is at https://dev.mysql.com/downloads/, and on Linux `sudo apt install mysql-server` works as well. MariaDB, a compatible fork, is equally fine for everything in this book.

1.4.6 Checking everything

Open a **new** terminal and ask each tool for its version.

```
code --version
dotnet --list-sdks
dotnet --list-runtimes
php --version
composer --version
java -version
mvn -version
```

On the machine that built this book the .NET, PHP, Composer, Java and Maven answers were as follows (the JAVA_TOOL_OPTIONS noise of that machine is left out).

```
10.0.112 [/usr/lib/dotnet/sdk]
Microsoft.AspNetCore.App 10.0.12 [/usr/lib/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.NETCore.App 10.0.12 [/usr/lib/dotnet/shared/Microsoft.NETCore.App]
PHP 8.4.21 (cli) (built: May  7 2026 18:44:14) (NTS)
Composer version 2.8.12 2025-09-19 13:41:59
openjdk version "21.0.10" 2026-01-20
Apache Maven 3.9.11 (3e54c93a704957b63ee3494413a2b544fd3d825b)
```

Two things are worth reading in this answer. First, `Microsoft.AspNetCore.App 10.0.12` is the ASP.NET Core **shared framework**: the whole of ASP.NET Core (the Kestrel server, routing, MVC, Razor, JSON) is installed with the SDK, so a web project needs nothing from the internet. Second, the build machine ran PHP 8.4 and Java 21, one version below what this chapter installs. Everything in Chapters 1 to 4 uses only features those versions share with PHP 8.5 and Java 25, so the outputs you will see are the same. The build machine also had one important limitation, which this book states openly wherever it matters: it **could not download packages** from NuGet, Packagist or Maven Central. Every program in Chapters 1 to 4 was therefore written to need only what the SDKs themselves contain, and every one of them was compiled and run; the output printed in these chapters is the real output. Laravel and Spring Boot themselves are packages, so their first projects (Section 1.6.3) could not be created on that machine; their commands are given, and Chapters 7 and 11 build them in full.

1.5 Building It Without a Framework

Before we let a framework do the work, we do it by hand once, so that we know exactly what we are giving away. The task is Bu Wati's price list in its smallest form, with four addresses:

- `/` answers with a short home page that links to the price list;
- `/services` answers with a table of the twelve services and their prices;
- `/services/S0004` answers with one service, or with `404 Not Found` if the code does not exist;
- every other address answers `404`, and every method except `GET` answers `405 Method Not Allowed`.

The data come from `data/Service.csv`, an export of the `Service` table of the Profitina Laundry database. Its first lines are shown below; note the quotes around descriptions that contain a comma.

```
id,description,price
S0001,"Wash, dry & fold (per kg)",8000.00
S0002,"Express wash & fold, 1 day (per kg)",12000.00
S0003,Ironing only (per kg),6000.00
S0004,Dry cleaning - shirt/blouse (per item),20000.00
```

1.5.1 In C#, with `HttpListener`

.NET has contained a small HTTP server class, `System.Net.HttpListener`, since long before ASP.NET Core [21]. It is not a framework: it accepts a connection and hands us the raw request, and everything else is ours. **Listing 1.1** is the whole program. It is a *console* project (`Sdk="Microsoft.NET.Sdk"`, not `.Web`), so ASP.NET Core is not involved at all.

Listing 1.1 `nofw-csharp/Program.cs`: the price list with no framework. The comments (1) to (6) mark the jobs a framework would do.

```
// Profitina Laundry's price list with NO framework: only HttpListener.
// Everything a framework would do is written out by hand, and marked.
using System.Globalization;
using System.Net;
using System.Text;

var services = LoadServices("../data/Service.csv");
var listener = new HttpListener();
listener.Prefixes.Add("http://127.0.0.1:4010/");
listener.Start();
Console.WriteLine("Listening on http://127.0.0.1:4010/ (Ctrl+C stops)");

while (true)
{
    var context = listener.GetContext();         // wait for ONE request
    string method = context.Request.HttpMethod;
    string path = context.Request.Url!.AbsolutePath;
    int status = 200;
    string body;

    // (1) Routing by hand: compare the path with every page we know.
    if (method != "GET")
    {
        status = 405;
        body = Page("Not allowed", "<p>This site only answers GET.</p>");
    }
    else if (path == "/")
    {
        body = Page("Profitina Laundry",
            "<p>See <a href=\"/services\">our services and prices</a>.</p>");
    }
    else if (path == "/services")
    {
        var rows = new StringBuilder();
        foreach (var s in services)
        {
            rows.Append($"<tr><td><a href=\"/services/{s.Id}\">{s.Id}</a>"
                + $"</td><td>{Escape(s.Description)}</td>"
                + $"<td>{Rp(s.Price)}</td></tr>");
        }
        body = Page("Our services", $"<table>{rows}</table>");
    }
    else if (path.StartsWith("/services/") && path.Count(c => c == '/') == 2)
    {
        // (2) Parameter extraction by hand: cut the id out of the path.
        string id = Uri.UnescapeDataString(path["/services/".Length..]);
```

Listing 1.1 (Continued) *nofw-csharp/Program.cs: the price list with no framework. The comments (1) to (6) mark the jobs a framework would do.*

```
        var found = services.FirstOrDefault(s => s.Id == id);
        if (found is null)
        {
            status = 404;
            body = Page("Not found", $"<p>No service {Escape(id)}.</p>");
        }
        else
        {
            body = Page(found.Id, $"<p>{Escape(found.Description)}: "
                + $"{Rp(found.Price)}</p>");
        }
    }
    else
    {
        status = 404;
        body = Page("Not found", $"<p>Nothing at {Escape(path)}.</p>");
    }

    // (3) The response by hand: status, header, bytes, close.
    byte[] bytes = Encoding.UTF8.GetBytes(body);
    context.Response.StatusCode = status;
    context.Response.ContentType = "text/html; charset=utf-8";
    context.Response.ContentLength64 = bytes.Length;
    context.Response.OutputStream.Write(bytes);
    context.Response.Close();
    Console.WriteLine($"{method} {path} -> {status}");
}

// (4) The page frame, repeated for every page, by hand.
static string Page(string title, string content) =>
    "<!DOCTYPE html><html><head><meta charset=\"utf-8\">"
    + $"<title>{Escape(title)}</title></head><body>"
    + $"<h1>{Escape(title)}</h1>{content}</body></html>";

// (5) HTML escaping by hand: forget it once and the site is attackable.
static string Escape(string text) => WebUtility.HtmlEncode(text);

static string Rp(decimal amount) =>
    "Rp " + amount.ToString("N0", CultureInfo.InvariantCulture)
        .Replace(",", ".");
```

Listing 1.1 (Continued) *nofw-csharp/Program.cs: the price list with no framework. The comments (1) to (6) mark the jobs a framework would do.*

```
// (6) Reading the data by hand: a CSV line with "quoted, commas".
static List<Service> LoadServices(string file)
{
    var list = new List<Service>();
    foreach (var line in File.ReadLines(file).Skip(1))
    {
        var f = new List<string>();
        var cur = new StringBuilder();
        bool quoted = false;
        foreach (char c in line)
        {
            if (c == '"') quoted = !quoted;
            else if (c == ',' && !quoted) { f.Add(cur.ToString()); cur.Clear(); }
            else cur.Append(c);
        }
        f.Add(cur.ToString());
        list.Add(new Service(f[0], f[1],
            decimal.Parse(f[2], CultureInfo.InvariantCulture)));
    }
    return list;
}

record Service(string Id, string Description, decimal Price);
```

Read the listing once from top to bottom and count the jobs. (1) The routing is a chain of `if`/`else if` that compares the path with every page we know. (2) The code `S0004` is cut out of the path with string arithmetic, `path["/services/".Length..]`, after checking that the path has exactly two slashes. (3) The response is assembled by hand: status, content type, length, bytes, close. (4) The page frame, `<!DOCTYPE html>` to `</html>`, is a function that every page calls. (5) Every value from outside passes through `Escape`, which calls `WebUtility.HtmlEncode`; forgetting it once, in one page, is how Bu Wati's old site printed a `<script>` back to a visitor. (6) The CSV file is parsed character by character, because a description may contain a comma inside quotes.

There is also a job the listing does not do at all: the `while (true)` loop handles **one request at a time**. While it builds one page, every other visitor waits. For Bu Wati's twelve services that is harmless; for a real site it is not. A framework's server handles thousands of requests at the same time, and we will not write that part by hand.

To run it, open a terminal in `Chapter01/nofw-csharp` and type:

```
dotnet run
Listening on http://127.0.0.1:4010/ (Ctrl+C stops)
```

Open `http://127.0.0.1:4010/services` in a browser; the result is a plain, unstyled table. Section 1.7 asks all four programs the same questions and compares the answers.

1.5.2 In PHP, with the built-in server

PHP was made for the web, so the PHP version is shorter: PHP itself parses the request into the arrays `$_SERVER`, `$_GET` and `$_POST`, and `echo` writes to the response. Still, every job of Section 1.2.1 is there. **Listing 1.2** uses PHP's built-in development server, which sends *every* request to the one script named on its command line; that script is therefore also the router.

Listing 1.2 `nofw-php/index.php`: the same price list in PHP without a framework. The script is at once router, controller, model and view.

```php
<?php
// Profitina Laundry's price list with NO framework: PHP's built-in server.
// Run from this folder:  php -S 127.0.0.1:4011 index.php
declare(strict_types=1);

function page(string $title, string $content): string
{
    $t = htmlspecialchars($title);
    return "<!DOCTYPE html><html><head><meta charset=\"utf-8\">"
        . "<title>$t</title></head><body><h1>$t</h1>$content</body></html>";
}

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

// Reading the data by hand (fgetcsv understands "quoted, commas").
$services = [];
$file = fopen(__DIR__ . '/../data/Service.csv', 'r');
fgetcsv($file, escape: '');                          // skip the header
while (($f = fgetcsv($file, escape: '')) !== false) {
    $services[$f[0]] = ['id' => $f[0], 'description' => $f[1],
                        'price' => (float) $f[2]];
}
fclose($file);

// Routing by hand.
$method = $_SERVER['REQUEST_METHOD'];
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$status = 200;
if ($method !== 'GET') {
    $status = 405;
    $body = page('Not allowed', '<p>This site only answers GET.</p>');
} elseif ($path === '/') {
    $body = page('Profitina Laundry',
        '<p>See <a href="/services">our services and prices</a>.</p>');
} elseif ($path === '/services') {
    $rows = '';
    foreach ($services as $s) {
        $rows .= "<tr><td><a href=\"/services/{$s['id']}\">{$s['id']}</a></td>"
            . '<td>' . htmlspecialchars($s['description']) . '</td>'
            . '<td>' . rp($s['price']) . '</td></tr>';
    }
    $body = page('Our services', "<table>$rows</table>");
} elseif (preg_match('#^/services/([^/]+)$#', $path, $m)) {
```

Listing 1.2 (Continued) *nofw-php/index.php: the same price list in PHP without a framework. The script is at once router, controller, model and view.*

```
    $id = urldecode($m[1]);
    if (!isset($services[$id])) {
        $status = 404;
        $body = page('Not found', '<p>No service ' . htmlspecialchars($id)
            . '.</p>');
    } else {
        $s = $services[$id];
        $body = page($s['id'], '<p>' . htmlspecialchars($s['description'])
            . ': ' . rp($s['price']) . '</p>');
    }
} else {
    $status = 404;
    $body = page('Not found', '<p>Nothing at ' . htmlspecialchars($path)
        . '.</p>');
}

// The response by hand.
http_response_code($status);
header('Content-Type: text/html; charset=utf-8');
echo $body;
```

Two lines deserve a comment. `parse_url(..., PHP_URL_PATH)` cuts the query string off the requested URI, because `$_SERVER['REQUEST_URI']` contains both. And `preg_match('#^/services/([^/]+)$#', ...)` is routing with a regular expression: it matches `/services/` followed by one or more characters that are not a slash, and captures them in `$m[1]`. That one line is a small router; a framework's router is the same idea, generalized and tested. Run it from `Chapter01/nofw-php`:

```
php -S 127.0.0.1:4011 index.php
```

1.5.3 In Java, with `com.sun.net.httpserver`

The JDK has shipped a small HTTP server since Java 6 in the package `com.sun.net.httpserver`. It is a supported part of the JDK's `jdk.httpserver` module, and, like `HttpListener`, it is a library, not a framework. **Listing 1.3** registers one handler for every path and routes inside it.

Listing 1.3 `nofw-java/RawLaundry.java`: the price list in Java without a framework, on the JDK's built-in HTTP server.

```java
// Profitina Laundry's price list with NO framework: the JDK's HttpServer.
// Run from this folder:  java RawLaundry.java
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

public class RawLaundry {
    record Service(String id, String description, long price) {}

    static List<Service> services = new ArrayList<>();

    public static void main(String[] args) throws IOException {
        List<String> lines = Files.readAllLines(Path.of("../data/Service.csv"));
        for (String line : lines.subList(1, lines.size())) {   // skip header
            List<String> f = splitCsv(line);
            services.add(new Service(f.get(0), f.get(1),
                Math.round(Double.parseDouble(f.get(2)))));
        }
        HttpServer server = HttpServer.create(
            new InetSocketAddress("127.0.0.1", 4012), 0);
        server.createContext("/", RawLaundry::handle);  // one handler for all
        server.start();
        System.out.println("Listening on http://127.0.0.1:4012/ (Ctrl+C stops)");
    }

    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        int status = 200;
        String body;
        if (!ex.getRequestMethod().equals("GET")) {             // routing by hand
            status = 405;
            body = page("Not allowed", "<p>This site only answers GET.</p>");
        } else if (path.equals("/")) {
            body = page("Profitina Laundry",
                "<p>See <a href=\"/services\">our services and prices</a>.</p>");
        } else if (path.equals("/services")) {
            StringBuilder rows = new StringBuilder();
            for (Service s : services) {
```

Listing 1.3 (Continued) *nofw-java/RawLaundry.java: the price list in Java without a framework, on the JDK's built-in HTTP server.*

```
                rows.append("<tr><td><a href=\"/services/" + s.id() + "\">")
                    .append(s.id()).append("</a></td><td>")
                    .append(escape(s.description())).append("</td><td>")
                    .append(rp(s.price())).append("</td></tr>");
            }
            body = page("Our services", "<table>" + rows + "</table>");
        } else if (path.matches("/services/[^/]+")) {
            String id = URLDecoder.decode(path.substring(10),
                StandardCharsets.UTF_8);
            Service found = services.stream().filter(s -> s.id().equals(id))
                .findFirst().orElse(null);
            if (found == null) {
                status = 404;
                body = page("Not found", "<p>No service " + escape(id) + ".</p>");
            } else {
                body = page(found.id(), "<p>" + escape(found.description())
                    + ": " + rp(found.price()) + "</p>");
            }
        } else {
            status = 404;
            body = page("Not found", "<p>Nothing at " + escape(path) + ".</p>");
        }
        byte[] bytes = body.getBytes(StandardCharsets.UTF_8);  // response by hand
        ex.getResponseHeaders().set("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(status, bytes.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(bytes);
        }
        System.out.println(ex.getRequestMethod() + " " + path + " -> " + status);
    }

    static String page(String title, String content) {
        return "<!DOCTYPE html><html><head><meta charset=\"utf-8\"><title>"
            + escape(title) + "</title></head><body><h1>" + escape(title)
            + "</h1>" + content + "</body></html>";
    }

    static String escape(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;")
            .replace(">", "&gt;").replace("\"", "&quot;");
    }

    static String rp(long amount) {
        return "Rp " + String.format("%,d", amount).replace(',', '.');
    }
```

Listing 1.3 (Continued) *nofw-java/RawLaundry.java: the price list in Java without a framework, on the JDK's built-in HTTP server.*

```
    static List<String> splitCsv(String line) {       // "quoted, commas" too
        List<String> fields = new ArrayList<>();
        StringBuilder cur = new StringBuilder();
        boolean quoted = false;
        for (char c : line.toCharArray()) {
            if (c == '"') quoted = !quoted;
            else if (c == ',' && !quoted) {
                fields.add(cur.toString());
                cur.setLength(0);
            }
            else cur.append(c);
        }
        fields.add(cur.toString());
        return fields;
    }
}
```

Since Java 11, a single source file can be run without compiling it first, so from `Chapter01/nofw-java` the command is simply:

```
java RawLaundry.java
```

Unlike `HttpListener`, the JDK server can use a thread pool (`server.setExecutor(...)`); we left the default, which again handles one request at a time. The HTML escaping in `escape` is written by hand, four `replace` calls; the Java standard library has no HTML encoder, which is one more job a framework's template engine takes over.

1.6 The Same Price List With a Framework

1.6.1 ASP.NET Core minimal APIs

Now the same four addresses with ASP.NET Core. The project file differs in one word: `Sdk="Microsoft.NET.Sdk.Web"` instead of `Sdk="Microsoft.NET.Sdk"`. That word brings in the ASP.NET Core shared framework. **Listing 1.4** is the whole file.

Listing 1.4 `aspnet/HelloLaundry.csproj`. The `.Web` SDK makes this an ASP.NET Core project; `net10.0` targets .NET 10.

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

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

</Project>
```

ASP.NET Core offers two styles for handling requests: **controllers** with views (MVC, from Chapter 3 on), and **minimal APIs**, where each route is one line that maps a method and a URL pattern to a function [22]. Minimal APIs are the shortest way to show what the framework does, so **Listing 1.5** uses them.

Listing 1.5 `aspnet/Program.cs`: the price list with ASP.NET Core minimal APIs. Routing, parameter extraction, status codes, JSON and the server are the framework's.

```
// The same price list WITH a framework: ASP.NET Core minimal APIs.
using System.Globalization;
using System.Text.RegularExpressions;

var app = WebApplication.CreateBuilder(args).Build();

// A CSV field is either "quoted, maybe with commas" or plain text.
var field = new Regex("\"([^\"]*)\"|([^,]+)");
var services = File.ReadLines("../data/Service.csv").Skip(1)
    .Select(line => field.Matches(line)
        .Select(m => m.Groups[1].Success ? m.Groups[1].Value : m.Value)
        .ToArray())
    .Select(f => new Service(f[0], f[1],
        decimal.Parse(f[2], CultureInfo.InvariantCulture)))
    .ToList();

app.MapGet("/", () => "Profitina Laundry. Prices at /services");
app.MapGet("/services", () => services);
app.MapGet("/services/{id}", (string id) =>
    services.FirstOrDefault(s => s.Id == id) is { } found
        ? Results.Ok(found)
        : Results.NotFound(new { error = $"No service {id}" }));

app.Run();

record Service(string Id, string Description, decimal Price);
```

Compare it with **Listing 1.1**, line by line.

- `WebApplication.CreateBuilder(args).Build()` creates the application with the Kestrel server, logging and configuration already set up. There is no `HttpListener`, no prefix, no loop: `app.Run()` starts the framework's loop, which handles many requests at once.
- `app.MapGet("/services/{id}", (string id) => ...)` is routing *and* parameter extraction in one line. `{id}` is a **route parameter**: the router matches any one path segment there and hands its value to the function's parameter of the same name. Nothing is cut out by hand, and a request for `/services/S0004/extra` simply matches no route.
- `app.MapGet("/services", () => services)` returns a list. The framework notices that the result is not a string and serializes it to **JSON**, with the header `Content-Type: application/json; charset=utf-8`.
- `Results.Ok(found)` and `Results.NotFound(...)` choose the status code by name, not by number.
- There is no 404 page for unknown paths and no 405 for `POST`: the router answers both by itself.
- The CSV parsing is still ours (the framework does not know our file), but it shrank to a regular expression, `"([^"]*)"|([^,]+)`, which reads "either a quoted field or a run of characters without a comma". Chapter 5 replaces the file with a database and an ORM.

The CSV part aside, the program is **26 lines**, against **110** for **Listing 1.1**, **66** for the PHP version and **108** for the Java version. More important than the count is *which* lines disappeared: exactly the amber ones of **Figure 1.1**, where the bugs live.

The one thing the minimal API version does *not* do is produce HTML. It answers with JSON, which is what a mobile app or another program wants, and what minimal APIs do best. Turning data into HTML pages with a layout, links and forms is the job of MVC and Razor, and Chapter 3 builds Bu Wati's complete site that way.

1.6.2 Running it

```
cd Chapter01/aspnet
dotnet run --urls http://127.0.0.1:4013
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:4013
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /home/claude/kit/FBP/Code_new/Chapter01/aspnet
```

Four facts are in this log. The application **is** the web server: Kestrel is inside it. `--urls` chose the address; without it, ASP.NET Core would use `http://localhost:5000` or the address in `Properties/launchSettings.json`. The **hosting environment** is `Production`, because nothing said otherwise; Chapter 3 shows how `Development` changes the error pages. And the **content root** is the folder the program reads its files from, which is why the relative path `../data/Service.csv` works when you start the program from its own folder. The small file `appsettings.json` next to the program sets the log level of ASP.NET Core's own messages to `Warning`, so that only our own messages and the start-up lines appear.

Ask it for one service with `curl`, the command-line HTTP client that ships with Windows 10 and 11, macOS and nearly every Linux (`-i` shows the status line and the headers):

```
curl -i http://127.0.0.1:4013/services/S0004
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

{"id":"S0004","description":"Dry cleaning - shirt/blouse (per item)","price":20000.00}
```

(The `Date` header, which only tells the time of the request, is left out here and in the rest of the book.) The property names were turned into *camelCase* (`Id` became `id`), the JSON convention of the web; the `decimal` price keeps its two decimals. We wrote none of this.

1.6.3 A first real project in each framework

The four programs above are deliberately tiny. A real framework project starts from a **template** or a **generator** that creates the folders, the configuration and a working home page. These are the commands; run them in a folder where you keep projects.

ASP.NET Core MVC, with the template that Chapter 3 starts from:

```
dotnet new mvc -o FirstMvc
cd FirstMvc
dotnet run
The template "ASP.NET Core Web App (Model-View-Controller)" was created successfully.
This template contains technologies from parties other than Microsoft, see
https://aka.ms/aspnetcore/10.0-third-party-notices for details.

Processing post-creation actions...
Restoring …/FirstMvc/FirstMvc.csproj:
  Determining projects to restore...
  Restored …/FirstMvc/FirstMvc.csproj (in 135 ms).
Restore succeeded.
```

This really ran on the build machine: the template needs no package beyond the shared framework, so *restore* succeeds offline. `dotnet run` then starts the site; its home page is titled *Home Page - FirstMvc*. Chapter 3 takes this project apart file by file.

Laravel 13, with the installer of Section 1.4.3 [15]:

```
laravel new first-laravel
cd first-laravel
php artisan serve
```

The installer asks a few questions (a starter kit, a testing framework; choose *None* and *Pest* or *PHPUnit* for now), downloads Laravel and its packages from Packagist, creates the SQLite database `database/database.sqlite` and runs the first migrations. `php artisan serve` starts PHP's development server with Laravel behind it, by default at `http://localhost:8000`. This needs the internet, so it could not run on the build machine; Chapter 7 builds Profitina Laundry in Laravel step by step.

Spring Boot 4, with Spring Initializr [18]. Open https://start.spring.io, choose *Maven*, *Java*, the current Spring Boot 4.1 version, Java 25, the artifact name `first-spring` and the dependencies *Spring Web* and *Thymeleaf*, and press *Generate*; or ask for the same zip from the terminal:

```
curl -o first-spring.zip "https://start.spring.io/starter.zip?type=maven-
project&language=java&javaVersion=25&artifactId=first-spring&dependencies=web,thymeleaf"
```

Unzip it, open a terminal in the folder, and start it with the Maven wrapper: `./mvnw spring-boot:run` on macOS and Linux, `mvnw.cmd spring-boot:run` on Windows. The first run downloads Spring and its dependencies from Maven Central and then serves the application at `http://localhost:8080`. It also needs the internet, and Chapter 11 builds it in full. In VS Code, the Spring Boot Extension Pack offers the same generator as the command *Spring Initializr: Create a Maven Project*.

1.7 Running the Four Programs Side by Side

A comparison is only fair if every program is asked exactly the same questions. The script `Chapter01/verify.sh` starts each of the four servers in turn, sends it five requests with `curl`, prints the status code, the path and the start of the answer (with the HTML tags removed, so that it fits on a line), and stops it again. The five requests are the home page, an existing service, a missing service, a path that contains HTML (`%3Cb%3EHi%3C%2Fb%3E` is `<b>Hi</b>` written with *percent-encoding*, the way a browser sends such characters) and a `POST`. This is its real output.

```
===== C#, no framework (HttpListener), port 4010
200  /                              Profitina Laundry: See our services and prices
200  /services/S0004                S0004: Dry cleaning - shirt/blouse (per item):
404  /services/S0099                Not found: No service S0099.
404  /services/%3Cb%3EHi%3C%2Fb%3E  Not found: No service &lt;b&gt;Hi&lt;/b&gt;.
411  POST /services
===== PHP, no framework (built-in server), port 4011
200  /                              Profitina Laundry: See our services and prices
200  /services/S0004                S0004: Dry cleaning - shirt/blouse (per item):
404  /services/S0099                Not found: No service S0099.
404  /services/%3Cb%3EHi%3C%2Fb%3E  Not found: No service &lt;b&gt;Hi&lt;/b&gt;.
405  POST /services
===== Java, no framework (com.sun.net.httpserver), port 4012
200  /                              Profitina Laundry: See our services and prices
200  /services/S0004                S0004: Dry cleaning - shirt/blouse (per item):
404  /services/S0099                Not found: No service S0099.
404  /services/%3Cb%3EHi%3C%2Fb%3E  Not found: Nothing at /services/&lt;b&gt;Hi&lt
405  POST /services
===== C# with ASP.NET Core (minimal APIs), port 4013
200  /                              Profitina Laundry. Prices at /services
200  /services/S0004                {"id":"S0004","description":"Dry cleaning - sh
404  /services/S0099                {"error":"No service S0099"}
404  /services/%3Cb%3EHi%3C%2Fb%3E  {"error":"No service <b>Hi<%2Fb>"}
405  POST /services
```

On the easy questions the four agree, which is what we wanted: the same page for `/`, the same service for `S0004`, `404` for `S0099`. The interesting lines are the ones where they *disagree*, because each disagreement is a decision that someone had to make, and that a framework makes for you, consistently, everywhere.

The `POST` without a body. Our C# program never saw that request. `curl -X POST` sent no body and no `Content-Length` header, and `HttpListener` itself rejected it with `411 Length Required` before our `if (method != "GET")` could answer `405`. The server log confirms it: the C# program printed a line for every `GET` and none for the `POST`. Even the thinnest server library makes decisions of its own, and when you write everything by hand you must know them all.

The encoded slash. The Java program answered "Nothing at …" where the others answered "No service …". The JDK server decodes the whole path before our code sees it, so `%2F` became a real `/`, the path had three slashes, and our routing sent it to the "not found" branch for unknown paths. The C# and PHP programs cut the id out first and decoded it afterwards. ASP.NET Core does something more careful still: it decodes the path segment but leaves an encoded slash encoded (`<b>Hi<%2Fb>`), because a slash inside a value must not change which route matches. Three hand-written servers, three behaviors; the framework has one, documented.

Escaping. All three hand-written programs printed `<b>`, the escaped form, so the browser shows the text `<b>Hi</b>` instead of making "Hi" bold; each did so only because we called an escaping function on that one value. ASP.NET Core printed the angle brackets as they were, which is correct for JSON: the answer is sent as `application/json`, which a browser never runs as HTML. When ASP.NET Core does produce HTML (Chapter 3), its Razor views escape every value automatically, unless you explicitly ask them not to. OWASP, the web security community, lists "escape all untrusted output, automatically, in the template" as the first defense against cross-site scripting [23]; a framework makes that the default instead of a thing to remember.

The answers' form. The hand-written servers answered HTML, the framework JSON. That is not a property of frameworks but of the style we chose: minimal APIs aim at programs; Chapter 3's MVC aims at people.

1.8 Making It Better: What We Still Did by Hand

Even the framework version left several jobs to us, and naming them now gives a map of the coming chapters.

- **The data file.** We parsed a CSV file with a regular expression. A real shop keeps its data in a database; Chapter 5 connects ASP.NET Core to SQLite through Entity Framework Core, an ORM, and the parsing disappears.
- **HTML pages.** Minimal APIs answered JSON. Chapter 3 builds the pages with controllers, Razor views, a shared layout and tag helpers, and escaping becomes automatic.
- **Input from forms.** Nothing here accepted input except the path. Chapter 3 binds a form's fields to a C# object; Chapter 6 adds validation.
- **Logins and roles.** Anybody can see everything. Chapter 6 adds ASP.NET Core Identity; Chapters 10 and 14 do the same in Laravel and Spring Boot.
- **Tests.** We checked the programs with `curl` by hand, through a script. Chapters 10 and 14 write automated tests that run on every change.

Chapter 2 does something unusual first: before using the real frameworks, it builds a tiny framework of its own, about three hundred lines, with a router, a dependency-injection container, an object-relational mapper and a template engine. It is not meant for real use; it is meant to take away the mystery, so that when ASP.NET Core, Laravel or Spring Boot do the same things at scale, you know what is happening inside.

1.9 Using It for Real

1.9.1 Your own shop, clinic or farm

Every small business that Bu Wati's story reminds you of has a price list, and most have the same wish: one change in one place. If you want to adapt this chapter's programs to a business you know, start with the data file. A tailor's `Service.csv` might list "hemming trousers" and "school uniform, set"; a small clinic might list its consultation and laboratory fees; a fruit and durian orchard selling from its gate might list its fruits per kilogram by season. Change the file, keep its three columns (`id`, `description`, `price`), and all four programs work unchanged. Then change the words on the home page. Which of the four programs was easiest to adapt? Almost certainly the one with the fewest lines that are not about the business.

1.9.2 Choosing a framework for a real project

For a real project, the choice among the three is usually made by the people and the place rather than by the framework. A team that already knows C# and works with Microsoft products picks ASP.NET Core; a small agency whose customers run shared PHP hosting picks Laravel; a bank or an enterprise with a Java estate picks Spring Boot. All three can build anything in this book. Chapter 15 compares them in depth, with measurements, once you have built the same features in each.

1.9.3 Profitina in practice

Profitina, the AI visibility platform after which the laundry of this book is named, is itself built on a web framework, but on a fourth one: **FastAPI**, a Python framework, served by the Uvicorn server [24]. The fact is instructive, because FastAPI has every feature of this chapter under different names. Its routes are Python functions decorated with the method and the path; its

route parameters are function parameters, converted to the declared type automatically; its dependency injection (`Depends`) supplies the database session and the current user; its answers are JSON by default. The dashboard of a Profitina user is collected by one route, `GET /api/me/dashboard`, whose handler asks several model functions for notifications, followed channels and rank, and returns one JSON document; the browser page renders it. That is the minimal API of **Listing 1.5**, grown up. When you move between frameworks, look for the same eight steps of Section 1.2.1 under new names; you will always find them.

1.10 Common Mistakes, and How to Avoid Them

Installing a runtime instead of the SDK. The .NET download page offers the SDK, the .NET runtime and the ASP.NET Core runtime. Only the SDK has the `dotnet new`, `dotnet build` and `dotnet run` commands for developers. If `dotnet --list-sdks` prints nothing, install the SDK.

An old terminal. A terminal opened *before* an installation does not know the new commands. After installing anything, close the terminal (and VS Code's terminals) and open a new one before concluding that the installation failed.

Two Javas. `java -version` answers 17 or 21 although you installed 25: the older JDK comes first on the `PATH`. Set `JAVA_HOME` to the JDK 25 folder and put `%JAVA_HOME%\bin` (Windows) or `$JAVA_HOME/bin` (macOS, Linux) at the *front* of the `PATH`.

Running a program from the wrong folder. All four programs read `../data/Service.csv`, a path relative to the folder you start them from. Started elsewhere, they stop with `FileNotFoundException` (C#), a PHP warning or `NoSuchFileException` (Java). Start each one from its own folder, as the commands in this chapter do.

A port that is still busy. If a server says the address is already in use, a previous run is still alive, often in another terminal or in VS Code. Stop it with **Ctrl+C** in its terminal, or choose another port.

Thinking the framework is magic. When a route does not match or a value is not bound, beginners try random changes. Instead, ask which of the eight steps of Section 1.2.1 failed, and look at what the framework logs for that step. Every framework can be made to explain its routing decision; Chapter 4 shows how in ASP.NET Core.

Forgetting to escape in hand-written HTML. Whenever you build HTML with string concatenation, every value from outside must be escaped, every single time. Better: let a template engine do it (Chapter 3), and treat any code that switches escaping off (`Html.Raw` in Razor, `{!! !!}` in Blade, `th:utext` in Thymeleaf) as a place to review twice.

1.11 Chapter Summary

- Every request to a web application goes through the same steps: wait, route, extract, fetch, decide, render, respond, survive. Only fetching and deciding are about the business.
- A library is called by your code; a framework calls your code. This inversion of control is what lets a framework do the plumbing once, correctly, for every application.
- Web frameworks organize your code as Model–View–Controller: the model holds data and rules, the view renders, the controller handles one request. A router chooses the controller action. Keep controllers thin.

- This book uses ASP.NET Core 10 (C#), Laravel 13 (PHP 8.5) and Spring Boot 4 (Java 25), which together represent the three big families of server-side development and which the 2025 Stack Overflow survey found in use by 21.3%, 9.3% and 15.6% of professional developers.
- The workbench is VS Code with a few extensions, the .NET 10 SDK, PHP 8.5 with Composer and the Laravel installer, JDK 25 with Maven (or the Maven wrapper), SQLite and MySQL 9.7 LTS.
- Written by hand, the price list took 110 lines in C#, 66 in PHP and 108 in Java, and each version made its own small decisions about slashes, methods and escaping. With ASP.NET Core minimal APIs it took 26 lines, and the framework made those decisions consistently.

1.12 Exercises

1. Classify each of the following as a library or a framework, and give the reason in one sentence: `HttpListener`; PHP's `fgetcsv`; ASP.NET Core; `System.Text.Json`; Laravel; the JDK's `com.sun.net.httpserver`; Spring Boot.

2. Follow the request `GET /services/S0099` through **Listing 1.1**. For each of the eight steps of Section 1.2.1, name the line or lines that perform it, or say that the program does not perform it.

3. Add an address `/services/under/{max}` to the minimal API of **Listing 1.5** that returns, as JSON, every service whose price is at most `max` rupiah. `/services/under/cheap` must answer `404`. Show the answer for `/services/under/7000`.

4. Add the same address to the hand-written C# program of **Listing 1.1**, answering an HTML table. How many lines did you add, compared with Exercise 3?

5. In Section 1.7, `HttpListener` answered `411` to `curl -X POST`, while the other three answered `405`. Which `curl` option makes the request reach our code, and what does it then answer? Why is this detail a good argument for a framework?

6. Remove the call to `Escape` from the 404 message of **Listing 1.1** (the line with `No service`), run the program, and open `http://127.0.0.1:4010/services/%3Cb%3EHi%3C%2Fb%3E` in a browser. What do you see, and why is this dangerous? Put `Escape` back afterwards.

7. Write down, for your own computer, the answers of the seven version commands of Section 1.4.6. Which line proves that ASP.NET Core itself is installed, and not only .NET?

1.13 Solutions to the Exercises

1. `HttpListener` is a library: our loop calls `GetContext()` whenever it wants the next request. `fgetcsv` is a library function that our script calls. ASP.NET Core is a framework: after `app.Run()` it owns the loop and calls our route handlers. `System.Text.Json` is a library we (or the framework) call to serialize. Laravel is a framework: its `public/index.php` receives every request and calls our routes and controllers. `com.sun.net.httpserver` is a library, although it calls our handler back: it only accepts connections and parses the request line and headers, and leaves routing, rendering and every other decision to us. The line between the two is not always sharp; the test is who owns the flow of a request. Spring Boot is a framework: `SpringApplication.run` starts the server and calls our controllers.

2. *Wait*: `listener.GetContext()`. *Route*: the chain of `if`/`else if` on `path`; the branch `path.StartsWith("/services/") && path.Count(...) == 2` is taken. *Extract*: `Uri.UnescapeDataString(path["/services/".Length..])` gives `"S0099"`. *Fetch*: `services.FirstOrDefault(s => s.Id == id)` returns `null`. *Decide*: `if (found is null)` sets

`status = 404`. *Render*: `Page("Not found", ...)` with `Escape(id)`. *Respond*: the five lines under comment (3). *Survive*: only `Console.WriteLine` logs the result; there is no error handling, so an exception (for example a missing CSV file) would stop the whole server.

3. One route, placed after the other `MapGet` calls:

```
app.MapGet("/services/under/{max:int}", (int max) =>
    services.Where(s => s.Price <= max));
```

`{max:int}` is a *route constraint*: the route matches only if the segment is an integer, and the value arrives already converted to `int`. For `cheap` no route matches, so the framework answers `404` with an empty body. The answer for `/services/under/7000`, from a run of exactly this change:

```
[{"id":"S0003","description":"Ironing only (per kg)","price":6000.00}]
```

4. One working version, inserted just before the branch for `/services/{id}`. (Here the order happens not to matter, because that branch requires exactly two slashes; in general, a more specific test must come before a more general one in an `if` chain.)

```
    else if (path.StartsWith("/services/under/")
             && int.TryParse(path["/services/under/".Length..], out int max))
    {
        var rows = new StringBuilder();
        foreach (var s in services.Where(s => s.Price <= max))
        {
            rows.Append($"<tr><td>{s.Id}</td><td>{Escape(s.Description)}</td>"
                + $"<td>{Rp(s.Price)}</td></tr>");
        }
        body = Page($"Up to {Rp(max)}", $"<table>{rows}</table>");
    }
```

That is eleven lines against two, and it contains every job of Section 1.2.1 again: the routing test, the extraction with `TryParse`, the escaping, the page frame. A run answered `/services/under/7000` with one table row, `S0003 Ironing only (per kg) Rp 6.000`, and `/services/under/cheap` with `404`, because `TryParse` fails and no other branch matches.

5. `curl -X POST -d "" http://127.0.0.1:4010/services` sends an empty body with `Content-Length: 0`. Then `HttpListener` passes the request on, and our program answers `405`, as the others did; a run gave exactly `405` with `-d ""` and `411` without it. The detail matters because the behavior of a hand-written server depends on rules of the underlying library that you may never have read. A framework's router answers `405 Method Not Allowed` for every route that exists with another method, whatever the client sends.

6. The browser shows "No service **Hi**." with "Hi" in bold: the text from the address became HTML. With `<script>` instead of `<b>`, the browser would run the script, in the name of Bu Wati's site; that is a *reflected cross-site scripting* attack, and a link containing it can be sent to any customer [23]. Escaping turns `<` into `<`, so the browser displays the characters instead of obeying them.

7. The answers depend on your computer; compare them with Section 1.4.6. The line that proves ASP.NET Core is installed is the one of `dotnet --list-runtimes` that starts with `Microsoft.AspNetCore.App 10.0.`: that is the shared framework. A computer with only `Microsoft.NETCore.App` can run console programs but not web applications.

References

[1] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, 1988.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[3] M. Fowler, "Inversion of control containers and the dependency injection pattern," martinfowler.com, Jan. 2004. [Online]. Available: https://martinfowler.com/articles/injection.html

[4] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[5] Microsoft, ".NET 10.0 Update — September 8, 2026," Microsoft Support, Sep. 2026. [Online]. Available: https://support.microsoft.com/en-us/servicing/dotnet/net-10/2026/net-10-0-update-september-8-2026

[6] B. Crozat, "Laravel versions: latest release and support status," accessed Sep. 2026. [Online]. Available: https://benjamincrozat.com/laravel-versions

[7] PHP.Watch, "PHP 8.5 releases," accessed Sep. 2026. [Online]. Available: https://php.watch/versions/8.5/releases

[8] eosl.date, "Spring Boot end of life (EOL) dates," Aug. 2026 (4.1.1 release). [Online]. Available: https://eosl.date/eol/product/spring-boot/

[9] Oracle, "Consolidated JDK 25 release notes," accessed Sep. 2026. [Online]. Available: https://www.oracle.com/java/technologies/javase/25all-relnotes.html

[10] Stack Overflow, "2025 Developer Survey: Technology — Web frameworks and technologies," 2025. [Online]. Available: https://survey.stackoverflow.co/2025/technology

[11] Microsoft, "Getting started with C# in VS Code," *Visual Studio Code Documentation*, accessed Sep. 2026. [Online]. Available: https://code.visualstudio.com/docs/csharp/get-started

[12] Laravel, "Laravel," Visual Studio Code Marketplace, accessed Sep. 2026. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=laravel.vscode-laravel

[13] Microsoft, "Spring Boot in Visual Studio Code," *Visual Studio Code Documentation*, accessed Sep. 2026. [Online]. Available: https://code.visualstudio.com/docs/java/java-spring-boot

[14] Microsoft, "Download .NET 10.0 (Linux, macOS, and Windows)," accessed Sep. 2026. [Online]. Available: https://dotnet.microsoft.com/en-us/download/dotnet/10.0

[15] Laravel, "Installation," *Laravel 13.x Documentation*, 2026. [Online]. Available: https://laravel.com/docs/13.x/installation

[16] Composer, "Download Composer," accessed Sep. 2026. [Online]. Available: https://getcomposer.org/download/

[17] The Apache Software Foundation, "Download Apache Maven," accessed Sep. 2026. [Online]. Available: https://maven.apache.org/download.cgi

[18] Spring, "Spring Initializr," accessed Sep. 2026. [Online]. Available: https://start.spring.io

[19] SQLite, "SQLite release 3.53.0 on 2026-04-09," 2026. [Online]. Available: https://sqlite.org/releaselog/3_53_0.html

[20] InfoQ, "MySQL 9.7: first major LTS since 8.4 brings enterprise features to Community Edition," May 2026. [Online]. Available: https://www.infoq.com/news/2026/05/mysql-97-lts/

[21] Microsoft, "HttpListener class," *.NET API Reference*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/dotnet/api/system.net.httplistener

[22] Microsoft, "Minimal APIs overview," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis

[23] OWASP Foundation, "Cross Site Scripting prevention cheat sheet," *OWASP Cheat Sheet Series*, accessed Sep. 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[24] S. Ramírez, "FastAPI," documentation, accessed Sep. 2026. [Online]. Available: https://fastapi.tiangolo.com

Chapter 2

Foundations Shared by Modern Frameworks: MVC, Routing, ORM, DI, Templates

ASP.NET Core, Laravel and Spring Boot look very different at first sight: different languages, different folders, different words. Underneath, they are built on the same four ideas, arranged around the same Model–View–Controller pattern: a **router** that maps an address to one action, a **dependency-injection container** that builds the objects each action needs, an **object-relational mapper** that turns table rows into objects and conditions into SQL, and a **template engine** that turns objects into HTML inside a shared layout. This chapter builds all four from scratch, in about three hundred and fifty lines of plain C#, as a tiny framework called MiniFrame, and then writes Profitina Laundry's price list on top of it. With the machinery open in front of you, the chapter then shows the same feature in ASP.NET Core, Laravel and Spring Boot, so that you learn to recognize each idea under its three names. By the end of the chapter you will be able to explain how a router matches a URL against route templates and extracts parameters, including typed constraints; explain what a DI container does, what *singleton* and *transient* lifetimes mean, and why constructor injection makes code easier to change and test; explain how an ORM maps rows to objects and how a condition written in code becomes a `WHERE` clause; explain layouts, loops and automatic escaping in templates; and read a controller, a model and a view in any of the three frameworks.

2.1 Where We Left Off

Chapter 1 wrote the price list of Profitina Laundry four times. Three hand-written versions needed between 66 and 110 lines each, and every one of them made its own small decisions about routing, parameters and escaping. The ASP.NET Core version needed 26 lines, because the framework took over the routing, the extraction of parameters, the status codes, the JSON and the server itself.

That comparison left an uncomfortable question. *How* does the framework do it? When `app.MapGet("/services/{id}", (string id) => …)` receives `/services/S0004`, something must split the path, compare it with the template, notice that `{id}` is a parameter, and pass `"S0004"` to a function it has never seen. When a Laravel controller declares a repository in its constructor, something must build that repository first. When a Spring repository has a method called `findByPriceLessThanEqual`, something must write the SQL. Without an answer, a framework feels like magic, and magic is impossible to debug.

Bu Wati's nephew, Dimas, ran into exactly this when he opened the three frameworks for the first time. In the ASP.NET Core template he found `[HttpGet("{id}")]`; in Laravel `Route::get('/services/{id}', …)`; in Spring `@GetMapping("/{id}")`. Three syntaxes, one idea. In ASP.NET Core a controller received an `ILaundryCatalog` in its constructor; in Laravel a controller asked for a `ServiceRepository` the same way; in Spring the constructor took a `ServiceRepository` and no code anywhere created one. Again three syntaxes, one idea. This

chapter names the four ideas once, builds each by hand, and then points at them in all three frameworks.

2.2 The Idea in Plain Words

Figure 2.1 places the four foundations along the path of one request. The router (1) chooses the action. The container (2) builds the controller and everything the controller needs. The controller asks a repository for data, and the ORM (3) turns that request into SQL and the rows into objects. The template engine (4) puts the data into a page template, the page into the layout, and sends HTML back.

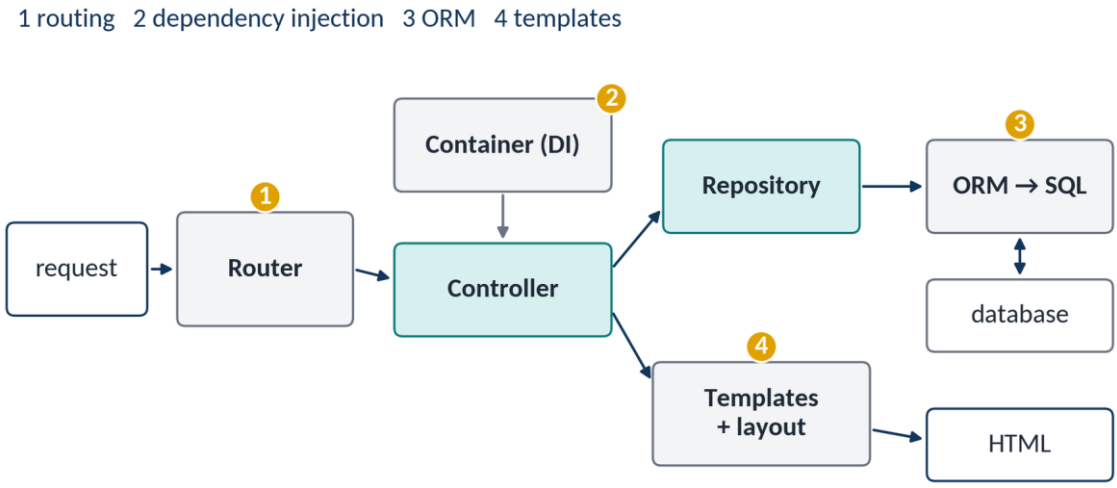


Figure 2.1 The four foundations along one request: routing chooses the action, dependency injection builds the controller, the ORM turns objects into SQL and rows into objects, and the templates turn objects into HTML.

> **DEFINITION.** A **route** is a rule that pairs an HTTP method and a URL *template* such as `/services/{id}` with an action. A **router** keeps the routes in a *route table* and, for each request, finds the route whose template matches the path, extracting the values of its *parameters* (the parts in braces). A **constraint** such as `{max:int}` restricts what a parameter may match.
>
> **DEFINITION. Dependency injection** (DI) means that a class does not create the objects it depends on; it declares them, usually as constructor parameters, and receives them from outside. A **DI container** is the part of a framework that knows, from *registrations* made at start-up, which concrete class to create for each requested type, creates the whole graph of dependencies in the right order, and decides each object's *lifetime*: one object for the whole application (*singleton*), a new one every time (*transient*), or one per request (*scoped*).
>
> **DEFINITION.** An **object-relational mapper** (ORM) translates between the rows and columns of a relational database and the objects and properties of a program. It reads a row into an object by matching column names with property names, and it translates a query written in the programming language (a *condition* such as `s.Price <= 15000`) into SQL. A **migration** is a versioned description of one change to the database schema, written in code so that it can be replayed on every copy of the database.
>
> **DEFINITION.** A **template** is a text file, usually HTML, with *holes* that a **template engine** fills with values from a *model*. Good template engines escape every value by default, so that text from outside cannot become markup, and support a **layout**, a template for the frame that every page shares, into which each page's own content is inserted.

These four definitions are all the theory of this chapter. The rest is watching them work.

2.3 Building a Tiny Framework: MiniFrame

MiniFrame is a teaching framework: small enough to read in one sitting, complete enough to serve Bu Wati's price list over real HTTP, and honest about what it leaves out. It is a console project with no package references at all; it uses only the .NET class library. **Table 2.1** lists its files.

Table 2.1 The files of MiniFrame in `Chapter02/miniframe`. The four foundations take 226 lines together.

File	Lines	Foundation	What it contains
`Router.cs`	55	Routing	A route table, template matching, `{name}` and `{name:int}`
`Container.cs`	45	Dependency injection	Registrations, singleton and transient lifetimes, constructor injection
`Mapper.cs`	90	ORM	Rows to objects by name; a C# condition translated to SQL text
`Template.cs`	36	Templates	`{{ x }}` escaped, `{{{ x }}}` raw, `{{#each}}` loops, a layout
`Laundry.cs`	66	(the application)	Model, repository and controller of the price list
`Program.cs`	63	(start-up)	Registrations, routes, an offline demo, a small HTTP server
`templates/*.html`		(the views)	`layout.html`, `services.html`, `service.html`, `missing.html`

2.3.1 Routing

The router's job is to turn `GET /services/under/15000` into "call this handler with `max = 15000`". **Listing 2.1** does it in the simplest way that still has the essential features: an ordered list of routes, templates split into segments, parameters in braces, and one constraint.

Listing 2.1 `Router.cs`: a route table, template matching and parameter extraction.

```
namespace MiniFrame;

// Foundation 1, ROUTING: from "GET /services/S0004" to one handler
// plus the values cut out of the address ({ id = "S0004" }).
public class Router
{
    private record Route(string Method, string[] Parts,
        Func<Dictionary<string, string>, Response> Handler);

    private readonly List<Route> routes = [];

    public void Add(string method, string template,
        Func<Dictionary<string, string>, Response> handler) =>
        routes.Add(new Route(method, template.Trim('/').Split('/'), handler));

    public (Func<Dictionary<string, string>, Response>? Handler,
            Dictionary<string, string> Values) Match(string method, string path)
    {
        string[] parts = path.Trim('/').Split('/');
        foreach (var route in routes)
        {
            var values = new Dictionary<string, string>();
            if (route.Method == method && Fits(route.Parts, parts, values))
            {
                return (route.Handler, values);
            }
        }
        return (null, []);
    }

    // "services/{id}" fits "services/S0004"; "{max:int}" needs digits.
    private static bool Fits(string[] template, string[] parts,
        Dictionary<string, string> values)
    {
        if (template.Length != parts.Length) return false;
        for (int i = 0; i < parts.Length; i++)
        {
            string t = template[i];
            if (t.StartsWith('{') && t.EndsWith('}'))
            {
                string[] nameAndRule = t[1..^1].Split(':');
                if (nameAndRule.Length == 2 && nameAndRule[1] == "int"
                    && !parts[i].All(char.IsAsciiDigit)) return false;
                values[nameAndRule[0]] = Uri.UnescapeDataString(parts[i]);
            }
            else if (!t.Equals(parts[i], StringComparison.OrdinalIgnoreCase))
```

Listing 2.1 (Continued) *Router.cs: a route table, template matching and parameter extraction.*

```
            {
                return false;
            }
        }
        return true;
    }
}

public record Response(int Status, string ContentType, string Body);
```

`Add` stores each route as its method, its template split at the slashes, and a *handler*, a function that receives the extracted values as a dictionary and returns a `Response`. `Match` splits the requested path the same way and tries the routes **in the order they were added**. `Fits` compares a template with a path segment by segment. The two must have the same number of segments. A literal segment such as `services` must equal the path's segment (ignoring upper and lower case). A

segment in braces matches anything, and its value is stored under its name; if the name carries the constraint `:int`, the segment must consist of digits only, or the route does not fit. `Uri.UnescapeDataString` turns `%3Cb%3E` back into `<b>`, so that the handler receives the value as the user meant it.

Figure 2.2 traces the request `GET /services/under/15000` through the three routes of Bu Wati's price list. The first template has one segment and the path has three, so it cannot fit. The second has two. The third has three segments; `services` and `under` equal the path's segments, and `{max:int}` accepts `15000` because it is all digits. The handler is called with `max = 15000`.

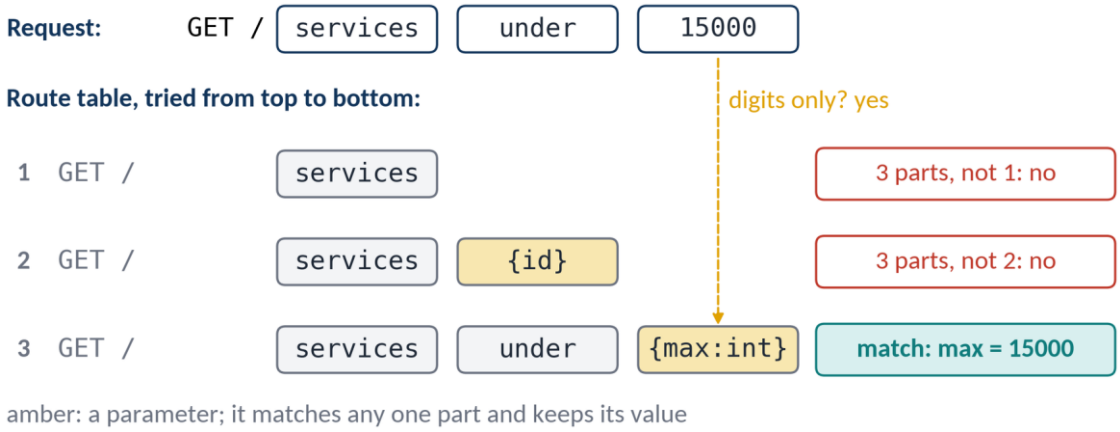


Figure 2.2 Matching `GET /services/under/15000` against the route table: the segment counts rule out the first two routes, and the third fits because its parameter `{max:int}` accepts digits.

Real routers add much to this: they compile the whole table into a decision tree so that matching does not try every route; they allow optional parameters (`{id?}`), default values (`{action=Index}`) and catch-all parameters (`{**path}`); they offer many constraints (`int`, `guid`, `min(1000)`, `regex(...)`); and, most importantly, they do not depend on the order of registration. ASP.NET Core ranks templates by *precedence*: a literal segment beats a parameter, a constrained parameter beats an unconstrained one [1]. MiniFrame's first-match rule is simpler and more dangerous, as Exercise 1 shows.

2.3.2 Dependency injection

The price list needs three kinds of object: a *table* that knows the services, a *repository* that answers questions about them, and a *controller* that turns answers into pages. Each depends on the one before. Without a container, the start-up code would build them by hand, in the right order: `new ServicesController(new CsvServiceRepository(new Table<Service>(...)), new Templates(...))`. That works for three classes, and becomes a burden at thirty. Worse, every class that creates its own dependencies with `new` is welded to one concrete class, so that replacing the CSV file by a database, or by a fake for a test, means editing every class that builds a repository.

A DI container reverses this, as Martin Fowler described in the article that gave the pattern its name [2]. Each class *declares* what it needs in its constructor. At start-up, the application *registers*, once, which concrete class answers each need. When an object is requested, the container looks at its constructor, resolves each parameter the same way, and builds the whole graph. **Listing 2.2** is a complete, if small, container.

Listing 2.2 `Container.cs`: registrations, two lifetimes and constructor injection through reflection.

```
namespace MiniFrame;

// Foundation 2, DEPENDENCY INJECTION: classes say what they need in their
// constructor; the container builds them, and what they need, in order.
public class Container
{
    private readonly Dictionary<Type, (Type Impl, bool Single)> map = [];
    private readonly Dictionary<Type, object> singles = [];

    public void Singleton<TService, TImpl>() where TImpl : TService =>
        map[typeof(TService)] = (typeof(TImpl), true);

    public void Transient<TService, TImpl>() where TImpl : TService =>
        map[typeof(TService)] = (typeof(TImpl), false);

    public void Instance<T>(T obj) where T : notnull
    {
        map[typeof(T)] = (obj.GetType(), true);
        singles[typeof(T)] = obj;
    }

    public T Resolve<T>() => (T)Resolve(typeof(T));

    private object Resolve(Type type)
    {
        if (!map.TryGetValue(type, out var entry))
        {
            throw new InvalidOperationException(
                $"Nothing registered for {type.Name}.");
        }
        if (entry.Single && singles.TryGetValue(type, out var made))
        {
            return made;
        }
        // Take the constructor with the most parameters, resolve each one.
        var ctor = entry.Impl.GetConstructors()
            .OrderByDescending(c => c.GetParameters().Length).First();
        object[] args = ctor.GetParameters()
            .Select(p => Resolve(p.ParameterType)).ToArray();
        object obj = ctor.Invoke(args);
        Console.WriteLine($"  [container] built {entry.Impl.Name}");
        if (entry.Single) singles[type] = obj;
        return obj;
    }
}
```

The dictionary `map` records, for each requested type, the class to create and whether it is a singleton. `Instance` registers an object that already exists. `Resolve` is the heart: it finds the registration, returns the stored object if the type is a singleton that was built before, and otherwise takes the constructor with the most parameters, resolves every parameter type *recursively*, and calls the constructor through reflection. The line that prints `[container] built …` lets us watch it work.

Figure 2.3 shows the registrations of the price list and the graph that `Resolve<ServicesController>()` builds from them. The controller needs an `IServiceRepository` and a `Templates`; the registration says that an `IServiceRepository` is a `CsvServiceRepository`, which needs a `Table<Service>`; the table and the templates were registered as instances.

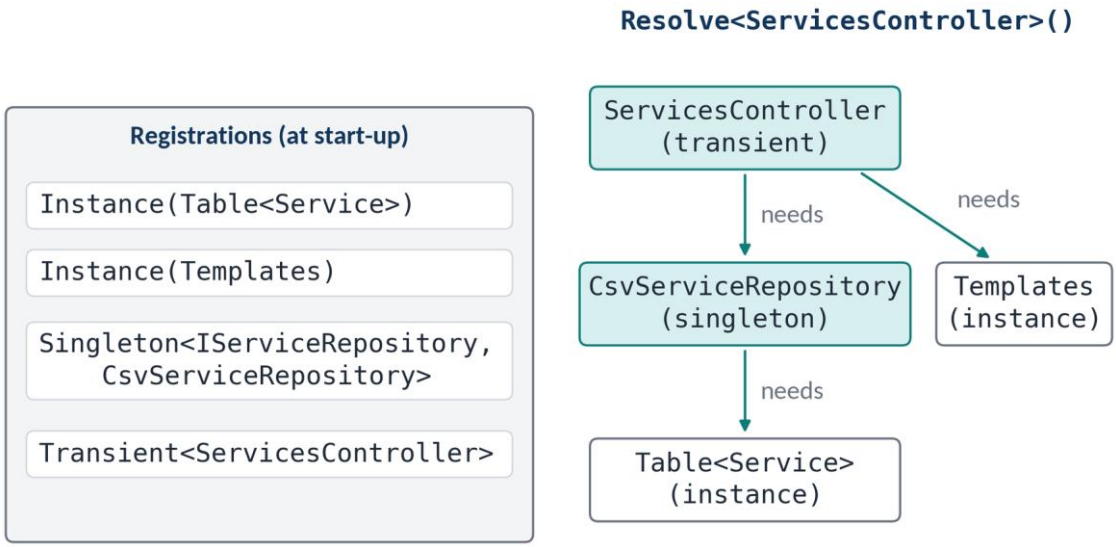


Figure 2.3 The registrations made at start-up, and the graph the container builds when a controller is requested. Only the transient controller is new on every request.

Two ideas in the figure matter in every framework. The first is that the controller depends on the *interface* `IServiceRepository`, not on the class `CsvServiceRepository`. Chapter 5 replaces the CSV file by a database; the only line that changes is the registration. The second is the **lifetime**. The repository holds the data and is safe to share, so it is a *singleton*: built once, used by every request. The controller is *transient*: a fresh one for each request, so that nothing one visitor did can leak into the page of the next. Real containers add a third lifetime, *scoped*, one object per request shared by everything that request touches; it is the natural lifetime of a database connection, and Chapter 5 uses it.

2.3.3 Object-relational mapping

The data of Profitina Laundry live in tables; the program thinks in objects. An ORM stands between the two. Real ORMs are large (Entity Framework Core, Eloquent and Hibernate are each hundreds of thousands of lines), but the two central tricks fit in a page, and **Listing 2.3** shows both.

Listing 2.3 `Mapper.cs`: rows become objects by matching names, and a C# condition becomes the text of an SQL `WHERE` clause.

```
using System.Globalization;
using System.Linq.Expressions;
using System.Reflection;

namespace MiniFrame;

// Foundation 3, OBJECT-RELATIONAL MAPPING: rows become objects, and a
// condition written in C# becomes the WHERE clause of an SQL query.
public class Table<T>(string name, List<Dictionary<string, string>> rows)
{
    // Row -> object: match each constructor parameter with a column.
    public List<T> Where(Expression<Func<T, bool>> condition)
    {
        Console.WriteLine($"  [mapper] SQL: SELECT * FROM {name} "
            + $"WHERE {Sql(condition.Body)}");
        var test = condition.Compile();
        return rows.Select(ToObject).Where(test).ToList();
    }

    public List<T> All()
    {
        Console.WriteLine($"  [mapper] SQL: SELECT * FROM {name}");
        return rows.Select(ToObject).ToList();
    }

    private static T ToObject(Dictionary<string, string> row)
    {
        var ctor = typeof(T).GetConstructors()[0];
        object[] args = ctor.GetParameters().Select(p => Convert.ChangeType(
            row[p.Name!.ToLowerInvariant()], p.ParameterType,
            CultureInfo.InvariantCulture)).ToArray();
        return (T)ctor.Invoke(args);
    }

    // Translates s => s.Price <= 20000 && s.Id != "S0003" into SQL text.
    private static string Sql(Expression e) => e switch
    {
        BinaryExpression b => $"{Sql(b.Left)} {Op(b.NodeType)} {Sql(b.Right)}",
        MemberExpression { Expression: ParameterExpression } m =>
            m.Member.Name.ToLowerInvariant(),
        MemberExpression m => Literal(Expression.Lambda(m).Compile()
            .DynamicInvoke()),                  // a captured local variable
        ConstantExpression c => Literal(c.Value),
        _ => throw new NotSupportedException(e.NodeType.ToString())
    };
```

Listing 2.3 (Continued) *Mapper.cs: rows become objects by matching names, and a C# condition becomes the text of an SQL WHERE clause.*

```
    private static string Op(ExpressionType t) => t switch
    {
        ExpressionType.Equal => "=", ExpressionType.NotEqual => "<>",
        ExpressionType.LessThan => "<", ExpressionType.LessThanOrEqual => "<=",
        ExpressionType.GreaterThan => ">",
        ExpressionType.GreaterThanOrEqual => ">=",
        ExpressionType.AndAlso => "AND", ExpressionType.OrElse => "OR",
        _ => throw new NotSupportedException(t.ToString())
    };

    private static string Literal(object? v) => v switch
    {
        string s => "'" + s.Replace("'", "''") + "'",
        IFormattable f => f.ToString(null, CultureInfo.InvariantCulture),
        _ => "NULL"
    };
}

public static class Csv
{
    // Reads a CSV file into rows keyed by the (lower-case) header names.
    public static List<Dictionary<string, string>> Read(string path)
    {
        var lines = File.ReadAllLines(path);
        string[] header = Split(lines[0]);
        return lines.Skip(1).Select(Split)
            .Select(f => header.Zip(f).ToDictionary(
                p => p.First.ToLowerInvariant(), p => p.Second))
            .ToList();
    }

    private static string[] Split(string line)
    {
        var fields = new List<string> { "" };
        bool quoted = false;
        foreach (char c in line)
        {
            if (c == '"') quoted = !quoted;
            else if (c == ',' && !quoted) fields.Add("");
            else fields[^1] += c;
        }
        return fields.ToArray();
    }
}
```

Rows to objects. `ToObject` takes the first constructor of the class, here the record `Service(string Id, string Description, decimal Price)`, and for each parameter looks up the column with the same name in lower case (`id`, `description`, `price`), converting the text to the parameter's type with `Convert.ChangeType`. This *mapping by convention* is exactly what the real ORMs do by default: a property `Price` maps to a column `price` unless you say otherwise.

Conditions to SQL. The method `Where` does not take an ordinary function; it takes an `Expression<Func<T, bool>>`. When a parameter has that type, the C# compiler does not compile the lambda `s => s.Price <= 15000` into machine code only; it also builds a data structure, an **expression tree**, that describes it [3]: a node `LessThanOrEqual` whose left child is the member `Price` of the parameter `s` and whose right child is the constant `15000`. `Sql` walks that tree recursively and prints each node in SQL: a binary node as "left operator right", a member of the parameter as its column name, a constant or a captured variable as an SQL literal. **Figure 2.4** shows both directions of the mapping.

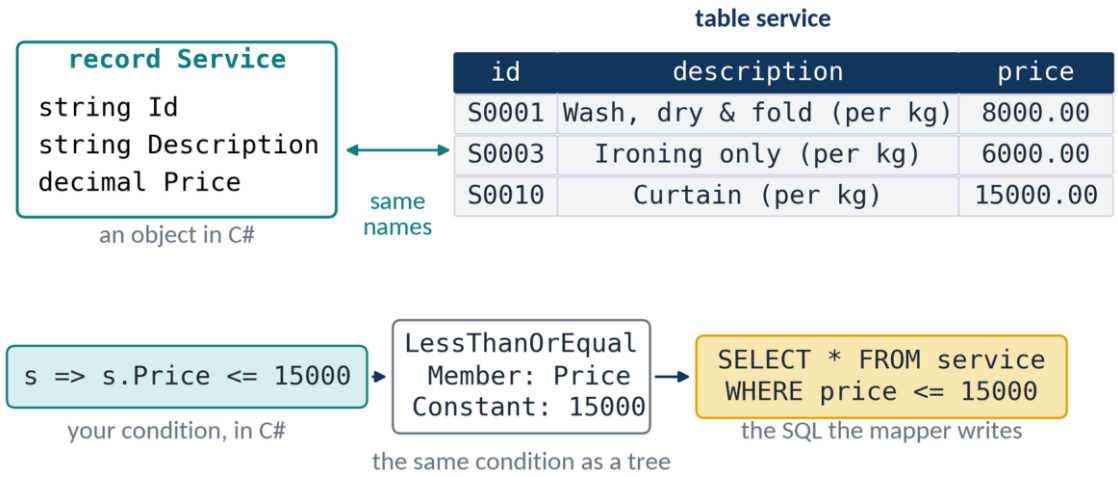


Figure 2.4 Above, a row of the table `service` becomes a `Service` object because column names equal property names. Below, the condition `s => s.Price <= 15000` is read as an expression tree and printed as SQL.

MiniFrame cheats in one place: after printing the SQL, it does not send it to a database. There is none (the data are the CSV file), so it compiles the same condition into ordinary code, `condition.Compile()`, and filters the rows in memory. Entity Framework Core does the same translation, with far more node types (method calls such as `Contains`, joins, ordering, grouping), and sends the SQL to SQLite, SQL Server or MySQL [4]. The idea, a query written in C# and translated from its expression tree, is LINQ, and Chapter 5 uses it every day.

2.3.4 Templates

The last foundation turns data into HTML. **Listing 2.4** is a template engine with the three features every real one has: holes that are escaped by default, loops, and a layout.

Listing 2.4 `Template.cs`: holes, loops and a layout, with escaping by default.

```
using System.Net;
using System.Text.RegularExpressions;

namespace MiniFrame;

// Foundation 4, TEMPLATES: HTML files with holes. {{ name }} is filled
// with an escaped value, {{{ name }}} with raw HTML, {{#each list}}...
// {{/each}} repeats, and every page goes inside layout.html.
public partial class Templates(string folder)
{
    public string Page(string file, Dictionary<string, object> model)
    {
        string page = Fill(File.ReadAllText(Path.Combine(folder, file)), model);
        model["body"] = page;                  // layout.html says {{{ body }}}
        return Fill(File.ReadAllText(Path.Combine(folder, "layout.html")),
            model);
    }

    private static string Fill(string text, Dictionary<string, object> model)
    {
        text = EachBlock().Replace(text, m =>
            string.Concat(((IEnumerable<Dictionary<string, object>>)
                model[m.Groups[1].Value]).Select(item => Fill(m.Groups[2].Value,
                    item))));
        return Hole().Replace(text, m => m.Groups[1].Success
            ? model[m.Groups[1].Value].ToString()!          // {{{ raw }}}
            : WebUtility.HtmlEncode(model[m.Groups[2].Value].ToString() ?? ""));
    }

    [GeneratedRegex(@"\{\{#each (\w+)\}\}(.*?)\{\{/each\}\}",
        RegexOptions.Singleline)]
    private static partial Regex EachBlock();

    [GeneratedRegex(@"\{\{\{ *(\w+) *\}\}\}|\{\{ *(\w+) *\}\}")]
    private static partial Regex Hole();
}
```

`Page` fills a page template with the model, then stores the result as `body` and fills `layout.html` with the same model. `Fill` works in two passes with regular expressions. The first replaces each `{{#each list}}…{{/each}}` block by the block's inside, filled once for every item of the list. The second replaces each hole. The difference between the two kinds of hole is the most important line of the file: `{{ name }}` is filled with `WebUtility.HtmlEncode(value)`, the escaped value, while `{{{ name }}}` is filled with the value exactly as it is. Only the layout uses the raw form, for `{{{ body }}}`, because the body is HTML that our own template produced. Everything that came from data or from a URL goes through the double braces and is escaped automatically; Chapter 1's worry, "every value, every time", became the default. **Listing 2.5** and **Listing 2.6** are the layout and the list page.

Listing 2.5 `templates/layout.html`: the frame every page shares; the page's own HTML goes where `{{{ body }}}` is.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>{{ title }} - Profitina Laundry</title>
</head>
<body>
  <header><a href="/services">Profitina Laundry</a></header>
  <main>
{{{ body }}}
  </main>
</body>
</html>
```

Listing 2.6 `templates/services.html`: a loop over the rows of the model, with every value escaped.

```
<h1>{{ title }}</h1>
<table>
{{#each rows}}  <tr><td><a href="/services/{{ id }}">{{ id }}</a></td>
      <td>{{ description }}</td><td>{{ price }}</td></tr>
{{/each}}</table>
```

Figure 2.5 follows the list page from template to browser: the controller's model fills `services.html` (1), the result is put into the layout where it says `{{{ body }}}` (2), and the complete page is sent (3). The same three steps happen in Razor, Blade and Thymeleaf, as Section 2.5.4 shows.

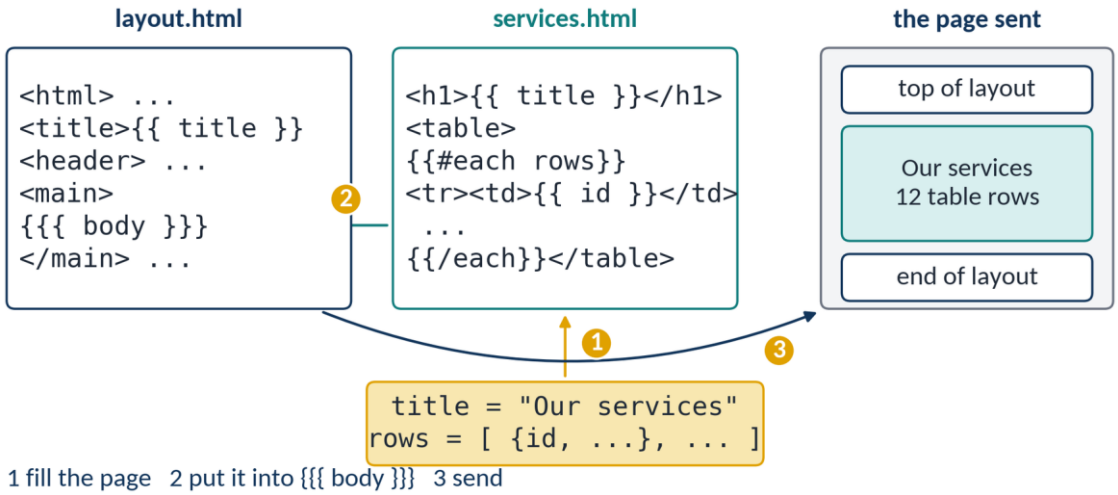


Figure 2.5 From model to page: the page template is filled with the model, the result is inserted into the layout, and the whole page is sent.

2.3.5 The application on top

With the four foundations in place, the application itself is short. **Listing 2.7** contains its model (the record `Service`), a repository interface and its CSV implementation, and the controller.

Listing 2.7 `Laundry.cs`: the model, the repository and the controller of the price list. No routing, parsing, SQL or HTML escaping appears here.

```
using System.Globalization;
using MiniFrame;

// The application code: the only part a framework user writes.
public record Service(string Id, string Description, decimal Price);

public interface IServiceRepository
{
    List<Service> All();
    Service? Find(string id);
    List<Service> UpTo(decimal max);
}

// The M of MVC: data access through the mapper, no SQL strings here.
public class CsvServiceRepository(Table<Service> table) : IServiceRepository
{
    public List<Service> All() => table.All();

    public Service? Find(string id) =>
        table.Where(s => s.Id == id).FirstOrDefault();

    public List<Service> UpTo(decimal max) => table.Where(s => s.Price <= max);
}

// The C of MVC: gets what it needs through its constructor.
public class ServicesController(IServiceRepository repository,
    Templates views)
{
    public Response Index() => List("Our services", repository.All());

    public Response UpTo(int max) =>
        List($"Services up to {Rp(max)}", repository.UpTo(max));

    public Response Details(string id)
    {
        var s = repository.Find(id);
        if (s is null)
        {
            return Html(404, views.Page("missing.html",
                new() { ["title"] = "Not found", ["id"] = id }));
        }
        return Html(200, views.Page("service.html", new()
        {
            ["title"] = s.Id, ["description"] = s.Description,
            ["price"] = Rp(s.Price)
        }));
```

Listing 2.7 (Continued) *Laundry.cs: the model, the repository and the controller of the price list. No routing, parsing, SQL or HTML escaping appears here.*

```
    }

    private Response List(string title, List<Service> list) =>
        Html(200, views.Page("services.html", new()
        {
            ["title"] = title,
            ["rows"] = list.Select(s => new Dictionary<string, object>
            {
                ["id"] = s.Id, ["description"] = s.Description,
                ["price"] = Rp(s.Price)
            })
        }));

    private static Response Html(int status, string body) =>
        new(status, "text/html; charset=utf-8", body);

    private static string Rp(decimal amount) =>
        "Rp " + amount.ToString("N0", CultureInfo.InvariantCulture)
            .Replace(",", ".");
}
```

Read the controller as a table of contents. `Index` asks the repository for all services and chooses the list page. `UpTo` asks for the services up to a price. `Details` asks for one service and chooses either the details page or, with status 404, the "missing" page. The repository contains no SQL strings; it states conditions in C#, `s => s.Id == id` and `s => s.Price <= max`, and leaves the translation to the mapper. Nothing in this file knows about HTTP, the CSV format or escaping. That is what "thin controller" means.

Listing 2.8 is the start-up code, the part that a real framework mostly writes for you: the registrations, the routes, and a loop that serves HTTP. It also contains an offline demo, started with `--demo`, that sends five requests straight to the router without a network, so that the log of every foundation can be read in one place.

Listing 2.8 `Program.cs`: registrations, routes, the offline demo and a minimal HTTP server.

```
using System.Net;
using System.Text;
using MiniFrame;

// Wiring: what a framework does for you when the application starts.
var container = new Container();
container.Instance(new Table<Service>("service", Csv.Read("data/Service.csv")));
container.Instance(new Templates("templates"));
container.Singleton<IServiceRepository, CsvServiceRepository>();
container.Transient<ServicesController, ServicesController>();

var router = new Router();
router.Add("GET", "/services",
    _ => container.Resolve<ServicesController>().Index());
router.Add("GET", "/services/{id}",
    v => container.Resolve<ServicesController>().Details(v["id"]));
router.Add("GET", "/services/under/{max:int}",
    v => container.Resolve<ServicesController>().UpTo(int.Parse(v["max"])));

Response Handle(string method, string path)
{
    var (handler, values) = router.Match(method, path);
    string shown = string.Join(", ", values.Select(p => $"{p.Key}={p.Value}"));
    Console.WriteLine($"  [router] {method} {path} -> "
        + (handler is null ? "no route" : $"handler {{ {shown} }}"));
    return handler is null
        ? new Response(404, "text/plain; charset=utf-8", $"Nothing at {path}")
        : handler(values);
}

if (args.Length > 0 && args[0] == "--demo")
{
    // Answers a few requests without a network, to show each foundation.
    foreach (string path in new[] { "/services/S0004", "/services/under/15000",
                 "/services/under/cheap", "/services/%3Cb%3E", "/prices" })
    {
        Console.WriteLine($"REQUEST GET {path}");
        var r = Handle("GET", path);
        string Inside(string tag) => r.Body.Contains($"<{tag}>")
            ? r.Body.Split($"<{tag}>")[1].Split($"</{tag}>")[0] : "";
        Console.WriteLine($"RESPONSE {r.Status} {r.ContentType}");
        Console.WriteLine(r.ContentType.StartsWith("text/plain")
            ? $"  {r.Body}\n"
            : $"  <h1>{Inside("h1")}</h1> <p>{Inside("p")}</p> "
              + $"rows: {r.Body.Split("<tr>").Length - 1}\n");
    }
```

Listing 2.8 (Continued) *Program.cs: registrations, routes, the offline demo and a minimal HTTP server.*

```
    return;
}

var listener = new HttpListener();
listener.Prefixes.Add("http://127.0.0.1:4020/");
listener.Start();
Console.WriteLine("MiniFrame on http://127.0.0.1:4020/services (Ctrl+C stops)");
while (true)
{
    var context = listener.GetContext();
    var r = Handle(context.Request.HttpMethod, context.Request.Url!.AbsolutePath);
    byte[] bytes = Encoding.UTF8.GetBytes(r.Body);
    context.Response.StatusCode = r.Status;
    context.Response.ContentType = r.ContentType;
    context.Response.OutputStream.Write(bytes);
    context.Response.Close();
}
```

Notice that the handlers do not create controllers with `new`; they ask the container, `container.Resolve<ServicesController>()`. That is how a real framework calls your controllers too: it resolves them from its container, which is why a controller can simply declare what it needs.

2.4 Running It

Open a terminal in `Chapter02/miniframe` and run the demo:

```
dotnet run -- --demo
```

The two dashes separate the arguments for `dotnet` from the arguments for our program. This is the real output; the indented lines are what each foundation printed while it worked.

```
REQUEST GET /services/S0004
  [router] GET /services/S0004 -> handler { id=S0004 }
  [container] built CsvServiceRepository
  [container] built ServicesController
  [mapper] SQL: SELECT * FROM service WHERE id = 'S0004'
RESPONSE 200 text/html; charset=utf-8
  <h1>S0004</h1> <p>Dry cleaning - shirt/blouse (per item): Rp 20.000</p> rows: 0

REQUEST GET /services/under/15000
  [router] GET /services/under/15000 -> handler { max=15000 }
  [container] built ServicesController
  [mapper] SQL: SELECT * FROM service WHERE price <= 15000
RESPONSE 200 text/html; charset=utf-8
  <h1>Services up to Rp 15.000</h1> <p></p> rows: 4

REQUEST GET /services/under/cheap
  [router] GET /services/under/cheap -> no route
RESPONSE 404 text/plain; charset=utf-8
  Nothing at /services/under/cheap

REQUEST GET /services/%3Cb%3E
  [router] GET /services/%3Cb%3E -> handler { id=<b> }
  [container] built ServicesController
  [mapper] SQL: SELECT * FROM service WHERE id = '<b>'
RESPONSE 404 text/html; charset=utf-8
  <h1>Not found</h1> <p>There is no service with the code &lt;b&gt;.</p> rows: 0

REQUEST GET /prices
  [router] GET /prices -> no route
RESPONSE 404 text/plain; charset=utf-8
  Nothing at /prices
```

Each block tells a small story.

- **`/services/S0004`.** The router chose the details route and extracted `id=S0004`. The container built the repository (first use of a singleton) and then the controller. The repository's condition became `WHERE id = 'S0004'`. The page's heading and paragraph show the service.
- **`/services/under/15000`.** This time the container built only the controller: the repository is a singleton and already existed. The condition `s.Price <= max` became `price <= 15000`: the mapper replaced the captured variable `max` by its value. Four services cost at most Rp 15.000: S0001, S0002, S0003 and S0010.
- **`/services/under/cheap`.** No route fits: the third route's constraint refuses `cheap`, and the other two have the wrong number of segments. Nobody was built, nothing was queried, and the answer is a plain 404.

- **`/services/%3Cb%3E`.** The router decoded the value to `<b>`; the repository found nothing; the "missing" page printed the code as `<b>`, escaped by the template engine without the controller doing anything. Note also the SQL: the value `<b>` is inside quotes as a literal. Section 2.6 explains why real ORMs send such values as *parameters* instead.
- **`/prices`.** No route at all.

The same classes also serve real HTTP. `dotnet run` without `--demo` starts the server on port 4020; `Chapter02/verify.sh` asks it four questions with `curl`. The status codes were `200` for `/services`, `/services/S0010` and `/services/under/7000`, and `404` for `/services/under/x`, and the body for `/services/under/7000` was the following page, the layout wrapped around the list with one row.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Services up to Rp 7.000 - Profitina Laundry</title>
</head>
<body>
  <header><a href="/services">Profitina Laundry</a></header>
  <main>
<h1>Services up to Rp 7.000</h1>
<table>
  <tr><td><a href="/services/S0003">S0003</a></td>
      <td>Ironing only (per kg)</td><td>Rp 6.000</td></tr>
</table>

  </main>
</body>
</html>
```

2.5 The Same Feature in the Three Real Frameworks

Now that every foundation has been built once by hand, we can read the real frameworks. This section writes the same three addresses (the list, the details, the services up to a price) in Laravel and in Spring Boot, and points to the ASP.NET Core version that Chapter 3 builds in full. The Laravel and Spring files are in `Chapter02/snippets`. As Chapter 1 explained, the machine that built this book could not download Laravel or Spring, so these files were **checked but not run**: the PHP files with `php -l`, which reports syntax errors, and the Java files with `javac` against a handful of tiny stand-in types that imitate the few Spring and Jakarta names they use (`snippets/spring/stubs`), which checks syntax and types. Chapters 7 and 11 turn them into running applications. They follow the current documentation of Laravel 13 and Spring Boot 4 [5], [6].

2.5.1 Routing

In **Laravel**, routes live in one central file, `routes/web.php`, and read almost like sentences. **Listing 2.9** is the whole routing of the price list.

Listing 2.9 `routes/web.php` in Laravel: three routes, each pointing to a controller method; `whereNumber` is the constraint.

```php
<?php

use App\Http\Controllers\ServiceController;
use Illuminate\Support\Facades\Route;

Route::get('/services', [ServiceController::class, 'index']);
Route::get('/services/under/{max}', [ServiceController::class, 'under'])
    ->whereNumber('max');
Route::get('/services/{id}', [ServiceController::class, 'show']);
```

`Route::get` pairs the method `GET` and a template with an action, written as an array of the controller class and the method name. Parameters are in braces, as in MiniFrame, and `->whereNumber('max')` is the constraint that MiniFrame spelled `{max:int}` [5]. Laravel matches routes in the order they are registered, like MiniFrame; that is why the more specific `/services/under/{max}` is written before `/services/{id}`, even though here the segment counts already keep them apart.

In **Spring Boot**, routes are annotations on the controller's methods, in the style ASP.NET Core calls *attribute routing*: `@RequestMapping("/services")` on the class gives a prefix, and `@GetMapping("/{id}")` on a method adds the rest [7]. A constraint is written as a regular expression after a colon, `{max:\\d+}`, "one or more digits". Spring does not depend on the order: it chooses the most specific matching pattern.

In **ASP.NET Core**, both styles exist. Chapter 3 uses a *conventional* route, one central pattern `{controller=Home}/{action=Index}/{id?}` that maps `/Outlets/Details/O01` to the method `Details` of `OutletsController` with `id = "O01"`, and *attribute routes* such as `[HttpGet("under/{max:int:min(1000)}")]` on the methods of a controller marked `[Route("services")]` [1].

2.5.2 Dependency injection

All three frameworks have a container, and in all three a controller receives its dependencies through its constructor. The difference is in how the registrations are made.

- **ASP.NET Core** registers services explicitly in `Program.cs`, with the lifetime in the method's name: `builder.Services.AddSingleton<ILaundryCatalog, CsvCatalog>()`, `AddScoped<…>()`, `AddTransient<…>()` [8]. Controllers are created by the framework for each request and receive what they declare.
- **Laravel**'s *service container* resolves most classes without any registration, by reflecting on their constructors exactly as MiniFrame does (it calls this *zero configuration resolution*); an interface needs one line in a service provider, `$this->app->singleton(ServiceRepository::class, EloquentServiceRepository::class)` [9].
- **Spring** discovers classes marked `@Component`, `@Service`, `@Repository` or `@Controller` by scanning the application's packages, creates one of each (singleton is Spring's default *scope*) and injects them through constructors; when a class has a single constructor, no annotation is even needed on it [10]. In **Listing 2.18** below, the constructor `ServiceController(ServiceRepository repository)` is the whole of the injection.

The words differ; the idea is MiniFrame's `Container`: declare what you need, register once what answers the need, let the container build the graph.

2.5.3 Object-relational mapping

Two families of ORM exist, named by Martin Fowler in *Patterns of Enterprise Application Architecture* [11]. In the **Active Record** pattern, a model object knows how to load and save itself:

`Service::find('S0004')`, `$service->save()`. In the **Data Mapper** pattern, the model is a plain object, and a separate object (a *context*, a *session*, a *repository*) moves it to and from the database. Laravel's Eloquent is an Active Record ORM; Entity Framework Core and Spring Data JPA (with Hibernate underneath) are Data Mappers. MiniFrame's `Table<T>` is a small Data Mapper.

Listing 2.10 is the Eloquent model of the service table. It is almost empty, because Eloquent maps by convention: the class `Service` would map to a table `services` with an auto-increment integer key, and the four lines inside only state where Profitina Laundry's database differs from that convention [12].

Listing 2.10 `app/Models/Service.php`: an Eloquent model. The properties state only what differs from Laravel's conventions.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

// One row of the table "service" becomes one Service object.
class Service extends Model
{
    protected $table = 'service';
    protected $keyType = 'string';     // ids look like "S0004"
    public $incrementing = false;
    public $timestamps = false;        // the table has no created_at/updated_at
}
```

The Spring version has two parts. **Listing 2.11** is the *entity*, a plain class whose annotations tell JPA which table and which key; **Listing 2.12** is the *repository*, which is only an interface.

Listing 2.11 `Service.java`: a JPA entity mapped to the table `service`.

```java
package com.profitina.laundry;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import java.math.BigDecimal;

// One row of the table "service" becomes one Service object.
@Entity
@Table(name = "service")
public class Service {
    @Id
    private String id;
    private String description;
    private BigDecimal price;

    protected Service() {}                    // JPA needs an empty constructor

    public String getId() { return id; }
    public String getDescription() { return description; }
    public BigDecimal getPrice() { return price; }
}
```

Listing 2.12 `ServiceRepository.java`: a Spring Data repository. The query is derived from the method's name.

```java
package com.profitina.laundry;

import java.math.BigDecimal;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

// No class to write: Spring Data reads the method name and writes the query.
public interface ServiceRepository extends JpaRepository<Service, String> {
    List<Service> findByPriceLessThanEqual(BigDecimal max);
}
```

Nobody writes a class that implements `ServiceRepository`. When the application starts, Spring Data creates one: `findAll` and `findById` come from `JpaRepository`, and for `findByPriceLessThanEqual` it *parses the method name*, "find by `price`, less than or equal", and derives the query `… where s.price <= ?1` [6]. It is the same translation as MiniFrame's expression tree, starting from a name instead of a lambda. Entity Framework Core starts from a lambda, exactly like MiniFrame: `db.Services.Where(s => s.Price <= max)` becomes SQL [4].

2.5.4 Templates

The three frameworks have three template languages, and each has the three features of **Listing 2.4**: escaped holes, loops and a layout.

Blade, Laravel's engine, writes an escaped hole as `{{ $value }}` and a raw one as `{!! $value !!}`; directives such as `@foreach` and `@if` start with `@`; a page names its layout with `@extends` and fills the layout's `@yield` slots with `@section` [13]. **Listing 2.13** and **Listing 2.14** are the layout and the list page.

Listing 2.13 `resources/views/layouts/app.blade.php`: the Blade layout; `@yield` marks the slots.

```blade
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>@yield('title') - Profitina Laundry</title>
</head>
<body>
  <header><a href="/services">Profitina Laundry</a></header>
  <main>
    @yield('content')
  </main>
</body>
</html>
```

Listing 2.14 `resources/views/services/index.blade.php`: the list page, filling the layout's two slots.

```blade
@extends('layouts.app')

@section('title', $title)

@section('content')
  <h1>{{ $title }}</h1>
  <table>
    @foreach ($services as $s)
      <tr><td><a href="/services/{{ $s->id }}">{{ $s->id }}</a></td>
          <td>{{ $s->description }}</td><td>{{ $s->price }}</td></tr>
    @endforeach
  </table>
@endsection
```

The controller that feeds this view is **Listing 2.15**. Each method returns `view('services.index', [...])`, the name of the template and its model, exactly like MiniFrame's `views.Page("services.html", model)`.

Listing 2.15 `app/Http/Controllers/ServiceController.php`: the Laravel controller. `findOrFail` answers 404 by itself when the code does not exist.

```php
<?php

namespace App\Http\Controllers;

use App\Models\Service;
use Illuminate\View\View;

class ServiceController extends Controller
{
    public function index(): View
    {
        return view('services.index', [
            'title' => 'Our services',
            'services' => Service::all(),
        ]);
    }

    public function under(int $max): View
    {
        return view('services.index', [
            'title' => 'Services up to Rp ' . number_format($max, 0, ',', '.'),
            'services' => Service::where('price', '<=', $max)->get(),
        ]);
    }

    public function show(string $id): View
    {
        return view('services.show', ['service' => Service::findOrFail($id)]);
    }
}
```

Thymeleaf, the engine used with Spring Boot in this book, takes a different road: its templates are valid HTML files whose attributes, all starting with `th:`, tell the engine what to replace [14]. `th:text` fills an element with an escaped value (`th:utext` would not escape), `th:each` repeats an element, and `th:replace` inserts a *fragment* from another file, which is how Thymeleaf builds layouts. The placeholder text inside the elements (`Our services`, `S0001`) is shown when a designer opens the file directly in a browser, and replaced when Spring renders it. **Listing 2.16** holds the shared fragments and **Listing 2.17** is the list page.

Listing 2.16 `templates/layout.html`: two Thymeleaf fragments, the head (with a parameter) and the header.

```html
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:fragment="head(title)">
  <meta charset="utf-8">
  <title th:text="${title} + ' - Profitina Laundry'">Profitina Laundry</title>
</head>
<body>
  <header th:fragment="header">
    <a th:href="@{/services}">Profitina Laundry</a>
  </header>
</body>
</html>
```

Listing 2.17 `templates/services/index.html`: the list page in Thymeleaf; `th:each` is the loop, `th:text` the escaped hole.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="~{layout :: head(${title})}"></head>
<body>
  <header th:replace="~{layout :: header}"></header>
  <main>
    <h1 th:text="${title}">Our services</h1>
    <table>
      <tr th:each="s : ${services}">
        <td><a th:href="@{/services/{id}(id=${s.id})}"
               th:text="${s.id}">S0001</a></td>
        <td th:text="${s.description}">Wash, dry &amp;amp; fold (per kg)</td>
        <td th:text="${s.price}">8000.00</td>
      </tr>
    </table>
  </main>
</body>
</html>
```

Listing 2.18 is the Spring controller. Its methods put the model into a `Model` object and return the template's name.

Listing 2.18 `ServiceController.java`: the Spring MVC controller, with its repository injected through the constructor.

```
package com.profitina.laundry;

import java.math.BigDecimal;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.server.ResponseStatusException;

@Controller
@RequestMapping("/services")
public class ServiceController {
    private final ServiceRepository repository;

    public ServiceController(ServiceRepository repository) {   // injected
        this.repository = repository;
    }

    @GetMapping
    public String index(Model model) {
        model.addAttribute("title", "Our services");
        model.addAttribute("services", repository.findAll());
        return "services/index";
    }

    @GetMapping("/under/{max:\\d+}")
    public String under(@PathVariable int max, Model model) {
        model.addAttribute("title", "Services up to Rp " + max);
        model.addAttribute("services",
            repository.findByPriceLessThanEqual(BigDecimal.valueOf(max)));
        return "services/index";
    }

    @GetMapping("/{id}")
    public String show(@PathVariable String id, Model model) {
        Service service = repository.findById(id).orElseThrow(
            () -> new ResponseStatusException(HttpStatus.NOT_FOUND));
        model.addAttribute("service", service);
        return "services/show";
    }
}
```

Razor, ASP.NET Core's engine, is the subject of Chapter 3. It marks C# with a single `@`: `@Model.Name` is an escaped hole, `@foreach (var s in Model) { … }` a loop, `@Html.Raw(x)` the deliberate exception, and a shared `_Layout.cshtml` holds the frame, into which `@RenderBody()` inserts each page [15].

2.5.5 One feature, three frameworks

Table 2.2 collects the section into one table, with MiniFrame as the first column. Read it by rows: each row is one idea, which you now know from the inside, spelled four ways.

Table 2.2 The four foundations in MiniFrame and in the three frameworks of this book.

Foundation	MiniFrame	ASP.NET Core	Laravel	Spring Boot
Route table	`router.Add("GET", ...)`	`MapControllerRoute`, `[HttpGet]`	`Route::get(...)` in `routes/web.php`	`@GetMapping`
Parameter	`{id}`	`{id}`	`{id}`	`{id}`
Constraint	`{max:int}`	`{max:int}`	`->whereNumber('max')`	`{max:\\d+}`
Registration	`container.Singleton<I, C>()`	`builder.Services.AddSingleton<I, C>()`	`$this->app->singleton(I::class, C::class)`	`@Repository`, `@Service` (scanned)
Injection	constructor	constructor	constructor	constructor
Row to object	`Table<Service>`	EF Core entity	Eloquent model	JPA `@Entity`
Query	`table.Where(s => s.Price <= max)`	`db.Services.Where(s => s.Price <= max)`	`Service::where('price', '<=', $max)`	`findByPriceLessThanEqual(max)`
Escaped hole	`{{ x }}`	`@x`	`{{ $x }}`	`th:text="${x}"`
Raw hole	`{{{ x }}}`	`@Html.Raw(x)`	`{!! $x !!}`	`th:utext="${x}"`
Loop	`{{#each rows}}`	`@foreach`	`@foreach`	`th:each`
Layout	`layout.html`, `{{{ body }}}`	`_Layout.cshtml`, `@RenderBody()`	`@extends`, `@yield`	fragments, `th:replace`

2.6 Making It Better: What MiniFrame Leaves Out

MiniFrame is correct for what it does, but a real framework does a great deal more, and each missing piece is a lesson about why frameworks are large.

- **Parameters, not literals, in SQL.** MiniFrame prints values into the SQL text (`WHERE id = '<b>'`), doubling any quote. Real ORMs send the SQL with placeholders (`WHERE id = @p0`) and the values separately, so that no value, however strange, can ever change the meaning of the statement. Parameterized queries are the primary defense against SQL injection [16].
- **Route precedence and speed.** MiniFrame tries routes in order, one by one. Real routers compile the route table into a tree and rank templates, so that the result does not depend on the order of registration and matching stays fast with thousands of routes [1].
- **Scoped lifetime and disposal.** MiniFrame knows singletons and transients. Real containers add the scoped lifetime and dispose of objects (closing database connections, for example) at the end of each request [8].
- **Concurrency.** MiniFrame's server handles one request at a time, and its singleton dictionary is not safe for simultaneous access. Real frameworks are built for thousands of simultaneous requests.

- **Compiled templates.** MiniFrame reads and parses each template file on every request. Razor compiles templates into C# classes when the project is built; Blade and Thymeleaf cache the compiled form.
- **Migrations.** MiniFrame cannot create or change a table. Each real ORM has migrations (Chapters 5, 9 and 11).
- **Everything else** in the table of Section 1.2.4: model binding, validation, security, configuration, error pages, logging.

Adding these to MiniFrame would be an excellent way to spend a year and a poor way to run a laundry. That is the point: the machinery is understandable, and it is also a great deal of work to do well, which is why we use frameworks built and tested by thousands of people.

2.7 Using It for Real

2.7.1 Reading any framework through the four foundations

The four foundations are a reading guide for any web framework you meet, including ones this book does not teach. Faced with new documentation, ask four questions. *Where is the route table, and how are parameters and constraints written? Where are services registered, with what lifetimes, and how does a class receive them? How is a table mapped to a class, and how is a query written? What are the escaped hole, the raw hole, the loop and the layout called?* Answer those four and you can read most of the framework's example code on the first day. **Table 2.2** is the answer sheet for our three.

2.7.2 Profitina in practice

Profitina's FastAPI application answers the four questions in its own way. Its routes are Python functions decorated with the path, `@router.get("/api/me/dashboard")`, with parameters and types taken from the function's signature. Its dependency injection is the `Depends(...)` marker on a parameter: the handler that builds a dashboard declares that it needs the current user and a database session, and FastAPI supplies both for each request, the session with a per-request lifetime. Its data access goes through an ORM layer that writes parameterized SQL for PostgreSQL in production and SQLite in parts of the test suite, and a guard in the test suite keeps the two SQL dialects honest. Its HTML pages use a template engine with escaping by default. None of this is exotic; it is Section 2.2 with Python names.

2.7.3 Planning a new feature with the four questions

The four questions also work in the other direction, for *planning*. When Bu Wati asked Dimas for a page that lists the three outlets with their addresses, Dimas did not start by typing code. He wrote four short lines on paper, one per foundation. *Route*: `GET /outlets` for the list and `GET /outlets/{id}` for one outlet, where the id looks like `O01`. *Dependencies*: the controller needs something that knows the outlets; that is a new method on the existing catalog, not a new class, so no new registration is needed. *Data*: the table `Outlet` has the columns `id`, `name`, `address`, `phone` and `opened`, so the model is a record with five properties of the same names, and the only query is "all outlets" or "the outlet with this id". *Template*: a list page that loops over the outlets and a details page, both inside the existing layout, with every value in an escaped hole.

With those four lines, the work in any of the three frameworks became a matter of spelling. In ASP.NET Core the plan turned into the `OutletsController` of Chapter 3, reached through the conventional route; in Laravel it would be two lines in `routes/web.php`, an Eloquent model

`Outlet` and two Blade views; in Spring it would be an entity, a repository interface with no methods of its own, a controller and two Thymeleaf templates. The plan did not change between the three; only the file names did. Planning in foundations first also exposes questions early. Should an outlet that closed still be listed? That is a data question and belongs in the query, not in the template. Should the phone number be a clickable `tel:` link? That is a template question and changes nothing else. Keeping each decision in its own foundation is Bu Wati's rule of Chapter 1, "one change in one place", applied to design.

2.8 Common Mistakes, and How to Avoid Them

Registering routes in the wrong order in an order-sensitive router. In MiniFrame and in Laravel, a general route registered before a specific one swallows its requests (Exercise 1). Register the specific routes first, or add constraints so that the general one cannot match.

Creating dependencies with `new` inside a class that the container builds. It works, and it quietly defeats the container: the class can no longer be given a different implementation or a test double. Declare the dependency in the constructor instead.

Choosing the wrong lifetime. A singleton that holds per-request data (the current user, a shopping basket) mixes one visitor's data into another's page. A transient that opens an expensive resource on every use wastes it. Ask of every registration: *may two requests share this object?*

Building SQL by concatenating values. Even with an ORM, most have an escape hatch for raw SQL (`FromSqlRaw`, `DB::raw`, `@Query(nativeQuery = true)`). Never paste a value from a request into such a string; use the parameter mechanism the ORM offers [16].

Switching escaping off to "fix" a display problem. When `&` appears on a page, the cause is usually text that was escaped twice, not escaping itself. Find where the second escaping happens; do not reach for the raw hole.

Mistaking the syntax for the idea. Beginners often learn `@GetMapping`, `Route::get` and `[HttpGet]` as three unrelated facts. Learn the idea once, and each syntax becomes a detail.

2.9 Chapter Summary

- Beneath their different syntax, ASP.NET Core, Laravel and Spring Boot share Model–View–Controller and four foundations: routing, dependency injection, object-relational mapping and templates.
- A router matches the method and path of a request against templates made of literal segments and parameters, applies constraints, and passes the extracted values to one action. Real routers rank templates by precedence instead of trying them in order.
- A DI container builds objects from registrations, resolving constructor parameters recursively. Singleton, scoped and transient lifetimes say how long an object lives and who shares it. Depending on interfaces makes implementations replaceable.
- An ORM maps rows to objects by matching names, and turns a query written in the programming language into SQL, through an expression tree (EF Core, MiniFrame), a fluent builder (Eloquent) or a method name (Spring Data). Real ORMs always send values as parameters.
- A template engine fills escaped holes, repeats blocks for lists and inserts pages into a shared layout. Escaping is the default; raw output is the exception that must be justified.

- MiniFrame implements all four in 226 lines. Its omissions (parameters in SQL, precedence, scoped lifetimes, concurrency, compiled templates, migrations) show why real frameworks are large.

2.10 Exercises

1. Add a route `GET /services/cheapest` to MiniFrame that answers the plain text `cheapest`. First register it *after* the route `/services/{id}` and run the demo with this address; then register it *before* that route. Explain both results. How would ASP.NET Core behave?

2. What SQL does MiniFrame's mapper print for the condition `s => s.Price > 10000 && s.Price < 40000`, and which services does the condition select?

3. Remove the registration of `IServiceRepository` from `Program.cs` and request a `ServicesController`. What happens, and at which line of **Listing 2.2**?

4. In `templates/services.html`, change `{{ description }}` to `{{{ description }}}` and open `/services`. What changes in the HTML of the first row? When would the difference become dangerous?

5. The demo printed `[container] built ServicesController` for every request but `[container] built CsvServiceRepository` only once. Explain from **Listing 2.2** and **Listing 2.8**. What would go wrong if the repository were registered as transient and read a large file in its constructor?

6. Write the query "services priced between Rp 10.000 and Rp 40.000" in Eloquent and as a Spring Data method name, using the documentation of each [12], [6].

2.11 Solutions to the Exercises

1. Registered after `/services/{id}`, the new route is never reached: the router tries `/services/{id}` first, it fits (two segments, and `{id}` accepts anything), and the demo prints `handler { id=cheapest }`, the SQL `WHERE id = 'cheapest'`, and a 404 page "There is no service with the code cheapest." Registered before, the router meets the literal template first and the demo prints `handler {  }` and `RESPONSE 200 text/plain` with the text `cheapest`. Both results come from real runs of the modified program. ASP.NET Core does not depend on the order: a literal segment has higher precedence than a parameter, so `/services/cheapest` would reach the literal route in both cases [1].

2. The mapper prints `SELECT * FROM service WHERE price > 10000 AND price < 40000`: the `&&` node became `AND`, and each comparison became column, operator and literal. A run selected S0002 (12.000), S0004 (20.000), S0008 (35.000), S0009 (25.000), S0010 (15.000) and S0012 (30.000). S0001 and S0003 are too cheap, and S0005, S0006, S0007 and S0011 are too expensive; S0007 costs exactly 40.000, which `<` excludes.

3. `Resolve` throws `InvalidOperationException: Nothing registered for IServiceRepository.` It happens inside the recursive call for the controller's first constructor parameter: `map.TryGetValue` fails for `IServiceRepository` and the `throw` at the top of `Resolve` runs. A real container gives the same kind of message, naming the type it could not resolve and the class that needed it; reading that message is the fastest way to find a missing registration.

4. The first row's description changes from `Wash, dry & fold (per kg)` to `Wash, dry & fold (per kg)`. A browser happens to show both the same way, because a lone `&` followed by a

space is tolerated. The difference becomes dangerous as soon as a description contains markup: if the owner (or anyone with access to the data) saved a description containing `<script>`, the raw hole would send it to every visitor as code. The escaped hole is the safe default; the raw hole is only for HTML the program itself produced, such as the page body inside the layout.

5. The controller is registered with `Transient`, so `Resolve` never stores it in `singles` and builds a new one for every request. The repository is registered with `Singleton`; the first `Resolve` stores it, and later requests find it in `singles` and return it without building. If the repository were transient and read a large file in its constructor, every request would read the file again, making every page slow; and since each request would get its own copy, a change made through one copy (for example a price update kept in memory) would be invisible to the others.

6. In Eloquent, `Service::whereBetween('price', [10000, 40000])->get();` `whereBetween` includes both ends, as SQL's `BETWEEN` does [12]. In Spring Data, the method `List<Service> findByPriceBetween(BigDecimal min, BigDecimal max);` in `ServiceRepository`; the keyword `Between` in the name derives `where s.price between ?1 and ?2` [6]. Neither was run on the build machine; Chapters 7 and 11 run such queries against the real database.

References

[1] Microsoft, "Routing in ASP.NET Core," *ASP.NET Core Documentation*, for ASP.NET Core 10.0, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/routing

[2] M. Fowler, "Inversion of control containers and the dependency injection pattern," martinfowler.com, Jan. 2004. [Online]. Available: https://martinfowler.com/articles/injection.html

[3] Microsoft, "Expression trees," *C# Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/dotnet/csharp/advanced-topics/expression-trees/

[4] Microsoft, "Entity Framework Core overview," *EF Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/

[5] Laravel, "Routing," *Laravel 13.x Documentation*, 2026. [Online]. Available: https://laravel.com/docs/13.x/routing

[6] Spring, "JPA query methods," *Spring Data JPA Reference Documentation*, accessed Sep. 2026. [Online]. Available: https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

[7] Spring, "Mapping requests," *Spring Framework Reference Documentation: Web MVC*, accessed Sep. 2026. [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html

[8] Microsoft, "Dependency injection in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/dependency-injection

[9] Laravel, "Service container," *Laravel 13.x Documentation*, 2026. [Online]. Available: https://laravel.com/docs/13.x/container

[10] Spring, "Dependency injection," *Spring Framework Reference Documentation: The IoC Container*, accessed Sep. 2026. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html

[11] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002.

[12] Laravel, "Eloquent: getting started," *Laravel 13.x Documentation*, 2026. [Online]. Available: https://laravel.com/docs/13.x/eloquent

[13] Laravel, "Blade templates," *Laravel 13.x Documentation*, 2026. [Online]. Available: https://laravel.com/docs/13.x/blade

[14] The Thymeleaf Team, “Tutorial: Using Thymeleaf,” version 3.1, accessed Sep. 2026. [Online]. Available: https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.html

[15] Microsoft, “Razor syntax reference for ASP.NET Core,” *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/views/razor

[16] OWASP Foundation, “SQL injection prevention cheat sheet,” *OWASP Cheat Sheet Series*, accessed Sep. 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

Chapter 3

ASP.NET Core MVC I: Project, Pipeline, Routing, Controllers and Razor Views

This chapter builds Profitina Laundry's public website with ASP.NET Core MVC: a home page, the list of services with their prices, a page per service, the services up to a chosen price, a price-quote form that applies the shop's real rules, a page per outlet, an "about" page, friendly error pages and a small JSON endpoint for other programs. Along the way it takes a new ASP.NET Core project apart file by file. By the end of the chapter you will be able to create an MVC project with the `dotnet` command and explain every file in it; explain the request pipeline and write middleware of your own; register services in the dependency-injection container and receive them in controllers and views; route requests with a conventional route and with attribute routes, including constraints; write controller actions that return views, JSON, redirects and status codes; bind a form to a C# object and report input errors; and write Razor views with a layout, a partial view, sections and tag helpers. Everything in this chapter was built and run on the machine that produced the book; every output shown is real, and the screenshots were taken from the running application.

3.1 Bu Wati's Website

Chapter 1 left Bu Wati with a price list that other programs could read as JSON, and Chapter 2 opened up the machinery that frameworks use. Now she wants what her customers will actually see. She wrote the wish list on the back of an order ticket:

- a **home page** that says what the shop is and shows the three services people ask for most;
- the **list of services** with prices, and a **page for each service**;
- a quick way to see **the services up to a price**, for customers on a budget;
- a **price quote**: choose a service, type the weight or the number of items, tick "pickup and delivery", and see what it will cost, *with the shop's rules*: at least 3 kg for services charged by weight, items counted whole, pickup Rp 10.000 but free from Rp 100.000;
- the **outlets**, each with its address and phone number;
- an **about** page with the opening hours;
- no ugly error pages: a mistyped address should get a polite page in the shop's style;
- and, for Dimas, who dreams of a mobile app, the service data as **JSON**.

Bu Wati also has two rules of her own, both learned the hard way with the old site. The price rules must live in one place, so that the day the minimum changes to 2.5 kg only one line changes. And the site must look the same on every page without anyone copying a header. Chapter 1 called the first rule "keep controllers thin" and Chapter 2 showed that the second is what a layout is for. This chapter builds all of it with ASP.NET Core 10, using nothing but the shared framework that comes with the SDK. The data are the same CSV exports as before (`data/Service.csv` and `data/Outlet.csv`); Chapter 5 moves them into a database.

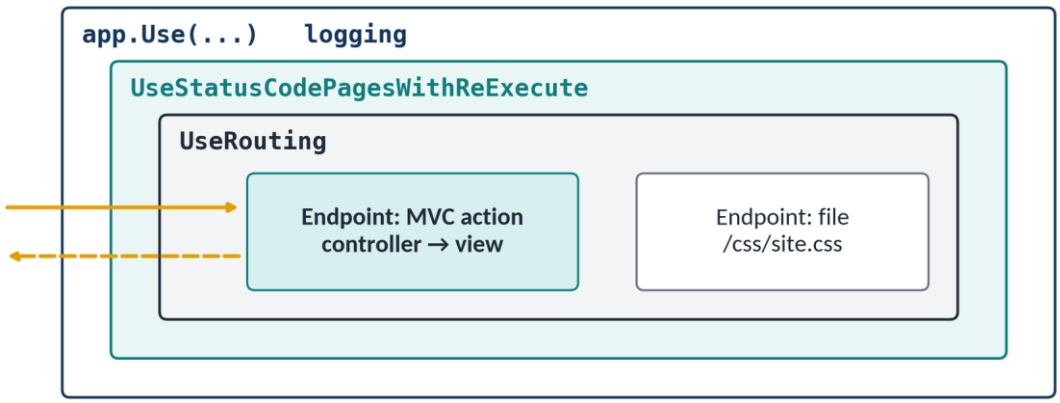
3.2 The Idea in Plain Words

3.2.1 The host and the pipeline

An ASP.NET Core application is a console program that builds a **host**, the object that owns the web server (Kestrel), the configuration, the logging and the dependency-injection container, and then runs it. Every request that Kestrel receives is turned into an `HttpContext` object, which holds the request and the response being built, and is passed through the **request pipeline**.

> **DEFINITION.** The **request pipeline** is an ordered chain of **middleware** components. Each middleware receives the `HttpContext` and a reference to the *next* middleware; it may do something before calling the next one, call it (or not), and do something after it returns. The last element of the pipeline is an **endpoint**: the code chosen to produce the response, such as a controller action or a static file.

Figure 3.1 shows the pipeline of this chapter's application as nested layers. A request enters through every layer from the outside in, reaches an endpoint, and the response goes back out through the same layers in reverse. The outermost layer logs every request with its status and duration, which it can only know on the way *out*. The next layer, status-code pages, notices responses such as 404 on the way out and replaces them with a friendly page. The routing layer chooses the endpoint. Because middleware wraps middleware, **order matters**: a layer can only see and change what passes through it [1].

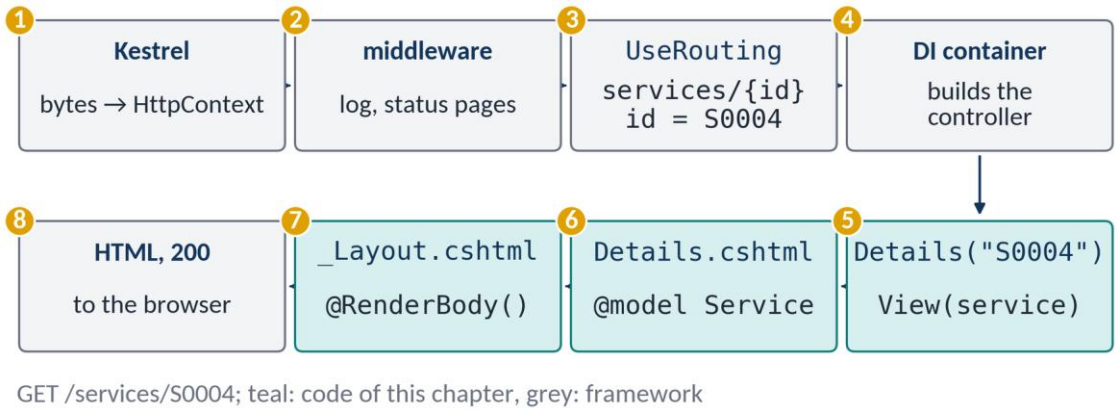


amber: the request goes in through every layer; the response comes back out through the same layers (dashed)

Figure 3.1 The request pipeline of LaundryMvc: every request passes through the logging, status-code and routing middleware to an endpoint, and the response comes back through the same layers.

3.2.2 One request through MVC

Figure 3.2 follows a single request, `GET /services/S0004`, through all the pieces this chapter writes. Kestrel turns bytes into an `HttpContext` (1); the middleware of **Figure 3.1** lets it through (2); routing matches the template `services/{id}` and records `id = "S0004"` (3); the dependency-injection container builds a `ServicesController`, handing it the catalog and the price calculator (4); the framework calls the action `Details("S0004")`, which returns `View(service)` (5); Razor renders `Details.cshtml` with the service as its model (6) inside `_Layout.cshtml` (7); and the HTML goes back with status 200 (8). Only steps 5, 6 and 7 are code of this chapter [2]; the rest is the framework, doing what MiniFrame did in Chapter 2, at industrial strength.



GET /services/S0004; teal: code of this chapter, grey: framework

Figure 3.2 One request, `GET /services/S0004`, through LaundryMvc: from bytes to `HttpContext`, through middleware and routing, to a controller built by the container, a view and the layout.

3.3 Creating the Project

3.3.1 The MVC template

Chapter 1 created a project with `dotnet new mvc -o FirstMvc`. We start the same way, calling the project `LaundryMvc`:

```
dotnet new mvc -o LaundryMvc
cd LaundryMvc
```

The template creates the files of **Table 3.1**. Two folders, `bin` and `obj`, appear after the first build and hold compiled output; they are never edited and never committed to version control.

Table 3.1 The files created by `dotnet new mvc`, and what each one is for.

File or folder	What it is for
`LaundryMvc.csproj`	The project: the web SDK, the target `net10.0`, nullable checking, implicit `using`s
`Program.cs`	Builds the host: registers services, then builds the pipeline, then runs
`appsettings.json`, `appsettings.Development.json`	Configuration, such as log levels; the second overrides the first in Development
`Properties/launchSettings.json`	How `dotnet run` and VS Code start the app: address, environment
`Controllers/HomeController.cs`	A first controller with the actions `Index`, `Privacy` and `Error`
`Models/ErrorViewModel.cs`	A model for the template's error page
`Views/Home/Index.cshtml`, `Privacy.cshtml`	The views of `HomeController`
`Views/Shared/_Layout.cshtml`	The layout: the frame around every page
`Views/Shared/Error.cshtml`, `_ValidationScriptsPartial.cshtml`	Shared views
`Views/_ViewImports.cshtml`, `_ViewStart.cshtml`	Settings for every view in the folder: namespaces, tag helpers, the layout
`wwwroot/`	Static files served as they are: `css/`, `js/`, `favicon.ico`, and `lib/` with Bootstrap and jQuery (9.5 MB)

The folder names are not decoration; they are the **conventions** Chapter 1 mentioned. A class in `Controllers` whose name ends in `Controller` is a controller. The view for the action `Details` of `ServicesController` is looked for first in `Views/Services/Details.cshtml` and then in `Views/Shared/Details.cshtml`. Files starting with an underscore are not pages of their own but pieces used by pages. Learning these conventions is most of learning MVC.

3.3.2 What we change in the template

LaundryMvc keeps the template's structure but makes four changes, each for a reason.

- **No Bootstrap, no jQuery.** The template copies 9.5 MB of third-party libraries into `wwwroot/lib`. Bu Wati's site needs one small style sheet, `wwwroot/css/site.css`, of fewer than forty lines in the house colors, so `lib/` is deleted and the layout links only our CSS. Without jQuery, the template's client-side validation scripts do not apply; the server checks every input anyway (Section 3.6.4), which it must do in any case.
- **No HTTPS redirection in development.** The template's pipeline starts with `app.UseHttpsRedirection()`, which sends every `http://` request to `https://`. On a development machine without a trusted certificate that only produces a warning, so LaundryMvc serves plain HTTP on `http://localhost:4030`; Chapter 6 deploys it behind HTTPS.
- **A fixed port.** `Properties/launchSettings.json` is reduced to one profile, `http`, with the address `http://localhost:4030` and the environment `Development`, shown in **Listing 3.1**.
- **Our own controllers, models and views**, replacing `Privacy` and `ErrorViewModel`, as the rest of this chapter describes.

Listing 3.1 `Properties/launchSettings.json`: `dotnet run` starts the app at `http://localhost:4030` in the Development environment.

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:4030",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Listing 3.2 is the configuration file. It sets the log level of the framework's own messages (categories starting with `Microsoft.AspNetCore`) to `Warning`, so that the console shows the start-up lines and our own request log, but not the framework's detailed messages for every request. `AllowedHosts` restricts which host names the app answers to; `*` means any, which is fine behind a proxy that checks them.

Listing 3.2 `appsettings.json`: log levels and allowed hosts.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

The project file is exactly the one of Chapter 1's minimal API (Section 1.6.1) with another name: `Sdk="Microsoft.NET.Sdk.Web"`, `net10.0`, `Nullable` and `ImplicitUsings`. `Nullable` makes the compiler distinguish `string` (never null) from `string?` (may be null) and warn when a null could slip through, which you will see in the `?` marks throughout this chapter's code.

3.4 `Program.cs`: Services, Then Pipeline

Listing 3.3 is the whole start-up code of LaundryMvc. It has two halves, and the order of the halves is the order of every ASP.NET Core application: first register what the application *has* (services), then describe what it *does* with every request (the pipeline).

Listing 3.3 `Program.cs`: the services are registered on the builder, then the pipeline is built on the application.

```
using System.Diagnostics;
using LaundryMvc.Data;
using LaundryMvc.Models;

var builder = WebApplication.CreateBuilder(args);

// 1. Services: everything the application needs, registered once.
builder.Services.AddControllersWithViews();
builder.Services.AddSingleton<ILaundryCatalog>(_ => new CsvCatalog(
    Path.Combine(builder.Environment.ContentRootPath, "data")));
builder.Services.AddSingleton<PriceCalculator>();

var app = builder.Build();

// 2. Pipeline: what happens to every request, in this order.
app.Use(async (context, next) =>
{
    var clock = Stopwatch.StartNew();
    await next(context);                // everything later runs in here
    app.Logger.LogInformation("{Method} {Path}{Query} -> {Status} in {Ms} ms",
        context.Request.Method, context.Request.Path,
        context.Request.QueryString, context.Response.StatusCode,
        clock.ElapsedMilliseconds);
});

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/error/500");
}
app.UseStatusCodePagesWithReExecute("/error/{0}");

app.UseRouting();

app.MapStaticAssets();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}")
    .WithStaticAssets();

app.Run();
```

The builder. `WebApplication.CreateBuilder(args)` prepares the host: it reads `appsettings.json` and the environment's settings, sets up logging to the console, and creates the service collection, the list of registrations for the container. `builder.Environment.ContentRootPath` is the folder the application was started from, the project folder when you use `dotnet run`.

The services. `AddControllersWithViews()` registers everything MVC needs: controller discovery, model binding, Razor view compilation and rendering, tag helpers, antiforgery tokens. The next two lines register our own services, and they are MiniFrame's `Container` in its real form [3]. `AddSingleton<ILaundryCatalog>(_ => new CsvCatalog(...))` says: "whenever anyone asks for an `ILaundryCatalog`, give them the one `CsvCatalog`, built once by this function". A function is needed because the constructor takes a folder name, a string the container cannot guess. `AddSingleton<PriceCalculator>()` registers the price rules as a class that the container can build by itself, since its constructor takes nothing. Both are singletons because both are safe to share: the catalog is read once and never changes, and the calculator has no state at all.

The pipeline. `builder.Build()` creates the application, and each `app.Use…` call appends one middleware, in the order written.

- The first middleware is written inline with `app.Use(async (context, next) => …)`. It starts a stopwatch, calls `await next(context)`, which runs *the whole rest of the pipeline*, and after it returns, when the status code is known, logs one line: method, path, query string, status and milliseconds. Everything a middleware does after `await next` happens on the way out.
- `UseExceptionHandler("/error/500")` is added only outside Development. If a later component throws an exception, it catches it, logs it and re-runs the request as `/error/500`, so that the visitor sees our error page. In Development, ASP.NET Core instead shows a detailed *developer exception page* with the stack trace [4], which is what a programmer wants and a customer must never see.
- `UseStatusCodePagesWithReExecute("/error/{0}")` watches the way out. If a response has a status between 400 and 599 and *no body*, it re-runs the pipeline with the path `/error/404` (the `{0}` becomes the status code), so that an empty 404 becomes our friendly page, still with the status 404 [5].
- `UseRouting()` matches the request against all endpoints and stores the chosen one in the `HttpContext`. Middleware placed after it can ask which endpoint was chosen; Chapter 4 uses that.
- `MapStaticAssets()` adds endpoints for the files in `wwwroot`, with compression and cache headers, based on a manifest produced at build time. `.WithStaticAssets()` on the controller route lets views link to those files with a fingerprint in the URL, which we will see in the layout [6].
- `MapControllerRoute` adds the **conventional route**, discussed in Section 3.6.1, and `app.Run()` starts Kestrel and blocks until the application stops.

Where are the controllers' endpoints themselves executed? After the last middleware: ASP.NET Core adds the endpoint-executing middleware at the end of the pipeline automatically. Our order therefore is: log, (handle exceptions), status pages, routing, endpoint. **Figure 3.1** is this list drawn as layers.

3.5 The Model: Data and Rules

"Model" in MVC means everything that is about the business and not about HTTP: the data, the rules and the code that fetches the data. LaundryMvc keeps the data classes and the rules in `Models/` and the data access in `Data/`.

3.5.1 Services, outlets and money

Listing 3.4 is the service. It is a C# **record**, a class whose identity is its values, declared with a *primary constructor*: the three parameters become three read-only properties, `Id`, `Description` and `Price`.

Listing 3.4 `Models/Service.cs`: a service with its code, description and price, and three properties derived from the description.

```
namespace LaundryMvc.Models;

// One row of data/Service.csv: something the laundry sells, and its price.
public record Service(string Id, string Description, decimal Price)
{
    // "Wash, dry & fold (per kg)" -> "Wash, dry & fold"
    public string Name => Description.Contains(" (")
        ? Description[..Description.LastIndexOf(" (")] : Description;

    public bool PerKg => Description.EndsWith("(per kg)");

    public string Unit => PerKg ? "kg"
        : Description.EndsWith("(per pair)") ? "pair" : "item";
}
```

The three computed properties turn the database's description into what the pages need. `Name` cuts off the unit in parentheses, `"Wash, dry & fold (per kg)"` becoming `"Wash, dry & fold"`; `PerKg` says whether the service is charged by weight; `Unit` gives `kg`, `pair` or `item`. They are computed each time they are read, so they can never disagree with the description. **Listing 3.5** is the outlet, a record with five properties; `DateOnly` holds a date without a time.

Listing 3.5 `Models/Outlet.cs`: an outlet of Profitina Laundry.

```
namespace LaundryMvc.Models;

// One row of data/Outlet.csv: a shop where customers bring their laundry.
public record Outlet(string Id, string Name, string Address, string Phone,
    DateOnly Opened);
```

Indonesian prices are written with a dot between the thousands, `Rp 36.000`. .NET could do this with the culture `id-ID`, but culture data depend on the operating system's installation, so LaundryMvc formats rupiah with the invariant culture and swaps the comma for a dot, in one place, **Listing 3.6**.

Listing 3.6 `Models/Money.cs`: rupiah amounts in the Indonesian style, formatted in one place.

```
using System.Globalization;

namespace LaundryMvc.Models;

public static class Money
{
    // 36000 -> "Rp 36.000" (Indonesian style: a dot between thousands)
    public static string Rp(decimal amount) =>
        "Rp " + amount.ToString("N0", CultureInfo.InvariantCulture)
            .Replace(",", ".");
}
```

3.5.2 The price rules

Listing 3.7 is where Bu Wati's rules live, and only there. `Calculate` takes a service, a quantity and the pickup choice, and returns a `Quote`, a record with everything a page needs to show: what was billed, the subtotal, the pickup fee, the total, and the notes that explain each rule that was applied.

Listing 3.7 `Models/PriceCalculator.cs`: the shop's pricing rules, with an explanation for every rule applied.

```
namespace LaundryMvc.Models;

public record Quote(Service Service, decimal Quantity, decimal Billed,
    decimal Subtotal, decimal PickupFee, decimal Total, List<string> Notes);

// The shop's pricing rules live here, not in a controller or a view.
public class PriceCalculator
{
    public const decimal MinimumKg = 3m;
    public const decimal PickupFee = 10_000m;
    public const decimal FreePickupFrom = 100_000m;

    public Quote Calculate(Service service, decimal quantity, bool pickup)
    {
        var notes = new List<string>();
        decimal billed = quantity;
        if (service.PerKg && quantity < MinimumKg)
        {
            billed = MinimumKg;
            notes.Add($"Minimum {MinimumKg} kg, so {quantity} kg is billed "
                + $"as {MinimumKg} kg.");
        }
        else if (!service.PerKg && quantity != Math.Ceiling(quantity))
        {
            billed = Math.Ceiling(quantity);
            notes.Add($"Items are counted whole: {quantity} becomes {billed}.");
        }
        decimal subtotal = billed * service.Price;
        decimal fee = 0m;
        if (pickup)
        {
            fee = subtotal >= FreePickupFrom ? 0m : PickupFee;
            notes.Add(fee == 0m
                ? $"Pickup and delivery are free from {Money.Rp(FreePickupFrom)}."
                : $"Pickup and delivery cost {Money.Rp(PickupFee)}.");
        }
        return new Quote(service, quantity, billed, subtotal, fee,
            subtotal + fee, notes);
    }
}
```

Work one quote by hand before trusting the code. A customer brings 2.5 kg for “wash, dry and fold” (Rp 8.000 per kg) and wants pickup. The service is per kg and 2.5 is less than the minimum 3, so `billed` becomes 3 and a note says so. The subtotal is $3 \times 8000 = 24000$. Pickup was asked for; 24.000 is below the 100.000 threshold, so the fee is 10.000 and a second note says so. The total is $24000 + 10000 = 34000$. Section 3.8 shows that the running site gives exactly this.

Why a separate class, instead of a few lines inside the controller action? Because the rules are the business. The same rules will be needed by the counter screen of Chapter 5, by a JSON endpoint for Dimas’s app, and by automated tests; none of them can call a controller action conveniently, and all of them can call `PriceCalculator.Calculate`. The constants at the top are the one place where the minimum, the fee and the threshold are written, which is Bu Wati’s first rule in C#.

3.5.3 View models and the form model

A **view model** is a class shaped for one view, holding exactly what that view shows. The home page shows three popular services and two counts, so **Listing 3.8** is a record with those three things and nothing else.

Listing 3.8 `Models/HomeViewModel.cs`: exactly what the home page shows.

```
namespace LaundryMvc.Models;

// Exactly what the home page shows, and nothing more.
public record HomeViewModel(IReadOnlyList<Service> Popular,
    int ServiceCount, int OutletCount);
```

The quote form needs a class that model binding can fill from the request, **Listing 3.9**. Its properties have *setters*, because the binder creates the object and sets them one by one; `ServiceId` and `Quantity` are nullable so that "not given" can be told apart from "given as zero". The `Result` property is filled by the controller, never from the request; the attribute `[BindNever]` tells the binder to leave it alone even if someone adds `?Result=…` to the address.

Listing 3.9 `Models/QuoteForm.cs`: the input of the price-quote form, filled by model binding.

```
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace LaundryMvc.Models;

// Filled by model binding from the query string of /services/quote.
public class QuoteForm
{
    public string? ServiceId { get; set; }
    public decimal? Quantity { get; set; }
    public bool Pickup { get; set; }

    [BindNever]                          // set by the controller, never by a URL
    public Quote? Result { get; set; }
}
```

3.5.4 The catalog: data access behind an interface

Controllers must not know where the data come from. They depend on the interface **Listing 3.10**; today its implementation reads CSV files, in Chapter 5 a database.

Listing 3.10 `Data/ILaundryCatalog.cs`: what controllers may ask about services and outlets.

```
using LaundryMvc.Models;

namespace LaundryMvc.Data;

// What the controllers may ask for. Chapter 5 swaps the CSV files
// behind it for a real database without touching a controller.
public interface ILaundryCatalog
{
    IReadOnlyList<Service> Services();
    Service? FindService(string id);
    IReadOnlyList<Outlet> Outlets();
    Outlet? FindOutlet(string id);
}
```

Listing 3.11 reads both files once, in its constructor, into lists. Because it is registered as a singleton, that happens once per run of the application. The `Find` methods compare codes without regard to case, so `/services/s0004` finds `S0004`. The small CSV reader is the one of Chapter 1, written as an *iterator* (`yield return`) that produces one row at a time.

Listing 3.11 `Data/CsvCatalog.cs`: the catalog implemented on the two CSV files, read once at start-up.

```
using System.Globalization;
using System.Text;
using LaundryMvc.Models;

namespace LaundryMvc.Data;

// Reads data/Service.csv and data/Outlet.csv once, at start-up.
public class CsvCatalog : ILaundryCatalog
{
    private readonly List<Service> services;
    private readonly List<Outlet> outlets;

    public CsvCatalog(string folder)
    {
        services = Read(Path.Combine(folder, "Service.csv"))
            .Select(f => new Service(f[0], f[1],
                decimal.Parse(f[2], CultureInfo.InvariantCulture)))
            .ToList();
        outlets = Read(Path.Combine(folder, "Outlet.csv"))
            .Select(f => new Outlet(f[0], f[1], f[2], f[3],
                DateOnly.Parse(f[4], CultureInfo.InvariantCulture)))
            .ToList();
    }

    public IReadOnlyList<Service> Services() => services;

    public Service? FindService(string id) => services.FirstOrDefault(
        s => s.Id.Equals(id, StringComparison.OrdinalIgnoreCase));

    public IReadOnlyList<Outlet> Outlets() => outlets;

    public Outlet? FindOutlet(string id) => outlets.FirstOrDefault(
        o => o.Id.Equals(id, StringComparison.OrdinalIgnoreCase));

    // Splits each line on commas, except commas inside "double quotes".
    private static IEnumerable<string[]> Read(string path)
    {
        foreach (var line in File.ReadLines(path).Skip(1))  // skip header
        {
            var fields = new List<string>();
            var field = new StringBuilder();
            bool quoted = false;
            foreach (char c in line)
            {
                if (c == '"') quoted = !quoted;
                else if (c == ',' && !quoted)
```

Listing 3.11 (Continued) *Data/CsvCatalog.cs: the catalog implemented on the two CSV files, read once at start-up.*

```
                {
                    fields.Add(field.ToString());
                    field.Clear();
                }
                else field.Append(c);
            }
            fields.Add(field.ToString());
            yield return fields.ToArray();
        }
    }
}
```

3.6 Controllers and Routing

A **controller** is a class whose public methods, called **actions**, handle requests. In ASP.NET Core MVC a controller usually derives from `Controller`, which gives it helper methods such as `View()`, `NotFound()`, `Ok()` and `RedirectToAction()`, and properties such as `Request`, `Response`, `ModelState` and `ViewData`. An action returns an `IActionResult`, an object that *describes* the response (a view to render, a status, JSON, a redirect) and that the framework then executes. LaundryMvc has three controllers, and together they show both ways of routing.

3.6.1 Conventional routing: `HomeController` and `OutletsController`

The route registered in `Program.cs`,

```
{controller=Home}/{action=Index}/{id?}
```

is a **conventional route**: one template, defined centrally, that fits every controller that follows the naming convention [7]. Its first segment names the controller (without the word `Controller`), the second the action, and the third an optional `id`. The `=Home` and `=Index` are defaults, used when a segment is missing, and `?` makes `id` optional. **Table 3.2** shows what it makes of some addresses; all of them are real answers of LaundryMvc (Section 3.8).

Table 3.2 How the conventional route `{controller=Home}/{action=Index}/{id?}` maps addresses to actions. Matching ignores upper and lower case.

Address	Controller	Action	id	Status
`/`	`Home` (default)	`Index` (default)	none	200
`/Home`	`Home`	`Index` (default)	none	200
`/home/index`	`Home`	`Index`	none	200
`/Home/About`	`Home`	`About`	none	200
`/Outlets`	`Outlets`	`Index` (default)	none	200
`/Outlets/Details/O02`	`Outlets`	`Details`	`O02`	200
`/Outlets/Details/O09`	`Outlets`	`Details`	`O09`	404 (no such outlet)
`/nothing/here`	no `NothingController`			404

Listing 3.12 is the home controller. It uses a **primary constructor**, `HomeController(ILaundryCatalog catalog)`, available for classes since C# 12 [8]: the parameter `catalog` is in scope in the whole class, and the container supplies it when the framework creates the controller for a request. There is no field and no assignment to write.

Listing 3.12 `Controllers/HomeController.cs`: the home page, the about page, and the page shown for error status codes.

```
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;

namespace LaundryMvc.Controllers;

// Reached through the conventional route {controller=Home}/{action=Index}.
public class HomeController(ILaundryCatalog catalog) : Controller
{
    public IActionResult Index()
    {
        string[] popular = ["S0001", "S0002", "S0008"];
        var model = new HomeViewModel(
            popular.Select(id => catalog.FindService(id)!).ToList(),
            catalog.Services().Count, catalog.Outlets().Count);
        return View(model);
    }

    public IActionResult About() => View();

    // An attribute route: only /error/404, /error/500, ... reach this action.
    [Route("/error/{code:int}")]
    public IActionResult Error(int code)
    {
        // The status-code middleware re-ran the request as /error/404;
        // this feature remembers the address the visitor really asked for.
        var original = HttpContext.Features.Get<IStatusCodeReExecuteFeature>();
        ViewData["Path"] = original?.OriginalPath ?? Request.Path.Value;
        Response.StatusCode = code;
        return View(code);
    }
}
```

`Index` builds the view model from three popular codes and the two counts, and returns `View(model)`: "render the view with this action's name, `Views/Home/Index.cshtml`, with this model". `About` has no data and returns `View()`. The third action, `Error`, is different. It carries an **attribute route**, `[Route("/error/{code:int}")]`, so it is reachable only at `/error/404`, `/error/500` and so on, never through the conventional route. This is the address that the status-code middleware re-runs. When it does, the request's path *is* `/error/404`; the address the visitor actually typed is kept in a *feature* of the context, `IStatusCodeReExecuteFeature`, and the action copies its `OriginalPath` into `ViewData` for the view. It also sets `Response.StatusCode = code`, so that the friendly page still says 404 to browsers and search engines.

Listing 3.13 is the outlet controller, the simplest possible: a list, and one outlet or `NotFound()`.

Listing 3.13 `Controllers/OutletsController.cs`: the outlets, reached through the conventional route.

```
using LaundryMvc.Data;
using Microsoft.AspNetCore.Mvc;

namespace LaundryMvc.Controllers;

// Conventional routing: /Outlets and /Outlets/Details/O01.
public class OutletsController(ILaundryCatalog catalog) : Controller
{
    public IActionResult Index() => View(catalog.Outlets());

    public IActionResult Details(string id)
    {
        var outlet = catalog.FindOutlet(id);
        if (outlet is null)
        {
            return NotFound();
        }
        return View(outlet);
    }
}
```

`NotFound()` produces a 404 response with no body. That is exactly the kind of response the status-code middleware waits for: on the way out it sees “404, empty” and re-runs the pipeline with `/error/404`. The controller did not need to know that a friendly page exists.

3.6.2 Attribute routing: `ServicesController`

The conventional route produces addresses like `/Services/Details/S0004`, which are correct but not pretty, and it cannot express `/services/under/20000`. For such addresses ASP.NET Core offers **attribute routing**: the route is written on the controller and its actions [7]. **Listing 3.14** is the services controller, the heart of the site.

Listing 3.14 `Controllers/ServicesController.cs`: attribute routes, a constraint, a form action and a JSON action.

```
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Mvc;

namespace LaundryMvc.Controllers;

// Attribute routing: every URL of this controller starts with /services.
[Route("services")]
public class ServicesController(ILaundryCatalog catalog,
    PriceCalculator calculator) : Controller
{
    [HttpGet("")]                                   // /services
    public IActionResult Index() => View(catalog.Services());

    [HttpGet("{id}")]                               // /services/S0004
    public IActionResult Details(string id)
    {
        var service = catalog.FindService(id);
        return service is null ? NotFound() : View(service);
    }

    [HttpGet("under/{max:int:min(1000)}")]          // /services/under/20000
    public IActionResult Under(int max)
    {
        ViewData["Title"] = $"Services up to {Money.Rp(max)}";
        var cheap = catalog.Services().Where(s => s.Price <= max).ToList();
        return View("Index", cheap);
    }

    [HttpGet("quote")]                    // /services/quote?serviceId=...
    public IActionResult Quote(QuoteForm form)
    {
        if (form.Quantity is null)
        {
            return View(form);                      // first visit: empty form
        }
        var service = catalog.FindService(form.ServiceId ?? "");
        if (service is null)
        {
            ModelState.AddModelError(nameof(form.ServiceId),
                "Choose a service from the list.");
        }
        if (form.Quantity <= 0)
        {
            ModelState.AddModelError(nameof(form.Quantity),
                "Enter a quantity above zero.");
```

Listing 3.14 (Continued) *Controllers/ServicesController.cs: attribute routes, a constraint, a form action and a JSON action.*

```
        }
        if (service is not null && ModelState.IsValid)
        {
            form.Result = calculator.Calculate(service, form.Quantity.Value,
                form.Pickup);
        }
        return View(form);
    }

    [HttpGet("/api/services/{id}")]                // JSON for other programs
    public IActionResult Api(string id)
    {
        var service = catalog.FindService(id);
        return service is null
            ? NotFound(new { error = $"No service with code {id}." })
            : Ok(service);
    }
}
```

`[Route("services")]` on the class gives every action the prefix `services`. Each `[HttpGet("...")]` adds the rest of the template and restricts the action to the `GET` method. Once a controller has attribute routes, the conventional route no longer reaches it. **Table 3.3** lists the templates the controller defines.

Table 3.3 The attribute routes of `ServicesController`. A template starting with `/` ignores the controller's prefix.

Template	Action	Example	Result
`services`	`Index`	`/services`	the list
`services/{id}`	`Details`	`/services/S0004`	one service, or 404
`services/under/{max:int:min(1000)}`	`Under`	`/services/under/20000`	the list, filtered
`services/quote`	`Quote`	`/services/quote?serviceId=S0001&quantity=2.5`	the form, and its result
`/api/services/{id}`	`Api`	`/api/services/S0004`	JSON, or 404 with a JSON error

Three details deserve attention.

Constraints. `{max:int:min(1000)}` combines two constraints: the segment must be an integer, and at least 1000. `/services/under/abc` and `/services/under/500` therefore match *no* route, and the answer is 404 without the action ever running. A constraint is not validation of user input; it is a statement about which addresses exist. The action receives `max` already converted to `int`.

Precedence. Both `services/{id}` and `services/quote` could match `/services/quote`. ASP.NET Core does not try the routes in the order written, as MiniFrame did in Chapter 2. It ranks them: a literal segment (`quote`) is more specific than a parameter (`{id}`), so `/services/quote` always reaches `Quote` [9]. Move the methods around in the file and nothing changes.

Routes outside the prefix. The template of `Api` starts with a slash, `/api/services/{id}`, which makes it absolute: the class prefix is ignored. The same action style serves two audiences, `/services/S0004` for people and `/api/services/S0004` for programs.

3.6.3 What actions return

The four kinds of result in **Listing 3.14** cover most of what an action ever does.

- `View(model)` renders a view. `View("Index", cheap)` in `Under` renders a view with a *different* name, the list view of `Index`, reusing it for the filtered list; the title is passed through `ViewData["Title"]`.
- `NotFound()` sends 404 without a body, which the status-code middleware turns into the friendly page. `NotFound(new { error = … })` sends 404 *with* a JSON body; because the body is not empty, the middleware leaves it alone, and Dimas's app receives a machine-readable error instead of an HTML page.
- `Ok(service)` sends 200 with the object serialized to JSON, using the same camelCase names as Chapter 1.
- The fourth kind, a redirect, appears in Chapter 4 (`RedirectToAction`).

3.6.4 Model binding: the quote form

The action `Quote(QuoteForm form)` never reads the query string itself. Before calling it, MVC's **model binding** creates a `QuoteForm` and fills each property from the request: for `ServiceId` it looks for a value named `serviceId` (ignoring case) in the route values, the query string and the form fields; it finds `?serviceId=S0001`; it converts the text to the property's type [10]. For `Quantity=2.5` it converts to `decimal?`; query-string values are converted with the invariant culture, so the decimal separator is always a point. For the checkbox `Pickup` it finds `true`.

When a value cannot be converted, model binding does not throw. It records an error in `ModelState`, a dictionary of errors per property, and leaves the property unset. `?quantity=abc` produces the error "The value 'abc' is not valid for Quantity." without any code of ours. The action adds its own errors with `ModelState.AddModelError` (an unknown service, a quantity of zero or less), and calculates only if `ModelState.IsValid`. A form with errors is simply shown again, with the messages at the top, and whatever the customer typed is still in the fields.

Why is the quote a `GET` and not a `POST`? Because calculating a price changes nothing on the server: it is a question, not an order, and HTTP defines `GET` as the *safe* method for exactly such requests [11]. A `GET` form puts the question in the address, so a quote can be bookmarked, shared and reloaded safely. Placing an order *will* change data, and Chapter 5 does that with `POST` and an antiforgery token.

3.7 Razor Views

A **Razor view** is an HTML file with the extension `.cshtml` in which C# is marked by a single `@` [12]. `@Model.Name` writes the value of an expression, **HTML-encoded**; `@{ … }` is a block of C# statements; `@if`, `@foreach` and `@switch` are C# statements whose bodies are HTML again. Razor

compiles each view into a C# class when the project is built, so a typo in a view is a compile error, not a surprise in front of a customer.

3.7.1 Settings for all views: `_ViewImports` and `_ViewStart`

Listing 3.15 is read before every view in the `Views` folder. Its `@using` lines make our namespaces available, so views can write `Service` instead of `LaundryMvc.Models.Service`, and `@addTagHelper` switches on the built-in tag helpers (Section 3.7.4).

Listing 3.15 `Views/_ViewImports.cshtml`: namespaces and tag helpers for every view.

```
@using LaundryMvc
@using LaundryMvc.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Listing 3.16 runs before every view and sets its layout; a view can override it, or set `Layout = null` to have none.

Listing 3.16 `Views/_ViewStart.cshtml`: every view uses the layout `_Layout`.

```
@{
    Layout = "_Layout";
}
```

3.7.2 The layout

Listing 3.17 is the frame around every page, Bu Wati's second rule [13]. It is written once; every page appears where it says `@RenderBody()`.

Listing 3.17 `Views/Shared/_Layout.cshtml`: the shared frame, with a menu that marks the current page.

```
@{
    // "here" marks the menu link of the page being shown.
    string? Here(string controller, string action) =>
        ViewContext.RouteData.Values["controller"] as string == controller
        && ViewContext.RouteData.Values["action"] as string == action
            ? "here" : null;
}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Profitina Laundry</title>
    <link rel="icon" href="data:," />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
</head>
<body>
    <header>
        <a class="brand" asp-controller="Home" asp-action="Index">
            Profitina Laundry</a>
        <nav>
            <a class="@Here("Home", "Index")"
               asp-controller="Home" asp-action="Index">Home</a>
            <a class="@Here("Services", "Index")"
               asp-controller="Services" asp-action="Index">Services</a>
            <a class="@Here("Services", "Quote")"
               asp-controller="Services" asp-action="Quote">Price quote</a>
            <a class="@Here("Outlets", "Index")"
               asp-controller="Outlets" asp-action="Index">Outlets</a>
            <a class="@Here("Home", "About")"
               asp-controller="Home" asp-action="About">About</a>
        </nav>
    </header>
    <main>
        @RenderBody()
    </main>
    <footer>
        @await RenderSectionAsync("Footer", required: false)
        <p>&copy; @DateTime.Now.Year Profitina Laundry, Surabaya</p>
    </footer>
</body>
</html>
```

The `@{ … }` block at the top defines a small local function, `Here`, that returns the CSS class `"here"` when the current request's controller and action (from `ViewContext.RouteData`) are the ones given, and `null` otherwise. Razor has a useful rule for attributes: when an attribute's value is `null`, it leaves the attribute out entirely, so the links of other pages carry no `class` at all. The title comes from `ViewData["Title"]`, which each view sets. `@await RenderSectionAsync("Footer", required: false)` marks an optional **section**: a view may send some HTML to this place in the layout, and the About page does. `asp-append-version="true"` on the style sheet adds a fingerprint of the file's content to its URL, so that browsers fetch the new CSS as soon as it changes.

3.7.3 Pages, a partial view and a section

Listing 3.18 is the home page. `@model HomeViewModel` declares the type of `Model`, so that `Model.Popular` is checked by the compiler. The three cards are drawn by a **partial view**, a piece of a page kept in its own file (**Listing 3.19**), called once per service with `<partial name="_ServiceCard" model="service" />` [14].

Listing 3.18 `Views/Home/Index.cshtml`: the home page, with a partial view for each card.

```
@model HomeViewModel
@{
    ViewData["Title"] = "Home";
}
<h1>Clean clothes, honest prices</h1>
<p>We wash, iron and dry-clean in @Model.OutletCount outlets across
   Surabaya. Choose from @Model.ServiceCount services, or
   <a asp-controller="Services" asp-action="Quote">work out a price</a>
   before you come.</p>
<h2>Most asked for</h2>
<div class="cards">
    @foreach (var service in Model.Popular)
    {
        <partial name="_ServiceCard" model="service" />
    }
</div>
```

Listing 3.19 `Views/Shared/_ServiceCard.cshtml`: one card, written once and used for every popular service.

```
@model Service
<div class="card">
    <h3>@Model.Name</h3>
    <p class="price">@Money.Rp(Model.Price) <span>per @Model.Unit</span></p>
    <a asp-controller="Services" asp-action="Details"
       asp-route-id="@Model.Id">More about @Model.Id</a>
</div>
```

The About page, **Listing 3.20**, shows a **section**: `@section Footer { … }` sends its content to the place in the layout where `RenderSectionAsync("Footer")` stands, below every other footer line of that page.

Listing 3.20 `Views/Home/About.cshtml`: a page that adds the opening hours to the layout's footer through a section.

```
@{
    ViewData["Title"] = "About us";
}
<h1>About us</h1>
<p>Profitina Laundry opened its first outlet in Rungkut in 2019. Every
   order is weighed or counted in front of you, gets a rack number, and
   is ready on the day we promise.</p>
@section Footer {
    <p class="hours">All outlets are open every day, 07.00 to 21.00.</p>
}
```

The list of services, **Listing 3.21**, serves two actions. `ViewData["Title"] ??= "Our services"` sets the title only if the action did not set one already, so the same view says "Our services" for `/services` and "Services up to Rp 20.000" for `/services/under/20000`.

Listing 3.21 `Views/Services/Index.cshtml`: the table of services, shared by the actions `Index` and `Under`.

```
@model IReadOnlyList<Service>
@{
    ViewData["Title"] ??= "Our services";
}
<h1>@ViewData["Title"]</h1>
<p>@Model.Count services. Prices are per kilogram, per item or per pair.</p>
<table class="list">
    <thead>
        <tr><th>Code</th><th>Service</th><th class="num">Price</th><th></th></tr>
    </thead>
    <tbody>
    @foreach (var s in Model)
    {
        <tr>
            <td>@s.Id</td>
            <td>@s.Name</td>
            <td class="num">@Money.Rp(s.Price) / @s.Unit</td>
            <td><a asp-action="Details" asp-route-id="@s.Id">Details</a></td>
        </tr>
    }
    </tbody>
</table>
<p>On a budget? See the services
   <a asp-action="Under" asp-route-max="20000">up to Rp 20.000</a>.</p>
```

Listing 3.22 `Views/Services/Details.cshtml`: one service, with a link to a quote for it.

```
@model Service
@{
    ViewData["Title"] = Model.Name;
}
<h1>@Model.Name</h1>
<dl class="facts">
    <dt>Code</dt><dd>@Model.Id</dd>
    <dt>Price</dt><dd>@Money.Rp(Model.Price) per @Model.Unit</dd>
    <dt>Charged by</dt><dd>@(Model.PerKg ? "weight" : "count")</dd>
</dl>
<p><a asp-action="Quote" asp-route-serviceId="@Model.Id">Get a quote
   for this service</a> or <a asp-action="Index">see all services</a>.</p>
```

Listing 3.22 shows one service. The expression `@(Model.PerKg ? "weight" : "count")` needs parentheses, because Razor would otherwise stop reading C# at the first space. The outlet views, **Listing 3.23** and **Listing 3.24**, follow the same pattern; `ToString("d MMMM yyyy")` formats the opening date as, for example, “1 February 2022”.

Listing 3.23 `Views/Outlets/Index.cshtml`: the list of outlets.

```
@model IReadOnlyList<Outlet>
@{
    ViewData["Title"] = "Outlets";
}
<h1>Our outlets</h1>
<ul class="outlets">
    @foreach (var o in Model)
    {
        <li><a asp-action="Details" asp-route-id="@o.Id">@o.Name</a><br />
            @o.Address</li>
    }
</ul>
```

Listing 3.24 `Views/Outlets/Details.cshtml`: one outlet.

```
@model Outlet
@{
    ViewData["Title"] = Model.Name;
}
<h1>@Model.Name</h1>
<dl class="facts">
    <dt>Address</dt><dd>@Model.Address</dd>
    <dt>Phone</dt><dd>@Model.Phone</dd>
    <dt>Open since</dt><dd>@Model.Opened.ToString("d MMMM yyyy")</dd>
</dl>
<p><a asp-action="Index">All outlets</a></p>
```

The error view, **Listing 3.25**, receives the status code as its model and the original path through `ViewData`; the path is written with `@`, so even an address full of `<script>` is displayed harmlessly.

Listing 3.25 `Views/Shared/Error.cshtml`: the friendly page for 404 and for other error statuses.

```
@model int
@{
    ViewData["Title"] = Model == 404 ? "Not found" : "Something went wrong";
}
<h1>@ViewData["Title"]</h1>
@if (Model == 404)
{
    <p>We could not find the page
        <code>@ViewData["Path"]</code>.</p>
}
else
{
    <p>The server could not finish your request (status @Model).</p>
}
<p><a asp-controller="Home" asp-action="Index">Back to the home page</a></p>
```

3.7.4 Tag helpers and the quote form

Tag helpers are C# classes that take part in rendering ordinary-looking HTML elements, recognized by attributes that start with `asp-` [15]. They are why the views of this chapter contain almost no hand-written URLs.

- `<a asp-controller="Services" asp-action="Details" asp-route-id="@s.Id">` asks the *routing system* for the URL of that action with that `id`. Because `ServicesController` uses attribute routes, the answer is `/services/S0004`; for `OutletsController`, which uses the conventional route, the same kind of link becomes `/Outlets/Details/O02`. If a route changes, every link follows. When the controller is not named, the current one is meant, as in the services views.
- `<form asp-action="Quote" method="get">` produces the `action` attribute from the route, `/services/quote`.
- `<label asp-for="Quantity">`, `<input asp-for="Quantity" />` and `<select asp-for="ServiceId">` bind an element to a property of the model: they write the `name` and `id` the model binder will look for, put the current value back into the field, and add the CSS class `input-validation-error` when `ModelState` holds an error for that property. For a `bool` property, `<input asp-for="Pickup" />` becomes a checkbox plus a hidden field `Pickup=false` at the end of the form, so that an unticked box still sends a value.
- `<div asp-validation-summary="All">` lists every error in `ModelState`.

Listing 3.26 is the quote view, where all of these work together.

Listing 3.26 `Views/Services/Quote.cshtml`: the quote form, built with tag helpers, and the result table.

```
@model QuoteForm
@inject LaundryMvc.Data.ILaundryCatalog Catalog
@{
    ViewData["Title"] = "Price quote";
    var choices = Catalog.Services().Select(s => new SelectListItem(
        $"{s.Id} {s.Name} ({Money.Rp(s.Price)}/{s.Unit})", s.Id));
}
<h1>Price quote</h1>
<form asp-action="Quote" method="get" class="quote">
    <div asp-validation-summary="All" class="errors"></div>
    <label asp-for="ServiceId">Service</label>
    <select asp-for="ServiceId" asp-items="choices">
        <option value="">-- choose a service --</option>
    </select>
    <label asp-for="Quantity">Quantity (kg, items or pairs)</label>
    <input asp-for="Quantity" />
    <label class="check"><input asp-for="Pickup" /> Pickup and delivery</label>
    <button type="submit">Calculate</button>
</form>
@if (Model.Result is { } q)
{
    <table class="list quote-result">
        <tr><th>Service</th><td>@q.Service.Name</td></tr>
        <tr><th>Billed</th>
            <td>@q.Billed @q.Service.Unit x @Money.Rp(q.Service.Price)</td></tr>
        <tr><th>Subtotal</th><td class="num">@Money.Rp(q.Subtotal)</td></tr>
        <tr><th>Pickup and delivery</th>
            <td class="num">@Money.Rp(q.PickupFee)</td></tr>
        <tr class="total"><th>Total</th><td class="num">@Money.Rp(q.Total)</td></tr>
    </table>
    <ul class="notes">
        @foreach (var note in q.Notes)
        {
            <li>@note</li>
        }
    </ul>
}
```

Two more features appear here. `@inject LaundryMvc.Data.ILaundryCatalog Catalog` asks the DI container for a service *inside the view*: the list of choices for the drop-down is a matter of presentation, so the view fetches it itself rather than burdening the controller. And the result table is shown only when `Model.Result is { } q`, a C# *pattern* that is true when `Result` is not null and names it `q` at the same time.

3.8 Running It

3.8.1 In a terminal or in VS Code

From the folder `Chapter03/aspnet/LaundryMvc`:

```
dotnet run
Using launch settings from …/LaundryMvc/Properties/launchSettings.json...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:4030
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: …/Chapter03/aspnet/LaundryMvc
```

(The build machine started the app with `--urls http://127.0.0.1:4030`, which is why it listens on `127.0.0.1` instead of `localhost`; the port and everything else are the same.) The environment is **Development**, set by the launch profile. In VS Code, open the folder `LaundryMvc` with **File → Open Folder…**; C# Dev Kit recognizes the project, and **Run → Start Debugging** (F5) starts it with the debugger attached, so that a breakpoint in `PriceCalculator.Calculate` stops the program while a quote is calculated.

3.8.2 What the browser shows

Figure 3.3 is the home page. The menu marks "Home" with the amber underline that the layout's `Here` function chose; the three cards are the partial view, called three times; the counts come from the view model.

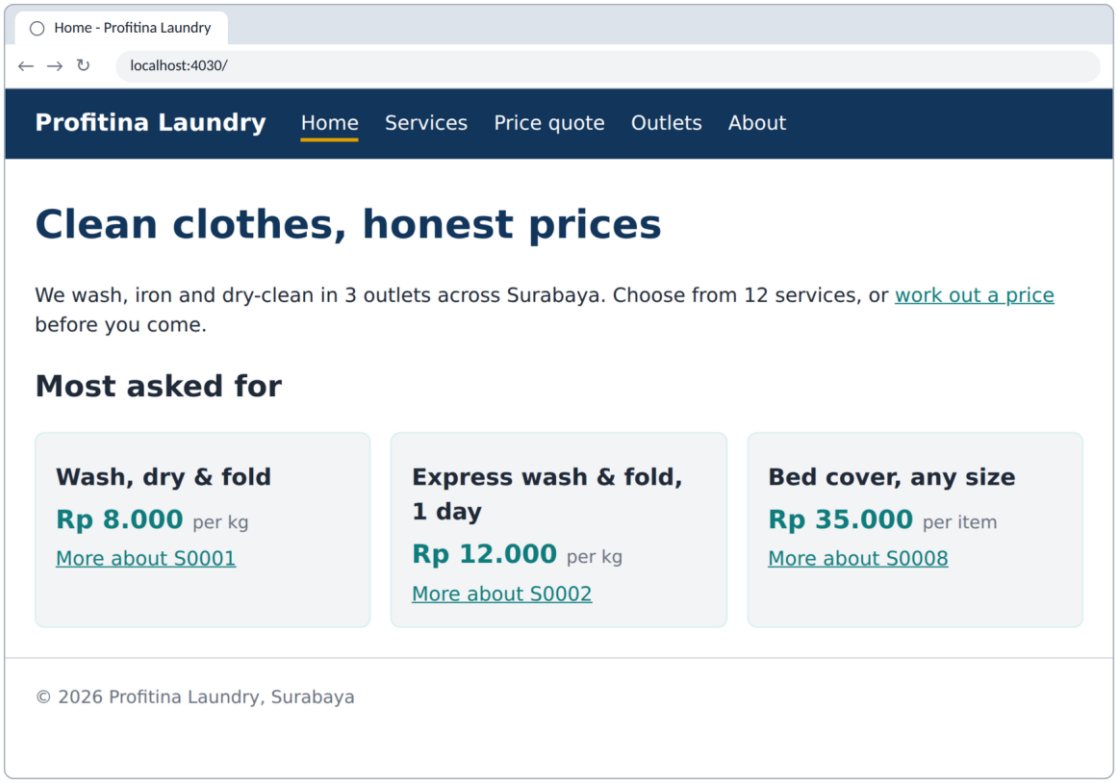


Figure 3.3 The home page of LaundryMvc: the layout's header and menu, the view's text with counts from the view model, and three cards drawn by the partial view `_ServiceCard`.

Figure 3.4 is `/services`: the list view with all twelve services, each price formatted by `Money.Rp` and each "Details" link generated by a tag helper from the attribute route.

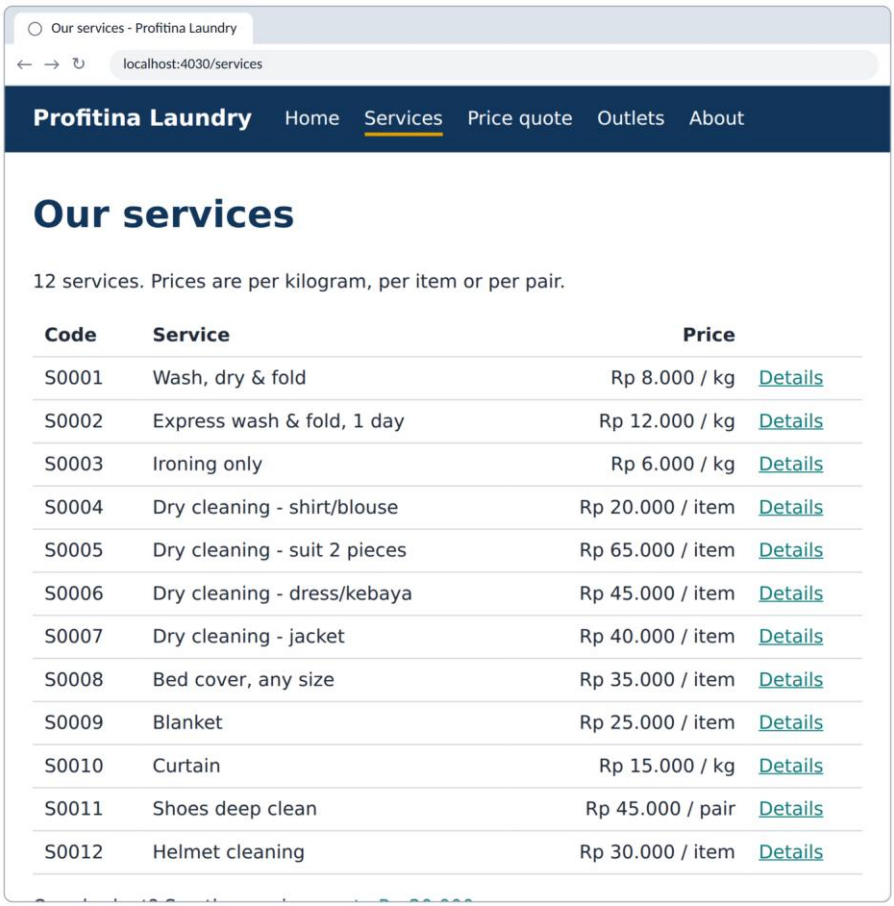


Figure 3.4 The list of services at `/services`, rendered by `Views/Services/Index.cshtml`.

Figure 3.5 is the quote for 2.5 kg of “wash, dry and fold” with pickup, the example worked by hand in Section 3.5.2. The site agrees with the hand calculation: 3 kg billed, subtotal Rp 24.000, pickup Rp 10.000, total Rp 34.000, and the two notes explain why. The form still holds the choices, because the tag helpers wrote the bound values back into the fields.

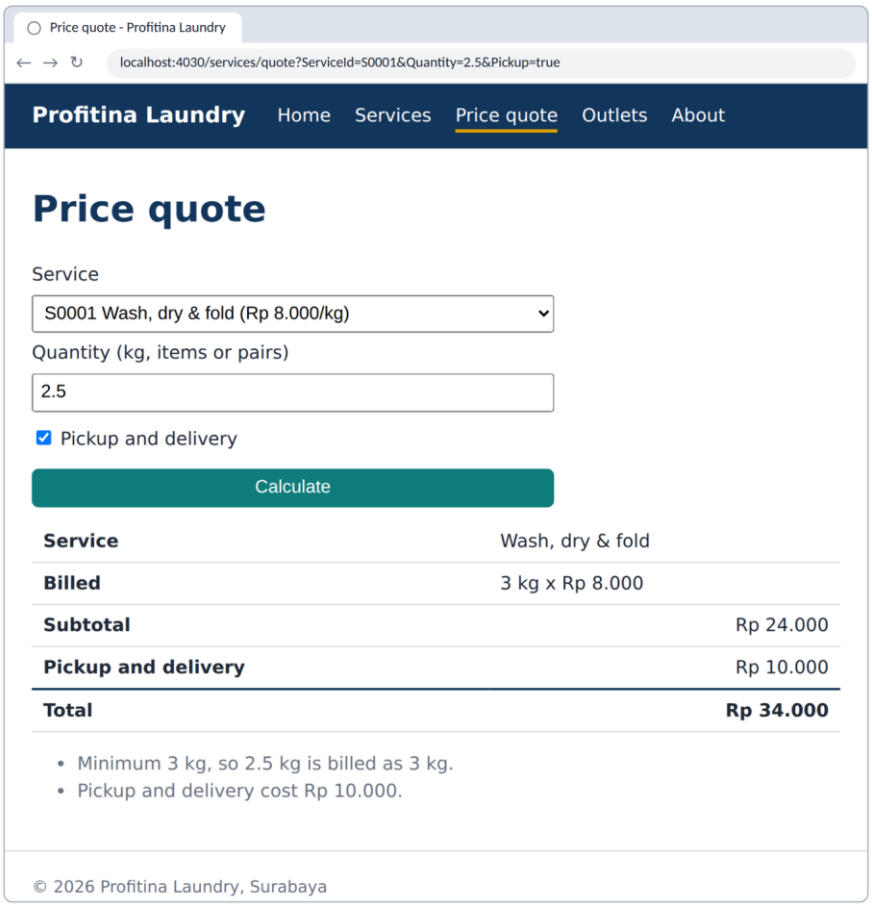


Figure 3.5 A price quote: model binding filled `QuoteForm` from the address, `PriceCalculator` applied the minimum weight and the pickup fee, and the view shows the result with its explanations.

Figure 3.6 is the same form with `Quantity=abc`. The action never saw a broken number: model binding recorded the error in `ModelState`, the validation summary lists it, and the input field carries the red border of the class `input-validation-error`.

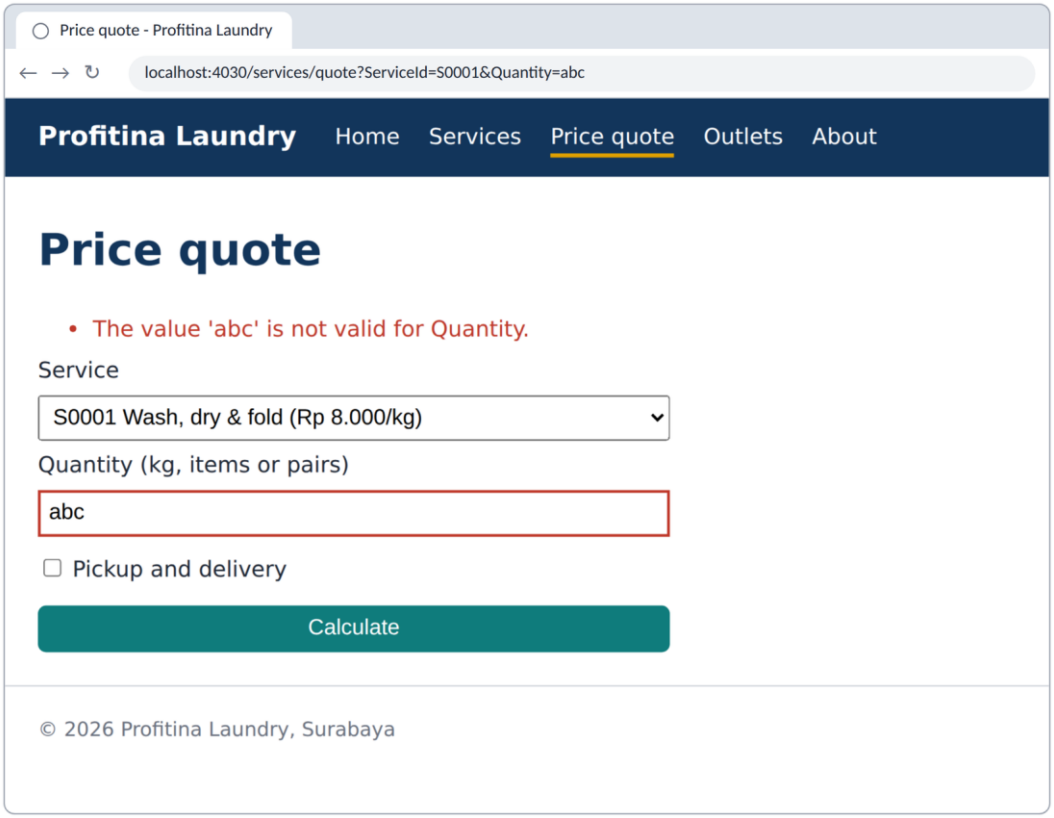


Figure 3.6 The quote form after `Quantity=abc`: the conversion error from model binding appears in the validation summary, and the field is marked.

Figure 3.7 is what a mistyped address gets: `/services/under/abc` matches no route because of the `int` constraint, the empty 404 is replaced by the error view on the way out, and the page still has the shop's layout.

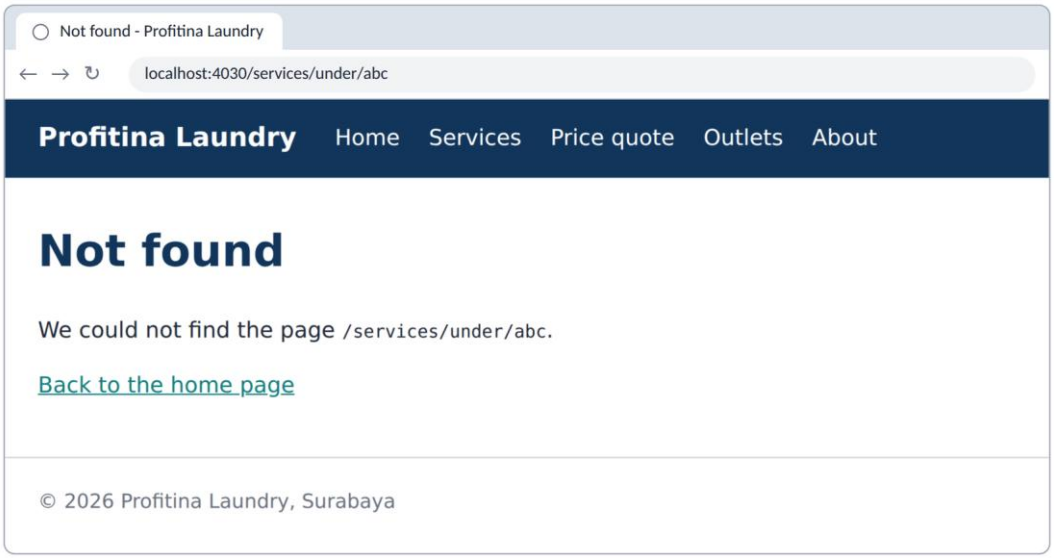


Figure 3.7 The friendly 404 page for `/services/under/abc`, produced by the status-code middleware and `HomeController.Error`.

3.8.3 Every address, checked

The script `Chapter03/verify.sh` builds the application, starts it, and asks it with `curl` for every address this chapter mentions. **Table 3.4** is the first part of its real output, the status code of each address; it confirms both routing tables of Section 3.6.

Table 3.4 The status codes returned by the running LaundryMvc for every address of this chapter, from `verify.sh`.

Status	Address	Why
200	`/`, `/Home`, `/home/index`	conventional route with defaults; matching ignores case
200	`/services`, `/services/S0004`, `/services/s0004`	attribute routes; `FindService` ignores case
200	`/services/quote`	the literal `quote` wins over `{id}`
200	`/services/under/20000`	`int` and `min(1000)` satisfied
404	`/services/under/abc`, `/services/under/500`	constraint fails, no route matches
200	`/Outlets`, `/Outlets/Details/O02`, `/Home/About`	conventional route
404	`/Outlets/Details/O09`	the action returned `NotFound()`
404	`/nothing/here`	no controller named `Nothing`
200	`/api/services/S0004`	absolute attribute route, JSON

The JSON endpoint answers like this, with the computed properties included:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

{"id":"S0004","description":"Dry cleaning - shirt/blouse (per
item)","price":20000.00,"name":"Dry cleaning - shirt/blouse","perKg":false,"unit":"item"}

HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

{"error":"No service with code S9999."}
```

The second answer shows the difference between the two `NotFound` calls: a body was given, so the status-code middleware did not replace it with HTML. For the quote form, the script extracts only the table cells, the notes and the error messages from each page. Six questions, six real answers:

```
== quote serviceId=S0001&quantity=2.5&pickup=true
Service Wash, dry &amp;amp; fold
Billed
3 kg x Rp 8.000
Subtotal Rp 24.000
Pickup and delivery
Rp 10.000
Total Rp 34.000
Minimum 3 kg, so 2.5 kg is billed as 3 kg.
Pickup and delivery cost Rp 10.000.
== quote serviceId=S0005&quantity=1.5&pickup=true
Service Dry cleaning - suit 2 pieces
Billed
2 item x Rp 65.000
Subtotal Rp 130.000
Pickup and delivery
Rp 0
Total Rp 130.000
Items are counted whole: 1.5 becomes 2.
Pickup and delivery are free from Rp 100.000.
== quote serviceId=S0002&quantity=9
Service Express wash &amp;amp; fold, 1 day
Billed
9 kg x Rp 12.000
Subtotal Rp 108.000
Pickup and delivery
Rp 0
Total Rp 108.000
== quote serviceId=S0099&quantity=4
Choose a service from the list.
== quote serviceId=S0001&quantity=abc
The value &amp;#x27;abc&amp;#x27; is not valid for Quantity.
== quote serviceId=S0001&quantity=0
Enter a quantity above zero.
```

Check each against the rules. The suit is counted per item, so 1.5 becomes 2 and $2 \times 65000 = 130000$, above the threshold, so pickup is free. Nine kilograms of express service are $9 \times 12000 = 108000$; no pickup was asked for, so the fee is zero and no pickup note appears. An unknown code, a non-number and a zero each produce exactly one message. The `&` and `'` are Razor's HTML encoding of `&` and `'`, seen here because the script shows the raw HTML; a browser displays them as the characters.

One line of this output is not quite right: "2 item" should be "2 items". It is a real, small bug of the kind that only running the program reveals, and Exercise 1 fixes it.

3.8.4 The log

While `verify.sh` asked its questions, the console of the application printed one line per request, written by our first middleware. An excerpt:

```
info: LaundryMvc[0]
      GET /services/under/20000 -> 200 in 5 ms
info: LaundryMvc[0]
      GET /services/under/abc -> 404 in 36 ms
info: LaundryMvc[0]
      GET /Outlets/Details/O09 -> 404 in 4 ms
info: LaundryMvc[0]
      GET /nothing/here -> 404 in 1 ms
info: LaundryMvc[0]
      GET /services/quote?serviceId=S0001&quantity=2.5&pickup=true -> 200 in 12 ms
```

Look at `/nothing/here`: the log says 404, the path the visitor typed, although the body came from `/error/404`. The logging middleware is *outside* the status-code middleware, so it sees the original

request go in and the resulting status come out, and never sees the re-executed request. Exercise 4 moves it inside and shows the difference. The first request after start-up takes longer (the first line of the full log shows about 200 ms) because the views and the catalog are loaded on first use; the times after that are a few milliseconds.

3.9 Making It Better

3.9.1 The fat-controller version, and why we did not write it

It is instructive to see the version of `Quote` that a beginner often writes first. The price rules are typed straight into the action:

```
// Tempting, and wrong: the rules live inside the controller action.
decimal billed = form.Quantity.Value;
if (service.Description.EndsWith("(per kg)") && billed < 3) billed = 3;
decimal total = billed * service.Price;
if (form.Pickup && total < 100000) total += 10000;
```

It works, and it looks shorter. But the constants 3, 100000 and 10000 are now in a controller, where no other page, no JSON endpoint and no test can reach them; the explanations (“billed as 3 kg”) would have to be built in the view by repeating the same tests; and the day the counter screen of Chapter 5 needs the same calculation, the rules will be copied, and the copies will drift apart, which is exactly how Bu Wati’s old site showed two prices for ironing. `PriceCalculator` costs one small class and removes all of that.

3.9.2 Friendly errors that keep their status

A friendly error page must not pretend that everything is fine. `HomeController.Error` sets `Response.StatusCode = code`, so the 404 page is sent with status 404: browsers, search engines and monitoring tools see the truth, and people see a page in the shop’s style. The JSON endpoint deliberately answers errors with JSON bodies so that programs are not handed HTML they cannot read. The development and production difference was tested too: an action that throws an exception returns, in Development, the developer exception page with the stack trace, and, when started with `ASPNETCORE_ENVIRONMENT=Production` and `--no-launch-profile`, our error view with “Something went wrong” and status 500, while the full exception goes to the server’s log (Exercise 6).

3.9.3 What is still missing

The site reads its data from CSV files and cannot change them. Chapter 5 connects Entity Framework Core and SQLite, adds the counter screen that records orders, and turns the catalog into a database-backed implementation of the same `ILaundryCatalog`. The quote form checks its input by hand; Chapter 6 introduces validation attributes that describe the rules on the model, and adds logins for staff. Chapter 4 first practices this chapter’s material on a complete small project.

3.10 Using It for Real

3.10.1 Your own business on LaundryMvc

LaundryMvc is a good skeleton for any small business with a price list, branches and a rule or two. Replace `data/Service.csv` and `data/Outlet.csv` with your own (keep the columns, or change

the records and the catalog together), change the popular codes in `HomeController.Index`, and rewrite `PriceCalculator` with your own rules: a tailor's minimum charge, a printing shop's price per page with a discount above one hundred pages, a clinic's registration fee added once. Because the rules live in one class, the pages do not change. When you change the look, change `site.css` and `_Layout.cshtml` only.

3.10.2 A preview of deployment

For a quick test on another computer in the same network, `dotnet run --urls http://0.0.0.0:4030` makes the app listen on all network interfaces, and a phone on the same Wi-Fi can open `http://<the computer's address>:4030`. That is fine for trying things and wrong for production, which needs a published build (`dotnet publish -c Release`), HTTPS, a process manager and a reverse proxy; Chapter 6 does it properly.

3.10.3 Profitina in practice

Profitina's own web application follows the same shape as this chapter, in a different framework. Its request pipeline has middleware that runs for every request (for example, the login rate limit that slows down repeated failed sign-ins, required by its UU PDP compliance work), then routing, then handlers. Its handlers are thin: the dashboard handler collects results from several service functions and returns them, and the business rules, such as when a subscription renews or how a community post is screened for unsafe links, live in their own modules where tests can reach them. Its public pages are rendered from templates with automatic escaping, which matters on a site where users post text that other users read. The words are Python's; the architecture is the one you have just built.

3.11 Common Mistakes, and How to Avoid Them

A controller that the framework cannot find. The class must be public, not abstract, and its name must end in `Controller` (or it must carry `[Controller]`). `ServiceController` and `ServicesController` are different controllers with different default URLs; decide on one.

A view that cannot be found. The error lists every place it searched, for example `/Views/Services/Detail.cshtml` and `/Views/Shared/Detail.cshtml`. Compare the action's name with the file name letter by letter; `Detail` is not `Details`.

Mixing conventional and attribute routes on one controller. As soon as a controller has an attribute route, the conventional route no longer reaches its actions. Put attribute routes on every action of that controller, as `ServicesController` does.

Middleware in the wrong order. Status-code pages placed after routing and endpoints never see the responses; a logging middleware placed inside the status-code middleware logs every error twice. Write the pipeline from the outside in: logging and error handling first, then status pages, then routing, then endpoints.

Reading the query string by hand. `Request.Query["quantity"]` works, but it bypasses conversion, `ModelState` and the tag helpers' round trip. Declare a parameter or a model class and let model binding do it.

Business rules in the controller or the view. If a number such as a minimum weight or a fee appears in a controller or a `.cshtml` file, move it into the model. The test: could a JSON endpoint and a unit test use the rule without an HTTP request? If not, it is in the wrong place.

`@Html.Raw` **to "fix" a display problem.** It switches off Razor's encoding. Use it only for HTML your own program produced and trusts, never for anything that came from a user or a database field that users can edit.

3.12 Chapter Summary

- An ASP.NET Core application builds a host, registers services in the DI container, builds a request pipeline of middleware, and runs Kestrel. `Program.cs` is exactly those steps, in that order.
- Middleware wraps middleware; each can act before and after the rest of the pipeline, so order matters. LaundryMvc logs, handles exceptions outside Development, re-executes error statuses as friendly pages, routes, and executes endpoints.
- The model holds data and rules: records for services and outlets, a `PriceCalculator` for the shop's rules, view models shaped for views, a form model for binding, and data access behind the `ILaundryCatalog` interface registered as a singleton.
- Controllers are reached through a conventional route (`{controller=Home}/{action=Index}/{id?}`) or through attribute routes with prefixes, constraints and precedence. Actions return results: views, 404s with or without bodies, JSON, redirects.
- Model binding fills action parameters and form models from the route, the query string and the form; conversion errors and the action's own checks collect in `ModelState`.
- Razor views mix HTML and C# with `@`, encode every value by default, share one layout with sections, reuse partial views, and use tag helpers to generate URLs from routes and bind form fields to the model.
- The running site matched the hand calculations, every route answered as predicted, and the only surprise, "2 item", became an exercise.

3.13 Exercises

1. Make the quote say "2 items" and "1 item", but keep "3 kg" and "2.5 kg". Change as few files as possible.

2. Add the constraint `length(5)` to the route of `Details`, so that `/services/S04` never reaches the action. What status does `/services/S04` get, and what does the log show?

3. Add an action `Cheapest` at `/services/cheapest` that shows the cheapest service with the `Details` view. Does it matter whether you write it above or below `Details` in the file? Compare with Exercise 1 of Chapter 2.

4. Move the logging middleware in `Program.cs` so that it comes *after* `UseStatusCodePagesWithReExecute`. Request `/services/S04` (after Exercise 2) and `/nothing/here`. How does the log change, and why?

5. Without running the program, write down the `href` that each of these tag helpers produces, and explain the difference: `<a asp-controller="Services" asp-action="Details" asp-route-id="S0004">` and `<a asp-controller="Outlets" asp-action="Details" asp-route-id="O03">`.

6. Add an action at `/services/boom` that throws `InvalidOperationException("Boom")`. What does a request to it return when the app runs with `dotnet run`, and what when it runs with

`ASPNETCORE_ENVIRONMENT=Production dotnet run --no-launch-profile`? Why does the second command need `--no-launch-profile`? Remove the action afterwards.

7. Bu Wati decides that pickup is free from Rp 150.000 instead of Rp 100.000. Which files change? What does the quote for 1.5 suits (Exercise output above) become?

3.14 Solutions to the Exercises

1. Only the model and one line of one view change. Add a method to `Service`:

```
    // "1 item", "2 items", but "2.5 kg" (units such as kg have no plural)
    public string UnitFor(decimal quantity) =>
        PerKg || quantity == 1 ? Unit : Unit + "s";
```

and in `Quote.cshtml` write `@q.Service.UnitFor(q.Billed)` instead of `@q.Service.Unit`. A run of exactly this change answered `2 items x Rp 65.000` for 1.5 suits and `1 item x Rp 65.000` for one suit; kilograms are unchanged. Putting the rule in `Service` rather than in the view means that every page that shows a quantity can use it.

2. Write `[HttpGet("{id:length(5)}")]`. `/services/S04` now matches no route, the status is 404, and the friendly page is shown; `/services/S0004` still works. The log shows `GET /services/S04 -> 404 in 0 ms`: the action never ran, because the router rejected the address. Before the change the action ran, searched the catalog in vain and returned `NotFound()`; the visitor saw the same page, but the work was done for nothing.

3.

```
    [HttpGet("cheapest")]
    public IActionResult Cheapest() =>
        View("Details", catalog.Services().MinBy(s => s.Price));
```

`/services/cheapest` shows “Ironing only” (Rp 6.000 per kg), and it does so wherever the method stands in the file, as a run confirmed. ASP.NET Core ranks the literal `cheapest` above the parameter `{id}` [9]. In MiniFrame (Chapter 2, Exercise 1) the order of registration decided, and the wrong order sent “cheapest” to `Details` as a service code.

4. With the logging middleware inside the status-code middleware, each error appears *twice* in the log. The real lines were:

```
GET /services/S04 -> 404 in 0 ms
GET /error/404 -> 404 in 13 ms
GET /nothing/here -> 404 in 0 ms
GET /error/404 -> 404 in 3 ms
```

Re-execution runs the part of the pipeline *after* the status-code middleware again, with the path `/error/404`; our logger is now in that part, so it logs the original request and then the re-executed one. Outside, it saw each request once, with the path the visitor typed. For a request log, outside is right; for measuring how long error pages take to render, inside can be useful.

5. The first gives `href="/services/S0004"`, the second `href="/Outlets/Details/O03"`. Tag helpers ask the routing system for a URL that would reach the named action. `ServicesController.Details` is reachable only through its attribute route `services/{id}`; `OutletsController.Details` only through the conventional route `{controller}/{action}/{id?}`, which fills the segments with the controller and action names as written in the class. That is also why the menu links in the layout read `/Outlets` and `/Home/About` but `/services`.

6. With `dotnet run` the launch profile sets `ASPNETCORE_ENVIRONMENT=Development`, and the response is status 500 with the developer exception page; asked by `curl`, which does not ask for HTML, it is plain text beginning `System.InvalidOperationException: Boom`, followed by the stack trace. With the Production command, the response is status 500 with our page "Something went wrong" and "status 500", produced by `UseExceptionHandler("/error/500")`, and the log contains a `fail:` line from `ExceptionHandlerMiddleware` with the exception. The option `--no-launch-profile` is needed because a launch profile's environment variables override those of the terminal; without it the first command's Development setting would win. Both behaviors were observed on the build machine.

7. Exactly one line changes: `FreePickupFrom = 150_000m` in `PriceCalculator.cs`. The note on the quote page is built from the constant, so it changes by itself to "Pickup and delivery are free from Rp 150.000." For 1.5 suits the subtotal stays Rp 130.000, which is now below the threshold, so the pickup fee becomes Rp 10.000, the total Rp 140.000, and the note reads "Pickup and delivery cost Rp 10.000." Nothing in any controller or view changes, which is Bu Wati's first rule kept.

References

[1] Microsoft, "ASP.NET Core middleware," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/middleware/

[2] Microsoft, "Overview of ASP.NET Core MVC," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/overview

[3] Microsoft, "Dependency injection in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/dependency-injection

[4] Microsoft, "Use multiple environments in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/environments

[5] Microsoft, "Handle errors in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/error-handling

[6] Microsoft, "Static files in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/static-files

[7] Microsoft, "Routing to controller actions in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/controllers/routing

[8] Microsoft, "Declare primary constructors for classes and structs," *C# Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/dotnet/csharp/whats-new/tutorials/primary-constructors

[9] Microsoft, "Routing in ASP.NET Core," *ASP.NET Core Documentation*, for ASP.NET Core 10.0, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/routing

[10] Microsoft, "Model binding in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/models/model-binding

[11] R. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, Internet Engineering Task Force, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[12] Microsoft, "Razor syntax reference for ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/views/razor

[13] Microsoft, "Layout in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/views/layout

[14] Microsoft, "Partial views in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/views/partial

[15] Microsoft, "Tag Helpers in ASP.NET Core," *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/views/tag-helpers/intro

Chapter 4

Evaluation and Discussion I: The Profitina Laundry Showcase Site, Solved in Full

This chapter adds no new part of ASP.NET Core. Instead it takes what Chapters 1 to 3 built (the request pipeline, routing, controllers, dependency injection, Razor views and configuration) and uses it on one small but complete real project: the showcase website that Profitina Laundry shows to investors, partners and job candidates, with a page where customers can track their orders. The project is split into five problems. The first two build the site and its tracking page; the other three look closely at the machinery underneath: predicting a route table by hand, adding a response header from middleware, and switching the whole site into maintenance mode without breaking its style sheet. Every problem is solved in full, in the same seven parts: the problem, understanding it by hand, designing it on paper, the solution, running it, a tempting wrong solution with its real output, and a variant to check yourself. By the end of the chapter you will have a routine for approaching any framework problem, a finished application you can adapt to your own business, and first-hand experience of five mistakes that every ASP.NET Core developer makes once.

4.1 Where We Left Off

Before solving anything, let us lay out the toolkit. Each problem of this chapter reaches for a handful of tools from the first three chapters, and naming the tool is already half of the work.

From **Chapter 1** you have the workbench (the .NET 10 SDK, VS Code, `dotnet new`, `dotnet run`, `curl`) and the big picture: a framework calls your code; every request is waited for, routed, extracted, fetched, decided, rendered, answered and survived; and Model–View–Controller divides your part of that work.

From **Chapter 2** you have the four foundations from the inside: a router that matches templates segment by segment and applies constraints; a container that builds objects from registrations; a mapper that turns rows into objects; a template engine that escapes by default and wraps pages in a layout.

From **Chapter 3** you have ASP.NET Core MVC itself: `Program.cs` with services first and the pipeline second; middleware that acts before and after `await next`; conventional and attribute routes with constraints and precedence; actions that return views, 404s and JSON; model binding and `ModelState`; Razor views with a layout, partial views, sections and tag helpers; and the status-code and exception pages.

Table 4.1 shows which tools each problem exercises most. Use it as a map: if middleware still feels vague, start with Problems 4.4 and 4.5; if routing does, start with 4.3.

Table 4.1 The five problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most.

Problem	Topic	Main tools from Chapters 1–3
4.1	The showcase site	attribute routes with a prefix, a layout, DI of content and data, view models
4.2	Where is my laundry?	a `regex` route constraint, a `GET` form, `RedirectToAction`, 404 pages, privacy
4.3	The route table by hand	route matching, precedence, case, conventional versus attribute routes
4.4	A response-time header	middleware, the moment the response starts, `OnStarting`
4.5	Maintenance mode	configuration and environment variables, middleware order, `UseRouting` and endpoints

4.2 How to Approach a Framework Problem

A beginner handed a framework problem usually opens the editor and starts creating controllers. An experienced developer first writes down *addresses*. The difference is not talent but a routine, and **Figure 4.1** shows the one this chapter follows for every problem. It is the problem-solving routine George Pólya described for mathematics (understand, plan, carry out, look back) [1], adapted to web applications.

Figure 4.1 The six steps used for every problem in this chapter; the dashed arrow is the loop between running the program and fixing it.

Step 1: Read and restate. Read the statement twice and write in one or two sentences what the visitor asks for and what the site must answer. Underline every address and every rule; they are your first tests.

Step 2: Write the address table. For every page, write the method, the address, the action that will handle it, and the status it must return, including the addresses that must *fail* (a wrong code, a wrong type, a wrong method). This table is the specification of the routing, and in step 5 it becomes the list of things to check.

Step 3: Sketch the pages and the data. Draw each page as a box with what it shows, and write next to it where each piece comes from: fixed text (content), numbers computed from data (model), or the request (binding). Anything computed belongs in the model or a service, not in the view.

Step 4: Write a thin controller, the views and the rules. Controllers route, ask and choose; rules go into classes the container provides; views display. Write one action and its view at a time, and run after each.

Step 5: Run and probe. Run the application and ask it every address of the table from step 2, in the browser and with `curl -i`, which shows the status and the headers. When a result differs from the table, do not guess: read the log and find the first place where reality differs from the plan. That is the dashed loop of **Figure 4.1**.

Step 6: Look back. Ask three questions. Which address did I not try? What is the most tempting wrong way to solve this problem, and what exactly happens when I try it? Can I solve a slightly different version? These are the last two parts of every problem below.

> **WHY THE ADDRESS TABLE COMES BEFORE THE CODE.** A web application is, from the outside, nothing but a set of addresses and their answers. If you cannot write the table, you do not yet know what to build; if you can, the routing almost writes itself, and checking the finished site is a matter of going down the list.

The code of all five problems is one application, `Chapter04/aspnet/LaundryShowcase`, which grows problem by problem; the five tempting wrong solutions are tiny separate projects in `Chapter04/aspnet/Tempting`, so that each mistake can be seen alone. Everything was built with the .NET 10 SDK and run; `Chapter04/verify.sh` starts the solution and each wrong solution, asks every address of this chapter with `curl` and prints the answers, and every output below comes from it. Like the application of Chapter 3, LaundryShowcase uses only the ASP.NET Core shared framework.

4.3 Problem 4.1: The Showcase Site

4.3.1 The problem

> **PROBLEM 4.1.** Profitina Laundry wants a website that presents the business to people who might invest in it, work with it or work for it. It must have a home page, a profile of the business, a description of the company, a description of its services, and a page that explains what the business offers and asks of investors, partners and job candidates. The addresses must be: `/laundry` for the home page, `/laundry/profile`, `/laundry/company`, `/laundry/services` and `/laundry/collaboration`. The site root `/` must lead to the home page. All pages share one look. The words of the site must be editable without recompiling, and every number shown must come from the shop's data, not be typed by hand.

4.3.2 Understanding it by hand

In plain words: five pages under a common prefix, one shared frame, text from a file, numbers from the data. Step 2 turns the statement into **Table 4.2**; the last three rows are addresses that must *not* work, which a statement rarely lists and a developer must always add.

Table 4.2 The address table of Problem 4.1, including three addresses that test the edges.

Method	Address	Action	Expected
GET	`/`	redirect	302 to `/laundry`
GET	`/laundry`	`Home`	200, tagline and live counts
GET	`/laundry/profile`	`Profile`	200, story and promises
GET	`/laundry/company`	`Company`	200, mission, numbers, team
GET	`/laundry/services`	`Services`	200, price list
GET	`/laundry/collaboration`	`Collaboration`	200, three audiences
GET	`/profile`	none	404: pages live under `/laundry`
GET	`/Showcase/Profile`	none	404: no conventional route
GET	`/laundry/Profile`	`Profile`	200: matching ignores case

Figure 4.2 draws the same table as a site map, together with the tracking addresses that Problem 4.2 adds.

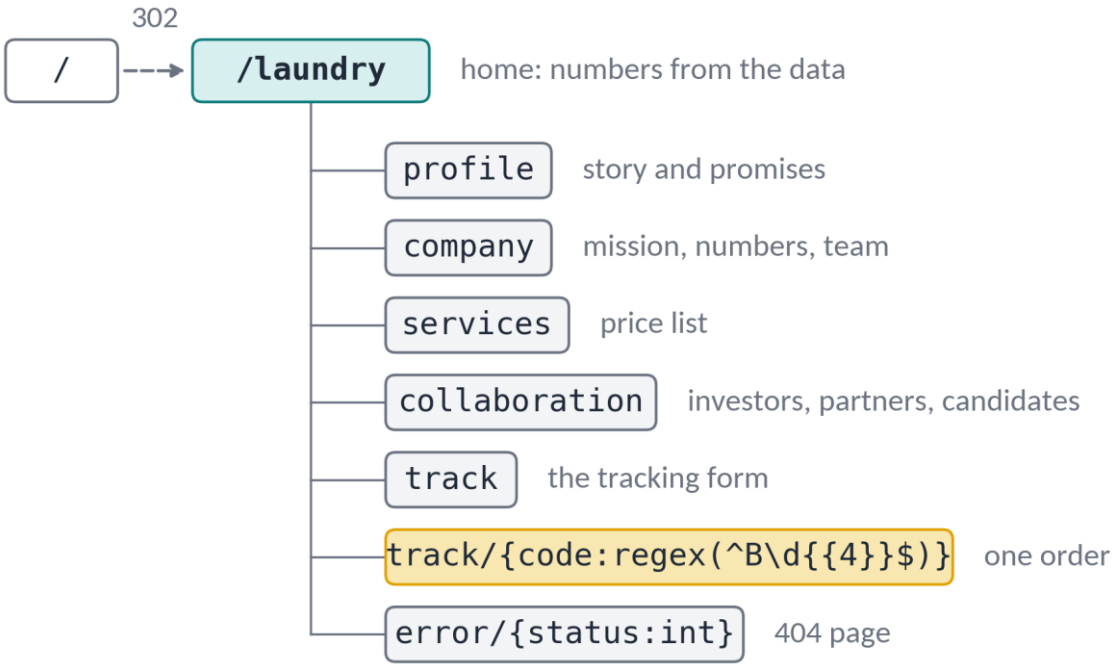


Figure 4.2 The site map of LaundryShowcase: every address is an attribute route of one controller under the prefix `/laundry`; the tracking route with a constraint belongs to Problem 4.2.

4.3.3 Designing it on paper

Routing. All pages share the prefix `laundry`, which is what `[Route("laundry")]` on a controller expresses. One controller, `ShowcaseController`, with one action per page, is enough; five controllers of one action each would scatter a single idea over five files. The site root is not a page of the showcase but a door to it, so it is a one-line minimal-API redirect in `Program.cs`. The application registers `MapControllers()` (attribute routes only) and no conventional route, so `/Showcase/Profile` cannot exist.

Content. "Editable without recompiling" means the words live in a file that the application reads at start-up: `content/site.json`, deserialized into C# records with `System.Text.Json`, which is part of .NET. The content object is registered in the container as a singleton, so every controller and view can receive it.

Data. "Numbers from the data" means reading the shop's exports (`Service.csv`, `Outlet.csv`, `Bill.csv`) once and computing the facts: the number of orders, the first and last order dates, the average order and the orders per outlet. A class `LaundryData` does both, also registered as a singleton.

Views. One layout with the menu (the current page marked, as in Chapter 3), and one view per action. The layout needs the site's name on every page, so instead of passing it from every action, `_ViewImports.cshtml` injects the content object into *every* view with `@inject SiteContent Site`.

4.3.4 The solution

Listing 4.1 is the content file. The texts are kept short so that each line fits; a real site would have longer paragraphs, and the program would not change.

Listing 4.1 `content/site.json`: every word of the showcase that is not computed from data.

```
{
  "name": "Profitina Laundry",
  "tagline": "Clean clothes, honest prices, ready on the promised day.",
  "city": "Surabaya",
  "story": [
    "We began in January 2019 as one counter in Rungkut.",
    "Every order is weighed or counted in front of the customer.",
    "The notebook became a database; the promise stayed the same."
  ],
  "values": [
    {
      "title": "Weighed in front of you",
      "text": "The scale faces the customer, and the price follows from it."
    },
    {
      "title": "A rack for every order",
      "text": "Finished laundry waits on a numbered rack, found in seconds."
    },
    {
      "title": "The promised day",
      "text": "Express is ready the next day, regular in two to three days."
    }
  ],
  "mission": "Give every household a few hours back each week, at a fair price.",
  "team": [
    { "role": "Counter and transactions", "people": 5 },
    { "role": "Washing and drying", "people": 5 },
    { "role": "Ironing and folding", "people": 3 },
    { "role": "Owner and supervision", "people": 1 }
  ],
  "collaboration": [
    {
      "audience": "Investors",
      "offer": "Two more outlets in east Surabaya, same system, same prices.",
      "ask": "Capital for machines and the first six months of rent.",
      "contact": "invest@profitina-laundry.example"
    },
    {
      "audience": "Partners",
      "offer": "Boarding houses and clinics get a weekly pickup, one invoice.",
      "ask": "A fixed pickup day and one contact person.",
      "contact": "partners@profitina-laundry.example"
    },
    {
      "audience": "Job candidates",
      "offer": "Paid training at the counter or in the washing room.",
```

Listing 4.1 (Continued) *content/site.json: every word of the showcase that is not computed from data.*

```
      "ask": "Care, honesty and a good sense of time.",
      "contact": "jobs@profitina-laundry.example"
    }
  ]
}
```

Listing 4.2 declares records whose shape mirrors the JSON. With `JsonSerializerOptions.Web`, the deserializer [2] matches the JSON's camelCase names (`tagline`) to the C# properties (`Tagline`) and reads numbers into `int`. Note the record `Opening`: a first version called it `Offer`, which failed to compile because it also has a property named `Offer`, and a member cannot have the same name as its enclosing type.

Listing 4.2 `Models/SiteContent.cs`: the shape of `site.json` as C# records.

```
namespace LaundryShowcase.Models;

// The words of the site, read from content/site.json at start-up.
public record SiteContent(string Name, string Tagline, string City,
    List<string> Story, List<Value> Values, string Mission,
    List<TeamRole> Team, List<Opening> Collaboration);

public record Value(string Title, string Text);
public record TeamRole(string Role, int People);
public record Opening(string Audience, string Offer, string Ask, string Contact);
```

Listing 4.3 holds the data records. `Service` and `Outlet` are those of Chapter 3; `Order` is one row of `Bill.csv`, the table the database calls *bill*. The comment on `Order` records a deliberate decision that Problem 4.2 explains: the customer is not part of the record at all. `ShopFacts` is the view model of the numbers.

Listing 4.3 `Models/Records.cs`: services, outlets, orders without customers, and the facts computed from them.

```
namespace LaundryShowcase.Models;

public record Service(string Id, string Description, decimal Price)
{
    public string Name => Description.Contains(" (")
        ? Description[..Description.LastIndexOf(" (")] : Description;

    public string Unit => Description.EndsWith("(per kg)") ? "kg"
        : Description.EndsWith("(per pair)") ? "pair" : "item";
}

public record Outlet(string Id, string Name, string Address, string Phone,
    DateOnly Opened);

// One row of data/Bill.csv. The customer is deliberately left out:
// the tracking page must not tell a stranger whose laundry this is.
public record Order(string Id, DateTime Arrived, DateTime? Finished,
    DateTime? Departed, string Status, string Rack, int Units,
    decimal Payment, string ServiceId, string OutletId);

// Numbers the profile and company pages compute from the data.
public record ShopFacts(int Orders, int Outlets, int Services,
    DateOnly FirstOrder, DateOnly LastOrder, decimal AverageTicket,
    Dictionary<string, int> OrdersPerOutlet);
```

`Money.Rp` is the same helper as in Chapter 3 and is not repeated here. **Listing 4.4** reads the three files once and computes the facts. The orders are kept in a dictionary keyed by code with a case-insensitive comparer, so that looking up `b0072` finds `B0072` in one step. `Facts()` uses LINQ: `Min` and `Max` of the arrival dates, `Average` of the payments rounded to whole rupiah, and `GroupBy` over the outlet code to count the orders per outlet.

Listing 4.4 `Data/LaundryData.cs`: the shop's data, loaded once, and the facts the pages show.

```
using System.Globalization;
using System.Text;
using LaundryShowcase.Models;

namespace LaundryShowcase.Data;

// Loads the three CSV files once and answers questions about them.
public class LaundryData
{
    private static readonly CultureInfo Inv = CultureInfo.InvariantCulture;

    public IReadOnlyList<Service> Services { get; }
    public IReadOnlyList<Outlet> Outlets { get; }
    private readonly Dictionary<string, Order> orders;

    public LaundryData(string folder)
    {
        Services = Csv(Path.Combine(folder, "Service.csv"))
            .Select(f => new Service(f[0], f[1], decimal.Parse(f[2], Inv)))
            .ToList();
        Outlets = Csv(Path.Combine(folder, "Outlet.csv"))
            .Select(f => new Outlet(f[0], f[1], f[2], f[3],
                DateOnly.Parse(f[4], Inv)))
            .ToList();
        orders = Csv(Path.Combine(folder, "Bill.csv"))
            .Select(f => new Order(f[0], DateTime.Parse(f[1], Inv), When(f[2]),
                When(f[3]), f[4], f[5], int.Parse(f[6], Inv),
                decimal.Parse(f[7], Inv), f[10], f[11]))
            .ToDictionary(o => o.Id, StringComparer.OrdinalIgnoreCase);
    }

    public Order? FindOrder(string code) =>
        orders.TryGetValue(code, out var order) ? order : null;

    public Service? FindService(string id) =>
        Services.FirstOrDefault(s => s.Id == id);

    public Outlet? FindOutlet(string id) =>
        Outlets.FirstOrDefault(o => o.Id == id);
```

Listing 4.4 (Continued) *Data/LaundryData.cs: the shop's data, loaded once, and the facts the pages show.*

```
    public ShopFacts Facts()
    {
        var all = orders.Values;
        return new ShopFacts(all.Count, Outlets.Count, Services.Count,
            DateOnly.FromDateTime(all.Min(o => o.Arrived)),
            DateOnly.FromDateTime(all.Max(o => o.Arrived)),
            Math.Round(all.Average(o => o.Payment)),
            all.GroupBy(o => o.OutletId)
               .ToDictionary(g => FindOutlet(g.Key)?.Name ?? g.Key,
                             g => g.Count()));
    }

    private static DateTime? When(string text) =>
        text == "" ? null : DateTime.Parse(text, Inv);

    // Splits each line on commas, except commas inside "double quotes".
    private static IEnumerable<string[]> Csv(string path)
    {
        foreach (var line in File.ReadLines(path).Skip(1))
        {
            var fields = new List<string>();
            var field = new StringBuilder();
            bool quoted = false;
            foreach (char c in line)
            {
                if (c == '"') quoted = !quoted;
                else if (c == ',' && !quoted)
                {
                    fields.Add(field.ToString());
                    field.Clear();
                }
                else field.Append(c);
            }
            fields.Add(field.ToString());
            yield return fields.ToArray();
        }
    }
}
```

Listing 4.5 is the complete `Program.cs`, including the parts that Problems 4.3 to 4.5 add; each is explained in its own problem. For Problem 4.1 the important lines are the two registrations, the redirect from `/` and `MapControllers()`. The content is read and deserialized once at start-up and registered as an *instance*, the real-framework form of MiniFrame's `container.Instance(...)`.

Listing 4.5 `Program.cs` of LaundryShowcase: content and data registered as singletons, a middleware pipeline, a redirect, the controllers, and a development-only route list.

```
using System.Text.Json;
using LaundryShowcase.Data;
using LaundryShowcase.Middleware;
using LaundryShowcase.Models;

var builder = WebApplication.CreateBuilder(args);
var root = builder.Environment.ContentRootPath;

// Services
builder.Services.AddControllersWithViews();
builder.Services.AddSingleton(JsonSerializer.Deserialize<SiteContent>(
    File.ReadAllText(Path.Combine(root, "content", "site.json")),
    JsonSerializerOptions.Web)!);
builder.Services.AddSingleton(new LaundryData(Path.Combine(root, "data")));

var app = builder.Build();

// Pipeline
app.UseMiddleware<ResponseTimeMiddleware>();
app.UseStatusCodePagesWithReExecute("/laundry/error/{0}");
app.UseRouting();

// Maintenance mode: after UseRouting, so the chosen endpoint is known and
// the style sheet can still be served while every page is closed.
app.Use(async (context, next) =>
{
    bool closed = app.Configuration.GetValue<bool>("Showcase:Maintenance");
    var endpoint = context.GetEndpoint() as RouteEndpoint;  // set by routing
    bool isAsset = endpoint?.RoutePattern.RawText?.StartsWith("css/") == true;
    if (closed && !isAsset)
    {
        context.Response.StatusCode = StatusCodes.Status503ServiceUnavailable;
        context.Response.Headers.RetryAfter = "3600";
        context.Response.ContentType = "text/html; charset=utf-8";
        await context.Response.WriteAsync("""
            <!DOCTYPE html><html lang="en"><head><meta charset="utf-8">
            <title>Back soon - Profitina Laundry</title>
            <link rel="stylesheet" href="/css/site.css"></head>
            <body><main><h1>Back soon</h1>
            <p>The website is being updated. The outlets are open as usual.</p>
            </main></body></html>
            """);
        return;
    }
    await next(context);
});
```

Listing 4.5 (Continued) *Program.cs of LaundryShowcase: content and data registered as singletons, a middleware pipeline, a redirect, the controllers, and a development-only route list.*

```
app.MapStaticAssets();
app.MapGet("/", () => Results.Redirect("/laundry"));
app.MapControllers();

if (app.Environment.IsDevelopment())
{
    // Lists every route the application knows (Problem 4.3).
    app.MapGet("/_routes", (EndpointDataSource source) => string.Join("\n",
        source.Endpoints.OfType<RouteEndpoint>()
            .Where(e => !e.RoutePattern.RawText!.StartsWith("css/"))
            .OrderBy(e => e.RoutePattern.RawText)
            .Select(e => $"{e.RoutePattern.RawText,-40} " + e.DisplayName?
                .Replace("LaundryShowcase.Controllers.", "").Split(" (")[0])));
}

app.Run();
```

Listing 4.6 is the one controller. Each page action is a single line: it hands the view the model it needs (the facts, the services or the collaboration list) and nothing else. The tracking and error actions belong to Problem 4.2.

Listing 4.6 `Controllers/ShowcaseController.cs`: one attribute-routed controller for the whole showcase.

```
using LaundryShowcase.Data;
using LaundryShowcase.Models;
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;

namespace LaundryShowcase.Controllers;

[Route("laundry")]
public class ShowcaseController(SiteContent site, LaundryData data)
    : Controller
{
    [HttpGet("")]                                   // /laundry
    public IActionResult Home() => View(data.Facts());

    [HttpGet("profile")]                            // /laundry/profile
    public IActionResult Profile() => View(data.Facts());

    [HttpGet("company")]                            // /laundry/company
    public IActionResult Company() => View(data.Facts());

    [HttpGet("services")]                           // /laundry/services
    public IActionResult Services() => View(data.Services);

    [HttpGet("collaboration")]                      // /laundry/collaboration
    public IActionResult Collaboration() => View(site.Collaboration);

    [HttpGet("track")]                              // the form, and its answer
    public IActionResult TrackForm(string? code)
    {
        if (string.IsNullOrWhiteSpace(code))
        {
            return View();
        }
        return RedirectToAction(nameof(Track), new { code = code.Trim() });
    }
```

Listing 4.6 (Continued) *Controllers/ShowcaseController.cs: one attribute-routed controller for the whole showcase.*

```
    [HttpGet(@"track/{code:regex(^B\d{{4}}$)}")]  // /laundry/track/B0072
    public IActionResult Track(string code)
    {
        var order = data.FindOrder(code);
        if (order is null)
        {
            return NotFound();
        }
        var service = data.FindService(order.ServiceId);
        ViewData["Service"] = service?.Name;
        ViewData["Unit"] = service?.Unit;
        ViewData["Outlet"] = data.FindOutlet(order.OutletId)?.Name;
        return View(order);
    }

    [HttpGet("error/{status:int}")]
    public IActionResult Error(int status)
    {
        var original = HttpContext.Features.Get<IStatusCodeReExecuteFeature>();
        ViewData["Path"] = original?.OriginalPath ?? Request.Path.Value;
        Response.StatusCode = status;
        return View(status);
    }
}
```

Listing 4.7 makes the content available to every view, and **Listing 4.8** is the layout. Because every action belongs to the same controller, the menu's `Here` function compares only the action name, and `asp-action` without `asp-controller` means "an action of the current controller".

Listing 4.7 `Views/_ViewImports.cshtml`: `@inject` puts the site's content into every view.

```
@using LaundryShowcase
@using LaundryShowcase.Models
@inject SiteContent Site
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Listing 4.8 `Views/Shared/_Layout.cshtml`: the showcase's frame and menu.

```
@{
    string? Here(string action) =>
        ViewContext.RouteData.Values["action"] as string == action
            ? "here" : null;
}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - @Site.Name</title>
    <link rel="icon" href="data:," />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
</head>
<body>
    <header>
        <a class="brand" asp-action="Home">@Site.Name</a>
        <nav>
            <a class="@Here("Home")" asp-action="Home">Home</a>
            <a class="@Here("Profile")" asp-action="Profile">Profile</a>
            <a class="@Here("Company")" asp-action="Company">Company</a>
            <a class="@Here("Services")" asp-action="Services">Services</a>
            <a class="@Here("Collaboration")"
               asp-action="Collaboration">Work with us</a>
            <a class="@Here("TrackForm")" asp-action="TrackForm">Track</a>
        </nav>
    </header>
    <main>
        @RenderBody()
    </main>
    <footer>
        <p>@Site.Name &middot; @Site.City &middot; @Site.Tagline</p>
    </footer>
</body>
</html>
```

The five page views follow: **Listing 4.9**, **Listing 4.10**, **Listing 4.11**, **Listing 4.12** and **Listing 4.13**. They contain no numbers and no rules, only where to show what: `@Model.Orders.ToString("N0")` formats 2953 as “2,953”, and in **Listing 4.11** the loop `@foreach (var (outlet, count) in Model.OrdersPerOutlet)` takes each dictionary entry apart into its two halves.

Listing 4.9 `Views/Showcase/Home.cshtml`: the tagline and counts computed from the data.

```
@model ShopFacts
@{
    ViewData["Title"] = "Home";
}
<section class="hero">
    <h1>@Site.Tagline</h1>
    <p>@Model.Outlets outlets in @Site.City, @Model.Services services,
       and @Model.Orders.ToString("N0") orders since
       @Model.FirstOrder.ToString("MMMM yyyy").</p>
    <p>
        <a class="button" asp-action="Services">See the prices</a>
        <a class="button" asp-action="TrackForm">Track an order</a>
        <a class="button light" asp-action="Collaboration">Work with us</a>
    </p>
</section>
```

Listing 4.10 `Views/Showcase/Profile.cshtml`: the story and the three promises, from `site.json`.

```
@model ShopFacts
@{
    ViewData["Title"] = "Profile";
}
<h1>Who we are</h1>
@foreach (var paragraph in Site.Story)
{
    <p>@paragraph</p>
}
<h2>What we promise</h2>
<div class="cards">
    @foreach (var value in Site.Values)
    {
        <div class="card"><h3>@value.Title</h3><p>@value.Text</p></div>
    }
</div>
```

Listing 4.11 `Views/Showcase/Company.cshtml`: the mission, the numbers from the data, and the team.

```
@model ShopFacts
@{
    ViewData["Title"] = "Company";
    int people = Site.Team.Sum(t => t.People);
}
<h1>The company</h1>
<p class="lead">@Site.Mission</p>
<h2>In numbers</h2>
<table class="list">
    <tr><th>Orders from @Model.FirstOrder.ToString("d MMM yyyy")
        to @Model.LastOrder.ToString("d MMM yyyy")</th>
        <td class="num">@Model.Orders.ToString("N0")</td></tr>
    <tr><th>Average order</th>
        <td class="num">@Money.Rp(Model.AverageTicket)</td></tr>
    @foreach (var (outlet, count) in Model.OrdersPerOutlet)
    {
        <tr><th>Orders at @outlet</th>
            <td class="num">@count.ToString("N0")</td></tr>
    }
</table>
<h2>The team: @people people</h2>
<ul>
    @foreach (var t in Site.Team)
    {
        <li>@t.Role: @t.People</li>
    }
</ul>
```

Listing 4.12 `Views/Showcase/Services.cshtml`: the price list.

```
@model IReadOnlyList<Service>
@{
    ViewData["Title"] = "Services";
}
<h1>Services and prices</h1>
<table class="list">
    <thead><tr><th>Service</th><th class="num">Price</th></tr></thead>
    <tbody>
    @foreach (var s in Model)
    {
        <tr><td>@s.Name</td><td class="num">@Money.Rp(s.Price) / @s.Unit</td></tr>
    }
    </tbody>
</table>
```

Listing 4.13 `Views/Showcase/Collaboration.cshtml`: what the business offers and asks of each audience.

```
@model List<Opening>
@{
    ViewData["Title"] = "Work with us";
}
<h1>Work with us</h1>
<div class="cards">
    @foreach (var o in Model)
    {
        <div class="card">
            <h3>@o.Audience</h3>
            <p><b>We offer:</b> @o.Offer</p>
            <p><b>We ask:</b> @o.Ask</p>
            <p><a href="mailto:@o.Contact">@o.Contact</a></p>
        </div>
    }
</div>
```

4.3.5 Running it

```
cd Chapter04/aspnet/LaundryShowcase
dotnet run
```

Figure 4.3 is `/laundry`: the tagline from `site.json` and three numbers computed from the data. **Figure 4.4** is the company page. The data are the sample data of the Profitina Laundry database, 2,953 orders from October 1, 2025 to September 30, 2026, with an average of Rp 48.836 per order; the Rungkut outlet, the oldest, took 1,312 of them, Keputih 996 and Wiyung, opened in January 2026, 645. None of these numbers appears anywhere in the code or the content file.

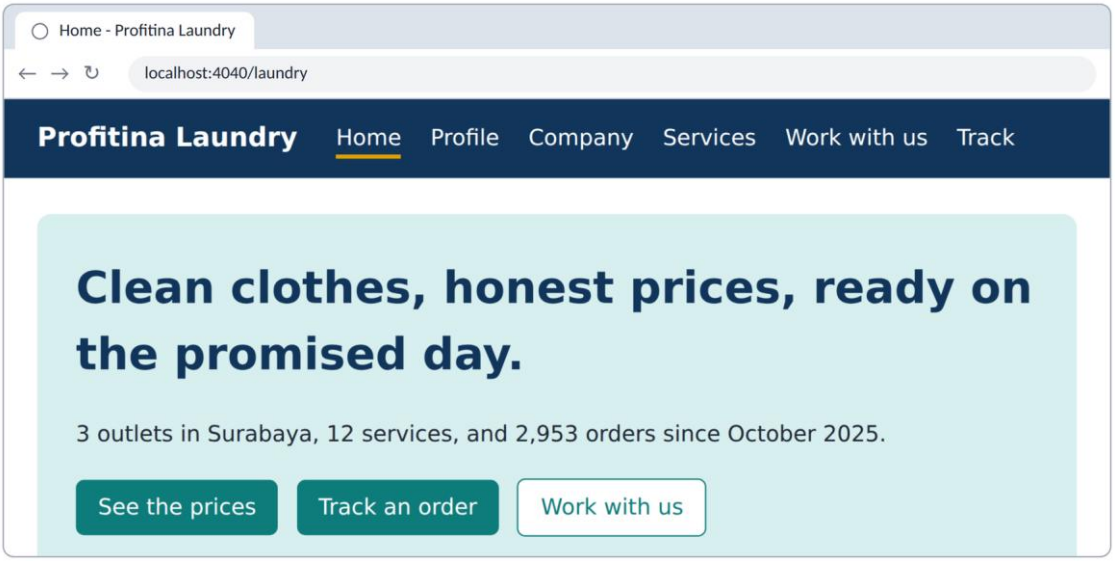


Figure 4.3 The showcase home page: the tagline comes from `site.json`, the counts from the data.

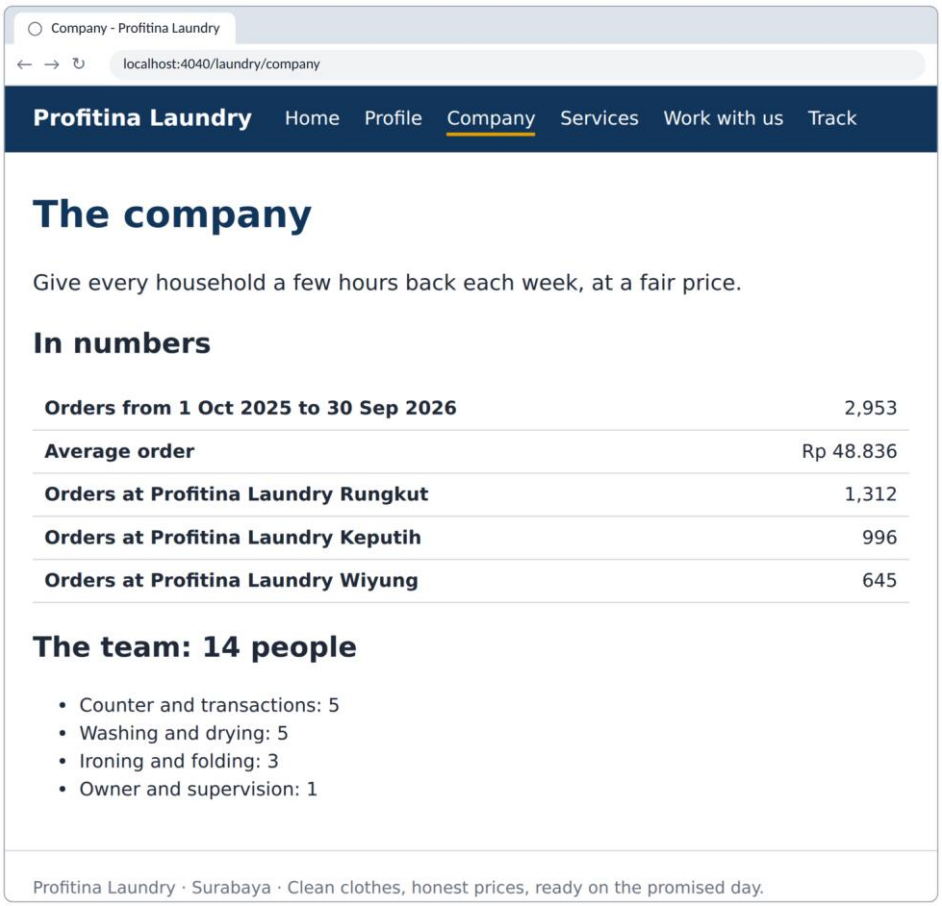


Figure 4.4 The company page: every number is computed by `LaundryData.Facts()` from the order data.

Figure 4.5 is the collaboration page, the three audiences of the problem statement side by side.

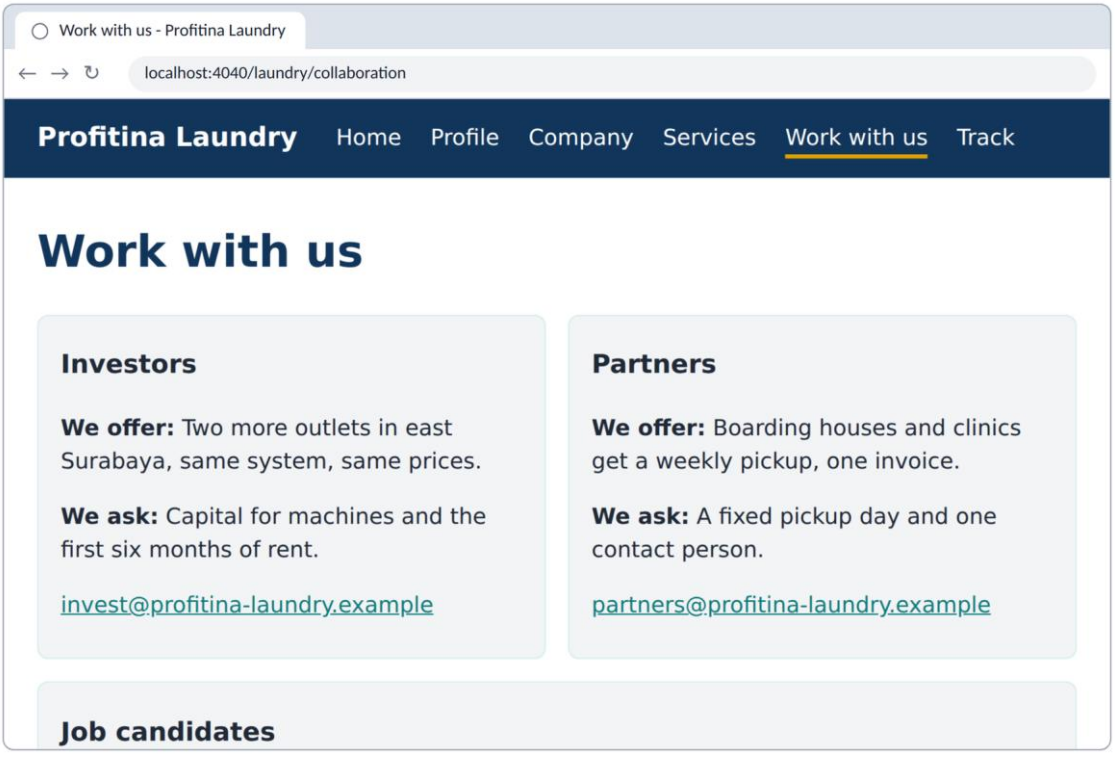


Figure 4.5 The collaboration page: offers and asks for investors, partners and job candidates, from `site.json`.

The address table is checked by `verify.sh`. The relevant lines of its output are:

```
302  /
200  /laundry
200  /laundry/profile
200  /laundry/company
200  /laundry/services
200  /laundry/collaboration
200  /laundry/Profile
404  /profile
404  /Showcase/Profile
```

and the redirect from the root carries the header `Location: /laundry`. Every row of **Table 4.2** holds.

4.3.6 A tempting wrong solution

The tempting mistake is a single character. A developer who writes the profile's route with a leading slash, `[HttpGet("/profile")]`, "because addresses start with a slash", has written an *absolute* route: it ignores the controller's prefix. **Listing 4.14** is the smallest program that shows it.

Listing 4.14 `Tempting/AbsoluteRoute/Program.cs`: one leading slash moves the profile out of `/laundry`.

```
// Tempting wrong solution for Problem 4.1: a leading "/" on an action's
// route throws away the controller's "laundry" prefix.
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
var app = builder.Build();
app.MapControllers();
app.Run();

[Route("laundry")]
public class ShowcaseController : Controller
{
    [HttpGet("")]
    public string Home() => "home";

    [HttpGet("/profile")]              // WRONG: absolute, not "profile"
    public string Profile() => "profile";
}
```

It compiles without a warning, and it runs:

```
200  /laundry  home
404  /laundry/profile
200  /profile  profile
```

The page exists, at the wrong address, and the menu's link (generated from the route) even points there, so clicking through the site shows nothing wrong. Only the address table catches it. Chapter 3 used a leading slash on purpose, for `/api/services/{id}` and `/error/{code:int}`; here it is an accident. The rule: inside a controller with `[Route(...)]`, an action's template starts with a slash only when it is meant to leave the prefix.

4.3.7 Check yourself

Variant. The owner wants the showcase at `/profitina-laundry` instead of `/laundry`, with the old addresses still working for a while. What changes?

Answer. One attribute: `[Route("profitina-laundry")]`. Every link in the views is generated from routes by tag helpers, so the menu follows by itself. To keep the old addresses, add a second attribute, `[Route("laundry")]`, on the class; a controller may have several route prefixes, and each action then has two addresses. The redirect in `Program.cs` should point to the new one.

4.4 Problem 4.2: Where Is My Laundry?

4.4.1 The problem

> **PROBLEM 4.2.** Every order ticket of Profitina Laundry carries a code: the letter B followed by four digits, such as B0072. Add to the showcase a page where a customer types that code and sees the status of the order (received, ready with the rack number, or collected), the service, the outlet, the dates and the amount. The form is at `/laundry/track`; each order has its own address, `/laundry/track/B0072`, so that the answer can be bookmarked. Codes may be typed in lower case. An address that cannot be an order code must not reach the tracking code at all; a code that could exist but does not must get a polite "not found" page with the status 404.

4.4.2 Understanding it by hand

Three kinds of order exist in the data, one per status. **Table 4.3** is worked out by hand from three rows of `Bill.csv`, and it is exactly what the page must show.

Table 4.3 Three orders from `Bill.csv`, one per status, worked out by hand. The amount of B0072 checks: $7 \times 8000 = 56000$.

Code	Status	Service, units	Outlet	Received	Finished	Collected	Amount
B0001	departed	S0008 Bed cover, 1	O01 Rungkut	1 Oct 2025, 09:46	4 Oct 2025, 07:16	5 Oct 2025, 07:24	35.000
B0072	finished (rack A01)	S0001 Wash, dry & fold, 7 kg	O01 Rungkut	14 Oct 2025, 12:11	16 Oct 2025, 07:45	not yet	56.000
B2953	queued	S0006 Dress/kebaya, 1	O01 Rungkut	30 Sep 2026, 20:49	not yet	not yet	45.000

The address table of the problem is **Table 4.4**. The interesting rows are the failures: `X12` and `B00721` can never be order codes, while `B9999` could be one but is not.

Table 4.4 The address table of Problem 4.2.

Address	Expected
`/laundry/track`	200, the form
`/laundry/track?code=b0072`	302 to `/laundry/track/b0072`
`/laundry/track/B0072`, `/laundry/track/b0001`, `/laundry/track/B2953`	200, the order
`/laundry/track/B9999`	404, friendly page: a possible code, no such order
`/laundry/track/X12`, `/laundry/track/B00721`	404: not a possible code, no route matches

4.4.3 Designing it on paper

"Cannot be an order code" is a route constraint. The rule "B followed by exactly four digits" is a regular expression, `^B\d{4}$`: `^` and `$` anchor it to the whole segment, `\d` is a digit, `{4}` means exactly four. As a route constraint, `{code:regex(...)}`, it makes addresses such as `X12` match no route at all, so the tracking action runs only for possible codes. ASP.NET Core evaluates regular-

expression constraints *ignoring case* [3], which takes care of lower-case codes without a second pattern.

One detail of syntax decides everything. Inside a route template, braces mean "parameter". The regular expression's own braces in `{4}` must therefore be doubled, `{{4}}`, and so must square brackets, `[[` and `]]`, because in attribute routes brackets mark tokens such as `[controller]` [3]. The template becomes `track/{code:regex(^B\d{{4}}$)}`, written as a verbatim string so that the backslash reaches the regular expression unchanged.

The form is a `GET` that redirects. The form at `/laundry/track` sends `?code=…` to the same address. Its action does not show the order itself; it answers with a **redirect** to `/laundry/track/{code}`, so that the address in the browser becomes the order's own address, as the statement asks. `RedirectToAction(nameof(Track), new { code })` asks the routing system for that address, exactly as tag helpers do.

Privacy. The tracking page is public: anybody who types a code sees the answer. A code is short and guessable (there are only 10,000 of them), so the page must show nothing that identifies the customer: no name, no phone number, no address. That is why `Order` in **Listing 4.3** has no customer field at all; what is not loaded cannot leak. Indonesia's Personal Data Protection Law, UU No. 27 Tahun 2022, requires exactly this kind of care with personal data [4]. The rack number is shown, because it helps the customer at the counter and tells a stranger nothing.

4.4.4 The solution

The two actions are in **Listing 4.6**. `TrackForm` shows the form when no code is given, and otherwise trims the code and redirects. `Track` looks the order up and returns `NotFound()` when there is none, which the status-code middleware turns into the friendly page with status 404; otherwise it puts the service's name and unit and the outlet's name into `ViewData` and shows the order. The `Error` action is Chapter 3's, under the prefix: the pipeline re-executes `/laundry/error/404` [5]. **Listing 4.15** is the form and **Listing 4.16** the order page.

Listing 4.15 `Views/Showcase/TrackForm.cshtml`: a `GET` form with one field.

```
@{
    ViewData["Title"] = "Track an order";
}
<h1>Where is my laundry?</h1>
<form asp-action="TrackForm" method="get" class="track">
    <label for="code">Order code on your ticket, for example B0072</label>
    <input id="code" name="code" maxlength="5" autofocus />
    <button type="submit">Track</button>
</form>
```

Listing 4.16 `Views/Showcase/Track.cshtml`: the status in words, then the facts of the order, with no personal data.

```
@model Order
@{
    ViewData["Title"] = $"Order {Model.Id}";
    string When(DateTime? t) => t?.ToString("ddd d MMM yyyy, HH:mm") ?? "-";
}
<h1>Order @Model.Id</h1>
<p class="status @Model.Status">
    @switch (Model.Status)
    {
        case "queued": <text>Received and waiting to be washed.</text> break;
        case "finished":
            <text>Ready: collect it from rack @Model.Rack.</text> break;
        default: <text>Collected. Thank you!</text> break;
    }
</p>
<table class="list">
    <tr><th>Service</th>
        <td>@ViewData["Service"], @Model.Units @ViewData["Unit"]</td></tr>
    <tr><th>Outlet</th><td>@ViewData["Outlet"]</td></tr>
    <tr><th>Received</th><td>@When(Model.Arrived)</td></tr>
    <tr><th>Finished</th><td>@When(Model.Finished)</td></tr>
    <tr><th>Collected</th><td>@When(Model.Departed)</td></tr>
    <tr><th>Amount</th><td>@Money.Rp(Model.Payment)</td></tr>
</table>
<p><a asp-action="TrackForm">Track another order</a></p>
```

The view shows two Razor features not seen before. `@switch` chooses the sentence for the status, and `<text>…</text>` marks plain text inside C# code, where Razor would otherwise expect a tag. The local function `When` formats a date that may be missing, printing “-” for `null`. The class of the status paragraph, `status @Model.Status`, lets the style sheet color a queued order amber. **Listing 4.17** is the error view.

Listing 4.17 `Views/Shared/Error.cshtml`: one sentence for a missing page or order, with the address that was asked for.

```
@model int
@{
    ViewData["Title"] = Model == 404 ? "Not found" : "Something went wrong";
}
<h1>@ViewData["Title"]</h1>
<p>@(Model == 404 ? "There is no page or order at" : "We could not answer")
   <code>@ViewData["Path"]</code>.</p>
<p><a asp-controller="Showcase" asp-action="Home">Back to the home page</a></p>
```

4.4.5 Running it

Figure 4.6 is the page for B0072, and it agrees with **Table 4.3** line by line.

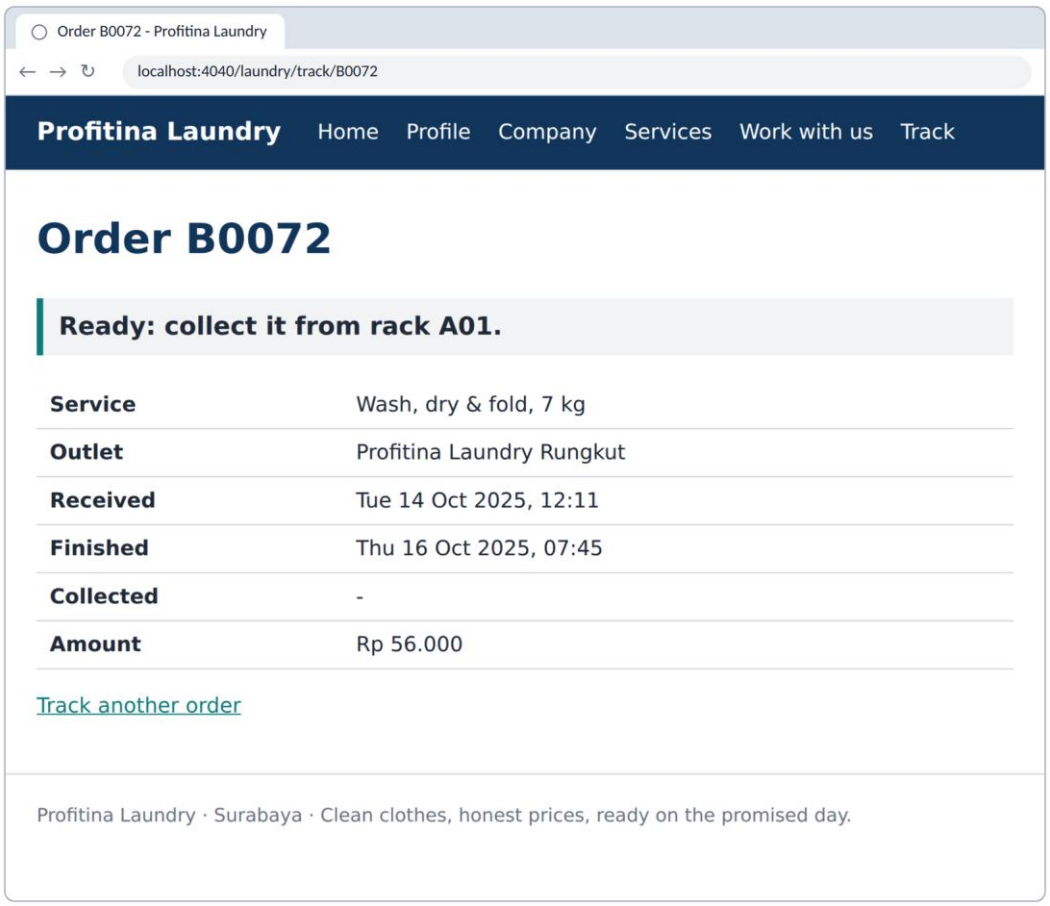


Figure 4.6 The tracking page for order B0072: ready on rack A01, with service, outlet, dates and amount, and nothing about the customer.

`verify.sh` asks the addresses of **Table 4.4**:

```
200  /laundry/track
302  /laundry/track?code=b0072
200  /laundry/track/B0072
200  /laundry/track/b0001
200  /laundry/track/B2953
404  /laundry/track/B9999
404  /laundry/track/X12
404  /laundry/track/B00721
```

The redirect is `Location: /laundry/track/b0072`, the code as typed: the route constraint accepts it, and the dictionary lookup ignores case. The texts of the three order pages, with tags removed, are:

```
== text of /laundry/track/B0072
Order B0072
Ready: collect it from rack A01.
Service
Wash, dry &amp;amp; fold, 7 kg
OutletProfitina Laundry Rungkut
ReceivedTue 14 Oct 2025, 12:11
FinishedThu 16 Oct 2025, 07:45
Collected-
AmountRp 56.000
Track another order
== text of /laundry/track/B2953
Order B2953
Received and waiting to be washed.
Service
Dry cleaning - dress/kebaya, 1 item
OutletProfitina Laundry Rungkut
ReceivedWed 30 Sep 2026, 20:49
Finished-
Collected-
AmountRp 45.000
Track another order
== text of /laundry/track/B0001
Order B0001
Collected. Thank you!
Service
Bed cover, any size, 1 item
OutletProfitina Laundry Rungkut
ReceivedWed 1 Oct 2025, 09:46
FinishedSat 4 Oct 2025, 07:16
CollectedSun 5 Oct 2025, 07:24
AmountRp 35.000
Track another order
== text of /laundry/track/B9999
Not found
There is no page or order at
/laundry/track/B9999.
Back to the home page
```

(The table cells run together, “OutletProfitina…”, only because the script removed the tags between them.) All three statuses and the missing order match the design.

4.4.6 A tempting wrong solution

The tempting mistake is to write the regular expression as one learns it, with single braces: `{code:regex(^B\d{4}$)}`. **Listing 4.18** does exactly that.

Listing 4.18 `Tempting/RegexBraces/Program.cs`: a regular expression with single braces inside a route template.

```
// Tempting wrong solution for Problem 4.2: single braces inside a regex
// constraint. In a route template { and } mean "parameter".
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
var app = builder.Build();
app.MapControllers();
app.Run();

[Route("laundry")]
public class ShowcaseController : Controller
{
    [HttpGet(@"track/{code:regex(^B\d{4}$)}")]     // WRONG: needs {{4}}
    public string Track(string code) => $"order {code}";
}
```

The compiler's route analyzer already warns, twice, while building:

```
Program.cs(14,28): warning ASP0017: Route issue: In a route parameter, '{' and '}' must be
escaped with '{{' and '}}'. (https://aka.ms/aspnet/analyzers)
Program.cs(14,41): warning ASP0017: Route issue: There is an incomplete parameter in the
route template. Check that each '{' character has a matching '}' character.
(https://aka.ms/aspnet/analyzers)
```

Warnings are easy to ignore, so the program is run anyway. It does not survive its first request: when routing builds the route table, the template parser meets `{4}` inside the parameter, and the application stops with an exception. `curl` gets no answer at all (its status is printed as `000`), and the log says:

```
      Application startup exception
      Microsoft.AspNetCore.Routing.Patterns.RoutePatternException: In a route parameter,
'{' and '}' must be escaped with '{{' and '}}'.
         at Microsoft.AspNetCore.Routing.Patterns.RoutePatternParser.Parse(String pattern)
```

This is a mistake that fails loudly, which is the best kind. The lesson has two halves: read the build warnings (the analyzer knew), and remember that route templates have their own syntax in which `{`, `}`, `[` and `]` are special.

4.4.7 Check yourself

Variant. The shop plans to move to five-digit codes (B00001) next year, while old tickets with four digits stay in circulation. Write the constraint.

Answer. `track/{code:regex(^B\d{{4,5}}$)}`: `{4,5}` means four or five digits, and both braces are doubled. Nothing else changes, because the dictionary lookup does not care about the length. A tempting alternative, `{code:length(5,6)}`, would also accept `XXXXX`.

4.5 Problem 4.3: The Route Table by Hand

4.5.1 The problem

> **PROBLEM 4.3.** Without running the program, use **Listing 4.5** and **Listing 4.6** to predict, for each of the twelve requests below, which endpoint handles it and which status code comes back: `GET /`, `/laundry`, `/laundry/Profile`, `/profile`, `/Showcase/Profile`, `/laundry/track`, `/laundry/track?code=b0072`, `/laundry/track/b0001`, `/laundry/track/B9999`, `/laundry/track/X12`, `/laundry/track/B00721`, `/css/site.css`. Then make the application show its own route table, and check every prediction against the running program.

4.5.2 Understanding it by hand

This problem is the router of Chapter 2, run in the head. For each path, list the templates that could match, apply the constraints, and when several match, apply precedence. The endpoints come from three places in `Program.cs`: `MapStaticAssets()` (the files in `wwwroot`), `MapGet("/", …)` (the redirect), and `MapControllers()` (every attribute route of `ShowcaseController`). There is no conventional route.

4.5.3 Designing it on paper

Table 4.5 is the prediction, one row per request, with the reason.

Table 4.5 The route table predicted by hand for twelve requests. Two different reasons hide behind the 404s: no route, or an action that returned `NotFound()`.

Request	Matching template	Endpoint	Predicted status
`/`	`/`	the redirect	302
`/laundry`	`laundry`	`Home`	200
`/laundry/Profile`	`laundry/profile` (case ignored)	`Profile`	200
`/profile`	none (no prefix)	none	404
`/Showcase/Profile`	none (no conventional route)	none	404
`/laundry/track`	`laundry/track`	`TrackForm`	200
`/laundry/track?code=b0072`	`laundry/track` (query ignored by routing)	`TrackForm`	302
`/laundry/track/b0001`	`laundry/track/{code:regex(...)}`, case ignored	`Track`	200
`/laundry/track/B9999`	the same; the order does not exist	`Track`	404
`/laundry/track/X12`	none: the constraint fails	none	404
`/laundry/track/B00721`	none: five digits	none	404
`/css/site.css`	a static asset	the file	200

Two rows deserve a second look. The query string plays no part in routing: `/laundry/track?code=b0072` matches the same template as `/laundry/track`, and the action decides what to do with `code`. And `B9999` and `X12` both end in 404, but for different reasons: `B9999` *reaches* the action, which looks the code up and fails; `X12` never reaches any action. The visitor sees the same friendly page; the log and the amount of work differ.

4.5.4 The solution: let the application show its routes

Routing keeps all endpoints in a service called `EndpointDataSource`. The last part of **Listing 4.5** maps a development-only address, `/_routes`, whose handler receives that service by injection (a minimal-API handler's parameters are resolved from the container), keeps the route endpoints, sorts them by template and prints each template next to the endpoint's name. It is registered only when `app.Environment.IsDevelopment()`, because the route table is an internal detail that a public site should not reveal.

4.5.5 Running it

```
curl http://localhost:4040/_routes
```

```
{**path:file}                              Fallback {**path:file}
/                                          HTTP: GET /
/_routes                                   HTTP: GET /_routes
laundry                                    ShowcaseController.Home
laundry/collaboration                      ShowcaseController.Collaboration
laundry/company                            ShowcaseController.Company
laundry/error/{status:int}                 ShowcaseController.Error
laundry/profile                            ShowcaseController.Profile
laundry/services                           ShowcaseController.Services
laundry/track                              ShowcaseController.TrackForm
laundry/track/{code:regex(^B\d{{4}}$)}     ShowcaseController.Track
```

This is the real route table, in the words of the router itself. The routes of the individual files in `wwwroot/css` were filtered out by the handler; the first line is a catch-all fallback that `MapStaticAssets` adds in Development so that files not yet in its build-time manifest can still be served. Every template of **Table 4.5** is there, with the constraint exactly as written. The status codes from `verify.sh` (Sections 4.3.5 and 4.4.5, and `200 /css/site.css`) agree with all twelve predictions.

4.5.6 A tempting wrong solution

Suppose a later request asks for pages per outlet at `/laundry/track/{outlet}`, and a developer adds it next to the order route without thinking about the table. **Listing 4.19** shows the smallest version: two templates of exactly the same shape.

Listing 4.19 `Tempting/Ambiguous/Program.cs`: two attribute routes that match exactly the same addresses.

```
// Tempting wrong solution for Problem 4.3: two actions whose routes
// match exactly the same addresses.
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
var app = builder.Build();
app.MapControllers();
app.Run();

[Route("laundry")]
public class ShowcaseController : Controller
{
    [HttpGet("track/{code}")]
    public string Track(string code) => $"order {code}";

    [HttpGet("track/{outlet}")]          // WRONG: same shape as the one above
    public string Outlet(string outlet) => $"outlet {outlet}";
}
```

Again the analyzer warns at build time (ASP0023, “Route ‘track/{code}’ conflicts with another action route”) [6]. At run time, the program starts, and the first request fails with status 500:

```
500  /laundry/track/B0072
fail: Microsoft.AspNetCore.Server.Kestrel[13]
      Connection id "…", Request id "…": An unhandled exception was thrown by the
application.
      Microsoft.AspNetCore.Routing.Matching.AmbiguousMatchException: The request matched
multiple endpoints. Matches:

      ShowcaseController.Track (Ambiguous)
      ShowcaseController.Outlet (Ambiguous)
```

Precedence cannot help: both templates have a literal `track` and one unconstrained parameter, so neither is more specific. The router refuses to guess, which is the right decision, and the fix is to make the addresses distinguishable: a constraint on one of them (`{code:regex(^B\d{{4}}$)}` and `{outlet:regex(^O\d{{2}}$)}`), or a different literal (`/laundry/outlets/{id}`). The address table of step 2 would have shown the conflict before any code was written.

4.5.7 Check yourself

Variant. After the fix with two regular expressions, which endpoint handles `/laundry/track/O01`, `/laundry/track/B0072` and `/laundry/track/Z99`?

Answer. `O01` matches only the outlet template, `B0072` only the order template, and `Z99` neither, so it gets 404 without reaching any action. Constraints that exclude each other remove the ambiguity entirely.

4.6 Problem 4.4: A Response-Time Header

4.6.1 The problem

> **PROBLEM 4.4.** To watch the site's speed from outside, Dimas wants every response to carry a header `X-Response-Time` with the number of milliseconds the server spent on it, for example `X-Response-Time: 12 ms`. Write it once, for every page, file and error.

4.6.2 Understanding it by hand

"For every response" means middleware, placed first so that it wraps everything. The duration is only known after the rest of the pipeline has done its work, so the natural plan is Chapter 3's logging middleware: start a stopwatch, `await next(context)`, then set the header. That plan is wrong, and understanding why is the point of the problem.

An HTTP response is sent as a *status line*, then the *headers*, then the *body* [7]. As soon as the first byte of the body leaves the server, the status and the headers have been sent and can no longer change. A Razor view writes its HTML into the response body while it renders, so by the time `await next(context)` returns, the response has usually *started*: `context.Response.HasStarted` is true, and the headers are read-only [8].

4.6.3 Designing it on paper

We need to run code at the last moment *before* the headers go out, whenever that is. ASP.NET Core offers exactly that hook: `context.Response.OnStarting(callback)` registers a function that the server calls just before it sends the headers [8]. The middleware starts the stopwatch, registers a callback that reads the stopwatch and sets the header, and then calls the rest of the pipeline. The time measured is from the start of the middleware to the moment the headers are sent, which is the time until the first byte, the number Dimas wants.

Writing it as a class, instead of a lambda in `Program.cs`, keeps `Program.cs` readable and makes the middleware reusable. An ASP.NET Core middleware class needs a constructor that takes the `RequestDelegate` for the next component, and a method `InvokeAsync(HttpContext)`; `app.UseMiddleware<T>()` adds it to the pipeline.

4.6.4 The solution

Listing 4.20 `Middleware/ResponseTimeMiddleware.cs`: the header is set in an `OnStarting` callback, just before the headers are sent.

```
using System.Diagnostics;

namespace LaundryShowcase.Middleware;

// Adds "X-Response-Time: 12 ms" to every response.
public class ResponseTimeMiddleware(RequestDelegate next)
{
    public async Task InvokeAsync(HttpContext context)
    {
        var clock = Stopwatch.StartNew();
        // Headers must be set before the first byte of the body is sent,
        // so we ask to be called back at exactly that moment.
        context.Response.OnStarting(() =>
        {
            context.Response.Headers["X-Response-Time"] =
                $"{clock.ElapsedMilliseconds} ms";
            return Task.CompletedTask;
        });
        await next(context);
    }
}
```

In **Listing 4.5** it is the first line of the pipeline, `app.UseMiddleware<ResponseTimeMiddleware>()`, so every response, including files, redirects and error pages, passes through it.

4.6.5 Running it

```
curl -s -D - -o /dev/null http://localhost:4040/laundry
```

`-D -` prints the response headers, `-o /dev/null` throws the body away (on Windows PowerShell write `-o NUL`). The real answer:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
X-Response-Time: 2 ms
```

The number changes from request to request, and the first request after start-up is slower, because views are loaded on first use.

4.6.6 A tempting wrong solution

Listing 4.21 is the natural plan of Section 4.6.2, in a minimal program whose only endpoint returns a short text.

Listing 4.21 `Tempting/LateHeader/Program.cs`: the header is set after the rest of the pipeline has already started the response.

```
// Tempting wrong solution for Problem 4.4: setting a header after the
// rest of the pipeline has already started sending the response.
using System.Diagnostics;

var app = WebApplication.CreateBuilder(args).Build();

app.Use(async (context, next) =>
{
    var clock = Stopwatch.StartNew();
    await next(context);
    context.Response.Headers["X-Response-Time"] =         // WRONG: too late
        $"{clock.ElapsedMilliseconds} ms";
});
app.MapGet("/laundry", () => "Clean clothes, honest prices.");
app.Run();
```

There is no build warning; this mistake is invisible until it runs. The request succeeds from the client's side (status 200 and the text arrive, without the header), and the server logs an exception for every request:

```
200  /laundry  Clean clothes, honest prices.
fail: Microsoft.AspNetCore.Server.Kestrel[13]
      Connection id "…", Request id "…": An unhandled exception was thrown by the
application.
      System.InvalidOperationException: Headers are read-only, response has already
started.
         at Program.<>c.<<Main>$>b__0_0>d.MoveNext() in
…/Tempting/LateHeader/Program.cs:line 11
```

This is the worst kind of bug: the site appears to work, the header is silently missing, and the log fills with errors that nobody reads. The same rule applies to everything that goes into headers, such as cookies, redirects and status codes: decide them before the body starts, or register an `OnStarting` callback.

4.6.7 Check yourself

Variant. Would the lambda version of **Listing 4.21** work if the endpoint returned `Results.NoContent()` (status 204, no body)?

Answer. Yes, and that is the trap. A run with `app.MapGet("/laundry", () => Results.NoContent())` answered `204 No Content` with `X-Response-Time: 9 ms` and logged no error: with no body, nothing forces the headers out before the middleware returns, so they are still writable. That is exactly why the mistake survives testing on simple endpoints and fails on real pages. Code that is correct only when the body happens to be empty is not correct; `OnStarting` is.

4.7 Problem 4.5: Maintenance Mode That Keeps Its Style

4.7.1 The problem

> **PROBLEM 4.5.** While the site is being updated, the owner wants to switch it into maintenance mode without editing code: every page must answer with status 503 ("Service Unavailable"), a `Retry-After` header of one hour and a short page saying that the outlets are open as usual. The maintenance page must be styled with the site's own style sheet, so the style sheet must keep working while everything else is closed.

4.7.2 Understanding it by hand

Three requirements hide in the statement. *Without editing code* means configuration: a setting that can be changed in `appsettings.json` or, better, from outside with an environment variable. *Every page* means middleware again. And *the style sheet must keep working* means the middleware must tell a request for `/css/site.css` from a request for a page. The status 503 with `Retry-After` is the standard HTTP way to say "temporarily unavailable, come back later", which search engines understand and do not treat as the site disappearing [7].

4.7.3 Designing it on paper

The switch. `appsettings.json` gets a section `"Showcase": { "Maintenance": false }`. The middleware reads `app.Configuration.GetValue<bool>("Showcase:Maintenance")` on every request, so the setting's current value always applies. ASP.NET Core also reads environment variables into the same configuration, with a double underscore standing for the colon: `Showcase__Maintenance=true` overrides the file without touching it [9].

Telling files from pages. The middleware could compare the path with `/css/`, but a cleaner answer uses what routing already knows. After `UseRouting()`, the chosen endpoint is available as `context.GetEndpoint()`, and for a static asset its route pattern starts with `css/`. So the maintenance middleware is placed **after** `UseRouting()`, asks for the endpoint, and lets static assets through. Placed before routing, it would find no endpoint at all.

4.7.4 The solution

The maintenance middleware is the inline `app.Use(...)` after `app.UseRouting()` in **Listing 4.5**. When the switch is on and the endpoint is not a CSS asset, it sets the status and the `Retry-After` header, writes a small self-contained HTML page that links `/css/site.css`, and returns *without calling `next`*, which ends the request there: this is a **short-circuit**. The page is written before any body has started, so setting the status and the header is safe, unlike in Problem 4.4. **Listing 4.22** is the configuration file.

Listing 4.22 `appsettings.json` of LaundryShowcase: the maintenance switch, off by default.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Showcase": {
    "Maintenance": false
  }
}
```

4.7.5 Running it

The switch is turned on from the terminal, with no file edited:

```
Showcase__Maintenance=true dotnet run            # macOS, Linux
$env:Showcase__Maintenance="true"; dotnet run    # Windows PowerShell
```

```
503  /laundry
503  /laundry/profile
200  /css/site.css
HTTP/1.1 503 Service Unavailable
Retry-After: 3600
<body><main><h1>Back soon</h1>
```

Pages answer 503 with the header, and the style sheet still answers 200, so the maintenance page is styled, as **Figure 4.7** shows.

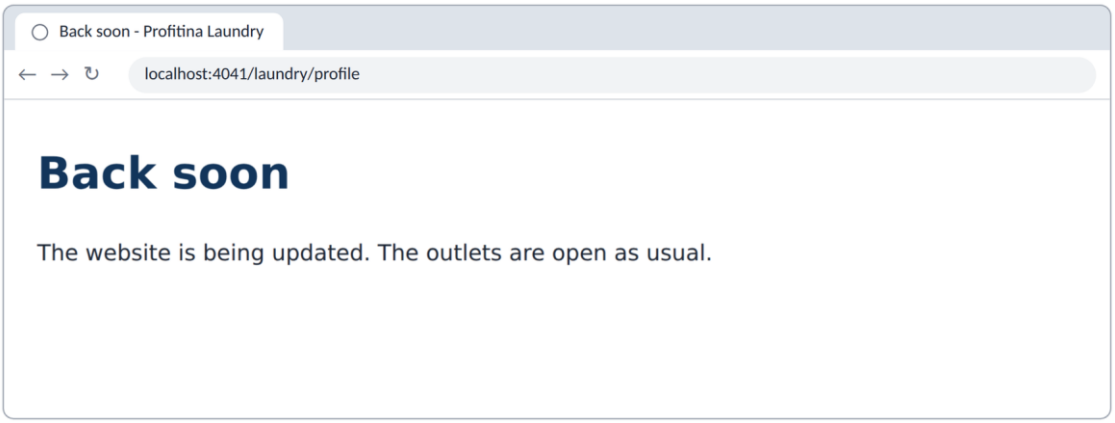


Figure 4.7 The maintenance page, answered with status 503 and styled by the site's own style sheet, which maintenance mode lets through.

4.7.6 A tempting wrong solution

The tempting mistake is to put the maintenance check where "checks" usually go, first in the pipeline. **Listing 4.23** does that, with everything else unchanged.

Listing 4.23 `Tempting/MaintenanceFirst/Program.cs`: the maintenance check before `UseRouting`, where no endpoint has been chosen yet.

```
// Tempting wrong solution for Problem 4.5: the maintenance check runs
// before UseRouting, so it cannot know that /css/site.css is a file.
var app = WebApplication.CreateBuilder(args).Build();

app.Use(async (context, next) =>                // WRONG: before UseRouting
{
    var endpoint = context.GetEndpoint() as RouteEndpoint;   // always null
    bool isAsset = endpoint?.RoutePattern.RawText?.StartsWith("css/") == true;
    if (!isAsset)
    {
        context.Response.StatusCode = StatusCodes.Status503ServiceUnavailable;
        await context.Response.WriteAsync("Back soon");
        return;
    }
    await next(context);
});
app.UseRouting();
app.MapStaticAssets();
app.MapGet("/laundry", () => "home");
app.Run();
```

```
503  /laundry  Back soon
503  /css/site.css  Back soon
```

Before routing, `GetEndpoint()` returns `null`, so every request, the style sheet included, looks like a page and is closed. In the browser, the maintenance page would appear unstyled, because its

style sheet is itself under maintenance. The code of the check is identical in both versions; only its position in the pipeline differs, which is Chapter 3's rule "order matters" in its purest form.

4.7.7 Check yourself

Variant. During maintenance, the owner wants customers to keep tracking their orders. What changes?

Answer. The test in the middleware: let through every endpoint whose pattern starts with `css/` *or* with `laundry/track`. Exercise 3 carries it out and shows the real answers.

4.8 Using It for Real

4.8.1 A showcase for your own business

LaundryShowcase was written so that most of a real adaptation happens in two files. Rewrite `content/site.json` for your own business: a startup's story, the promises it makes, its mission, its team, and what it offers and asks of investors, partners and future colleagues. Replace the files in `data/` with your own exports, or change `LaundryData.Facts()` to compute the numbers that matter for your business: customers served, harvests sold, patients seen. Change the prefix in `[Route("laundry")]`, and the colors in `site.css`. Because every link is generated from routes and every word comes from the content file, the controller and the views rarely need to change. The five problems of this chapter also make a good checklist for any small site: an address table with its failures, a constraint wherever an address has a format, privacy on every public page that looks something up, a header or two from middleware, and a maintenance switch that keeps the style sheet alive.

4.8.2 Tracking pages and personal data

Problem 4.2 kept personal data off a public page by never loading it. That is the simplest form of *data minimization*, a principle of Indonesia's Personal Data Protection Law (UU PDP), which requires personal data to be processed in a limited and specific way, for a clear purpose [4]. A real tracking page usually needs a second check as well, because four digits can be guessed: Profitina Laundry could print a short random verification word on the ticket and ask for it together with the code, and the tracking action would answer 404 unless both match. The pattern generalizes to clinics (a queue number plus a date of birth), schools (a registration number plus a PIN) and any other public look-up.

4.8.3 Profitina in practice

Profitina runs the same three mechanisms of this chapter in production. A maintenance switch is part of every staged release: during the few minutes a new version is installed on its Windows production host, public pages answer that the service will be back shortly, while the machinery that serves the site's assets keeps working. Middleware adds headers and enforces limits on every request, including the login rate limit that its UU PDP compliance work requires. And every public look-up is designed around what it must *not* show: the community's pages show a post, the public name of the member who wrote it and its replies, and nothing that the member did not choose to publish.

4.9 Common Mistakes, and How to Avoid Them

The five tempting wrong solutions are the five most instructive mistakes of this chapter; **Table 4.6** collects them with the symptom each one produces and how it was caught.

Table 4.6 The five tempting wrong solutions of this chapter, what each one does, and what catches it.

Mistake	Symptom	Caught by
Leading `/` on an action route	page answers at the wrong address, 404 at the right one	the address table (no warning)
Single braces in a `regex` constraint	startup exception, no request is answered	build warning ASP0017, then the crash
Two routes of the same shape	500 `AmbiguousMatchException` on the first matching request	build warning ASP0023, then the crash
Setting headers after `await next`	header silently missing, exception logged per request	reading the log; `curl -D -`
Endpoint check before `UseRouting`	every request closed, style sheet included	probing `/css/site.css` during maintenance

Three more mistakes appear often in projects like this one.

Hard-coding numbers that come from data. A showcase that says “almost 3,000 orders a year” in its text is wrong the day after it is written. Compute such numbers, as `LaundryData.Facts()` does, or leave them out.

Leaving development tools switched on in production. The route list of Problem 4.3 is useful to developers and useful to attackers. Map such endpoints only inside `if (app.Environment.IsDevelopment())`; a run in Production answered `/_routes` with 404 while the rest of the site worked normally.

Treating a friendly error page as success. An error view that forgets `Response.StatusCode = status` sends “Not found” with status 200, and search engines index it as a real page. Always keep the status.

4.10 Chapter Summary

- A routine for framework problems: read and restate, write the address table (with the addresses that must fail), sketch pages and data, write thin controllers with rules in services, run and probe with a browser and `curl`, look back for the tempting wrong solution.
- Problem 4.1 built a five-page site under one attribute-route prefix, with its words in a JSON file and its numbers computed from the data, both registered in the container and injected into controllers and views.
- Problem 4.2 added order tracking with a `regex` route constraint (braces doubled inside route templates, case ignored), a `GET` form that redirects to a bookmarkable address, 404 pages that keep their status, and no personal data on a public page.
- Problem 4.3 predicted a route table by hand and checked it against the router’s own list, taken from `EndpointDataSource` on a development-only route; ambiguous routes are refused with a 500.
- Problem 4.4 set a response header correctly with `Response.OnStarting`, because headers cannot change once the body has started.

- Problem 4.5 added a maintenance switch read from configuration and environment variables, placed after `UseRouting` so that it can let static assets through, answering 503 with `Retry-After`.

4.11 Exercises

1. Add a page `/laundry/outlets` to the showcase that lists the three outlets with address and phone, and add it to the menu.

2. During maintenance, let customers keep tracking their orders. Then request `/laundry/track/B9999` and `/laundry/track/X12` with maintenance on. What happens, and why? Fix it.

3. Start LaundryShowcase in the Production environment and request `/_routes`. What is the answer, and which line of **Listing 4.5** is responsible?

4. Add a second header, `X-Served-By: LaundryShowcase`, to every response, in the same middleware as `X-Response-Time`.

5. Write the address table for a new feature: one page per service at `/laundry/services/{id}`, where ids look like `S0004`. Is there a conflict with any existing route? Write the attribute.

6. The company page shows the average order. Which single place would you change to show the median instead, and why is the median sometimes the better number for a laundry?

4.12 Solutions to the Exercises

1. One action in `ShowcaseController`, one view and one menu line. The action:

```
    [HttpGet("outlets")]                          // /laundry/outlets
    public IActionResult Outlets() => View(data.Outlets);
```

The view `Views/Showcase/Outlets.cshtml`:

```
@model IReadOnlyList<Outlet>
@{
    ViewData["Title"] = "Outlets";
}
<h1>Our outlets</h1>
<table class="list">
    @foreach (var o in Model)
    {
        <tr><th>@o.Name</th><td>@o.Address<br />@o.Phone</td></tr>
    }
</table>
```

and in `_Layout.cshtml`, `<a class="@Here("Outlets")" asp-action="Outlets">Outlets</a>`. A run of the modified application answered `/laundry/outlets` with the three outlets: Rungkut, Jl. Kenari Raya No. 21, 031-0000-2101; Keputih, Jl. Flamboyan No. 8, 031-0000-2102; Wiyung, Jl. Cempaka Indah No. 45, 031-0000-2103.

2. In the maintenance middleware, replace the test with:

```
    string pattern = endpoint?.RoutePattern.RawText ?? "";
    bool open = pattern.StartsWith("css/") || pattern.StartsWith("laundry/track");
    if (closed && !open)
```

With maintenance on, a run gave `503` for `/laundry` and `/laundry/outlets`, and `200` for `/laundry/track`, `/laundry/track/B0072` and `/css/site.css`; `/laundry/track?code=B0072` gave its `302`. But `/laundry/track/B9999` and `/laundry/track/X12` both answered `503` with "Back soon" instead of the friendly 404. For `X12` no endpoint matches, so the pattern is empty and the request is closed. For `B9999` the tracking action runs and returns 404; the status-code middleware then re-executes the request as `/laundry/error/404`, and the re-executed request passes through the maintenance middleware again, where `laundry/error/{status:int}` is not open. The fix is to open the error route too, `|| pattern.StartsWith("laundry/error")`, and to treat "no endpoint" as a 404 rather than as a page: `if (closed && endpoint is not null && !open)`. A run with both changes answered `/laundry/track/B9999` and `/laundry/track/X12` with the friendly "Not found" page and status 404, `/laundry/track/B0072` with 200 and `/laundry` still with 503; an unknown address such as `/nothing` now gets 404 instead of 503, which is the honest answer. The lesson is Chapter 3's: re-execution runs the later part of the pipeline a second time, and every middleware in that part must be ready for it.

3. `404`, with the friendly "Not found" page, while `/laundry` answers `200` as usual; both from a run with `ASPNETCORE_ENVIRONMENT=Production dotnet run --no-launch-profile`. The route is mapped only inside `if (app.Environment.IsDevelopment())`, so in Production it does not exist.

4. Inside the `OnStarting` callback of **Listing 4.20**, add one line:

```
context.Response.Headers["X-Served-By"] = "LaundryShowcase";
```

Setting it directly before `await next(context)` would also work here, because nothing has been sent yet at that point; putting both headers in the callback keeps all of the middleware's header work in one place.

5.

Method	Address	Action	Expected
GET	`/laundry/services/S0004`	`ServiceDetails`	200, one service
GET	`/laundry/services/s0004`	`ServiceDetails`	200, case ignored
GET	`/laundry/services/S0099`	`ServiceDetails`	404, friendly page
GET	`/laundry/services/abc`	none	404, constraint fails

There is no conflict: the existing `laundry/services` has one segment fewer. The attribute is `[HttpGet(@"services/{id:regex(^S\d{{4}}$)}")]`, with the braces doubled as in Problem 4.2.

6. Only `LaundryData.Facts()`, which computes the number; the view shows whatever the model holds (and the label "Average order" in `Company.cshtml` would become "Typical order"). The median is the middle value when all orders are sorted by amount. A few very large orders, such as a hotel's weekly linen, pull the average up without changing what a typical household pays; the median is not moved by them, so it describes the typical customer better. Showing both is the most honest choice.

References

[1] G. Pólya, *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Princeton, NJ, USA: Princeton University Press, 1957.

[2] Microsoft, "JSON serialization and deserialization in .NET: overview," *.NET Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/dotnet/standard/serialization/system-text-json/overview

[3] Microsoft, “Routing in ASP.NET Core: route constraints and regular expressions,” *ASP.NET Core Documentation*, for ASP.NET Core 10.0, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/routing

[4] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara Republik Indonesia Tahun 2022 Nomor 196, Oct. 17, 2022. [Online]. Available: https://peraturan.bpk.go.id/Details/229798/uu-no-27-tahun-2022

[5] Microsoft, “Handle errors in ASP.NET Core,” *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/error-handling

[6] Microsoft, “ASP0023: Route conflict detected between controller actions,” *ASP.NET Core Diagnostics*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/diagnostics/asp0023

[7] R. Fielding, M. Nottingham, and J. Reschke, “HTTP Semantics,” RFC 9110, Internet Engineering Task Force, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[8] Microsoft, “Write custom ASP.NET Core middleware” and “ASP.NET Core middleware,” *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/middleware/write

[9] Microsoft, “Configuration in ASP.NET Core,” *ASP.NET Core Documentation*, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/configuration/

Chapter 5

ASP.NET Core MVC II: Entity Framework Core and Real CRUD

Chapter 3 gave Profitina Laundry a website that shows its services, its outlets and a price quote, all read from two CSV files. This chapter gives the shop a memory. By the end of it you will be able to explain what an object-relational mapper does and why Entity Framework Core (EF Core) is the one ASP.NET Core developers reach for; design a small relational database for customers, services, orders and payments; install EF Core 10 into the Chapter 3 project; describe the tables with C# classes and a `DbContext`; turn those classes into a real SQLite database with migrations; seed it from the Chapter 3 CSV files; write LINQ queries that EF Core translates into SQL; and build complete create, read, update and delete (CRUD) pages for customers, a price list for the owner, and an order counter that records orders, moves them through the laundry and takes payments. Along the way you will see the promise of Chapter 3 kept: the catalog controllers written there keep working, unchanged, on top of the database.

5.1 Where We Left Off: A Shop Without a Memory

Picture the counter of the Profitina Laundry outlet in Keputih at seven in the morning. A student brings a bag of clothes, the cashier weighs it on the digital scale (4.5 kg), writes the customer's name and phone number on a numbered carbon receipt, tears off the top copy for the customer and pushes the bag into a blue basket. By nine there are twenty baskets. By noon a customer is back asking whether her laundry is ready; the cashier flips through the carbon book, finds the page, and walks to the back to look. In the evening the owner adds up the money in the drawer and compares it, page by page, with the receipts. Some customers paid when they dropped off, some pay when they pick up, some paid half. When a page is torn or a sum is added wrongly, nobody knows who still owes what.

The website of Chapter 3 does not help with any of this. It shows the twelve services of the shop, from *Wash, dry & fold* at Rp 8.000 per kg to *Helmet cleaning*, the three outlets in Rungkut, Keputih and Wiyung, and a quote page that applies the shop's price rules. Everything it knows comes from `data/Service.csv` and `data/Outlet.csv`, read once when the program starts. It cannot remember a single customer, and when the owner changes a price she has to edit a file on the server and restart the program.

What the shop needs is the carbon book, the price board and the cash drawer inside the program: a place where every customer, every order and every payment is written down the moment it happens, can be found again in a second, and is still there tomorrow when the computer is switched back on. That place is a database, and this chapter connects the website to one.

The people it helps are easy to name. The cashier finds an order by the customer's name or phone number instead of flipping pages. The washer sees which baskets are queued and marks them finished. The customer hears the exact balance still owed. The owner sees at the counter page how many orders came in today and how much money was received, and changes a price once, in one place, for every page of the site.

Chapter 3 prepared for this moment deliberately. Its controllers never read the CSV files themselves; they asked an interface, `ILaundryCatalog`, for services and outlets, and the only class that knew about CSV files was `CsvCatalog`. Its comment made a promise: Chapter 5 swaps the CSV files behind it for a real database without touching a controller. Section 5.8 keeps that promise.

5.2 The Idea in Plain Words

5.2.1 Tables on one side, objects on the other

A relational database stores data in tables. A table has named columns and any number of rows; the `Customers` table might have the columns `Id`, `Name`, `Phone` and `Address`, and one row per customer. Rows of different tables are linked by values: an order row stores the `Id` of its customer in a column called `CustomerId`. The language for asking a relational database anything is SQL: `SELECT Name FROM Customers WHERE Phone = '081200000101'`.

A C# program, on the other hand, works with objects. A `Customer` object has properties `Name` and `Phone`, and an `Order` object can hold a reference to its `Customer` object directly, not just a number. The program would like to write `order.Customer.Name`, not a second query.

Chapter 2 introduced the tool that bridges the two worlds: an *object-relational mapper* (ORM). You describe your classes once, and the ORM writes the SQL, sends it with parameters, reads the rows that come back and turns them into objects. When you change an object and ask the ORM to save, it works out which SQL `INSERT`, `UPDATE` or `DELETE` statements are needed. EF Core is Microsoft's ORM for .NET, open source and maintained together with .NET itself; version 10 was released with .NET 10 in November 2025 and is supported, like .NET 10, as a long-term support release [1], [2].

> **DEFINITION.** In EF Core, an *entity* is a C# class whose objects are stored as rows of a table. A *DbContext* is the class that represents one working session with the database; it has a `DbSet<T>` property for every table, such as `DbSet<Customer> Customers`. A *migration* is a generated C# file that describes one change to the database structure (create a table, add a column) and can apply or undo it. The *change tracker* is the part of the DbContext that remembers which entities were loaded, added, changed or removed, so that `SaveChanges()` knows which SQL to send.

5.2.2 Two moments: design time and run time

It helps to separate two moments, shown in **Figure 5.1**. At *design time*, once for every change you make to the model, you write or edit the entity classes and the DbContext, run `dotnet ef migrations add` to let EF Core compare the model with the previous migration and write a new migration file, and run `dotnet ef database update` to apply it, which sends `CREATE TABLE` or `ALTER TABLE` statements to the database. At *run time*, on every request, a controller writes a LINQ query against a `DbSet`; EF Core translates it to one SQL statement with parameters, SQLite runs it, and the rows come back as objects. When the controller changes objects and calls `SaveChangesAsync()`, the change tracker turns the changes into `INSERT`, `UPDATE` and `DELETE` statements.

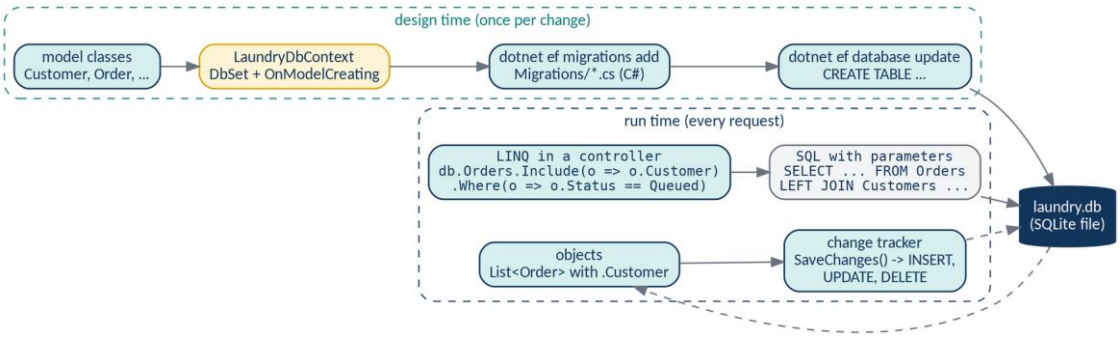


Figure 5.1 How EF Core connects the classes to the database: migrations at design time, LINQ translated into SQL and the change tracker at run time.

5.2.3 LINQ is the same language everywhere

LINQ (Language-Integrated Query) is the set of methods such as `Where`, `OrderBy`, `Select`, `GroupBy`, `Sum` and `Count` that every C# collection has. When you call them on a `List<Order>`, C# runs them in memory, one element at a time. When you call them on a `DbSet<Order>`, the same method calls are not run at all: EF Core reads them as an expression, translates the whole chain into SQL and lets the database do the work. The code looks the same; where it runs is different. That is why Section 5.4 starts without a database at all: the queries you learn on lists are the queries you will send to SQLite.

5.2.4 Why SQLite

SQLite is a complete SQL database in a single file, with no server to install. It is the default of the Laravel projects of Chapter 7 as well, and it is what a small shop with one computer at each counter really can use. EF Core supports many databases through *providers*; the SQLite provider is the NuGet package `Microsoft.EntityFrameworkCore.Sqlite`, and moving to SQL Server, PostgreSQL or MySQL later means changing that package and one line of `Program.cs` (Section 5.14).

5.3 Designing the Data

Before a line of code, the data. **Figure 5.2** shows the five tables of this chapter and how they are linked.

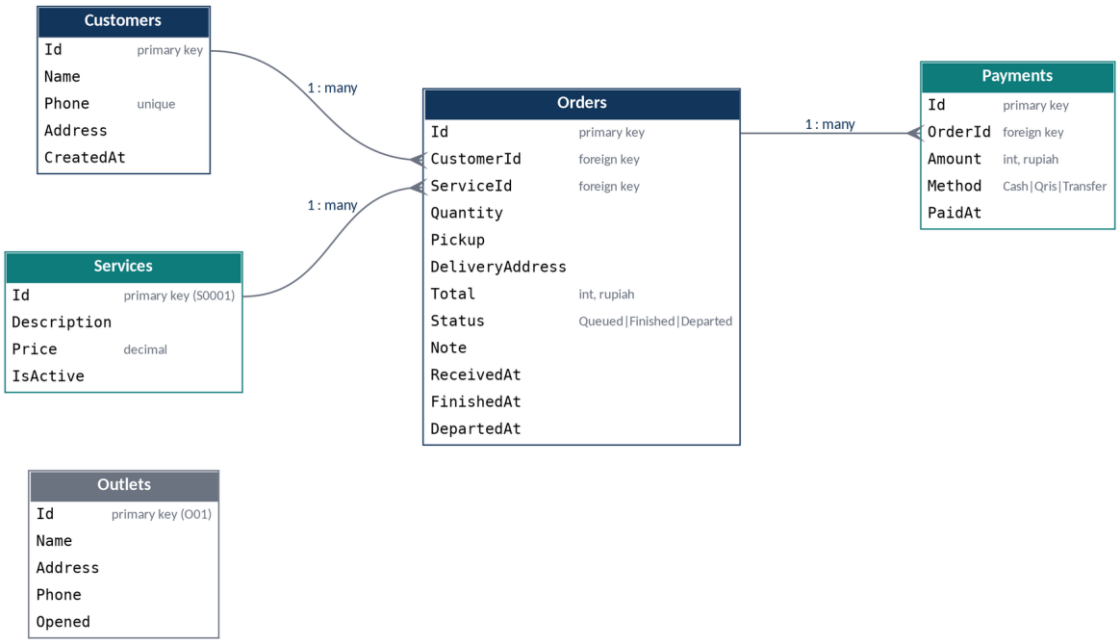


Figure 5.2 The Profitina Laundry database of this chapter: every order belongs to one customer and one service, and an order can have many payments.

Services and **Outlets** are the two CSV files of Chapter 3, now as tables. They keep the codes the shop already uses, `S0001` to `S0012` and `O01` to `O03`, as their primary keys: a *primary key* is the column whose value identifies a row, and a code that people already write on receipts is a good one. A new column `IsActive` lets the owner stop offering a service without deleting it, because old orders still point to it.

Customers get a number that the database hands out (`1`, `2`, `3`, …). The phone number is *unique*: two customers cannot share it, because the cashier looks people up by phone.

Orders are the heart of the shop. Each order stores the `Id` of its customer in `CustomerId` and the code of its service in `ServiceId`; such columns are *foreign keys*, and the database refuses an order whose `CustomerId` does not exist. An order also stores the quantity (kilograms, items or pairs), whether the shop picks up and delivers, the delivery address, a note, the total price, the status and three timestamps. The status follows the shop's canonical workflow: `Queued` when the bag is received, `Finished` when it is washed and folded, `Departed` when it has left the shop with its owner.

Payments belong to one order each. There can be several: a customer pays Rp 20.000 at drop-off and the rest at pickup. Each payment records the amount, the method (cash, QRIS or bank transfer) and the time.

Two design decisions deserve a sentence each. First, the order stores its `Total` at the moment it is created, instead of computing it again from the service's price every time. Prices change; last month's receipt must not. Second, what a customer has paid is not stored in the order at all; it is the sum of the order's payments, and the balance follows from it:

$$\text{Balance} = \text{Total} - \sum_{p \in \text{payments}} \text{Amount}_p \tag{5.1}$$

Storing it twice (a `Paid` column *and* the payments) would invite the two to disagree one day. Equation (5.1) is computed by the `Paid` and `Balance` properties of the `Order` class in Section 5.6.

Money is stored as whole rupiah in `int` columns. The rupiah has no coins below Rp 100 in daily use, so fractions never appear, and there is a practical reason as well: SQLite has no decimal type, and EF Core's SQLite provider can read, write and compare `decimal` values for equality but cannot sort them or add them up inside SQL [3]. With `int` columns, `SUM(Amount)` runs in the database. The service price keeps the `decimal` type it had in Chapter 3, because it is only read and written, never summed.

Table 5.1 lists how the pieces map onto each other.

Table 5.1 How the C# model and the SQLite tables correspond.

In C#	In the database	Example
class `Customer`	table `Customers`	
object `new Customer { Name = "Dewi Lestari" }`	one row	`3, 'Dewi Lestari', '081200000103', …`
property `Phone`	column `Phone TEXT NOT NULL`	`'081200000103'`
property `Orders` (a `List<Order>`)	nothing: the rows of `Orders` with `CustomerId = 3`	navigation property
property `CustomerId` in `Order`	foreign key column	`REFERENCES Customers(Id)`
enum `OrderStatus.Queued`	text `'Queued'`	value converter
`db.Customers.Add(c); SaveChanges()`	`INSERT INTO Customers …`	

5.4 LINQ First, Without a Database

The quickest way to trust LINQ is to watch it work on plain lists. `LinqDemo` is a console program that compiles the web app's own model classes (its project file includes `../LaundryMvc/Models/*.cs`) and runs the kinds of queries the controllers of this chapter will send to the database: a search, a filter by status, revenue grouped per service, and balances. **Listing 5.1** is the whole program.

Listing 5.1 `LinqDemo/Program.cs`: the chapter's queries on in-memory lists, with the web app's own model classes.

```
using LaundryMvc.Models;

// The same LINQ the controllers send to EF Core, run here on plain lists.
var fold = new Service { Id = "S0001", Price = 8000,
                         Description = "Wash, dry & fold (per kg)" };
var express = new Service { Id = "S0002", Price = 12000,
                            Description = "Express wash & fold, 1 day (per kg)" };
var cover = new Service { Id = "S0008", Price = 35000,
                          Description = "Bed cover, any size (per item)" };
var sari = new Customer { Id = 1, Name = "Sari Wulandari",
                          Phone = "081200000101" };
var budi = new Customer { Id = 2, Name = "Budi Santoso",
                          Phone = "081200000102" };
var dewi = new Customer { Id = 3, Name = "Dewi Lestari",
                          Phone = "081200000103" };

var orders = new List<Order>
{
    new() { Id = 1, Customer = sari, Service = fold, Quantity = 4.5m,
            Total = 36000, Status = OrderStatus.Finished,
            Payments = [ new() { Amount = 36000,
                                 Method = PaymentMethod.Qris } ] },
    new() { Id = 2, Customer = budi, Service = express, Quantity = 3m,
            Total = 36000,
            Payments = [ new() { Amount = 20000,
                                 Method = PaymentMethod.Cash } ] },
    new() { Id = 3, Customer = budi, Service = cover, Quantity = 2m,
            Pickup = true, Total = 80000 },
    new() { Id = 4, Customer = dewi, Service = fold, Quantity = 6m,
            Total = 48000, Status = OrderStatus.Departed,
            Payments = [ new() { Amount = 48000,
                                 Method = PaymentMethod.Transfer } ] },
};
var customers = new List<Customer> { sari, budi, dewi };

Console.WriteLine("1. Customers whose name or phone contains \"an\":");
var search = "an";
foreach (var c in customers.Where(c => c.Name.Contains(search)
                                    || c.Phone.Contains(search))
                           .OrderBy(c => c.Name))
{
    Console.WriteLine($"   {c.Name,-16} {c.Phone}");
}
```

Listing 5.1 (Continued) *LinqDemo/Program.cs: the chapter's queries on in-memory lists, with the web app's own model classes.*

```
Console.WriteLine("2. Orders still in the queue:");
foreach (var o in orders.Where(o => o.Status == OrderStatus.Queued))
{
    Console.WriteLine($"   #{o.Id} {o.Customer!.Name,-16} {o.Service!.Name}");
}

Console.WriteLine("3. Revenue per service (sum of order totals):");
var perService = orders
    .GroupBy(o => o.Service!.Name)
    .Select(g => new { Service = g.Key, Orders = g.Count(),
                       Revenue = g.Sum(o => o.Total) })
    .OrderByDescending(r => r.Revenue);
foreach (var r in perService)
{
    Console.WriteLine($"   {r.Service,-26} {r.Orders} order(s) "
                      + $"{Money.Rp(r.Revenue),10}");
}

Console.WriteLine("4. Orders with money still owed:");
foreach (var o in orders.Where(o => o.Balance > 0))
{
    Console.WriteLine($"   #{o.Id} total {Money.Rp(o.Total),9}"
                      + $"  paid {Money.Rp(o.Paid),9}"
                      + $"  owes {Money.Rp(o.Balance),9}");
}

Console.WriteLine($"5. Total owed to the shop: "
                  + Money.Rp(orders.Sum(o => o.Balance)));
```

Run it with `dotnet run` in the `LinqDemo` folder. This is its real output:

```
1. Customers whose name or phone contains "an":
   Budi Santoso     081200000102
   Sari Wulandari   081200000101
2. Orders still in the queue:
   #2 Budi Santoso     Express wash & fold, 1 day
   #3 Budi Santoso     Bed cover, any size
3. Revenue per service (sum of order totals):
   Wash, dry & fold           2 order(s)  Rp 84.000
   Bed cover, any size        1 order(s)  Rp 80.000
   Express wash & fold, 1 day 1 order(s)  Rp 36.000
4. Orders with money still owed:
   #2 total Rp 36.000  paid Rp 20.000  owes Rp 16.000
   #3 total Rp 80.000  paid      Rp 0  owes Rp 80.000
5. Total owed to the shop: Rp 96.000
```

Read the program slowly, because every idea returns later. `Where(c => c.Name.Contains(search) || c.Phone.Contains(search))` keeps the customers for which the lambda returns `true`; the lambda `c => …` is a small function written in place, with `c` standing for one customer. `OrderBy(c => c.Name)` sorts. `GroupBy(o => o.Service!.Name)` makes one group per service name, and `Select(g => new { … g.Count(), g.Sum(o => o.Total) })` turns each group into a small summary object. Query 4 uses `o.Balance`, a property computed from the order's payments, exactly as Equation (5.1) says: order 2 cost Rp 36.000, Rp 20.000 was paid, Rp 16.000 is still owed.

Notice what the output shows about the price rules of Chapter 3 as well: the bed-cover order (two items with pickup and delivery) costs Rp 80.000, which is 2 × Rp 35.000 plus the Rp 10.000 pickup fee, because the subtotal is below the Rp 100.000 at which pickup becomes free.

5.5 Installing Entity Framework Core

EF Core is not part of the ASP.NET Core shared framework; it is distributed as NuGet packages. Open a terminal in the `LaundryMvc` folder (in VS Code: Terminal → New Terminal) and add two packages and one tool:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite --version 10.0.12
dotnet add package Microsoft.EntityFrameworkCore.Design --version 10.0.12
dotnet tool install --global dotnet-ef --version 10.0.12
```

The first package is the SQLite provider, which brings EF Core itself with it. The second contains the design-time services that `dotnet ef` needs to read your model and write migrations; it is marked `PrivateAssets=all` so that it is not shipped with the published app. The third line installs the `dotnet ef` command once per computer. Version 10.0.12, released on September 8, 2026, was the latest patch of EF Core 10 when this chapter was written [4]; any later 10.0.x patch works the same way, and `dotnet tool update --global dotnet-ef` keeps the tool current. **Listing 5.2** shows the project file afterwards.

Listing 5.2 `LaundryMvc.csproj` after adding the two EF Core packages.

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

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

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite"
                      Version="10.0.12" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
                      Version="10.0.12">
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>

</Project>
```

The connection string, the text that tells the provider which database to open, goes into `appsettings.json` (**Listing 5.3**). `Data Source=laundry.db` means a file called `laundry.db` in the folder from which the app runs. The extra log category makes EF Core print every SQL statement it sends, which is the best way to learn what your LINQ becomes.

Listing 5.3 `appsettings.json`: the connection string and the SQL log.

```
{
  "ConnectionStrings": {
    "Laundry": "Data Source=laundry.db"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

5.6 The Model Classes

5.6.1 Service: from record to entity

In Chapter 3, `Service` was a positional record, `record Service(string Id, string Description, decimal Price)`. A record is perfect for data that is read once and never changes. An entity that the owner edits is different: EF Core loads it, the controller changes its `Price`, and the change tracker notices the change and sends an `UPDATE`. For that, the properties need setters. **Listing 5.4** is the new class. The computed properties `Name`, `PerKg` and `Unit` are copied unchanged from Chapter 3, so every view written there still works; because they have no setter, EF Core does not map them to columns.

Listing 5.4 `Models/Service.cs`: the Chapter 3 record turned into an entity class with setters and an `IsActive` flag.

```
namespace LaundryMvc.Models;

// Chapter 3 read services from a CSV file as read-only records. From
// Chapter 5 on they are rows the owner edits, so the properties have setters.
public class Service
{
    public string Id { get; set; } = "";           // S0001, S0002, ...
    public string Description { get; set; } = "";  // "Ironing only (per kg)"
    public decimal Price { get; set; }             // rupiah per kg/item/pair
    public bool IsActive { get; set; } = true;

    // Worked out from Description, as in Chapter 3; not stored.
    public string Name => Description.Contains(" (")
        ? Description[..Description.LastIndexOf(" (")] : Description;

    public bool PerKg => Description.EndsWith("(per kg)");

    public string Unit => PerKg ? "kg"
        : Description.EndsWith("(per pair)") ? "pair" : "item";
}
```

The only other Chapter 3 file that has to change is `CsvCatalog`, which created services with `new Service(id, description, price)`; it now uses an object initializer, `new Service { Id = f[0], Description = f[1], Price = … }`.

5.6.2 Customer, Order and Payment

Listing 5.5, **Listing 5.6** and **Listing 5.7** are the three new entities. EF Core recognizes a lot by *convention*, so the classes stay plain: a property called `Id` is the primary key, and because it is an `int`, SQLite numbers new rows automatically; a property `CustomerId` next to a property `Customer` of type `Customer` is a foreign key with its navigation property; a `List<Order>` in `Customer` is the other side of the same relationship. A non-nullable `string` becomes a `NOT NULL` column, and a `string?` a column that may be empty.

Listing 5.5 `Models/Customer.cs`: a customer and the navigation property to the customer's orders.

```
namespace LaundryMvc.Models;

public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public string Phone { get; set; } = "";
    public string? Address { get; set; }
    public DateTime CreatedAt { get; set; } = DateTime.Now;

    // Navigation property: every order this customer has brought in.
    public List<Order> Orders { get; set; } = new();
}
```

Listing 5.6 `Models/Order.cs`: an order with its foreign keys, status, timestamps and payments; `Paid` and `Balance` implement the balance formula.

```
namespace LaundryMvc.Models;

public enum OrderStatus { Queued, Finished, Departed }

public class Order
{
    public int Id { get; set; }

    public int CustomerId { get; set; }          // foreign key
    public Customer? Customer { get; set; }      // navigation property

    public string ServiceId { get; set; } = "";  // S0001, ...
    public Service? Service { get; set; }

    public decimal Quantity { get; set; }        // kg, items or pairs
    public bool Pickup { get; set; }             // pickup and delivery
    public string? DeliveryAddress { get; set; }
    public int Total { get; set; }               // rupiah, fixed at creation
    public OrderStatus Status { get; set; } = OrderStatus.Queued;
    public string? Note { get; set; }

    public DateTime ReceivedAt { get; set; } = DateTime.Now;
    public DateTime? FinishedAt { get; set; }
    public DateTime? DepartedAt { get; set; }

    public List<Payment> Payments { get; set; } = new();

    // Worked out from the loaded payments; not stored.
    public int Paid => Payments.Sum(p => p.Amount);
    public int Balance => Total - Paid;
}
```

Listing 5.7 `Models/Payment.cs`: one payment towards one order.

```
namespace LaundryMvc.Models;

public enum PaymentMethod { Cash, Qris, Transfer }

public class Payment
{
    public int Id { get; set; }
    public int OrderId { get; set; }
    public Order? Order { get; set; }
    public int Amount { get; set; }              // rupiah
    public PaymentMethod Method { get; set; }
    public DateTime PaidAt { get; set; } = DateTime.Now;
}
```

The `Customer?` and `Service?` navigation properties are nullable because they are only filled when you ask EF Core to load them (Section 5.10.2). `Paid` and `Balance` add up whatever payments are in the `Payments` list; if the payments were not loaded, the list is empty and the balance looks like the whole total. Keep that in mind; Chapter 8 turns it into a problem to solve.

5.7 The DbContext

Listing 5.8 is the DbContext of the laundry. It has one `DbSet` per table and one method, `OnModelCreating`, in which the *fluent API* says what the conventions cannot guess.

Listing 5.8 `Data/LaundryDbContext.cs`: the five tables and the rules the conventions cannot guess.

```
using LaundryMvc.Models;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Data;

public class LaundryDbContext(DbContextOptions<LaundryDbContext> options)
    : DbContext(options)
{
    public DbSet<Service> Services => Set<Service>();
    public DbSet<Outlet> Outlets => Set<Outlet>();
    public DbSet<Customer> Customers => Set<Customer>();
    public DbSet<Order> Orders => Set<Order>();
    public DbSet<Payment> Payments => Set<Payment>();

    protected override void OnModelCreating(ModelBuilder model)
    {
        model.Entity<Service>(e =>
        {
            e.Property(s => s.Id).HasMaxLength(5);
            e.Property(s => s.Description).HasMaxLength(60);
            e.Property(s => s.Price).HasPrecision(10, 2);
        });

        model.Entity<Outlet>(e =>
        {
            e.Property(o => o.Id).HasMaxLength(3);
            e.Property(o => o.Name).HasMaxLength(60);
            e.Property(o => o.Address).HasMaxLength(120);
            e.Property(o => o.Phone).HasMaxLength(20);
        });

        model.Entity<Customer>(e =>
        {
            e.Property(c => c.Name).HasMaxLength(80);
            e.Property(c => c.Phone).HasMaxLength(20);
            e.Property(c => c.Address).HasMaxLength(200);
            e.HasIndex(c => c.Phone).IsUnique();
        });
```

Listing 5.8 (Continued) *Data/LaundryDbContext.cs: the five tables and the rules the conventions cannot guess.*

```
        model.Entity<Order>(e =>
        {
            e.Property(o => o.Quantity).HasPrecision(6, 2);
            e.Property(o => o.Status).HasConversion<string>().HasMaxLength(12);
            e.Property(o => o.DeliveryAddress).HasMaxLength(200);
            e.Property(o => o.Note).HasMaxLength(200);
            e.HasIndex(o => o.Status);
            e.HasOne(o => o.Customer).WithMany(c => c.Orders)
             .HasForeignKey(o => o.CustomerId)
             .OnDelete(DeleteBehavior.Restrict);
            e.HasOne(o => o.Service).WithMany()
             .HasForeignKey(o => o.ServiceId)
             .OnDelete(DeleteBehavior.Restrict);
        });

        model.Entity<Payment>(e =>
        {
            e.Property(p => p.Method).HasConversion<string>().HasMaxLength(10);
            e.HasOne(p => p.Order).WithMany(o => o.Payments)
             .HasForeignKey(p => p.OrderId)
             .OnDelete(DeleteBehavior.Cascade);
        });
    }
}
```

Go through `OnModelCreating` one entity at a time.

- `HasMaxLength` gives every text column a length. SQLite ignores lengths, but SQL Server and MySQL do not, and the length documents the design.
- `HasIndex(c => c.Phone).IsUnique()` creates a unique index: the database itself refuses a second customer with the same phone number, even if a bug in a controller forgets to check.
- `HasPrecision(6, 2)` says that a quantity has at most six digits, two of them after the decimal point, so 4.5 kg and 1250.75 kg fit.
- `HasConversion<string>()` stores the enum `OrderStatus` as the text `'Queued'`, `'Finished'` or `'Departed'` instead of the numbers 0, 1 and 2. A number would break silently if someone ever reordered the enum; the text is also readable in any database viewer.
- `HasOne(o => o.Customer).WithMany(c => c.Orders).HasForeignKey(o => o.CustomerId)` spells out the relationship that the conventions would also find, and `OnDelete(DeleteBehavior.Restrict)` changes one important default. For a required relationship EF Core would otherwise use *cascade delete*: deleting a customer would delete all their orders, and with them the record of money received. `Restrict` makes the database refuse to delete a customer who has orders. Payments, on the other hand, use `Cascade`: they mean nothing without their order.

The primary constructor `LaundryDbContext(DbContextOptions<LaundryDbContext> options)` receives its options (which provider, which connection string) from dependency injection, the mechanism of Chapter 2. The DbContext never decides for itself which database to open, so the same class works with SQLite in development and with another database in production.

5.8 Seeding the Database and Keeping Chapter 3's Promise

5.8.1 Seed data

An empty database is useless for learning. EF Core 9 introduced `UseSeeding` and `UseAsyncSeeding`, two callbacks that run whenever the database is created or migrated, including by `dotnet ef database update` [5]. `SeedData` (**Listing 5.9**) fills an empty database in two steps: services and outlets from the very CSV files of Chapter 3, read by the unchanged `CsvCatalog`, then three sample customers with orders and payments. Each step first checks whether its table is empty, because the callbacks run on every migration, not just the first.

Listing 5.9 `Data/SeedData.cs`: services and outlets from the Chapter 3 CSV files, then sample customers, orders and payments.

```
using LaundryMvc.Models;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Data;

// Fills an empty database: services and outlets from the CSV files of
// Chapter 3, then a few customers with orders and payments to look at.
public static class SeedData
{
    public static void Apply(LaundryDbContext db, string dataFolder)
    {
        if (!db.Services.Any())
        {
            var csv = new CsvCatalog(dataFolder);
            db.Services.AddRange(csv.Services());
            db.Outlets.AddRange(csv.Outlets());
            db.SaveChanges();
        }
        if (!db.Customers.Any())
        {
            db.Customers.AddRange(SampleCustomers());
            db.SaveChanges();
        }
    }

    public static async Task ApplyAsync(LaundryDbContext db, string dataFolder,
                                        CancellationToken ct)
    {
        if (!await db.Services.AnyAsync(ct))
        {
            var csv = new CsvCatalog(dataFolder);
            db.Services.AddRange(csv.Services());
            db.Outlets.AddRange(csv.Outlets());
            await db.SaveChangesAsync(ct);
        }
        if (!await db.Customers.AnyAsync(ct))
        {
            db.Customers.AddRange(SampleCustomers());
            await db.SaveChangesAsync(ct);
        }
    }
```

Listing 5.9 (Continued) *Data/SeedData.cs: services and outlets from the Chapter 3 CSV files, then sample customers, orders and payments.*

```
    static List<Customer> SampleCustomers()
    {
        var today = DateTime.Today;
        return
        [
            new Customer
            {
                Name = "Sari Wulandari", Phone = "081200000101",
                Address = "Jl. Kertajaya 12, Surabaya",
                Orders =
                [
                    new Order
                    {
                        ServiceId = "S0001", Quantity = 4.5m, Total = 36000,
                        Status = OrderStatus.Finished,
                        ReceivedAt = today.AddDays(-1).AddHours(8),
                        FinishedAt = today.AddHours(9),
                        Payments = [ new Payment { Amount = 36000,
                            Method = PaymentMethod.Qris,
                            PaidAt = today.AddDays(-1).AddHours(8) } ]
                    }
                ]
            },
            new Customer
            {
                Name = "Budi Santoso", Phone = "081200000102",
                Orders =
                [
                    new Order
                    {
                        ServiceId = "S0002", Quantity = 3m, Total = 36000,
                        ReceivedAt = today.AddHours(7.5),
                        Payments = [ new Payment { Amount = 20000,
                            Method = PaymentMethod.Cash,
                            PaidAt = today.AddHours(7.5) } ]
                    },
                    new Order
                    {
                        ServiceId = "S0008", Quantity = 2m, Pickup = true,
                        DeliveryAddress = "Jl. Mulyosari 30, Surabaya",
                        Total = 80000, ReceivedAt = today.AddHours(10)
                    }
                ]
            },
            new Customer
            {
```

Listing 5.9 (Continued) *Data/SeedData.cs: services and outlets from the Chapter 3 CSV files, then sample customers, orders and payments.*

```
                Name = "Dewi Lestari", Phone = "081200000103",
                Address = "Jl. Manyar 7, Surabaya"
            }
        ];
    }
}
```

Note how a whole graph is added with one `AddRange`: each `Customer` object carries its `Orders`, and each order its `Payments`. EF Core inserts the customer, reads back the new `Id`, fills `CustomerId` in the orders, inserts them, and does the same for the payments, all inside one transaction.

5.8.2 The catalog on the database

Listing 5.10 is the class Chapter 3 promised. It implements the same interface as `CsvCatalog`, answering the same four questions from the database instead of from lists read at start-up. `Services()` now shows only the services that are offered; `FindService` upper-cases the code so that `/services/s0004` keeps working as it did with the CSV files.

Listing 5.10 `Data/EfCatalog.cs`: the Chapter 3 catalog interface answered by EF Core.

```
using LaundryMvc.Models;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Data;

// The same questions CsvCatalog answered in Chapter 3, now answered by the
// database. The controllers of Chapter 3 do not notice the difference.
public class EfCatalog(LaundryDbContext db) : ILaundryCatalog
{
    public IReadOnlyList<Service> Services() =>
        db.Services.AsNoTracking().Where(s => s.IsActive)
                   .OrderBy(s => s.Id).ToList();

    public Service? FindService(string id) =>
        db.Services.AsNoTracking()
                   .FirstOrDefault(s => s.Id == id.ToUpperInvariant());

    public IReadOnlyList<Outlet> Outlets() =>
        db.Outlets.AsNoTracking().OrderBy(o => o.Id).ToList();

    public Outlet? FindOutlet(string id) =>
        db.Outlets.AsNoTracking()
                  .FirstOrDefault(o => o.Id == id.ToUpperInvariant());
}
```

The interface of Chapter 3 is synchronous (`Services()` returns a list, not a `Task`), so `EfCatalog` uses the synchronous `ToList()` and `FirstOrDefault()`. That is fine for a catalog of twelve rows; the new controllers below use the asynchronous versions, which free the server thread while the database works.

5.8.3 Registering everything in Program.cs

Listing 5.11 is `Program.cs` after the change. Compared with Chapter 3, two things are new: `AddDbContext` registers `LaundryDbContext` with the SQLite provider, the connection string and the seeding callbacks, and `AddScoped<ILaundryCatalog, EfCatalog>()` replaces the singleton `CsvCatalog`.

Listing 5.11 `Program.cs`: the Chapter 3 pipeline with the DbContext registered and the catalog swapped.

```
using System.Diagnostics;
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// 1. Services: everything the application needs, registered once.
builder.Services.AddControllersWithViews();

// Chapter 5: one LaundryDbContext per request, on the SQLite file named in
// appsettings.json. The CSV files of Chapter 3 now only seed the database.
var dataFolder = Path.Combine(builder.Environment.ContentRootPath, "data");
builder.Services.AddDbContext<LaundryDbContext>(options =>
    options.UseSqlite(builder.Configuration.GetConnectionString("Laundry"))
           .UseSeeding((db, _) =>
               SeedData.Apply((LaundryDbContext)db, dataFolder))
           .UseAsyncSeeding((db, _, ct) =>
               SeedData.ApplyAsync((LaundryDbContext)db, dataFolder, ct)));
builder.Services.AddScoped<ILaundryCatalog, EfCatalog>();
builder.Services.AddSingleton<PriceCalculator>();

var app = builder.Build();

// 2. Pipeline: what happens to every request, in this order.
app.Use(async (context, next) =>
{
    var clock = Stopwatch.StartNew();
    await next(context);                 // everything later runs in here
    app.Logger.LogInformation("{Method} {Path}{Query} -> {Status} in {Ms} ms",
        context.Request.Method, context.Request.Path,
        context.Request.QueryString, context.Response.StatusCode,
        clock.ElapsedMilliseconds);
});

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/error/500");
}
app.UseStatusCodePagesWithReExecute("/error/{0}");

app.UseRouting();
```

Listing 5.11 (Continued) *Program.cs: the Chapter 3 pipeline with the DbContext registered and the catalog swapped.*

```
app.MapStaticAssets();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}")
    .WithStaticAssets();

app.Run();
```

Why *scoped*? `AddDbContext` registers the DbContext with a scoped lifetime: one instance per HTTP request, created when the request needs it and disposed when the request ends. A DbContext is not thread-safe and its change tracker grows with every entity it loads, so a single instance shared by all requests (a singleton) would be both wrong and slow. `EfCatalog` holds a DbContext, so it must be scoped too; registering it as a singleton would fail at start-up with an error about consuming a scoped service from a singleton, which is ASP.NET Core protecting you from exactly that mistake.

Now look at `HomeController`, `ServicesController` and `OutletsController` from Chapter 3. Not one line of them changed. They still ask `ILaundryCatalog` for services and outlets, and dependency injection now hands them an `EfCatalog`. The price quote page still uses the `PriceCalculator` of Chapter 3, and so will the order counter below. This is the practical meaning of "depend on an interface, not on a class".

5.9 Migrations: From Classes to Tables

With the model and the DbContext in place, create the first migration and apply it:

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

The first command builds the project, reads the model through `LaundryDbContext`, and writes three files into a new `Migrations` folder: `<timestamp>_InitialCreate.cs` with an `Up` method (create the five tables, their keys, foreign keys and indexes) and a `Down` method (drop them again), a designer file with metadata, and `LaundryDbContextModelSnapshot.cs`, a complete description of the model as it is now [6]. The second command opens `laundry.db` (creating the file if it does not exist), applies every migration not yet applied, records each applied migration in a table called `__EFMigrationsHistory`, and then runs the seeding callbacks.

Open `Migrations/<timestamp>_InitialCreate.cs` and read it; it is ordinary C#. You will find `migrationBuilder.CreateTable(name: "Customers", …)` with one `table.Column<…>` per property, `table.PrimaryKey("PK_Customers", x => x.Id)`, `table.ForeignKey(…, onDelete: ReferentialAction.Restrict)` for the order's customer, and `migrationBuilder.CreateIndex(name: "IX_Customers_Phone", …, unique: true)`. Every rule of Section 5.7 is there, written as a step that can be applied and undone.

The snapshot is what makes the *next* migration small. Suppose the owner later wants to record which rack a finished bag waits on. You add `public string? Rack { get; set; }` to `Order` and run `dotnet ef migrations add AddRackToOrders`. EF Core compares the model with the snapshot, sees exactly one difference, and writes a migration whose `Up` adds one column and whose `Down` removes it. `dotnet ef database update` applies only that migration, because the history table shows that `InitialCreate` has already run. Chapter 8 does exactly this in Problem 8.7.

> **KEY INSIGHT:** Migrations are to the database what Git commits are to the code. They are small, ordered, reviewable, and stored in the repository, so that every computer (a teammate's laptop, the test server, the shop's server) can be brought to exactly the same structure with one command. Never change a table by hand in a database viewer; change the model, add a migration, and let every copy of the database follow.

Two more commands are worth knowing. `dotnet ef migrations remove` deletes the last migration if it has not been applied yet, which is the normal way to fix a migration you just created by mistake. `dotnet ef migrations script` prints the SQL of all migrations, useful when a database administrator wants to read what will happen before it happens.

5.10 Real CRUD for Customers

CRUD means the four things every data page does: **C**reate, **R**ead, **U**pdate and **D**elete. **Table 5.2** lists the eight actions of `CustomersController` with their addresses and the SQL each one causes.

Table 5.2 The actions of `CustomersController`, their addresses, and the SQL they cause.

Action	Address and method	What EF Core sends
`Index(search)`	`GET /Customers?search=Sari`	`SELECT … WHERE instr(Name, @search) > 0 OR …`
`Details(id)`	`GET /Customers/Details/3`	one `SELECT` with a `LEFT JOIN` to `Orders` and `Services`
`Create()`	`GET /Customers/Create`	nothing: an empty form
`Create(customer)`	`POST /Customers/Create`	a phone check, then `INSERT INTO Customers …`
`Edit(id)`	`GET /Customers/Edit/3`	`SELECT … WHERE Id = @id`
`Edit(id, input)`	`POST /Customers/Edit/3`	`SELECT`, phone check, `UPDATE Customers SET …`
`Delete(id)`	`GET /Customers/Delete/3`	`SELECT` with orders, for the confirmation page
`DeleteConfirmed(id)`	`POST /Customers/Delete/3`	`SELECT`, an orders check, `DELETE FROM Customers …`

Listing 5.12 is the whole controller. Read it once from top to bottom before the explanation.

Listing 5.12 `Controllers/CustomersController.cs`: complete CRUD for customers with EF Core.

```
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Controllers;

public class CustomersController(LaundryDbContext db) : Controller
{
    // GET /Customers?search=Sari
    public async Task<IActionResult> Index(string? search)
    {
        var query = db.Customers.AsNoTracking();
        if (!string.IsNullOrWhiteSpace(search))
        {
            query = query.Where(c => c.Name.Contains(search)
                                  || c.Phone.Contains(search));
        }
        ViewData["Search"] = search;
        return View(await query.OrderBy(c => c.Name).ToListAsync());
    }

    // GET /Customers/Details/3
    public async Task<IActionResult> Details(int id)
    {
        var customer = await db.Customers
            .Include(c => c.Orders).ThenInclude(o => o.Service)
            .AsNoTracking()
            .FirstOrDefaultAsync(c => c.Id == id);
        return customer is null ? NotFound() : View(customer);
    }

    public IActionResult Create() => View(new Customer());

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(
        [Bind("Name,Phone,Address")] Customer customer)
    {
        await CheckPhoneIsFreeAsync(customer.Phone, customer.Id);
        if (!ModelState.IsValid) return View(customer);
```

Listing 5.12 (Continued) *Controllers/CustomersController.cs: complete CRUD for customers with EF Core.*

```
        db.Customers.Add(customer);
        await db.SaveChangesAsync();          // INSERT, then customer.Id is set
        TempData["Message"] = $"Customer {customer.Name} added.";
        return RedirectToAction(nameof(Details), new { id = customer.Id });
    }

    public async Task<IActionResult> Edit(int id)
    {
        var customer = await db.Customers.FindAsync(id);
        return customer is null ? NotFound() : View(customer);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Edit(int id,
        [Bind("Id,Name,Phone,Address")] Customer input)
    {
        if (id != input.Id) return BadRequest();
        var customer = await db.Customers.FindAsync(id);
        if (customer is null) return NotFound();

        await CheckPhoneIsFreeAsync(input.Phone, id);
        if (!ModelState.IsValid) return View(input);

        customer.Name = input.Name;           // the tracked entity is changed,
        customer.Phone = input.Phone;         // so SaveChanges sends an UPDATE
        customer.Address = input.Address;     // for exactly these columns
        await db.SaveChangesAsync();
        TempData["Message"] = $"Customer {customer.Name} saved.";
        return RedirectToAction(nameof(Details), new { id });
    }

    public async Task<IActionResult> Delete(int id)
    {
        var customer = await db.Customers.Include(c => c.Orders)
            .AsNoTracking().FirstOrDefaultAsync(c => c.Id == id);
        return customer is null ? NotFound() : View(customer);
    }
```

Listing 5.12 (Continued) *Controllers/CustomersController.cs: complete CRUD for customers with EF Core.*

```
    [HttpPost, ActionName("Delete")]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> DeleteConfirmed(int id)
    {
        var customer = await db.Customers.FindAsync(id);
        if (customer is null) return NotFound();
        if (await db.Orders.AnyAsync(o => o.CustomerId == id))
        {
            TempData["Error"] = "A customer with orders cannot be deleted.";
            return RedirectToAction(nameof(Details), new { id });
        }
        db.Customers.Remove(customer);
        await db.SaveChangesAsync();          // DELETE
        TempData["Message"] = $"Customer {customer.Name} deleted.";
        return RedirectToAction(nameof(Index));
    }

    async Task CheckPhoneIsFreeAsync(string phone, int ownId)
    {
        if (await db.Customers.AnyAsync(c => c.Phone == phone && c.Id != ownId))
        {
            ModelState.AddModelError(nameof(Customer.Phone),
                "Another customer already uses this phone number.");
        }
    }
}
```

5.10.1 Getting the DbContext

`public class CustomersController(LaundryDbContext db) : Controller` uses a primary constructor: ASP.NET Core creates the controller for every request and hands it the request's DbContext. The controller never writes `new LaundryDbContext(…)`, never opens or closes a connection, and never disposes anything; the DI container does all of that.

5.10.2 Read: queries, AsNoTracking and Include

`Index` builds a query step by step. `db.Customers.AsNoTracking()` is an `IQueryable<Customer>`: a *description* of a query, not yet a result. The `if` adds a `Where` only when a search word was given; `OrderBy` adds sorting; and only `ToListAsync()` sends the SQL and waits for the rows. Building the query in pieces like this is normal LINQ and costs nothing, because nothing is sent until the end.

`AsNoTracking()` tells EF Core not to register the loaded customers in the change tracker. A list page only displays; tracking would cost memory and time for nothing. Use it on every read that will not be followed by `SaveChanges`.

`Contains(search)` is translated by the SQLite provider into `instr(Name, @search) > 0`, and the value of `search` travels as a parameter, never pasted into the SQL text. That is what makes EF Core queries safe from SQL injection: a customer called `'; DROP TABLE Customers; --` is just a strange name. One side effect of `instr` is worth knowing now: it compares case-sensitively, so a search for `sari` does not find *Sari Wulandari*. Chapter 8 fixes that in Problem 8.4.

`Details` shows one customer with all their orders and, for each order, its service. Navigation properties are not loaded automatically; `Include(c => c.Orders)` asks for the orders, and `ThenInclude(o => o.Service)` for the service of each of those orders. EF Core sends one SQL statement with `LEFT JOIN`s and assembles the object graph from the rows [7]. `FirstOrDefaultAsync(c => c.Id == id)` returns `null` when there is no such customer, and the action answers with `NotFound()`, a 404 page from the status-code pages of Chapter 3.

5.10.3 Create: binding, checking, saving

The `GET` version of `Create` returns an empty form. The `POST` version receives the filled-in form as a `Customer` object, built by model binding from the form fields. The `[Bind("Name,Phone,Address")]` attribute lists the only properties that may be taken from the request. Without it, a visitor who adds a field `CreatedAt=2020-01-01` or `Id=7` to the form could set values the page never offered; this attack is called *overposting* [8]. Chapter 6 replaces `[Bind]` with a separate form class, which is the cleaner solution.

`CheckPhoneIsFreeAsync` asks the database whether another customer already has the phone number, with `AnyAsync`, which becomes `SELECT EXISTS(…)`. If so, it adds an error to `ModelState`, the dictionary that collects everything wrong with the request. `ModelState.IsValid` is `false` when anything was added, including errors that model binding found by itself; for example, `Name` is a non-nullable `string`, so ASP.NET Core already treats an empty name as an error. If the model is invalid the action shows the form again, with the values the user typed and the messages next to the fields.

When everything is valid, `db.Customers.Add(customer)` puts the new object into the change tracker in the state *Added*, and `await db.SaveChangesAsync()` sends the `INSERT`. SQLite assigns the new `Id`, EF Core copies it into `customer.Id`, and the action redirects to the new customer's page. The pattern *POST, then redirect, then GET* (PRG) means that pressing F5 on the result page reloads a page instead of submitting the form a second time. `TempData["Message"]` survives exactly that one redirect and is shown by the layout as a green bar.

5.10.4 Update: change a tracked entity

`Edit(id, input)` loads the customer with `FindAsync(id)`, which is tracked (no `AsNoTracking` here), copies the three editable values from the form onto it, and calls `SaveChangesAsync()`. The change tracker compares the current values with the values it saw when it loaded the row, finds the changed ones, and sends an `UPDATE` for exactly those columns [9]. If the user changed only the address, only the address is written.

The line `if (id != input.Id) return BadRequest();` guards against a form whose hidden `Id` field was changed to point at another customer. It is a small check that costs nothing.

5.10.5 Delete: confirm, then refuse or remove

Deleting takes two requests: a `GET` that shows a confirmation page and a `POST` that deletes. A link must never delete anything, because browsers, search engines and link previews follow links on their own. `[HttpPost, ActionName("Delete")]` lets the method be called `DeleteConfirmed` in C# (two methods called `Delete` with the same parameter would not compile) while still answering the address `/Customers/Delete/3`.

The action refuses to delete a customer who has orders and says why. The database would refuse as well, because of `DeleteBehavior.Restrict`, but with an exception rather than a friendly message; checking first gives the cashier a sentence instead of an error page.

5.10.6 The views

The views are Razor, as in Chapter 3. **Listing 5.13** is the list with its search form, **Listing 5.14** the fields shared by the Create and Edit pages, and **Listing 5.15** the customer's page with their orders.

Listing 5.13 `Views/Customers/Index.cshtml`: the customer list with a search box.

```
@model List<Customer>
@{
    ViewData["Title"] = "Customers";
}
<h1>Customers</h1>
<form method="get" class="search">
    <input name="search" value="@ViewData["Search"]"
           placeholder="Name or phone" />
    <button type="submit">Search</button>
    <a class="button" asp-action="Create">New customer</a>
</form>
<table class="list">
    <thead>
        <tr><th>Name</th><th>Phone</th><th>Address</th><th></th></tr>
    </thead>
    <tbody>
    @foreach (var c in Model)
    {
        <tr>
            <td><a asp-action="Details" asp-route-id="@c.Id">@c.Name</a></td>
            <td>@c.Phone</td>
            <td>@c.Address</td>
            <td><a asp-action="Edit" asp-route-id="@c.Id">Edit</a>
                <a asp-action="Delete" asp-route-id="@c.Id">Delete</a></td>
        </tr>
    }
    </tbody>
</table>
<p>@Model.Count customer(s).</p>
```

Listing 5.14 `Views/Customers/_Form.cshtml`: the fields shared by the Create and Edit pages.

```
@model Customer
<div asp-validation-summary="ModelOnly" class="errors"></div>
<label asp-for="Name"></label>
<input asp-for="Name" />
<span asp-validation-for="Name" class="error"></span>

<label asp-for="Phone"></label>
<input asp-for="Phone" />
<span asp-validation-for="Phone" class="error"></span>

<label asp-for="Address"></label>
<input asp-for="Address" />
<span asp-validation-for="Address" class="error"></span>
```

Listing 5.15 `Views/Customers/Details.cshtml`: one customer and the customer's orders, newest first.

```
@model Customer
@{
    ViewData["Title"] = Model.Name;
}
<h1>@Model.Name</h1>
<p>@Model.Phone · @(Model.Address ?? "no address")
   · customer since @Model.CreatedAt.ToString("d MMM yyyy")</p>
<p>
    <a class="button" asp-controller="Orders" asp-action="Create"
       asp-route-customerId="@Model.Id">New order</a>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a>
</p>
<h2>Orders</h2>
<table class="list">
    <thead>
        <tr><th>#</th><th>Received</th><th>Service</th><th>Total</th>
            <th>Status</th></tr>
    </thead>
    <tbody>
    @foreach (var o in Model.Orders.OrderByDescending(o => o.ReceivedAt))
    {
        <tr>
            <td><a asp-controller="Orders" asp-action="Details"
                   asp-route-id="@o.Id">@o.Id</a></td>
            <td>@o.ReceivedAt.ToString("dd/MM HH:mm")</td>
            <td>@o.Service?.Name</td>
            <td class="num">@Money.Rp(o.Total)</td>
            <td>@o.Status</td>
        </tr>
    }
    </tbody>
</table>
```

The tag helpers of Chapter 3 do the careful work. `asp-for="Name"` gives the input the right `name` and `id`, fills in the current value and, after a failed submission, the value the user typed; `asp-validation-for` prints the message for that field. A `<form method="post">` built with tag helpers also receives a hidden anti-forgery token automatically, and `[ValidateAntiForgeryToken]` on the `POST` actions rejects any submission that lacks it. That protects the shop against *cross-site request forgery*: a page on another site cannot make a signed-in cashier's browser submit a form to the laundry.

The Create and Edit views (**Listing 5.16** shows Create) only add the page title, the form tag and the buttons around the shared partial view.

Listing 5.16 `Views/Customers/Create.cshtml`: the new-customer page around the shared form.

```
@model Customer
@{
    ViewData["Title"] = "New customer";
}
<h1>New customer</h1>
<form asp-action="Create" method="post" class="form">
    <partial name="_Form" model="Model" />
    <button type="submit">Save</button>
    <a asp-action="Index">Cancel</a>
</form>
```

5.11 Orders, Payments and the Price List

5.11.1 The order counter

`OrdersController` (**Listing 5.17**) turns the carbon book into pages: a list that can be filtered by status, a page per order, a form for new orders, a button that moves an order along the workflow, and a small form that records a payment.

Listing 5.17 `Controllers/OrdersController.cs`: the order list, order pages, new orders priced by Chapter 3's calculator, the workflow and payments.

```
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Controllers;

public class OrdersController(LaundryDbContext db, PriceCalculator calculator)
    : Controller
{
    // GET /Orders or /Orders?status=Queued
    public async Task<IActionResult> Index(OrderStatus? status)
    {
        var query = db.Orders
            .Include(o => o.Customer)
            .Include(o => o.Service)
            .Include(o => o.Payments)
            .AsNoTracking();
        if (status is not null)
        {
            query = query.Where(o => o.Status == status);
        }
        ViewData["Status"] = status;
        return View(await query.OrderByDescending(o => o.ReceivedAt)
                               .ToListAsync());
    }

    public async Task<IActionResult> Details(int id)
    {
        var order = await db.Orders
            .Include(o => o.Customer)
            .Include(o => o.Service)
            .Include(o => o.Payments)
            .AsNoTracking()
            .FirstOrDefaultAsync(o => o.Id == id);
        return order is null ? NotFound() : View(order);
    }

    public async Task<IActionResult> Create(int? customerId)
    {
        await FillListsAsync();
        return View(new Order { CustomerId = customerId ?? 0, Quantity = 1 });
    }
```

Listing 5.17 (Continued) *Controllers/OrdersController.cs: the order list, order pages, new orders priced by Chapter 3's calculator, the workflow and payments.*

```
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create([Bind("CustomerId,ServiceId,"
        + "Quantity,Pickup,DeliveryAddress,Note")] Order order)
    {
        var service = await db.Services.FindAsync(order.ServiceId);
        if (service is null || !service.IsActive)
        {
            ModelState.AddModelError(nameof(Order.ServiceId), "Choose a service.");
        }
        if (!await db.Customers.AnyAsync(c => c.Id == order.CustomerId))
        {
            ModelState.AddModelError(nameof(Order.CustomerId),
                "Choose a customer.");
        }
        if (order.Quantity <= 0)
        {
            ModelState.AddModelError(nameof(Order.Quantity),
                "The quantity must be more than zero.");
        }
        if (!ModelState.IsValid || service is null)
        {
            await FillListsAsync();
            return View(order);
        }

        // The price rules of Chapter 3: minimum 3 kg, whole items, pickup fee.
        var quote = calculator.Calculate(service, order.Quantity, order.Pickup);
        order.Total = (int)decimal.Round(quote.Total,
                                         MidpointRounding.AwayFromZero);
        order.Status = OrderStatus.Queued;
        order.ReceivedAt = DateTime.Now;
        db.Orders.Add(order);
        await db.SaveChangesAsync();
        TempData["Message"] = $"Order #{order.Id} created: {Money.Rp(order.Total)}."
            + (quote.Notes.Count > 0 ? " " + string.Join(" ", quote.Notes) : "");
        return RedirectToAction(nameof(Details), new { id = order.Id });
    }
```

Listing 5.17 (Continued) *Controllers/OrdersController.cs: the order list, order pages, new orders priced by Chapter 3's calculator, the workflow and payments.*

```
    // Queued -> Finished -> Departed, each step with its time stamp.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Advance(int id)
    {
        var order = await db.Orders.FindAsync(id);
        if (order is null) return NotFound();
        switch (order.Status)
        {
            case OrderStatus.Queued:
                order.Status = OrderStatus.Finished;
                order.FinishedAt = DateTime.Now;
                break;
            case OrderStatus.Finished:
                order.Status = OrderStatus.Departed;
                order.DepartedAt = DateTime.Now;
                break;
            default:
                TempData["Error"] = "This order has already left the shop.";
                return RedirectToAction(nameof(Details), new { id });
        }
        await db.SaveChangesAsync();
        TempData["Message"] = $"Order #{id} is now {order.Status}.";
        return RedirectToAction(nameof(Details), new { id });
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> AddPayment(int id, int amount,
                                                PaymentMethod method)
    {
        if (!await db.Orders.AnyAsync(o => o.Id == id)) return NotFound();
        if (amount <= 0)
        {
            TempData["Error"] = "The amount must be more than zero.";
            return RedirectToAction(nameof(Details), new { id });
        }
        db.Payments.Add(new Payment
        {
            OrderId = id, Amount = amount, Method = method,
            PaidAt = DateTime.Now
        });
        await db.SaveChangesAsync();
        TempData["Message"] = $"Payment of {Money.Rp(amount)} recorded.";
        return RedirectToAction(nameof(Details), new { id });
    }
```

Listing 5.17 (Continued) *Controllers/OrdersController.cs: the order list, order pages, new orders priced by Chapter 3's calculator, the workflow and payments.*

```
    async Task FillListsAsync()
    {
        ViewBag.Customers = new SelectList(
            await db.Customers.OrderBy(c => c.Name).ToListAsync(),
            nameof(Customer.Id), nameof(Customer.Name));
        ViewBag.Services = new SelectList(
            await db.Services.Where(s => s.IsActive).OrderBy(s => s.Id)
                             .ToListAsync(),
            nameof(Service.Id), nameof(Service.Description));
    }
}
```

A few details deserve attention.

- `Index(OrderStatus? status)` shows model binding with an enum: `/Orders?status=Queued` fills `status` with `OrderStatus.Queued`; a missing or unknown value leaves it `null`, and the list shows everything. The three `Include` calls load each order's customer, service and payments, because the list shows the customer's name, the service and the balance.
- `Create` looks up the service and the customer before saving. The form offers only active services, but a form can be edited by anyone with the browser's developer tools, so the server checks again. The price comes from `calculator.Calculate(service, quantity, pickup)`, the very `PriceCalculator` of Chapter 3, injected into the controller. It applies the minimum of 3 kg for per-kg services, rounds items up to whole items, and adds the pickup fee of Rp 10.000 below Rp 100.000. The notes it returns ("Minimum 3 kg, so 2 kg is billed as 3 kg") are shown to the cashier after saving.
- `Advance` is the workflow: a `switch` moves a queued order to finished and a finished order to departed, and sets the matching timestamp. It uses `FindAsync`, because only the order row is needed.
- `AddPayment` adds a `Payment` row directly through `db.Payments.Add`, with the order's `Id` as the foreign key; there is no need to load the order and its list of payments to add one more.

The order page (**Listing 5.18**) shows the facts of the order, the button for the next step of the workflow, the payments, and, while money is owed, a payment form prefilled with the balance.

Listing 5.18 `Views/Orders/Details.cshtml`: one order, its workflow button, its payments and the payment form.

```
@model Order
@{
    ViewData["Title"] = $"Order #{Model.Id}";
}
<h1>Order #@Model.Id · @Model.Status</h1>
<dl class="facts">
    <dt>Customer</dt>
    <dd><a asp-controller="Customers" asp-action="Details"
           asp-route-id="@Model.CustomerId">@Model.Customer?.Name</a>
        (@Model.Customer?.Phone)</dd>
    <dt>Service</dt>
    <dd>@Model.Service?.Name, @Model.Quantity @Model.Service?.Unit</dd>
    <dt>Delivery</dt>
    <dd>@(Model.Pickup ? $"to {Model.DeliveryAddress}"
                       : "collected at the shop")</dd>
    <dt>Received</dt><dd>@Model.ReceivedAt.ToString("dd/MM/yyyy HH:mm")</dd>
    <dt>Finished</dt><dd>@Model.FinishedAt?.ToString("dd/MM/yyyy HH:mm")</dd>
    <dt>Departed</dt><dd>@Model.DepartedAt?.ToString("dd/MM/yyyy HH:mm")</dd>
    <dt>Note</dt><dd>@Model.Note</dd>
    <dt>Total</dt><dd>@Money.Rp(Model.Total)</dd>
    <dt>Paid</dt><dd>@Money.Rp(Model.Paid)</dd>
    <dt>Balance</dt><dd><strong>@Money.Rp(Model.Balance)</strong></dd>
</dl>

@if (Model.Status != OrderStatus.Departed)
{
    <form asp-action="Advance" asp-route-id="@Model.Id" method="post">
        <button type="submit">
            @(Model.Status == OrderStatus.Queued ? "Mark finished"
                                                 : "Mark picked up")
        </button>
    </form>
}

<h2>Payments</h2>
<table class="list">
    <thead><tr><th>Paid at</th><th>Method</th><th class="num">Amount</th></tr>
    </thead>
    <tbody>
    @foreach (var p in Model.Payments.OrderBy(p => p.PaidAt))
    {
        <tr><td>@p.PaidAt.ToString("dd/MM HH:mm")</td><td>@p.Method</td>
            <td class="num">@Money.Rp(p.Amount)</td></tr>
    }
    </tbody>
</table>
```

Listing 5.18 (Continued) *Views/Orders/Details.cshtml: one order, its workflow button, its payments and the payment form.*

```
@if (Model.Balance > 0)
{
    <form asp-action="AddPayment" asp-route-id="@Model.Id" method="post"
          class="inline">
        <input name="amount" type="number" min="1" value="@Model.Balance" />
        <select name="method"
                asp-items="Html.GetEnumSelectList<PaymentMethod>()"></select>
        <button type="submit">Record payment</button>
    </form>
}
```

`Html.GetEnumSelectList<PaymentMethod>()` builds the options *Cash*, *Qris* and *Transfer* from the enum, so a new payment method needs one new enum value and no new HTML.

The form for new orders (**Listing 5.19**) fills two drop-down lists from `ViewBag.Customers` and `ViewBag.Services`, which the controller's `FillListsAsync` method prepares as `SelectList` objects. `asp-items` turns a `SelectList` into `<option>` elements and selects the current value.

Listing 5.19 `Views/Orders/Create.cshtml`: the counter form for a new order.

```
@model Order
@{
    ViewData["Title"] = "New order";
}
<h1>New order</h1>
<form asp-action="Create" method="post" class="form">
    <div asp-validation-summary="ModelOnly" class="errors"></div>
    <label asp-for="CustomerId">Customer</label>
    <select asp-for="CustomerId" asp-items="ViewBag.Customers">
        <option value="0">- choose -</option>
    </select>
    <span asp-validation-for="CustomerId" class="error"></span>

    <label asp-for="ServiceId">Service</label>
    <select asp-for="ServiceId" asp-items="ViewBag.Services">
        <option value="">- choose -</option>
    </select>
    <span asp-validation-for="ServiceId" class="error"></span>

    <label asp-for="Quantity">Quantity (kg, items or pairs)</label>
    <input asp-for="Quantity" type="number" step="0.01" min="0.01" />
    <span asp-validation-for="Quantity" class="error"></span>

    <label class="check"><input asp-for="Pickup" /> Pickup and delivery</label>
    <label asp-for="DeliveryAddress">Delivery address</label>
    <input asp-for="DeliveryAddress" />

    <label asp-for="Note"></label>
    <input asp-for="Note" />

    <button type="submit">Create order</button>
    <a asp-action="Index">Cancel</a>
</form>
```

5.11.2 The owner's price list and the counter page

`PricesController` (**Listing 5.20**) lets the owner add a service and change a description, a price or the `IsActive` flag. It works on the same `Services` table that the public `/services` page of Chapter 3 now reads, so a changed price appears on the public page at once. A new service gets the next free code: `MaxAsync(s => (string?)s.Id)` finds the largest code (`S0012`), and `$"S{… + 1:D4}"` formats the next number with four digits (`S0013`). The cast to `string?` makes `MaxAsync` return `null` instead of throwing when the table is empty.

Listing 5.20 `Controllers/PricesController.cs`: the owner's price list, with automatic service codes.

```
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Controllers;

// The owner's price list: the same Service rows that /services shows.
public class PricesController(LaundryDbContext db) : Controller
{
    public async Task<IActionResult> Index() =>
        View(await db.Services.AsNoTracking().OrderBy(s => s.Id)
                              .ToListAsync());

    public IActionResult Create() => View(new Service());

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(
        [Bind("Description,Price,IsActive")] Service service)
    {
        if (!ModelState.IsValid) return View(service);

        // The next free code: S0012 -> S0013.
        var last = await db.Services.MaxAsync(s => (string?)s.Id) ?? "S0000";
        service.Id = $"S{int.Parse(last[1..]) + 1:D4}";
        db.Services.Add(service);
        await db.SaveChangesAsync();
        TempData["Message"] = $"{service.Id} {service.Name} added.";
        return RedirectToAction(nameof(Index));
    }

    public async Task<IActionResult> Edit(string id)
    {
        var service = await db.Services.FindAsync(id);
        return service is null ? NotFound() : View(service);
    }
```

Listing 5.20 (Continued) *Controllers/PricesController.cs: the owner's price list, with automatic service codes.*

```
    // A new price counts for new orders only: every order keeps its Total.
    // A service is never deleted; it is switched off with IsActive = false.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Edit(string id,
        [Bind("Id,Description,Price,IsActive")] Service input)
    {
        if (id != input.Id) return BadRequest();
        var service = await db.Services.FindAsync(id);
        if (service is null) return NotFound();
        if (!ModelState.IsValid) return View(input);

        service.Description = input.Description;
        service.Price = input.Price;
        service.IsActive = input.IsActive;
        await db.SaveChangesAsync();
        TempData["Message"] = $"{service.Id} {service.Name} saved.";
        return RedirectToAction(nameof(Index));
    }
}
```

There is deliberately no delete action. An old order points to its service; deleting the service would either be refused by the foreign key or, worse, lose the history. Switching `IsActive` off

hides the service from the counter and from the public list, while every old order still knows what it was.

Finally, `CounterController` (**Listing 5.21**) answers the owner's evening question in four numbers: orders in the queue, orders ready for pickup, orders received today, and money received today. Each number is one small SQL statement: `CountAsync` becomes `SELECT COUNT(*)` and `SumAsync` becomes `SELECT COALESCE(SUM("p"."Amount"), 0)`, which works because amounts are `int` columns (Section 5.3). "Today" is the range from midnight to the next midnight, `>= today && < tomorrow`, which the database can compare directly on the stored timestamps.

Listing 5.21 `Controllers/CounterController.cs`: four numbers for the counter, each one SQL statement.

```
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Controllers;

// GET /Counter: what the cashier needs to know at a glance.
public class CounterController(LaundryDbContext db) : Controller
{
    public async Task<IActionResult> Index()
    {
        var today = DateTime.Today;
        var tomorrow = today.AddDays(1);
        var model = new CounterViewModel(
            Queued: await db.Orders
                .CountAsync(o => o.Status == OrderStatus.Queued),
            ReadyForPickup: await db.Orders
                .CountAsync(o => o.Status == OrderStatus.Finished),
            ReceivedToday: await db.Orders
                .CountAsync(o => o.ReceivedAt >= today && o.ReceivedAt < tomorrow),
            PaidToday: await db.Payments
                .Where(p => p.PaidAt >= today && p.PaidAt < tomorrow)
                .SumAsync(p => p.Amount));
        return View(model);
    }
}
```

5.12 Running It

On your computer, with internet access, run the commands of Section 5.9 and then `dotnet run`, and open `http://localhost:4050`. The home page, the services and the outlets look exactly as in Chapter 3, now read from the database; the header has a second row of links for the staff pages. A good first tour:

1. **Counter** shows the sample data: two orders in the queue, one ready for pickup, two received today, and the money received today.
2. **Customers** lists Budi Santoso, Dewi Lestari and Sari Wulandari. Search for `Sari`, then for `sari`, and notice the difference described in Section 5.10.2.
3. Add a customer, then try to add a second one with the same phone number; the form comes back with the message from `CheckPhoneIsFreeAsync`.
4. **Orders → New order**: choose Dewi Lestari and *Wash, dry & fold*, enter 2 kg, and create. The message says the order costs Rp 24.000 and explains that 2 kg is billed as 3 kg.

5. On the order page, record a payment of Rp 10.000, then press **Mark finished** and **Mark picked up**, and watch the timestamps appear.
6. **Prices**: change the price of *Ironing only* and open `/services` in another tab; the new price is there. The old orders keep their totals.
7. Try to delete Budi Santoso; the page explains that a customer with orders cannot be deleted.

Keep an eye on the terminal while you click. Every request prints the timing line of Chapter 3's middleware and, before it, the SQL that EF Core sent, with its parameters shown by name, such as `@search` or `@p0` (their values are hidden unless you enable sensitive-data logging, which you should do only on your own machine). Reading that log is the fastest way to learn what LINQ becomes.

The project was written where NuGet was not reachable, so it could not be restored or run there. What was done there is stated in the chapter's README: `LinqDemo` was built and run (its output in Section 5.4 is real), and the web app's C# code and Razor views were compiled against small compile-only stand-ins for the EF Core API, a full type check that caught no errors. The steps above are what you will see on a normal computer; the only numbers quoted in them follow from the seed data and the price rules.

5.13 Making It Better

- **Read without tracking, write with tracking.** Every list and details page uses `AsNoTracking()`. Only the actions that change data load tracked entities.
- **Stay asynchronous.** The new controllers use `ToListAsync`, `FirstOrDefaultAsync`, `AnyAsync` and `SaveChangesAsync`. While SQLite reads the disk, the server thread serves another request. On a counter with three cashiers this hardly matters; on a website with thousands of visitors it decides how many requests one server can handle.
- **Index what you search.** The unique index on `Phone` speeds up phone searches as a side effect, and the index on `Orders.Status` helps the filtered order list. Add an index when a real query is slow, not in advance.
- **Watch the number of queries.** The order list uses three `Include`s and loads every payment of every order, just to show one balance. With 20 orders that is still one query, but it reads more than the page needs, and with thousands of orders the list itself becomes too long. Chapter 8 (Problems 8.3 and 8.4) replaces it with a projection and pages.
- **Mind the decimal separator.** Model binding reads form values with the server's current culture. On a Windows computer set to Indonesian, `4.5` from the browser's number field can become 45, because the comma is the decimal separator in `id-ID`. Keep the server culture at the default, or set `CultureInfo.DefaultThreadCurrentCulture` explicitly, and test with a value that has decimals.

5.14 Using It for Real

- **Your own shop.** Replace `data/Service.csv` and `data/Outlet.csv` with your own services and branches before the first `database update`, or change them later on the Prices page. A tailor would have services such as *Hemming (per item)*; a car wash *Motorbike wash (per item)*. The per-kg rule of Chapter 3's calculator only applies to descriptions ending in `(per kg)`.
- **Another database.** For SQL Server, replace the package with `Microsoft.EntityFrameworkCore.SqlServer`, change `UseSqlite` to `UseSqlServer`, change the connection string, delete the `Migrations` folder and create `InitialCreate` again, because

migrations are written for one provider. For MySQL or MariaDB, the community provider `Pomelo.EntityFrameworkCore.MySql` is the usual choice; check on NuGet that its version supports EF Core 10 before you switch.

- **Back up the file.** With SQLite the whole shop is one file, `laundry.db`. Copy it every evening to another disk or to cloud storage; a copy taken while the app is idle is a complete backup. Chapter 6 puts the file in a fixed folder on the server.
- **Personal data.** Names, phone numbers and addresses are personal data under Indonesia's Personal Data Protection Law (UU PDP, Law No. 27 of 2022). Collect only what the shop needs (this model does not ask for identity numbers or birthdays), keep the database file on a computer that is locked, and delete customers who ask for it and have no open orders; the delete page already supports that.

5.15 Profitina in Practice

Profitina's own platform is written in Python with FastAPI rather than C#, but it is built from the same parts as this chapter. Its data lives in PostgreSQL in production, while parts of its test path run on SQLite, exactly the split between a server database and a file database described in Sections 5.2.4 and 5.14; a guard in its test suite checks that the SQL used stays valid in both dialects, which is the problem that EF Core's providers solve for .NET developers. Its weekly score history per brand is a growing table that the dashboard queries with aggregates, the same kind of `GROUP BY` and `SUM` as the counter page and the LINQ demo. And, like Profitina Laundry, every change to its database structure goes through versioned migrations that the release process applies before the new version is allowed to answer requests.

5.16 Common Mistakes, and How to Avoid Them

- **Forgetting `SaveChangesAsync()`.** `Add`, `Remove` and changing a property only change the change tracker. Nothing reaches the database until `SaveChanges` or `SaveChangesAsync` is called.
- **Changing the model without a migration.** After you add a property, the database does not have the column. Since EF Core 9, `Migrate()` and `database update` stop with the error *"The model … has pending changes"*; run `dotnet ef migrations add <Name>` first.
- **Expecting navigation properties to fill themselves.** `order.Customer` is `null` and `order.Payments` is empty unless the query said `Include`. A balance computed from an empty list is simply wrong, without any error.
- **Tracking everything.** Reading a thousand rows without `AsNoTracking()` for a list page wastes memory. Conversely, calling `SaveChanges` after an `AsNoTracking` query saves nothing, because nothing was tracked.
- **Registering the DbContext as a singleton.** One DbContext for the whole app breaks as soon as two requests arrive together. `AddDbContext` already registers it correctly as scoped; leave it that way, and make services that use it scoped too.
- **Cascade delete by accident.** EF Core's default for required relationships is cascade delete. Decide deliberately, per relationship, whether deleting the parent may delete the children; for customers and orders, it must not.
- **Deleting through a link.** A `GET` request must never change data. Use a form with `method="post"` and an anti-forgery token for every action that writes.

- **Summing money in `decimal` on SQLite.** EF Core cannot translate `Sum` over a `decimal` column for SQLite and throws at run time. Keep amounts that you add up in integer columns, or convert them.

5.17 Chapter Summary

- An ORM maps classes to tables and objects to rows. EF Core 10 is the ORM of .NET 10; its SQLite provider and design tools are NuGet packages, and `dotnet ef` is a global tool.
- The model is plain C# classes. Conventions find keys, foreign keys and relationships; the fluent API in `OnModelCreating` adds lengths, unique indexes, value conversions and delete behavior.
- A `DbContext` is registered once with `AddDbContext` and injected, one per request, into controllers and services.
- Migrations turn model changes into ordered, reversible C# files; `database update` applies them and records them in `__EFMigrationsHistory`. `UseSeeding` fills an empty database, here from the CSV files of Chapter 3.
- LINQ on a `DbSet` is translated into parameterized SQL; nothing is sent until a method such as `ToListAsync` asks for the result. `Include` loads related data; `AsNoTracking` skips the change tracker for reads.
- Create adds and saves; update changes a tracked entity and saves, and EF Core writes only the changed columns; delete removes and saves, guarded by a check and by `Restrict` on the relationship.
- The catalog of Chapter 3 now reads the database through `EfCatalog`, and not one controller of Chapter 3 changed.

5.18 Exercises

1. By hand, apply Equation (5.1) to order 2 of the sample data after two more payments of Rp 10.000 and Rp 6.000. What are `Paid` and `Balance`, and what does the order page show in place of the payment form?
2. For each of the following, say which SQL statement EF Core sends and when: (a) `var q = db.Orders.Where(o => o.Status == OrderStatus.Queued);` (b) `var n = await q.CountAsync();` (c) `var list = await q.ToListAsync();` (d) `list[0].Note = "fragile"; await db.SaveChangesAsync();` in a controller whose query did *not* use `AsNoTracking`.
3. Write a LINQ query for the counter page that returns the three customers who have spent the most (sum of the totals of their orders), with the name and the sum, largest first. Test it by adding it to `LinqDemo`.
4. The owner wants to know at which outlet each order was received. Add the relationship to the model: which properties go into `Order`, what goes into `OnModelCreating`, which migration command follows, and what changes in the order form?
5. A cashier deletes a payment that was entered twice. Add an action `DeletePayment(int id, int paymentId)` to `OrdersController`, reachable only by `POST`, that removes the payment if it belongs to that order.
6. Explain in your own words why `EfCatalog` is registered as scoped while `PriceCalculator` stays a singleton. What would happen if both were singletons?

5.19 Solutions to the Exercises

1. The payments are Rp 20.000, Rp 10.000 and Rp 6.000, so `Paid` $= 20000 + 10000 + 6000 = 36000$ and, by Equation (5.1), `Balance` $= 36000 - 36000 = 0$. The details view shows the payment form only `@if (Model.Balance > 0)`, so the form disappears and the page shows a balance of Rp 0.

2. (a) Nothing: `q` is only a description of a query. (b) `SELECT COUNT(*) FROM "Orders" AS "o" WHERE "o"."Status" = 'Queued'`, sent when `CountAsync` runs; the enum is compared as text because of `HasConversion<string>()`. (c) `SELECT "o"."Id", "o"."CustomerId", … FROM "Orders" AS "o" WHERE "o"."Status" = 'Queued'`, sent when `ToListAsync` runs; the orders are tracked. (d) `UPDATE "Orders" SET "Note" = @p0 WHERE "Id" = @p1`, sent by `SaveChangesAsync` with only the changed column (SQLite's provider adds `RETURNING 1` to confirm that one row was changed).

3. Group the orders by customer, sum, sort descending and take three. On the database the same query works on `db.Orders` with `await … .ToListAsync()` at the end; `Total` is an `int`, so SQLite can sum it. **Listing 5.22** shows the lines for the demo.

Listing 5.22 Exercise 5.3: the three best customers, as lines to add to `LinqDemo/Program.cs`.

```
var top = orders
    .GroupBy(o => o.Customer!.Name)
    .Select(g => new { Customer = g.Key, Spent = g.Sum(o => o.Total) })
    .OrderByDescending(r => r.Spent)
    .Take(3);
foreach (var r in top)
{
    Console.WriteLine($"   {r.Customer,-16} {Money.Rp(r.Spent),10}");
}
```

With the demo data it prints Budi Santoso with Rp 116.000, Dewi Lestari with Rp 48.000 and Sari Wulandari with Rp 36.000.

4. Add `public string OutletId { get; set; } = "";` and `public Outlet? Outlet { get; set; }` to `Order`. In `OnModelCreating`, inside the `Order` block, add `e.HasOne(o => o.Outlet).WithMany().HasForeignKey(o => o.OutletId).OnDelete(DeleteBehavior.Restrict);`. Then run `dotnet ef migrations add AddOutletToOrders`. Because existing rows need a value for the new non-nullable column, either make it nullable (`string?`) for the migration, or give it a default in the migration (for example `O02` for the Keputih outlet where the system started) and edit the generated `AddColumn` call accordingly. In the order form, add `OutletId` to `[Bind]`, fill `ViewBag.Outlets` in `FillListsAsync` from `db.Outlets`, add a `<select asp-for="OutletId" asp-items="ViewBag.Outlets">`, and check in `Create` that the outlet exists. The ERD of **Figure 5.2** then gets a fourth relationship, from `Outlets` to `Orders`.

5. Look the payment up by both keys, so that a forged `paymentId` from another order is refused, then remove it (**Listing 5.23**):

Listing 5.23 Exercise 5.5: an action to add to `OrdersController` that removes one payment of one order.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeletePayment(int id, int paymentId)
{
    var payment = await db.Payments
        .FirstOrDefaultAsync(p => p.Id == paymentId && p.OrderId == id);
    if (payment is null) return NotFound();
    db.Payments.Remove(payment);
    await db.SaveChangesAsync();
    TempData["Message"] = $"Payment of {Money.Rp(payment.Amount)} removed.";
    return RedirectToAction(nameof(Details), new { id });
}
```

In the payments table of the details view, add to each row a small form: `<form asp-action="DeletePayment" asp-route-id="@Model.Id" asp-route-paymentId="@p.Id" method="post"><button type="submit">Remove</button></form>`.

6. `EfCatalog` holds a `LaundryDbContext`, and the DbContext is scoped: one per request, not safe to share between requests that run at the same time, and disposed at the end of its request. A singleton `EfCatalog` would keep the first request's DbContext forever; ASP.NET Core refuses this at start-up in Development with the message that a scoped service cannot be consumed from a singleton. `PriceCalculator` holds no data and no DbContext; its methods only compute from their arguments, so one shared instance is safe and cheapest. If both were singletons, the calculator would be fine and the catalog would be broken.

References

[1] Microsoft, "Entity Framework Core overview," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/

[2] Microsoft, ".NET and .NET Core support policy," accessed Sep. 2026. [Online]. Available: https://dotnet.microsoft.com/platform/support/policy/dotnet-core

[3] Microsoft, "SQLite EF Core database provider limitations," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/providers/sqlite/limitations

[4] NuGet Gallery, "Microsoft.EntityFrameworkCore.Sqlite 10.0.12," released Sep. 8, 2026. [Online]. Available: https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Sqlite

[5] Microsoft, "Data seeding," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/modeling/data-seeding

[6] Microsoft, "Migrations overview," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/managing-schemas/migrations/

[7] Microsoft, "Loading related data," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/querying/related-data/

[8] Microsoft, "Model binding in ASP.NET Core," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/models/model-binding

[9] Microsoft, "Change tracking in EF Core," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/change-tracking/

Chapter 6

ASP.NET Core MVC III: Validation, Identity and Deployment

At the end of Chapter 5 the laundry had a real database and real CRUD pages, and anybody who could reach the website could use all of them. This chapter makes the application fit to be trusted with a real shop. By the end of it you will be able to validate every form on the server with data annotation attributes, your own validation attribute and rules that involve several fields; accept forms through dedicated form classes instead of entities; explain the difference between authentication and authorization; add ASP.NET Core Identity so that staff sign in with hashed passwords, lockout and roles; protect the staff pages while the public pages of Chapter 3 stay open; and publish the application to a server behind Nginx or IIS, with production settings, a health endpoint, cookie keys that survive restarts and a database file that is backed up. Along the way you will run the app's own validation rules and Identity's password hasher in a console program and read the real output.

6.1 Where We Left Off: An Open Door

The first week with the Chapter 5 application went well, and then three small things happened. A new part-time cashier, trying to find the price of ironing, opened *Prices*, typed into the wrong field and saved *Ironing only* at Rp 60 per kg; nobody noticed until the evening. A customer's phone number was entered as `0812 0000 104`, with the letter O instead of a zero, so the WhatsApp message saying her laundry was ready never arrived. And an order with pickup and delivery was saved without a delivery address; the driver stood at the counter asking where to go.

None of these is a programming error in the usual sense. The code did exactly what it was told. What was missing is a set of rules that every request has to pass: the price of a service is at least Rp 1.000; a phone number is an Indonesian mobile number; a delivery needs an address; only the owner may change prices. And one more thing was missing: the application still ran on the owner's laptop, started by hand with `dotnet run`, and stopped whenever the laptop went to sleep.

This chapter adds the rules and moves the application to a server. It helps the cashier, who gets a clear message instead of a silent mistake; the customer, whose number is checked before it is saved; the driver, who always gets an address; and the owner, who alone decides the prices and who can stop worrying about whether the laptop is switched on.

6.2 The Idea in Plain Words

Three ideas carry the chapter.

Validation means checking that the data in a request makes sense before acting on it. The browser can help, by refusing to submit an empty required field, but the browser belongs to the user, and anybody can switch its checks off or send a request without a browser at all. The rule is

therefore simple: every rule is checked on the server; checks in the browser are only a convenience that saves a round trip.

Authentication and authorization are two different questions. Authentication asks *who are you?* and is answered by signing in: the user proves an identity, here with an e-mail address and a password, and receives a cookie that says so on every later request. Authorization asks *are you allowed to do this?* and is answered by rules: a signed-in cashier may record orders but may not change prices. A request that fails authentication is sent to the sign-in page; a request that fails authorization from a signed-in user is refused.

Deployment means running the application somewhere it can serve the shop every day: a computer that is always on, a web server in front of it that handles HTTPS, settings for production that differ from those on the developer's laptop, and a routine for updates and backups.

> **DEFINITION.** *ASP.NET Core Identity* is the membership system of ASP.NET Core: it stores users, hashes and verifies their passwords, locks out an account after repeated failed attempts, manages roles, and signs users in and out with an authentication cookie. A *role* is a named group of users, such as *Owner* or *Cashier*, that authorization rules can refer to. A *policy* is a named authorization rule; the *fallback policy* is the rule applied to every endpoint that states no rule of its own.

Figure 6.1 shows where each of these checks sits when a cashier submits the new-order form. The request passes the middleware of Chapter 3, is routed to `OrdersController.Create`, has its authentication cookie read, and is checked against the authorization rules; only then does model binding build the form object and validation check it, and only then does the action run and decide, from `ModelState.IsValid`, whether to save or to show the form again.

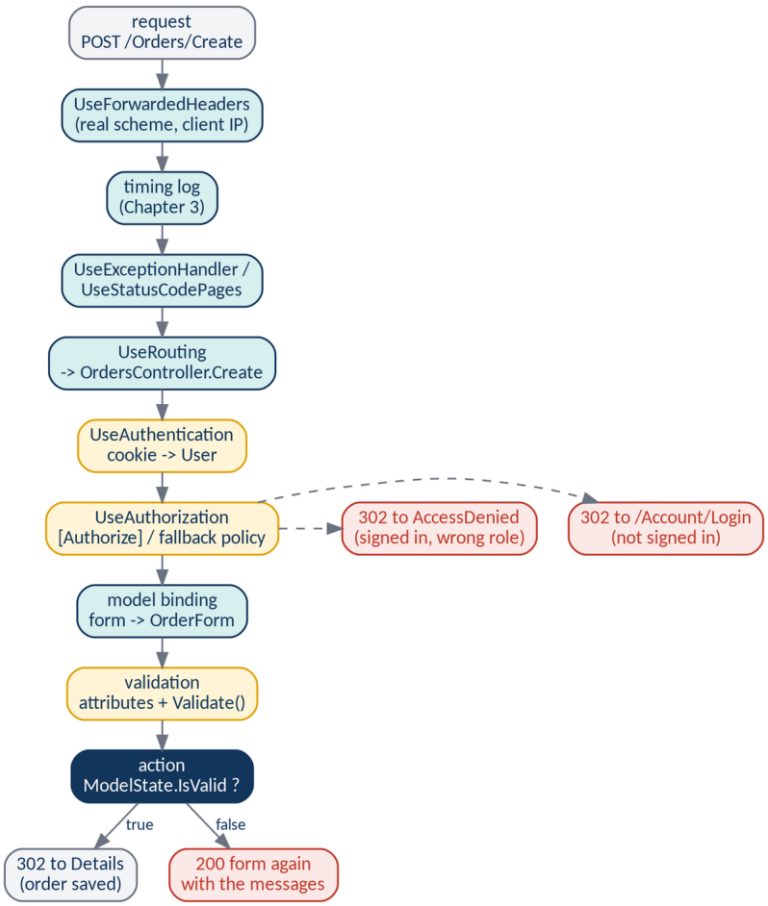


Figure 6.1 The path of a form submission through the Chapter 6 application: authentication and authorization before the controller, model binding and validation before the action, and the four ways the request can end.

6.3 Validation with Data Annotations

6.3.1 Attributes on the model

The simplest way to state a rule in ASP.NET Core is to put a *data annotation attribute* on a property. **Listing 6.1** is the `Customer` class of Chapter 5 with its rules added.

Listing 6.1 `Models/Customer.cs` with validation attributes: a required name of 2 to 80 characters and a required Indonesian mobile number.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryMvc.Models;

public class Customer
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Please enter the customer's name.")]
    [StringLength(80, MinimumLength = 2,
        ErrorMessage = "The name must be 2 to 80 characters long.")]
    public string Name { get; set; } = "";

    [Required(ErrorMessage = "Please enter a phone number.")]
    [IndonesianMobile]
    [Display(Name = "Phone (WhatsApp)")]
    public string Phone { get; set; } = "";

    [StringLength(200)]
    public string? Address { get; set; }

    public DateTime CreatedAt { get; set; } = DateTime.Now;

    public List<Order> Orders { get; set; } = new();
}
```

`[Required]` rejects a missing or empty value; `[StringLength(80, MinimumLength = 2)]` limits the length; `[Display(Name = "Phone (WhatsApp)")]` gives the label that `<label asp-for="Phone">` prints and that error messages use in place of `{0}`. Every attribute accepts an `ErrorMessage` written in the words of the shop, which is always better than the default text. **Table 6.1** lists the attributes used in this chapter.

Table 6.1 The validation attributes used in this chapter.

Attribute	Checks	Used on
`[Required]`	a value is present and not empty	customer name and phone, service description
`[StringLength(max, MinimumLength = min)]`	length of a text	names, addresses, notes
`[Range(min, max)]`	a number lies between two limits	price, quantity, payment amount
`[RegularExpression(pattern)]`	a text matches a pattern	service description ending in `(per kg)`
`[EmailAddress]`	a text looks like an e-mail address	sign-in form
`[EnumDataType(typeof(T))]`	a value is one of the enum's values	payment method
`[IndonesianMobile]` (ours)	an Indonesian mobile number	customer phone

The service gets rules for its description and price (**Listing 6.2**). The regular expression `^.{3,}\(per (kg|item|pair)\)$` accepts a description of at least three characters followed by a space and exactly one of `(per kg)`, `(per item)` or `(per pair)`; since the price rules of Chapter 3 read the unit from that ending, a description without it would silently turn a per-kg service into a per-item one. `[Range(1000, 1_000_000)]` would have stopped the Rp 60 ironing price of Section 6.1.

Listing 6.2 `Models/Service.cs` with rules for the description and the price.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryMvc.Models;

// Chapter 3 read services from a CSV file as read-only records. From
// Chapter 5 on they are rows the owner edits, so the properties have setters.
public class Service
{
    public string Id { get; set; } = "";           // S0001, S0002, ...

    [Required, StringLength(60)]
    [RegularExpression(@"^.{3,} \(per (kg|item|pair)\)$", ErrorMessage =
        "End the description with (per kg), (per item) or (per pair).")]
    public string Description { get; set; } = "";  // "Ironing only (per kg)"

    [Range(1000, 1_000_000,
        ErrorMessage = "The price must be between Rp 1.000 and Rp 1.000.000.")]
    [Display(Name = "Price (rupiah)")]
    public decimal Price { get; set; }             // rupiah per kg/item/pair

    public bool IsActive { get; set; } = true;

    // Worked out from Description, as in Chapter 3; not stored.
    public string Name => Description.Contains(" (")
        ? Description[..Description.LastIndexOf(" (")] : Description;

    public bool PerKg => Description.EndsWith("(per kg)");

    public string Unit => PerKg ? "kg"
        : Description.EndsWith("(per pair)") ? "pair" : "item";
}
```

6.3.2 A validation attribute of our own

No built-in attribute knows what an Indonesian mobile number looks like. `[Phone]` exists, but it accepts almost anything that contains digits. Writing an attribute takes a class derived from `ValidationAttribute` with one method, `IsValid` (**Listing 6.3**).

Listing 6.3 `Models/IndonesianMobileAttribute.cs`: a validation attribute for Indonesian mobile numbers, with a helper that stores every number in one spelling.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryMvc.Models;

// Accepts 0812-0000-0101, 0812 0000 0101, +62 812-0000-0101, 6281200000101.
public sealed class IndonesianMobileAttribute : ValidationAttribute
{
    public IndonesianMobileAttribute()
        : base("{0} must be a mobile number such as 0812-3456-7890.")
    {
    }

    public override bool IsValid(object? value)
    {
        if (value is not string text || text.Length == 0)
        {
            return true;                  // an empty value is [Required]'s job
        }
        var digits = Normalize(text);
        return digits.Length is >= 10 and <= 13
            && digits.StartsWith("08")
            && digits.All(char.IsAsciiDigit);
    }

    // "+62 812-0000-0101" -> "081200000101": one spelling in the database.
    public static string Normalize(string text)
    {
        var s = new string(text.Where(ch => !char.IsWhiteSpace(ch) && ch != '-')
                               .ToArray());
        if (s.StartsWith("+62")) return "0" + s[3..];
        if (s.StartsWith("62")) return "0" + s[2..];
        return s;
    }
}
```

People write the same number in many ways: `0812-0000-0101`, `0812 0000 0101`, `+62 812-0000-0101`. `Normalize` removes spaces and dashes and turns a leading `+62` or `62` into `0`; `IsValid` then accepts the result if it has 10 to 13 digits and starts with `08`. The attribute returns `true` for an empty value on purpose: whether a value is required is `[Required]`'s job, and keeping the two separate lets you reuse `[IndonesianMobile]` on an optional field. The controller calls `Normalize` before saving, so the database holds one spelling per number and the uniqueness check of Chapter 5 cannot be fooled by a space.

6.3.3 Running the rules without a web server

MVC runs these rules during model binding, but they are ordinary .NET: the class `Validator` in `System.ComponentModel.DataAnnotations` runs them anywhere. `ValidationDemo` (**Listing 6.4**) compiles the web app's model classes, as `LinqDemo` did in Chapter 5, validates seven objects, and then looks at what Identity stores instead of a password (Section 6.6.3).

Listing 6.4 `ValidationDemo/Program.cs`: the application's own validation rules and Identity's password hasher, run in a console.

```
using System.ComponentModel.DataAnnotations;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Identity;

// Runs the same checks that MVC model validation runs, and prints them.
static void Check(string title, object model)
{
    var errors = new List<ValidationResult>();
    var ok = Validator.TryValidateObject(model, new ValidationContext(model),
                                         errors, validateAllProperties: true);
    Console.WriteLine($"{title}: {(ok ? "valid" : "INVALID")}");
    foreach (var e in errors)
    {
        Console.WriteLine($"   [{string.Join(",", e.MemberNames)}] "
                          + e.ErrorMessage);
    }
}

Check("1. Customer 'A', phone 12345",
      new Customer { Name = "A", Phone = "12345" });
Check("2. Customer 'Rina Kartika', phone +62 812-0000-0104",
      new Customer { Name = "Rina Kartika", Phone = "+62 812-0000-0104" });
Console.WriteLine("   stored as "
                  + IndonesianMobileAttribute.Normalize("+62 812-0000-0104"));
Check("3. Service 'Carpet' at Rp 500, no unit",
      new Service { Description = "Carpet", Price = 500 });
Check("4. Order: no customer, no service, 0 kg",
      new OrderForm { Quantity = 0 });
Check("5. Order: pickup without an address",
      new OrderForm { CustomerId = 2, ServiceId = "S0001", Quantity = 4,
                      Pickup = true });
Check("6. Order: pickup to Jl. Mulyosari 30",
      new OrderForm { CustomerId = 2, ServiceId = "S0001", Quantity = 4,
                      Pickup = true, DeliveryAddress = "Jl. Mulyosari 30" });
Check("7. Payment of Rp 500",
      new PaymentForm { Amount = 500, Method = PaymentMethod.Cash });
```

Listing 6.4 (Continued) *ValidationDemo/Program.cs: the application's own validation rules and Identity's password hasher, run in a console.*

```
// What Identity stores instead of a password.
var hasher = new PasswordHasher<IdentityUser>();
var user = new IdentityUser { UserName = "kasir1@profitina-laundry.example" };
var hash = hasher.HashPassword(user, "Kasir#2026");
var bytes = Convert.FromBase64String(hash);
int ReadInt(int at) => (bytes[at] << 24) | (bytes[at + 1] << 16)
                     | (bytes[at + 2] << 8) | bytes[at + 3];
var saltLength = ReadInt(9);
Console.WriteLine("8. PasswordHasher<IdentityUser>");
Console.WriteLine($"   stored text      {hash.Length} characters, "
                  + $"starts {hash[..12]}...");
Console.WriteLine($"   format marker    0x{bytes[0]:X2}");
Console.WriteLine($"   PRF              {(KeyDerivationPrf)ReadInt(1)}");
Console.WriteLine($"   iterations       {ReadInt(5):N0}");
Console.WriteLine($"   salt, key bytes  {saltLength}, "
                  + $"{bytes.Length - 13 - saltLength}");
Console.WriteLine("   right password   "
    + hasher.VerifyHashedPassword(user, hash, "Kasir#2026"));
Console.WriteLine("   wrong password   "
    + hasher.VerifyHashedPassword(user, hash, "kasir#2026"));
Console.WriteLine("   same password, second hash equal? "
    + (hasher.HashPassword(user, "Kasir#2026") == hash));

enum KeyDerivationPrf { HMACSHA1, HMACSHA256, HMACSHA512 }
```

This is the real output of `dotnet run`:

```
1. Customer 'A', phone 12345: INVALID
   [Name] The name must be 2 to 80 characters long.
   [Phone] Phone (WhatsApp) must be a mobile number such as 0812-3456-7890.
2. Customer 'Rina Kartika', phone +62 812-0000-0104: valid
   stored as 081200000104
3. Service 'Carpet' at Rp 500, no unit: INVALID
   [Description] End the description with (per kg), (per item) or (per pair).
   [Price] The price must be between Rp 1.000 and Rp 1.000.000.
4. Order: no customer, no service, 0 kg: INVALID
   [CustomerId] Choose a customer.
   [ServiceId] Choose a service.
   [Quantity] The quantity must be between 0.1 and 100.
5. Order: pickup without an address: INVALID
   [DeliveryAddress] Pickup and delivery needs a delivery address.
6. Order: pickup to Jl. Mulyosari 30: valid
7. Payment of Rp 500: INVALID
   [Amount] A payment must be between Rp 1.000 and Rp 10.000.000.
8. PasswordHasher<IdentityUser>
   stored text      84 characters, starts AQAAAAIAAYag...
   format marker    0x01
   PRF              HMACSHA512
   iterations       100,000
   salt, key bytes  16, 32
   right password   Success
   wrong password   Failed
   same password, second hash equal? False
```

Case 1 fails twice, with the messages written in the attributes, and the `[Display]` name appears in the phone message. Case 2 passes and shows that `+62 812-0000-0104` is stored as `081200000104`. Case 3 is the kind of price-list accident of Section 6.1: no unit in the description and a price below the minimum. Cases 4 to 7 concern the form classes of the next section. Note in case 4 that the empty customer and service are caught by `[Range]` and `[Required]` rather than by a database lookup: rules that need no database run before the controller touches the database.

6.3.4 What MVC does with the rules

During model binding MVC runs the same checks and stores every failure in `ModelState` under the name of the property. The action asks `ModelState.IsValid`; the view shows each message with `<span asp-validation-for="Phone">` next to the field and the messages that belong to no single field with `<div asp-validation-summary="ModelOnly">`. That is why the Razor views of Chapter 5 already had these tags: they were waiting for rules.

Rules that need the database, such as “no other customer has this phone number” or “the service still exists and is offered”, cannot be attributes on a model class without dragging the DbContext into the model. They stay in the controller, which adds its own messages to `ModelState` with `AddModelError`, as `CheckPhoneIsFreeAsync` did in Chapter 5. From the view’s point of view there is no difference.

6.4 Form Classes Instead of Entities

6.4.1 Why a separate class

In Chapter 5 the new-order form was bound straight to the `Order` entity, and `[Bind("CustomerId,ServiceId,Quantity,Pickup,DeliveryAddress,Note")]` listed which properties the request was allowed to fill. It works, but it has two weaknesses. The list of names is

a string that the compiler cannot check, so a renamed property silently falls out of it. And the entity is the wrong place for rules that belong to one form: “a delivery needs an address” is a rule of the counter form, not of every order that will ever exist.

A *form class* (often called a view model or input model) solves both. It has exactly the properties the form may send, so overposting is impossible by construction: there is no `Total`, `Status` or `ReceivedAt` to overwrite. And it carries the rules of that form. **Listing 6.5** is the form class for new orders.

Listing 6.5 `Models/OrderForm.cs`: exactly what the new-order form may send, with its rules, including one that involves two fields.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryMvc.Models;

// What the counter form may send: nothing about Total, Status or dates.
public class OrderForm : IValidatableObject
{
    [Range(1, int.MaxValue, ErrorMessage = "Choose a customer.")]
    [Display(Name = "Customer")]
    public int CustomerId { get; set; }

    [Required(ErrorMessage = "Choose a service.")]
    [Display(Name = "Service")]
    public string? ServiceId { get; set; }

    [Range(typeof(decimal), "0.1", "100",
        ParseLimitsInInvariantCulture = true,
        ErrorMessage = "The quantity must be between 0.1 and 100.")]
    [Display(Name = "Quantity (kg, items or pairs)")]
    public decimal Quantity { get; set; } = 1;

    [Display(Name = "Pickup and delivery")]
    public bool Pickup { get; set; }

    [StringLength(200)]
    [Display(Name = "Delivery address")]
    public string? DeliveryAddress { get; set; }

    [StringLength(200)]
    public string? Note { get; set; }

    // A rule about two fields together: delivery needs an address.
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (Pickup && string.IsNullOrWhiteSpace(DeliveryAddress))
        {
            yield return new ValidationResult(
                "Pickup and delivery needs a delivery address.",
                [nameof(DeliveryAddress)]);
        }
    }
}
```

6.4.2 A rule about two fields

“Pickup and delivery needs a delivery address” involves two properties, so it cannot be an attribute on either of them. A class that implements `IValidatableObject` has a `Validate` method in which any rule can be written; it returns a `ValidationResult` for each failure, with the names of the properties the message belongs to. `yield return` produces the results one at a time, so a method with several rules simply has several `if` blocks.

The output of `ValidationDemo` shows two details worth remembering. In case 5 the address is missing and the message is attached to `DeliveryAddress`, so the form shows it next to that field. In case 4, where the quantity is zero and no customer is chosen, the `Validate` method is not run at all: `Validator` checks the attributes first and runs `IValidatableObject.Validate` only when they all pass. Keep that order in mind when you wonder why a message did not appear yet: it will, once the simpler errors are fixed.

`[Range(typeof(decimal), "0.1", "100", ParseLimitsInInvariantCulture = true)]` also deserves a word. Attribute arguments must be constants, and `decimal` values are not constants in C#, so the limits are written as text and converted. `ParseLimitsInInvariantCulture` makes that conversion use a dot as the decimal separator on every computer; without it, a server set to Indonesian would read `"0.1"` with a comma rule and get a different limit.

The payment form (**Listing 6.6**) is smaller: an amount between Rp 1.000 and Rp 10.000.000 and a method that must be one of the enum's values. `[EnumDataType]` matters because model binding happily puts any number into an enum: `Method=7` would otherwise become `(PaymentMethod)7` and be saved.

Listing 6.6 `Models/PaymentForm.cs`: the payment form and its rules.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryMvc.Models;

public class PaymentForm
{
    [Range(1000, 10_000_000,
        ErrorMessage = "A payment must be between Rp 1.000 and Rp 10.000.000.")]
    public int Amount { get; set; }

    [EnumDataType(typeof(PaymentMethod),
        ErrorMessage = "Choose cash, QRIS or transfer.")]
    public PaymentMethod Method { get; set; }
}
```

6.4.3 The controller with form classes

Listing 6.7 is `OrdersController` after the change. `Create` receives an `OrderForm`; by the time the first line runs, the attributes and `Validate` have been checked. The controller adds the two rules that need the database, and only if everything is valid does it build an `Order` entity from the form, field by field. The explicit copy is the price of a form class, and it is worth paying: every value that reaches the database passes through a line you can read.

Listing 6.7 `Controllers/OrdersController.cs` with `OrderForm` and `PaymentForm`.

```
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Controllers;

public class OrdersController(LaundryDbContext db, PriceCalculator calculator)
    : Controller
{
    // GET /Orders or /Orders?status=Queued
    public async Task<IActionResult> Index(OrderStatus? status)
    {
        var query = db.Orders
            .Include(o => o.Customer)
            .Include(o => o.Service)
            .Include(o => o.Payments)
            .AsNoTracking();
        if (status is not null)
        {
            query = query.Where(o => o.Status == status);
        }
        ViewData["Status"] = status;
        return View(await query.OrderByDescending(o => o.ReceivedAt)
                               .ToListAsync());
    }

    public async Task<IActionResult> Details(int id)
    {
        var order = await db.Orders
            .Include(o => o.Customer)
            .Include(o => o.Service)
            .Include(o => o.Payments)
            .AsNoTracking()
            .FirstOrDefaultAsync(o => o.Id == id);
        return order is null ? NotFound() : View(order);
    }

    public async Task<IActionResult> Create(int? customerId)
    {
        await FillListsAsync();
        return View(new OrderForm { CustomerId = customerId ?? 0 });
    }
```

Listing 6.7 (Continued) *Controllers/OrdersController.cs with OrderForm and PaymentForm.*

```
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(OrderForm form)
    {
        // The attributes on OrderForm have been checked already; these two
        // rules need the database, so the controller checks them.
        var service = await db.Services.FindAsync(form.ServiceId ?? "");
        if (form.ServiceId is not null && (service is null || !service.IsActive))
        {
            ModelState.AddModelError(nameof(form.ServiceId),
                "This service is not offered any more.");
        }
        if (form.CustomerId > 0
            && !await db.Customers.AnyAsync(c => c.Id == form.CustomerId))
        {
            ModelState.AddModelError(nameof(form.CustomerId),
                "This customer does not exist.");
        }
        if (!ModelState.IsValid || service is null)
        {
            await FillListsAsync();
            return View(form);
        }

        // The price rules of Chapter 3: minimum 3 kg, whole items, pickup fee.
        var quote = calculator.Calculate(service, form.Quantity, form.Pickup);
        var order = new Order
        {
            CustomerId = form.CustomerId,
            ServiceId = service.Id,
            Quantity = form.Quantity,
            Pickup = form.Pickup,
            DeliveryAddress = form.Pickup ? form.DeliveryAddress : null,
            Note = form.Note,
            Total = (int)decimal.Round(quote.Total,
                                       MidpointRounding.AwayFromZero),
            Status = OrderStatus.Queued,
            ReceivedAt = DateTime.Now
        };
        db.Orders.Add(order);
        await db.SaveChangesAsync();
        TempData["Message"] = $"Order #{order.Id} created: {Money.Rp(order.Total)}."
            + (quote.Notes.Count > 0 ? " " + string.Join(" ", quote.Notes) : "");
        return RedirectToAction(nameof(Details), new { id = order.Id });
    }
```

Listing 6.7 (Continued) *Controllers/OrdersController.cs with OrderForm and PaymentForm.*

```
    // Queued -> Finished -> Departed, each step with its time stamp.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Advance(int id)
    {
        var order = await db.Orders.FindAsync(id);
        if (order is null) return NotFound();
        switch (order.Status)
        {
            case OrderStatus.Queued:
                order.Status = OrderStatus.Finished;
                order.FinishedAt = DateTime.Now;
                break;
            case OrderStatus.Finished:
                order.Status = OrderStatus.Departed;
                order.DepartedAt = DateTime.Now;
                break;
            default:
                TempData["Error"] = "This order has already left the shop.";
                return RedirectToAction(nameof(Details), new { id });
        }
        await db.SaveChangesAsync();
        TempData["Message"] = $"Order #{id} is now {order.Status}.";
        return RedirectToAction(nameof(Details), new { id });
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> AddPayment(int id, PaymentForm payment)
    {
        if (!await db.Orders.AnyAsync(o => o.Id == id)) return NotFound();
        if (!ModelState.IsValid)
        {
            TempData["Error"] = ModelState.Values.SelectMany(v => v.Errors)
                                             .First().ErrorMessage;
            return RedirectToAction(nameof(Details), new { id });
        }
        db.Payments.Add(new Payment
        {
            OrderId = id, Amount = payment.Amount, Method = payment.Method,
            PaidAt = DateTime.Now
        });
        await db.SaveChangesAsync();
        TempData["Message"] = $"Payment of {Money.Rp(payment.Amount)} recorded.";
        return RedirectToAction(nameof(Details), new { id });
    }
```

Listing 6.7 (Continued) *Controllers/OrdersController.cs with OrderForm and PaymentForm.*

```
    async Task FillListsAsync()
    {
        ViewBag.Customers = new SelectList(
            await db.Customers.OrderBy(c => c.Name).ToListAsync(),
            nameof(Customer.Id), nameof(Customer.Name));
        ViewBag.Services = new SelectList(
            await db.Services.Where(s => s.IsActive).OrderBy(s => s.Id)
                             .ToListAsync(),
            nameof(Service.Id), nameof(Service.Description));
    }
}
```

Notice `DeliveryAddress = form.Pickup ? form.DeliveryAddress : null`. A cashier may type an address, change her mind and untick pickup; the address is then not stored, because it no

longer means anything. `AddPayment` receives a `PaymentForm`; if it is invalid, the first message is shown on the order page through `TempData["Error"]`, because the payment form is part of that page rather than a page of its own.

The view (**Listing 6.8**) is the Chapter 5 form with `@model OrderForm`, labels that come from the `[Display]` names, and a message slot under the delivery address.

Listing 6.8 `Views/Orders/Create.cshtml` bound to `OrderForm`.

```
@model OrderForm
@{
    ViewData["Title"] = "New order";
}
<h1>New order</h1>
<form asp-action="Create" method="post" class="form">
    <div asp-validation-summary="ModelOnly" class="errors"></div>
    <label asp-for="CustomerId"></label>
    <select asp-for="CustomerId" asp-items="ViewBag.Customers">
        <option value="0">- choose -</option>
    </select>
    <span asp-validation-for="CustomerId" class="error"></span>

    <label asp-for="ServiceId"></label>
    <select asp-for="ServiceId" asp-items="ViewBag.Services">
        <option value="">- choose -</option>
    </select>
    <span asp-validation-for="ServiceId" class="error"></span>

    <label asp-for="Quantity"></label>
    <input asp-for="Quantity" type="number" step="0.01" min="0.01" />
    <span asp-validation-for="Quantity" class="error"></span>

    <label class="check"><input asp-for="Pickup" /> Pickup and delivery</label>
    <label asp-for="DeliveryAddress"></label>
    <input asp-for="DeliveryAddress" />
    <span asp-validation-for="DeliveryAddress" class="error"></span>

    <label asp-for="Note"></label>
    <input asp-for="Note" />
    <span asp-validation-for="Note" class="error"></span>

    <button type="submit">Create order</button>
    <a asp-action="Index">Cancel</a>
</form>
```

6.5 Checks in the Browser, Too

The browser already does some checking with the HTML attributes that tag helpers write: `type="number"`, `min="0.01"` and `step="0.01"` on the quantity stop a letter or a negative number before the form is sent. ASP.NET Core can do more. For every validation attribute, the tag helpers also write `data-val-*` attributes into the HTML; for example, a `[Required]` property gets `data-val="true" data-val-required="Please enter a phone number."`. The small JavaScript libraries *jQuery Validation* and *jQuery Validation Unobtrusive* read those attributes and show the same messages as the user types, without a round trip. The project template of `dotnet new mvc` includes both libraries in `wwwroot/lib` and a partial view `_ValidationScriptsPartial.cshtml` that loads them; a form page switches them on with `@section Scripts { <partial name="_ValidationScriptsPartial" /> }` [1].

The Profitina Laundry project deliberately ships without them, as in Chapter 3, to keep the page free of frameworks and to make the point of Section 6.2 visible: every rule of this chapter is

checked on the server, and a page with client-side checks only saves the user a moment. Rules that need the database, and `IValidatableObject` rules, are checked on the server only in any case.

6.6 Identity: Who Is at the Counter?

6.6.1 What Identity provides

Writing sign-in yourself looks easy and is not: passwords must never be stored, only a slow salted hash of them; failed attempts must be counted and the account locked for a while; the cookie must be signed and encrypted so that it cannot be forged; a changed password must invalidate old cookies. ASP.NET Core Identity does all of this and has done it for years [2]. Most of Identity is part of the shared framework that every ASP.NET Core app already has; the piece that stores users and roles with EF Core is one more NuGet package, `Microsoft.AspNetCore.Identity.EntityFrameworkCore`, added like the EF Core packages in Chapter 5:

```
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore \
    --version 10.0.12
```

The DbContext then derives from `IdentityDbContext<IdentityUser>` instead of `DbContext`, which adds the Identity tables (`AspNetUsers`, `AspNetRoles`, `AspNetUserRoles` and a few more) to the model. **Listing 6.9** shows the changed lines; the call `base.OnModelCreating(model)` must come first, because it configures those tables.

Listing 6.9 `Data/LaundryDbContext.cs` deriving from `IdentityDbContext`, so that users and roles live in the same database as the orders.

```
using LaundryMvc.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Data;

public class LaundryDbContext(DbContextOptions<LaundryDbContext> options)
    : IdentityDbContext<IdentityUser>(options)
{
    public DbSet<Service> Services => Set<Service>();
    public DbSet<Outlet> Outlets => Set<Outlet>();
    public DbSet<Customer> Customers => Set<Customer>();
    public DbSet<Order> Orders => Set<Order>();
    public DbSet<Payment> Payments => Set<Payment>();

    protected override void OnModelCreating(ModelBuilder model)
    {
        base.OnModelCreating(model);      // the AspNet* tables of Identity

        model.Entity<Service>(e =>
        {
            e.Property(s => s.Id).HasMaxLength(5);
            e.Property(s => s.Description).HasMaxLength(60);
            e.Property(s => s.Price).HasPrecision(10, 2);
        });

        model.Entity<Outlet>(e =>
        {
            e.Property(o => o.Id).HasMaxLength(3);
            e.Property(o => o.Name).HasMaxLength(60);
            e.Property(o => o.Address).HasMaxLength(120);
            e.Property(o => o.Phone).HasMaxLength(20);
        });

        model.Entity<Customer>(e =>
        {
            e.Property(c => c.Name).HasMaxLength(80);
            e.Property(c => c.Phone).HasMaxLength(20);
            e.Property(c => c.Address).HasMaxLength(200);
            e.HasIndex(c => c.Phone).IsUnique();
        });
```

Listing 6.9 (Continued) *Data/LaundryDbContext.cs deriving from IdentityDbContext, so that users and roles live in the same database as the orders.*

```
        model.Entity<Order>(e =>
        {
            e.Property(o => o.Quantity).HasPrecision(6, 2);
            e.Property(o => o.Status).HasConversion<string>().HasMaxLength(12);
            e.Property(o => o.DeliveryAddress).HasMaxLength(200);
            e.Property(o => o.Note).HasMaxLength(200);
            e.HasIndex(o => o.Status);
            e.HasOne(o => o.Customer).WithMany(c => c.Orders)
             .HasForeignKey(o => o.CustomerId)
             .OnDelete(DeleteBehavior.Restrict);
            e.HasOne(o => o.Service).WithMany()
             .HasForeignKey(o => o.ServiceId)
             .OnDelete(DeleteBehavior.Restrict);
        });

        model.Entity<Payment>(e =>
        {
            e.Property(p => p.Method).HasConversion<string>().HasMaxLength(10);
            e.HasOne(p => p.Order).WithMany(o => o.Payments)
             .HasForeignKey(p => p.OrderId)
             .OnDelete(DeleteBehavior.Cascade);
        });
    }
}
```

A new migration (`dotnet ef migrations add AddIdentity`, or `InitialCreate` in a fresh project) creates the tables. Keeping users in the same SQLite file as the orders means one file to back up, and one place where a customer's data and the staff's accounts are protected together.

6.6.2 Program.cs with Identity, authorization and deployment settings

Listing 6.10 is the new `Program.cs`. Read it in the order of its comments.

Listing 6.10 `Program.cs`: Identity, the fallback authorization policy, and the settings that deployment needs.

```
using System.Diagnostics;
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.HttpOverrides;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// 1. Services.
builder.Services.AddControllersWithViews();
builder.Services.AddSingleton<PriceCalculator>();

var dataFolder = Path.Combine(builder.Environment.ContentRootPath, "data");
builder.Services.AddDbContext<LaundryDbContext>(options =>
    options.UseSqlite(builder.Configuration.GetConnectionString("Laundry"))
           .UseSeeding((db, _) =>
               SeedData.Apply((LaundryDbContext)db, dataFolder))
           .UseAsyncSeeding((db, _, ct) =>
               SeedData.ApplyAsync((LaundryDbContext)db, dataFolder, ct)));
builder.Services.AddScoped<ILaundryCatalog, EfCatalog>();

// Chapter 6: users, hashed passwords, roles and lockout, stored by EF Core.
builder.Services
    .AddIdentity<IdentityUser, IdentityRole>(options =>
    {
        options.Password.RequiredLength = 8;
        options.User.RequireUniqueEmail = true;
        options.Lockout.MaxFailedAccessAttempts = 5;
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(15);
    })
    .AddEntityFrameworkStores<LaundryDbContext>();

builder.Services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/Account/Login";
    options.AccessDeniedPath = "/Account/AccessDenied";
    options.ExpireTimeSpan = TimeSpan.FromHours(10);   // one working day
    options.SlidingExpiration = true;
});
```

Listing 6.10 (Continued) *Program.cs: Identity, the fallback authorization policy, and the settings that deployment needs.*

```
// Every page needs a signed-in user unless it says [AllowAnonymous].
builder.Services.AddAuthorizationBuilder()
    .SetFallbackPolicy(new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build());

// Deployment helpers: a health endpoint, cookie keys that survive restarts,
// and the real client address and scheme when running behind a proxy.
builder.Services.AddHealthChecks();
builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new DirectoryInfo(
        builder.Configuration["DataProtection:KeysPath"]
        ?? Path.Combine(builder.Environment.ContentRootPath, "keys")));
builder.Services.Configure<ForwardedHeadersOptions>(options =>
    options.ForwardedHeaders = ForwardedHeaders.XForwardedFor
                             | ForwardedHeaders.XForwardedProto);

var app = builder.Build();

await IdentitySeeder.SeedAsync(app.Services);

// 2. Pipeline.
app.UseForwardedHeaders();
app.Use(async (context, next) =>
{
    var clock = Stopwatch.StartNew();
    await next(context);
    app.Logger.LogInformation("{Method} {Path}{Query} -> {Status} in {Ms} ms",
        context.Request.Method, context.Request.Path,
        context.Request.QueryString, context.Response.StatusCode,
        clock.ElapsedMilliseconds);
});

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/error/500");
    app.UseHsts();
}
app.UseStatusCodePagesWithReExecute("/error/{0}");

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
```

Listing 6.10 (Continued) *Program.cs: Identity, the fallback authorization policy, and the settings that deployment needs.*

```
app.MapStaticAssets().AllowAnonymous();
app.MapHealthChecks("/health").AllowAnonymous();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}")
    .WithStaticAssets();

app.Run();
```

- `AddIdentity<IdentityUser, IdentityRole>` registers the user manager, the role manager and the sign-in manager, and `AddEntityFrameworkStores<LaundryDbContext>()` tells them to keep their data through our DbContext. The options require passwords of at least eight characters, one account per e-mail address, and a lockout of 15 minutes after five failed attempts.

- `ConfigureApplicationCookie` sets where users are sent to sign in and where a refusal is shown, and makes the cookie last ten hours, a working day, renewed while the user is active.
- `SetFallbackPolicy(…RequireAuthenticatedUser())` is the most important line of the chapter. It turns the default around: every endpoint now requires a signed-in user unless it says otherwise. A new controller that someone adds next year is protected even if they forget the attribute. The public pages of Chapter 3 opt out explicitly, as the next section shows.
- In the pipeline, `UseAuthentication()` reads the cookie and fills `User`, and `UseAuthorization()` applies the rules. Both come after `UseRouting()`, because authorization needs to know which endpoint was chosen and what attributes it has, and before the endpoints are executed.
- `MapStaticAssets().AllowAnonymous()` keeps the stylesheet reachable for visitors who are not signed in; without it, the fallback policy would also protect `site.css`, and the sign-in page would appear unstyled.

The remaining lines (forwarded headers, data protection, health checks, HSTS) belong to deployment and are explained in Section 6.8.

6.6.3 What is stored instead of a password

Part 8 of the `ValidationDemo` output shows what Identity's `PasswordHasher<IdentityUser>` produces from the password `Kasir#2026`: an 84-character text, the Base64 form of 61 bytes. The first byte, `0x01`, marks the format; then come the pseudo-random function (HMAC-SHA512), the iteration count (100,000), the salt length (16 bytes), the salt, and a 32-byte key derived with PBKDF2. Verifying the right password gives `Success` and a password that differs by one capital letter gives `Failed`. The last line is the most instructive: hashing the same password a second time gives a *different* text, because every hash gets a new random salt. Two users with the same password therefore have different hashes, and a table of precomputed hashes is useless to an attacker.

The 100,000 iterations are what makes the hash slow on purpose: checking one guess takes a noticeable fraction of a millisecond, which is nothing for a cashier who signs in once a day and a great deal for someone trying billions of guesses. The OWASP Password Storage Cheat Sheet recommends PBKDF2-HMAC-SHA512 only with a high iteration count, and prefers Argon2id where it is available [3]; Identity's hasher is replaceable through dependency injection if a shop's policy requires another algorithm. Whatever the algorithm, the rule of the chapter stands: the application stores and compares hashes, never passwords.

6.6.4 Roles and the first accounts

The shop has two roles, named once as constants (**Listing 6.11**) so that a typing mistake in a role name becomes a compiler error rather than a door that silently never opens.

Listing 6.11 `Models/AppRoles.cs`: the two roles of the shop.

```
namespace LaundryMvc.Models;

public static class AppRoles
{
    public const string Owner = "Owner";      // prices, deleting customers
    public const string Cashier = "Cashier";  // orders and payments
}
```

Somebody has to exist before anybody can sign in. `IdentitySeeder` (**Listing 6.12**) runs once at start-up, right after `builder.Build()`: it applies pending migrations (which also runs the seeding of Chapter 5), creates the two roles if they are missing, and creates every account listed under

`SeedUsers` in the configuration that does not exist yet. `UserManager.CreateAsync(user, password)` checks the password against the rules of `AddIdentity` and stores its hash; if the password is too short, start-up stops with a clear message instead of creating an account nobody can use.

Listing 6.12 `Data/IdentitySeeder.cs`: migrations, roles and the first accounts at start-up.

```
using LaundryMvc.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Data;

// Brings the database up to date and makes sure the roles and the accounts
// listed under "SeedUsers" in the configuration exist.
public static class IdentitySeeder
{
    public static async Task SeedAsync(IServiceProvider services)
    {
        using var scope = services.CreateScope();
        var sp = scope.ServiceProvider;

        var db = sp.GetRequiredService<LaundryDbContext>();
        await db.Database.MigrateAsync();     // also runs UseAsyncSeeding

        var roles = sp.GetRequiredService<RoleManager<IdentityRole>>();
        foreach (var role in new[] { AppRoles.Owner, AppRoles.Cashier })
        {
            if (!await roles.RoleExistsAsync(role))
            {
                await roles.CreateAsync(new IdentityRole(role));
            }
        }

        var users = sp.GetRequiredService<UserManager<IdentityUser>>();
        var config = sp.GetRequiredService<IConfiguration>();
        foreach (var account in config.GetSection("SeedUsers").GetChildren())
        {
            var email = account["Email"];
            var password = account["Password"];
            var role = account["Role"];
            if (email is null || password is null || role is null) continue;
            if (await users.FindByEmailAsync(email) is not null) continue;
```

Listing 6.12 (Continued) *Data/IdentitySeeder.cs: migrations, roles and the first accounts at start-up.*

```
            var user = new IdentityUser
            {
                UserName = email, Email = email, EmailConfirmed = true
            };
            var result = await users.CreateAsync(user, password);  // hashes it
            if (!result.Succeeded)
            {
                throw new InvalidOperationException(string.Join("; ",
                    result.Errors.Select(e => e.Description)));
            }
            await users.AddToRoleAsync(user, role);
        }
    }
}
```

The demo accounts live in `appsettings.Development.json` (**Listing 6.13**), a file that is read only when the environment is *Development*, which is what `dotnet run` uses through

`launchSettings.json`. On the production server the file is ignored, so the demo passwords can never become real ones. The first real owner account is created there from environment variables once (Section 6.8.4).

Listing 6.13 `appsettings.Development.json`: demo accounts for the developer's computer only.

```
{
  "SeedUsers": [
    { "Email": "owner@profitina-laundry.example", "Password": "Owner#2026",
      "Role": "Owner" },
    { "Email": "kasir1@profitina-laundry.example", "Password": "Kasir#2026",
      "Role": "Cashier" }
  ]
}
```

6.7 Signing In and Keeping Doors Closed

6.7.1 The sign-in page

`AccountController` (**Listing 6.14**) has four actions: the sign-in form, its submission, signing out and the refusal page. The whole class is marked `[AllowAnonymous]`, since a sign-in page that requires signing in would be a locked door with the key inside.

Listing 6.14 `Controllers/AccountController.cs`: sign in, sign out and the refusal page.

```
using LaundryMvc.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace LaundryMvc.Controllers;

[AllowAnonymous]
public class AccountController(SignInManager<IdentityUser> signIn) : Controller
{
    public IActionResult Login(string? returnUrl) =>
        View(new LoginForm { ReturnUrl = returnUrl });

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginForm form)
    {
        if (!ModelState.IsValid) return View(form);

        var result = await signIn.PasswordSignInAsync(form.Email, form.Password,
            form.RememberMe, lockoutOnFailure: true);
        if (result.Succeeded)
        {
            // Only ever return to a page of this site (no open redirect).
            return Url.IsLocalUrl(form.ReturnUrl)
                ? LocalRedirect(form.ReturnUrl)
                : RedirectToAction("Index", "Counter");
        }
        ModelState.AddModelError(string.Empty, result.IsLockedOut
            ? "Too many failed attempts. Please wait 15 minutes."
            : "The e-mail or the password is wrong.");
        return View(form);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Logout()
    {
        await signIn.SignOutAsync();
        return RedirectToAction("Index", "Home");
    }

    public IActionResult AccessDenied() => View();
}
```

`PasswordSignInAsync(email, password, rememberMe, lockoutOnFailure: true)` finds the user, verifies the password against the stored hash, counts a failure towards lockout, and on success writes the authentication cookie. With `RememberMe` the cookie survives closing the browser; without it, it lasts only for the browser session. The error message is deliberately the same for an unknown e-mail and a wrong password: telling an attacker which e-mail addresses exist would help them.

Two lines protect against subtler attacks. When the fallback policy sends an anonymous user to the sign-in page, it adds `?ReturnUrl=/Prices` so that the user returns there afterwards. An attacker could send a link with `ReturnUrl=https://evil.example/`, hoping the shop's own sign-in page would forward the cashier to a copy of it; `Url.IsLocalUrl` accepts only addresses inside this site. And sign-out is a `POST` with an anti-forgery token, so that another site cannot sign the cashier out with an image tag. **Listing 6.15** is the sign-in view; `autocomplete="current-password"` lets password managers help.

Listing 6.15 `Views/Account/Login.cshtml`: the staff sign-in form.

```
@model LoginForm
@{
    ViewData["Title"] = "Staff sign-in";
}
<h1>Staff sign-in</h1>
<form asp-action="Login" method="post" class="form">
    <div asp-validation-summary="ModelOnly" class="errors"></div>
    <input type="hidden" asp-for="ReturnUrl" />

    <label asp-for="Email"></label>
    <input asp-for="Email" autocomplete="username" />
    <span asp-validation-for="Email" class="error"></span>

    <label asp-for="Password"></label>
    <input asp-for="Password" autocomplete="current-password" />
    <span asp-validation-for="Password" class="error"></span>

    <label class="check"><input asp-for="RememberMe" /> Keep me signed in</label>
    <button type="submit">Sign in</button>
</form>
```

6.7.2 Who may do what

With the fallback policy in place, the rules are written as attributes. **Table 6.2** summarizes them.

Table 6.2 Who may use which pages after this chapter.

Pages	Anonymous visitor	Cashier	Owner
Home, `/services`, price quote, `/outlets`, error pages	yes	yes	yes
`/health`, stylesheet	yes	yes	yes
Counter, Orders, payments	sign-in page	yes	yes
Customers: list, add, edit	sign-in page	yes	yes
Customers: delete	sign-in page	refused	yes
Prices	sign-in page	refused	yes

The three controllers of Chapter 3 get one line each, `[AllowAnonymous]`, above the class: the shop's public face stays public. `PricesController` gets `[Authorize(Roles = AppRoles.Owner)]` on the class, so every action in it requires the Owner role. In `CustomersController`, only the two delete actions get the attribute, since cashiers do need to add and edit customers. A cashier who types `/Prices` into the address bar is signed in but not an owner; authorization fails, and the cookie handler redirects to `/Account/AccessDenied`, the page of **Listing 6.16**.

Listing 6.16 `Views/Account/AccessDenied.cshtml`: what a signed-in user sees when a page is not for their role.

```
@{
    ViewData["Title"] = "Not allowed";
}
<h1>Not allowed</h1>
<p>You are signed in as @User.Identity?.Name, but this page is for the
   shop owner. Ask the owner to make this change.</p>
<p><a asp-controller="Counter" asp-action="Index">Back to the counter</a></p>
```

Hiding a link is not protection, but it is good manners. The layout shows the staff links only to signed-in users and the *Prices* link only to owners, with `User.Identity?.IsAuthenticated` and `User.IsInRole(AppRoles.Owner)`. The attribute on the controller is what actually protects the page; the `if` in the layout only avoids showing a cashier a link that would lead to a refusal.

6.7.3 Policies: rules with names

Roles answer the question "is this user an owner?". Real shops soon have rules that are not a role: *only staff of the Keputih outlet may see Keputih's orders*, *refunds above Rp 100.000 need the owner*, *cashiers may record payments only while the shop is open*. ASP.NET Core expresses such rules as *policies*, named rules registered once in `Program.cs` and referred to by name [4]. The role rule of this chapter can itself be written as a policy:

```
builder.Services.AddAuthorizationBuilder().AddPolicy("OwnerOnly", p => p.RequireRole(AppRoles.Owner));
```

and a controller then says `[Authorize(Policy = "OwnerOnly")]`. The advantage is that the rule lives in one place. When the owner later decides that a trusted senior cashier may also edit prices, the policy becomes `p.RequireRole(AppRoles.Owner, "SeniorCashier")` and not one controller changes. A policy can also require a *claim*, a piece of information stored with the user, such as `p.RequireClaim("outlet", "O02")`; Identity stores claims in the `AspNetUserClaims` table, and `UserManager.AddClaimAsync` adds them. For rules that need data from the request or the database, such as the opening hours, a policy can be given a custom *requirement* and a *handler* class that decides; that is beyond this chapter, but the attribute on the controller would look exactly the same.

The fallback policy of Section 6.6.2 is simply the policy used when an endpoint names none. Together, the fallback policy, `[AllowAnonymous]` and named policies cover everything a small shop needs: closed by default, open where you say so, and specific rules with names that document themselves.

6.8 Deployment: From Laptop to Server

6.8.1 Environments and settings

ASP.NET Core reads its configuration in layers: `appsettings.json` first, then `appsettings.{Environment}.json`, then environment variables, then command-line arguments, each layer overriding the one before [5]. The environment is named by the variable `ASPNETCORE_ENVIRONMENT`: *Development* on the laptop (set in `launchSettings.json`), *Production* on the server (the default when the variable is not set). **Listing 6.17** is the production layer.

Listing 6.17 `appsettings.Production.json`: the database and the cookie keys in a fixed folder, and quieter logging.

```
{
  "ConnectionStrings": {
    "Laundry": "Data Source=/var/lib/laundrymvc/laundry.db"
  },
  "DataProtection": {
    "KeysPath": "/var/lib/laundrymvc/keys"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

It moves the database file to `/var/lib/laundrymvc`, a folder outside the program folder, so that installing a new version never touches the data. It keeps the cookie keys there too (Section 6.8.6)

and logs only warnings, plus the start and stop messages. On Windows the same values can be given as environment variables, where a double underscore stands for the colon: `ConnectionStrings__Laundry=Data Source=D:\Laundry\laundry.db`.

In Production, `Program.cs` also switches on `UseExceptionHandler`, so that an error shows the friendly page of Chapter 3 instead of a stack trace, and `UseHsts`, which tells browsers to use only HTTPS for the site from then on.

6.8.2 Publishing

`dotnet publish -c Release -o <folder>` compiles the application in Release mode and copies into one folder everything needed to run it: the application's DLL, the DLLs of its NuGet packages, the views compiled into the DLL, `wwwroot`, the settings files and a `.runtimeconfig.json` that names the .NET runtime to use [6]. The web app could not be restored where this book was written, so here is `ValidationDemo` published the same way; the lines are real, with the start of the paths shortened:

```
Determining projects to restore...
Restored …/aspnet/ValidationDemo/ValidationDemo.csproj (in 185 ms).
ValidationDemo -> …/aspnet/ValidationDemo/bin/Release/net10.0/ValidationDemo.dll
ValidationDemo -> …/aspnet/ValidationDemo/publish/
```

The folder `publish/` then contains `ValidationDemo.dll`, `ValidationDemo.deps.json`, `ValidationDemo.pdb`, `ValidationDemo.runtimeconfig.json` and the `ValidationDemo` launcher, and `dotnet ValidationDemo.dll` in that folder runs the program. The web app's `publish` folder looks the same with more files: `LaundryMvc.dll`, the EF Core and SQLite DLLs, `runtimes/` with the native SQLite library for each operating system, `wwwroot/`, `data/` and the `appsettings*.json` files.

This is a *framework-dependent* publish: the server needs the ASP.NET Core 10 runtime installed, and the folder stays small. Adding `--self-contained -r linux-x64` (or `win-x64`) produces a *self-contained* folder with the runtime inside, larger but independent of what is installed on the server. The .NET SDK can also build a container image directly with `dotnet publish /t:PublishContainer`, without writing a Dockerfile, for shops that run containers [6].

6.8.3 Migrations on the server

The database on the server must be brought to the current migration before the new version serves requests. This project does it at start-up: `IdentitySeeder` calls `MigrateAsync()`. For one small shop with one instance of the app, that is simple and safe. For larger systems Microsoft recommends applying migrations as a separate step of the release, for example with a *migration bundle*, a single executable produced by `dotnet ef migrations bundle` that applies the migrations to the database named in its connection string [7]. Either way, take a copy of `laundry.db` before applying a migration; a migration that drops a column cannot bring the data back.

6.8.4 A Linux server with Nginx and systemd

Figure 6.2 shows the usual arrangement on Linux. Kestrel, the web server built into every ASP.NET Core application, listens only on `127.0.0.1:5060`, so nobody outside the server can reach it directly. Nginx listens on port 443, holds the TLS certificate, and forwards requests to Kestrel. systemd starts the application when the server boots and restarts it if it stops [8].

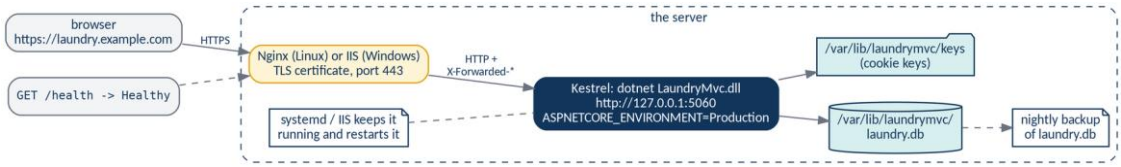


Figure 6.2 The laundry application on a Linux server: Nginx in front with the certificate, Kestrel behind it on the loopback address, systemd keeping it running, and the data folder with the database, the cookie keys and a nightly backup.

Listing 6.18 is the systemd unit. It runs the app as an unprivileged user `laundry`, sets the environment to Production and the listening address, and restarts the app five seconds after any exit.

Listing 6.18 `deploy/laundrymvc.service`: the systemd unit that keeps the app running.

```
# /etc/systemd/system/laundrymvc.service  (Linux server)
[Unit]
Description=Profitina Laundry (ASP.NET Core)
After=network.target

[Service]
WorkingDirectory=/opt/laundrymvc
ExecStart=/usr/bin/dotnet /opt/laundrymvc/LaundryMvc.dll
Restart=always
RestartSec=5
User=laundry
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=ASPNETCORE_URLS=http://127.0.0.1:5060
Environment=DOTNET_NOLOGO=1

[Install]
WantedBy=multi-user.target
```

Listing 6.19 is the Nginx site. The three `proxy_set_header` lines pass on the original host name, the visitor's address and the original scheme (`https`). `certbot --nginx` then obtains a free Let's Encrypt certificate and adds the `listen 443 ssl` lines.

Listing 6.19 `deploy/nginx-laundrymvc.conf`: Nginx forwarding to Kestrel with the forwarded headers.

```
# /etc/nginx/sites-available/laundrymvc  (the TLS lines are added by certbot)
server {
    listen 80;
    server_name laundry.example.com;

    location / {
        proxy_pass         http://127.0.0.1:5060;
        proxy_http_version 1.1;
        proxy_set_header   Host $host;
        proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
    }
}
```

Behind a proxy, Kestrel sees every request coming from `127.0.0.1` over plain HTTP. `UseForwardedHeaders()`, configured in `Program.cs` for `X-Forwarded-For` and `X-Forwarded-Proto`, puts the real visitor address and scheme back into the request, and it must run first in the pipeline so that everything after it sees the right values [9]. Without it, the authentication cookie's *secure* flag, redirects and the lockout logs would all believe the site runs on HTTP from localhost. By default the middleware trusts forwarded headers only from the loopback address, which is exactly where Nginx is.

The installation, in the order an administrator would type it on Ubuntu:

```
sudo apt install aspnetcore-runtime-10.0 nginx sqlite3
sudo useradd --system --home /var/lib/laundrymvc laundry
sudo mkdir -p /opt/laundrymvc /var/lib/laundrymvc
sudo chown laundry /var/lib/laundrymvc
sudo cp -r publish/* /opt/laundrymvc/
sudo cp deploy/laundrymvc.service /etc/systemd/system/
sudo systemctl edit laundrymvc      # add the first owner, see below
sudo systemctl enable --now laundrymvc
```

The `systemctl edit` step adds three lines to the unit for the first start only, `Environment=SeedUsers__0__Email=…`, `…__Password=…` and `…__Role=Owner`; after the first start has created the owner, remove them again. Then enable the Nginx site, run `certbot --nginx -d laundry.example.com`, and open the site.

6.8.5 Windows with IIS

A shop that already runs Windows Server can host the app in IIS. Install the *.NET 10 Hosting Bundle*, which contains the runtime and the ASP.NET Core Module; create a site whose physical path is the published folder; set its application pool to *No Managed Code*. The `web.config` that `dotnet publish` writes tells IIS to hand requests to the app through the module [10]. Settings such as the connection string go into environment variables of the application pool or into `appsettings.Production.json` with a Windows path. IIS itself holds the certificate and passes the original scheme on, so the forwarded-headers configuration does no harm.

6.8.6 The small things that make it last

- **Cookie keys.** Identity's cookie is encrypted with keys from ASP.NET Core Data Protection. If the keys are kept only in memory, every restart invalidates every cookie and all staff are signed out. `PersistKeysToFileSystem` in `Program.cs` stores them in `DataProtection:KeysPath`; the folder must be writable by the app and must not be deleted [11].
- **A health endpoint.** `AddHealthChecks()` and `MapHealthChecks("/health")` answer `Healthy` with status 200 when the app runs. A monitoring service, or the release script, asks `/health` after an update and alarms if the answer does not come.
- **Backups.** `sqlite3 /var/lib/laundrymvc/laundry.db ".backup /var/backups/laundry-$(date +%F).db"` in a nightly cron job makes a consistent copy even while the app runs. Copy the backups off the server as well.
- **Updates.** Publish the new version, stop the service, back up the database, copy the files, start the service, check `/health`. Six steps, always in that order.

6.8.7 A release checklist

Deployment goes wrong through forgotten steps, not difficult ones. **Table 6.3** is the checklist to keep next to the server, in the order in which the steps are done.

Table 6.3 The release checklist for the laundry server.

Step	Command or place	Why
1. Build and test on the laptop	`dotnet build`, click through Section 6.9	nothing broken reaches the shop
2. Publish	`dotnet publish -c Release -o publish`	one folder with everything
3. Back up the database	`sqlite3 … ".backup …"`	a migration cannot be undone with data
4. Stop the service	`sudo systemctl stop laundrymvc`	no half-written requests
5. Copy the files	`sudo cp -r publish/* /opt/laundrymvc/`	the new version
6. Start the service	`sudo systemctl start laundrymvc`	migrations run at start-up
7. Check health	`curl https://laundry.example.com/health`	`Healthy`, or roll back
8. Sign in once	the sign-in page	cookies, keys and roles still work

If step 7 or 8 fails, stop the service, copy back the previous `publish` folder and the backup of step 3, and start again: the shop is back where it was ten minutes earlier, and there is time to find the mistake.

6.9 Running It

On your computer, run the commands of the chapter's README: `dotnet restore`, `dotnet ef migrations add InitialCreate` and `dotnet run`, then open `http://localhost:4060`.

1. The home page, the services and the outlets open without signing in. Choose **Counter**; the browser is sent to `/Account/Login?ReturnUrl=%2FCounter`.
2. Sign in as the cashier with a wrong password five times; after the fifth attempt the page reports that the account is locked, and for the next 15 minutes even the right password is refused. Sign in as the owner instead.
3. As the owner, open **Prices**, edit *Ironing only*, remove `(per kg)` from its description and set the price to 60. The form comes back with the two messages of case 3 of the demo.
4. Add a customer with phone `+62 812-0000-0104`; the customer page shows `081200000104`. Add another with `0812 0000 1O4`; the message of `[IndonesianMobile]` appears.
5. Create an order with pickup and no address; the message appears under the address field.
6. Sign out and sign in as the cashier. The header has no Prices link; type `/Prices` into the address bar and the refusal page appears. Try to delete a customer; refused as well.
7. Open `http://localhost:4060/health` in a private window: `Healthy`, without signing in.
8. Stop the app and start it again: you are still signed in, because the cookie keys are in the `keys` folder.

Where this book was written, NuGet could not be reached, so the web app could not be restored or run there; its C# code and Razor views were compiled against compile-only stand-ins for the EF Core and Identity-EF APIs, and the rest (Identity itself, authorization, data protection, health checks) comes from the shared framework and was compiled for real. `ValidationDemo` was built, run and published, and its outputs above are real. The tour above follows from the code and from the documented behavior of each feature.

6.10 Making It Better

- **Limit sign-in attempts per address, too.** Lockout protects one account; an attacker trying one password against many accounts is not slowed by it. The rate-limiting middleware built into ASP.NET Core (`AddRateLimiter` with a fixed-window limiter on the login action) adds a limit per IP address.
- **Passkeys.** ASP.NET Core 10 added passkey (WebAuthn) support to Identity, so staff can sign in with the fingerprint reader of a phone or laptop instead of a password [12]. For a counter where several people share one computer, a password plus lockout is still the practical start.
- **Record who did what.** With `User.Identity.Name` available in every action, adding a `CreatedBy` column to orders and payments is a small migration, and it answers the question "who entered this payment?" that every shop eventually asks.
- **Use a real e-mail sender for password resets.** Identity can generate password-reset tokens; sending them needs an `IEmailSender` implementation, for example through the shop's e-mail provider. Until then, the owner resets a cashier's password by hand.

6.11 Using It for Real

- **Your roles.** A shop with a separate washer role adds `Washer` to `AppRoles`, gives the washer access to the order list and the *Mark finished* button only, and protects the payment actions with `[Authorize(Roles = AppRoles.Owner + "," + AppRoles.Cashier)]`.
- **Your rules.** Put your own rules where they belong: a rule about one value in an attribute on the form class, a rule about several values in `Validate`, a rule that needs the database in the controller. Write every message in the words your staff use at the counter.
- **Your server.** A small cloud virtual machine with 1 GB of memory runs this application comfortably; so does an old office PC with Linux in the back room, with a Cloudflare tunnel or the shop's router forwarding HTTPS to it.
- **Personal data, once more.** Staff accounts and customer records are personal data under UU PDP (Law No. 27 of 2022). Hashed passwords, lockout, access by role and HTTPS are part of the "appropriate technical measures" the law asks for; so is a backup that is itself protected. Give each person their own account, never a shared one, so that the records of who did what mean something.

6.12 Profitina in Practice

Profitina's production platform went through each step of this chapter, in Python rather than C#. Its account system hashes passwords with Argon2id, the algorithm OWASP prefers [3], asks for explicit consent, limits login attempts, and lets every user export their data, change their e-mail address and password, and delete their account completely, which is how it meets UU PDP 27/2022 in self-service. Its production host is a Windows 11 machine running the application as a service under NSSM (the Windows counterpart of the systemd unit in Section 6.8.4) behind a Cloudflare Tunnel that plays the role of Nginx and holds the certificate. And its release process is gated on health: a new version is staged, started, and only reported as released after it answers `/api/health`, the same idea as the `/health` endpoint of this chapter and the update routine of Section 6.8.6.

6.13 Common Mistakes, and How to Avoid Them

- **Trusting the browser.** A rule checked only in JavaScript is not a rule. Check everything on the server; add client-side checks only for comfort.
- **Forgetting `if (!ModelState.IsValid)`.** Validation fills `ModelState`, but it does not stop the action. An action that saves without asking saves invalid data.
- **Binding forms to entities.** Without `[Bind]` or a form class, a request can set any property of the entity, including prices and statuses. Use form classes.
- **Putting `UseAuthorization()` before `UseRouting()` or `UseAuthentication()`.** The order matters: routing chooses the endpoint, authentication reads the cookie, authorization checks the endpoint's rules. In the wrong order, the rules see no endpoint or no user.
- **Protecting pages one by one.** Forgetting `[Authorize]` on one new controller leaves it open. A fallback policy protects everything by default, and `[AllowAnonymous]` opens pages deliberately.
- **Hiding links instead of protecting pages.** A missing link does not stop anyone from typing the address. The attribute protects; the layout only tidies.
- **Demo passwords in production.** Keep seed accounts in `appsettings.Development.json`, never in `appsettings.json`, and create real accounts once from environment variables.
- **Losing the cookie keys.** Without persisted Data Protection keys every restart signs everyone out, and with several servers each one rejects the others' cookies.
- **Forgetting forwarded headers behind a proxy.** Redirects then point to `http://`, secure cookies misbehave, and logs show every visitor as `127.0.0.1`.
- **Updating without a backup.** A migration that drops or changes a column is not undone by `Down` in any way that restores data. Copy the database file first.

6.14 Chapter Summary

- Validation runs on the server for every request; data annotation attributes state rules about single values, `IValidatableObject` rules about several values, and the controller adds rules that need the database. All of them end up in `ModelState`, which the views display.
- Form classes carry exactly the fields a form may send and the rules of that form, which removes overposting by construction.
- Authentication establishes who the user is; authorization decides what they may do. ASP.NET Core Identity stores users in the application's database, hashes passwords with salted PBKDF2-HMAC-SHA512 at 100,000 iterations, locks out repeated failures, and signs users in with an encrypted cookie.
- A fallback policy that requires a signed-in user protects every page by default; `[AllowAnonymous]` opens the public pages, and `[Authorize(Roles = …)]` restricts pages to roles.
- Deployment means environments and layered configuration, `dotnet publish`, migrations applied before serving, a reverse proxy (Nginx or IIS) in front of Kestrel with forwarded headers, persisted Data Protection keys, a health endpoint, and backups.

6.15 Exercises

1. For each input, say which rule rejects it and which message the form shows, or that it is accepted and how it is stored: (a) phone `08-1200-0001-05`; (b) phone `+6281200000106`; (c)

service description `Carpet cleaning (per m2)` at Rp 25.000; (d) an order of 0.05 kg; (e) an order with pickup and the address `"   "` (three spaces).

2. Write a validation attribute `[NotInFuture]` for `DateTime` properties that rejects dates after today. On which date of an order does it belong: the date a late order was really received, or the date the laundry is promised for? Add that property to `OrderForm`.
3. The owner wants cashiers to be able to change the *notes* of an order but not its total. Explain how a form class makes this safe, and write the form class and the action.
4. Draw the order of middleware in `Program.cs` from `UseForwardedHeaders` to `MapControllerRoute` and explain, for each, what would go wrong if it were moved to the end.
5. A second owner joins the business. Without changing any code, how do you give her an Owner account on the production server? Give the steps.
6. The server's disk fails on a Friday night. List what you need to bring the shop back on a new server on Saturday morning, and where each item comes from.

6.16 Solutions to the Exercises

1. (a) `Normalize` removes the dashes: `081200000105`, 12 digits starting with `08`; accepted and stored as `081200000105`. (b) `+62` becomes `0`: `081200000106`; accepted and stored so. (c) The regular expression allows only `(per kg)`, `(per item)` or `(per pair)`, so `[RegularExpression]` rejects it with "End the description with (per kg), (per item) or (per pair)."; the price is within the range. (d) `[Range(typeof(decimal), "0.1", "100")]` rejects 0.05 with "The quantity must be between 0.1 and 100.". (e) `string.IsNullOrWhiteSpace("   ")` is `true`, so `Validate` returns "Pickup and delivery needs a delivery address." under the address field.

2. The attribute compares the value with today's date and, like `[IndonesianMobile]`, leaves empty values to `[Required]` (**Listing 6.20**).

Listing 6.20 Exercise 6.2: `Models/NotInFutureAttribute.cs`, a validation attribute of our own for dates.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryMvc.Models;

public sealed class NotInFutureAttribute : ValidationAttribute
{
    public NotInFutureAttribute() : base("{0} cannot be in the future.") { }

    public override bool IsValid(object? value) =>
        value is not DateTime date || date.Date <= DateTime.Today;
}
```

It belongs on the date the laundry was really received, for a cashier who enters yesterday's paper receipt today; a promised date lies in the future by its nature and would need the opposite rule. In `OrderForm` add `[NotInFuture] [Display(Name = "Received on")] public DateTime? ReceivedAt { get; set; }`, and in `Create` use `ReceivedAt = form.ReceivedAt ?? DateTime.Now`.

3. A form class with only `Note` (and `[StringLength(200)]` on it) means the request cannot carry `Total` at all; the action loads the tracked order, copies only the note, and saves. Owners can still change totals through a separate, protected page.

Listing 6.21 Exercise 6.3: a form class with one field and the action that uses it, to add to `OrdersController`.

```
public class OrderNoteForm
{
    [StringLength(200)]
    public string? Note { get; set; }
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditNote(int id, OrderNoteForm form)
{
    var order = await db.Orders.FindAsync(id);
    if (order is null) return NotFound();
    if (!ModelState.IsValid) return RedirectToAction(nameof(Details), new { id });
    order.Note = form.Note;
    await db.SaveChangesAsync();
    return RedirectToAction(nameof(Details), new { id });
}
```

The update that EF Core sends contains only the `Note` column, as in Section 5.10.4 (**Listing 6.21**).

4. `UseForwardedHeaders` → timing log → `UseExceptionHandler`/`UseHsts` → `UseStatusCodePagesWithReExecute` → `UseRouting` → `UseAuthentication` → `UseAuthorization` → `MapStaticAssets`/`MapHealthChecks`/`MapControllerRoute`. Moved to the end: forwarded headers would be applied after everything has already seen `http` and `127.0.0.1`; the timing log would measure nothing; the exception handler would not catch exceptions of the earlier middleware; status-code pages would miss the 404s produced before them; routing after authorization would leave authorization without an endpoint, so attribute rules would not apply; authentication after authorization would make every user look anonymous, so the fallback policy would send everyone to the sign-in page. The endpoint mappings are already at the end, where they belong.

5. Set three environment variables for the service once, `SeedUsers__0__Email=<her e-mail>`, `SeedUsers__0__Password=<a temporary password of at least 8 characters>` and `SeedUsers__0__Role=Owner` (with `sudo systemctl edit laundrymvc`), restart the service with `sudo systemctl restart laundrymvc`, and remove the variables again. `IdentitySeeder` creates accounts that do not exist yet and leaves existing ones alone, so the first owner is not touched. She changes the temporary password after her first sign-in (a password-change page is a small addition with `UserManager.ChangePasswordAsync`).

6. (1) The published application: rebuild it from the Git repository with `dotnet publish`, or keep the last published folder with the backups. (2) The database: the latest nightly backup of `laundry.db`, copied off the server. (3) The cookie keys from `/var/lib/laundrymvc/keys`: if they are in the backup, staff stay signed in; if not, everyone signs in again, which is harmless. (4) The configuration: `appsettings.Production.json` is in the repository; the systemd unit and the Nginx site are in `deploy/`. (5) A certificate: `certbot` issues a new one for the domain. (6) The DNS record, pointed at the new server's address. Then install the runtime and Nginx as in Section 6.8.4, copy the files and the database, start the service and check `/health`.

References

[1] Microsoft, "Model validation in ASP.NET Core MVC and Razor Pages," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/models/validation

[2] Microsoft, “Introduction to Identity on ASP.NET Core,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/authentication/identity

[3] OWASP Foundation, “Password Storage Cheat Sheet,” OWASP Cheat Sheet Series, accessed Sep. 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[4] Microsoft, “Policy-based authorization in ASP.NET Core,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/authorization/policies

[5] Microsoft, “Configuration in ASP.NET Core,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/configuration/

[6] Microsoft, “.NET application publishing overview,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/dotnet/core/deploying/

[7] Microsoft, “Applying migrations,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/managing-schemas/migrations/applying

[8] Microsoft, “Host ASP.NET Core on Linux with Nginx,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/host-and-deploy/linux-nginx

[9] Microsoft, “Configure ASP.NET Core to work with proxy servers and load balancers,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/host-and-deploy/proxy-load-balancer

[10] Microsoft, “Host ASP.NET Core on Windows with IIS,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/host-and-deploy/iis/

[11] Microsoft, “Key storage providers in ASP.NET Core,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/data-protection/implementation/key-storage-providers

[12] Microsoft, “What’s new in ASP.NET Core in .NET 10,” Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/release-notes/aspnetcore-10.0

Chapter 7

Laravel I: Artisan, Routing, Blade and Eloquent

Chapters 3, 5 and 6 built Profitina Laundry in ASP.NET Core. This chapter builds the same shop again in Laravel, the most widely used PHP framework, so that you can put the two side by side and see which ideas belong to one framework and which belong to every framework. By the end of it you will be able to install PHP, Composer and Laravel 13 on Windows, macOS or Linux; find your way around a new Laravel project; use Artisan to create models, migrations and controllers and to inspect routes; describe the laundry's tables in migrations and run them; write Eloquent models with relationships, casts and enums; seed the database from the same CSV files as Chapter 5; route requests with resource routes and route model binding; validate forms in controllers; and build the pages with Blade layouts, partials and form helpers. The result is the public catalog of Chapter 3 and the staff pages of Chapter 5 (counter, customers, prices, orders and payments), with the same data and the same price rules, in PHP.

7.1 Where We Left Off: The Same Shop, Another Language

The owner of Profitina Laundry has a nephew, Raka, who studies informatics and knows PHP from his web programming course. When the owner mentions that the Wiyung outlet might one day run its own copy of the staff pages on the cheap PHP hosting it already pays for, Raka offers to build it in Laravel. His first question is the right one: *how much of what you built in C# do I have to invent again?*

The answer this chapter gives is: very little of the thinking, and all of the typing. The database has the same five tables. The pages are the same pages. The workflow `queued → finished → departed` and the price rules (a minimum of 3 kg for per-kg services, whole items, a pickup fee of Rp 10.000 below Rp 100.000) are the same rules. What changes is where each piece lives, what it is called, and how much the framework does for you by convention. Learning a second framework is mostly learning a new map of old territory, and this chapter draws that map.

It helps Raka, who can start from the design instead of from nothing; the owner, who gets a second implementation she can host cheaply; and you, because seeing the same feature in two frameworks is the fastest way to understand what a framework is. Chapter 8 then solves the same seven problems in both.

7.2 The Idea in Plain Words

Laravel is an open-source PHP framework built around the same MVC pattern as ASP.NET Core MVC, with a strong preference for *convention over configuration*: if you name things the way Laravel expects, it connects them without being told. A model called `Customer` is stored in a table called `customers`; a foreign key to it is called `customer_id`; a controller method called `store` receives the form of `create`. Laravel 13, released on March 17, 2026, requires PHP 8.3 or newer and supports PHP up to 8.5; it receives bug fixes until the third quarter of 2027 and security fixes until March 17, 2028 [1].

> **DEFINITION.** *Artisan* is Laravel's command-line tool (`php artisan …`); it creates files, runs migrations and serves the application during development. *Eloquent* is Laravel's ORM; each model class represents one table, and each model object one row that can save itself. *Blade* is Laravel's template language; its files end in `.blade.php`, and `{{ … }}` prints a value with HTML escaping. A *route* maps an HTTP method and an address to a controller method; a *resource route* declares the seven usual CRUD routes of a resource in one line.

Figure 7.1 follows one request through a Laravel application: a cashier submits the new-customer form. Every request enters through `public/index.php`, the single *front controller*. `bootstrap/app.php` has configured routing, middleware and exception handling. The `web` middleware group starts the session, reads the cookies and checks the CSRF token of the form. The router finds the route `POST /customers` in `routes/web.php`, created by `Route::resource('customers', …)`, and calls `CustomerController@store`. The controller validates the input, Eloquent inserts the row, and the response is a redirect to the new customer's page, which the browser then requests with `GET` and which Blade renders. If validation fails, Laravel redirects back to the form with the error messages and the old input.

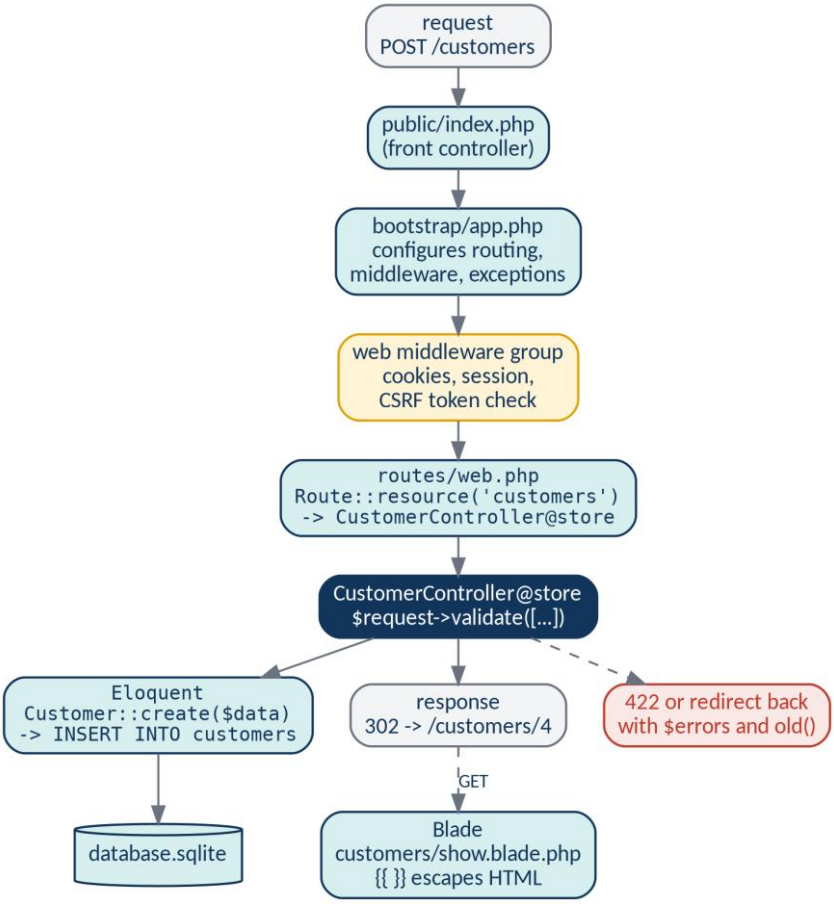


Figure 7.1 One request through the Laravel application: the front controller, the web middleware group, the router, the controller, Eloquent and Blade.

Compare the figure with the request pipeline of Chapter 6 (Section 6.2). The boxes have other names, but the order is the same: middleware, routing, controller, model, view. The biggest conceptual difference is in the model. EF Core follows the *data mapper* pattern: entities are plain classes, and a separate `DbContext` loads and saves them. Eloquent follows the *active record* pattern: a model object knows its table and saves itself, with `$customer->save()` or `Customer::create([...])`. Both are ORMs; they put the responsibility in different places.

7.3 Installing PHP, Composer and Laravel

Laravel needs three things: PHP, Composer (PHP's package manager, the counterpart of NuGet) and a database, which by default is SQLite and needs no installation. The simplest route on all three operating systems is the installer at `php.new`, which the Laravel documentation recommends: one command installs PHP, Composer and the Laravel installer [2]. Copy the command from the site rather than from a book, because it names the current PHP version:

- **macOS and Linux:** a single `curl … | bash` command, copied from `php.new` or from the Laravel installation guide.
- **Windows:** a single PowerShell command, run as administrator, copied from the same pages.
- **Any system, by hand:** PHP from `php.net` (Windows) or the system's package manager (`brew install php`, `sudo apt install php-cli php-sqlite3 php-mbstring php-xml php-curl`), then Composer from `getcomposer.org`.

Open a new terminal and check with `php -v` and `composer -V`. PHP 8.5, released on November 20, 2025, is the newest version, with the pipe operator `|>`, a new URI extension and many smaller improvements [3]; Laravel 13 runs on 8.3, 8.4 and 8.5, so a PHP that your distribution already ships is usually fine. In VS Code, install *PHP Intelephense* for PHP and the official *Laravel* extension for Blade and Artisan support.

Create the project:

```
composer create-project laravel/laravel profitina-laundry "^13.0"
cd profitina-laundry
php artisan serve --port=4070
```

Composer downloads the framework and its dependencies into `vendor/`, writes `.env` from `.env.example`, generates the application key, creates `database/database.sqlite` and runs the default migrations. `php artisan serve` starts PHP's development server; open `http://localhost:4070` and the Laravel welcome page appears. Stop it with Ctrl+C.

Table 7.1 is the map of the new project, next to the ASP.NET Core project of Chapters 3 to 6.

Table 7.1 Where things live in a Laravel 13 project, and their counterparts in ASP.NET Core.

Laravel	What it holds	In ASP.NET Core
`routes/web.php`	the routes of the web pages	`MapControllerRoute` and `[Route]` attributes
`app/Http/Controllers/`	controllers	`Controllers/`
`app/Models/`	Eloquent models	`Models/` (entities)
`database/migrations/`	migrations, written by hand	`Migrations/`, generated by `dotnet ef`
`database/seeders/`	seeders	`SeedData` with `UseSeeding`
`resources/views/`	Blade templates	`Views/` (Razor)
`public/`	the front controller and static files	`wwwroot/`
`bootstrap/app.php`	routing, middleware, exceptions	`Program.cs`
`.env`, `config/`	settings	`appsettings*.json`
`composer.json`, `vendor/`	packages	`.csproj`, NuGet cache

The chapter's code folder contains only the files this chapter adds or replaces; copy them over the new project as the README describes. The framework itself always comes from Composer.

7.4 Artisan, the Command Line That Writes Code

Almost every file of this chapter could be started with Artisan. **Table 7.2** lists the commands used.

Table 7.2 The Artisan commands used in this chapter.

Command	What it does
`php artisan make:model Customer -m`	a model and its migration
`php artisan make:controller CustomerController --resource --model=Customer`	a controller with the seven CRUD methods
`php artisan make:migration add_rack_to_orders_table --table=orders`	a migration that changes a table
`php artisan make:enum Enums/OrderStatus --string`	a backed enum
`php artisan migrate`	runs migrations not yet run
`php artisan migrate:fresh --seed`	drops all tables, migrates again and seeds
`php artisan db:seed`	runs `DatabaseSeeder`
`php artisan route:list --except-vendor`	the routes of the application
`php artisan tinker`	an interactive PHP shell inside the application
`php artisan serve --port=4070`	the development server

`tinker` deserves a special mention. It is a console in which you can type Eloquent directly: `App\Models\Customer::count()`, `App\Models\Order::with('payments')->first()`. It is the Laravel counterpart of the `LinqDemo` of Chapter 5, except that it works against the real database.

7.5 Migrations: The Tables Written in PHP

In EF Core the migrations were generated from the model. In Laravel it is the other way round: you write the migration, and the model follows the table. A migration is a class with `up()` and `down()` methods that use the *schema builder*, `Schema::create` with a `Blueprint` that has one method per column type [4]. Laravel runs migration files in the order of their names, which start with a date, and records each one in a table called `migrations`, together with a *batch* number.

The customers table (**Listing 7.1**) shows the basic vocabulary: `id()` is an auto-incrementing primary key, `string('name', 80)` a text column of at most 80 characters, `unique()` a unique index, `nullable()` a column that may be empty, and `timestamps()` the two columns `created_at` and `updated_at` that Eloquent fills automatically.

Listing 7.1 `2026_09_01_000001_create_customers_table.php`: the customers table.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {
        Schema::create('customers', function (Blueprint $table) {
            $table->id();
            $table->string('name', 80);
            $table->string('phone', 20)->unique();
            $table->string('address', 200)->nullable();
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('customers');
    }
};
```

The services and outlets keep their codes as primary keys, exactly as in Chapter 5, so `string('id', 5)->primary()` replaces `id()` (**Listing 7.2**). The price is an `unsignedInteger` in whole rupiah; since SQLite has no decimal type either, integer rupiah are the simplest correct choice in both frameworks.

Listing 7.2 `2026_09_01_000002_create_services_table.php`: services and outlets, with their codes as primary keys.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {
        Schema::create('services', function (Blueprint $table) {
            $table->string('id', 5)->primary();          // S0001, S0002, ...
            $table->string('description', 60);
            $table->unsignedInteger('price');            // rupiah per unit
            $table->boolean('is_active')->default(true);
            $table->timestamps();
        });

        Schema::create('outlets', function (Blueprint $table) {
            $table->string('id', 3)->primary();          // O01, O02, O03
            $table->string('name', 60);
            $table->string('address', 120);
            $table->string('phone', 20);
            $table->date('opened');
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('outlets');
        Schema::dropIfExists('services');
    }
};
```

The orders table (**Listing 7.3**) holds the relationships. `foreignId('customer_id')->constrained()` creates an unsigned big-integer column and a foreign key to `customers.id`, reading the table name from the column name by convention; `restrictOnDelete()` refuses to delete a customer who has orders, the same rule as `DeleteBehavior.Restrict` in Chapter 5. Because `service_id` refers to a text key, it is declared in two steps: the column, then `foreign(...)->references('id')->on('services')`. The status is a string with the default `'queued'` and an index, since the order list filters by it.

Listing 7.3 `2026_09_01_000003_create_orders_table.php`: orders with their foreign keys, status and timestamps.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {
        Schema::create('orders', function (Blueprint $table) {
            $table->id();
            $table->foreignId('customer_id')->constrained()->restrictOnDelete();
            $table->string('service_id', 5);
            $table->foreign('service_id')->references('id')->on('services')
                ->restrictOnDelete();
            $table->decimal('quantity', 6, 2);
            $table->boolean('pickup')->default(false);
            $table->string('delivery_address', 200)->nullable();
            $table->unsignedInteger('total');         // rupiah, fixed at creation
            $table->string('status', 12)->default('queued')->index();
            $table->string('note', 200)->nullable();
            $table->timestamp('received_at');
            $table->timestamp('finished_at')->nullable();
            $table->timestamp('departed_at')->nullable();
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('orders');
    }
};
```

The payments (**Listing 7.4**) use `cascadeOnDelete()`: payments mean nothing without their order.

Listing 7.4 `2026_09_01_000004_create_payments_table.php`: payments, deleted together with their order.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {
        Schema::create('payments', function (Blueprint $table) {
            $table->id();
            $table->foreignId('order_id')->constrained()->cascadeOnDelete();
            $table->unsignedInteger('amount');        // rupiah
            $table->string('method', 10);             // cash, qris, transfer
            $table->timestamp('paid_at');
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('payments');
    }
};
```

`php artisan migrate` runs every migration that is not yet in the `migrations` table, as one batch. `php artisan migrate:rollback` runs the `down()` methods of the last batch, newest first. Where EF Core compares the model with a snapshot to decide what a new migration contains, Laravel compares nothing: the file you write is the change. That makes Laravel migrations easy to read and easy to write for any database, and it puts the responsibility for keeping models and tables in step on you. `php artisan migrate:status` lists every migration file with *Ran* or *Pending*, the counterpart of EF Core's `__EFMigrationsHistory` table. `php artisan migrate:fresh` drops every table and runs everything again, which is what you want on your own computer while the design is still changing, and never on the shop's server.

7.6 Eloquent Models

7.6.1 Enums for the status and the payment method

PHP has had real enumerations since version 8.1 [5]. A *backed enum* has a fixed value for each case, which is what gets stored in the database. **Listing 7.5** is the order status, with a `label()` for the pages and a `next()` method that describes the workflow in one `match` expression; **Listing 7.6** is the payment method.

Listing 7.5 `app/Enums/OrderStatus.php`: the three statuses of an order and the workflow between them.

```php
<?php

namespace App\Enums;

enum OrderStatus: string
{
    case Queued = 'queued';
    case Finished = 'finished';
    case Departed = 'departed';

    public function label(): string
    {
        return ucfirst($this->value);
    }

    // The status that follows this one, or null when the order has left.
    public function next(): ?self
    {
        return match ($this) {
            self::Queued => self::Finished,
            self::Finished => self::Departed,
            self::Departed => null,
        };
    }
}
```

Listing 7.6 `app/Enums/PaymentMethod.php`: cash, QRIS and transfer.

```php
<?php

namespace App\Enums;

enum PaymentMethod: string
{
    case Cash = 'cash';
    case Qris = 'qris';
    case Transfer = 'transfer';

    public function label(): string
    {
        return match ($this) {
            self::Cash => 'Cash',
            self::Qris => 'QRIS',
            self::Transfer => 'Transfer',
        };
    }
}
```

Money is formatted in one place, `App\Support\Money` (**Listing 7.7**), which prints the same text as `Money.Rp` in the C# version: `Rp 36.000`, with a dot between thousands.

Listing 7.7 `app/Support/Money.php`: rupiah formatted the Indonesian way.

```php
<?php

namespace App\Support;

class Money
{
    // 36000 -> "Rp 36.000", the same text as Money.Rp in the C# version.
    public static function rp(int $amount): string
    {
        return 'Rp ' . number_format($amount, 0, ',', '.');
    }
}
```

Enums and the money helper are plain PHP, so they can be run without Laravel. `tools/enum_demo.php` (**Listing 7.8**) walks through the workflow and shows what Eloquent's enum cast will do with the text stored in the database.

Listing 7.8 `tools/enum_demo.php`: the enums and the money helper, run with plain PHP.

```php
<?php
// Runs with plain PHP (no Laravel needed): php tools/enum_demo.php

require __DIR__ . '/../app/Enums/OrderStatus.php';
require __DIR__ . '/../app/Enums/PaymentMethod.php';
require __DIR__ . '/../app/Support/Money.php';

use App\Enums\OrderStatus;
use App\Enums\PaymentMethod;
use App\Support\Money;

$status = OrderStatus::Queued;
while ($status !== null) {
    $next = $status->next();
    printf("%-9s -> %s\n", $status->label(), $next?->label() ?? '(end)');
    $status = $next;
}

// What Eloquent's cast does with the text stored in the database:
var_dump(OrderStatus::from('finished'));
var_dump(OrderStatus::tryFrom('washing'));        // unknown text -> null
echo PaymentMethod::Qris->label(), ' is stored as "',
    PaymentMethod::Qris->value, "\"\n";
echo Money::rp(36000), ' ', Money::rp(1250000), "\n";
```

This is its real output (`php tools/enum_demo.php`):

```
Queued    -> Finished
Finished  -> Departed
Departed  -> (end)
enum(App\Enums\OrderStatus::Finished)
NULL
QRIS is stored as "qris"
Rp 36.000 Rp 1.250.000
```

`OrderStatus::from('finished')` turns the stored text into the enum case, and `tryFrom('washing')` returns `null` for a text that is not a status instead of throwing. The order list uses `tryFrom` for exactly that reason: `/orders?status=nonsense` shows all orders instead of an error page.

7.6.2 The models

An Eloquent model is a class that extends `Model`; its table, key and timestamps follow from conventions unless the class says otherwise [6]. **Listing 7.9** is the simplest model. `#[Fillable(['name', 'phone', 'address'])]` lists the attributes that may be *mass assigned*, that is, set from an array as in `Customer::create($data)`. Everything else is refused, which is Laravel's protection against the overposting attack of Chapter 5. The attribute syntax is new in Laravel 13; the older `protected $fillable = [...]` property still works and means the same [7]. The method `orders()` declares the relationship: a customer *has many* orders, found through the `customer_id` column by convention [8].

Listing 7.9 `app/Models/Customer.php`: a customer and the relationship to the customer's orders.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

#[Fillable(['name', 'phone', 'address'])]
class Customer extends Model
{
    public function orders(): HasMany
    {
        return $this->hasMany(Order::class);
    }
}
```

The service model (**Listing 7.10**) has three details. `$keyType = 'string'` and `$incrementing = false` tell Eloquent that the key is a code such as `S0001` that the application chooses, not a number the database counts. The `casts()` method converts the price to an integer and `is_active` from SQLite's `0`/`1` to a real boolean. And the three *accessors* `name`, `perKg` and `unit` compute the same values as the computed properties of the C# `Service` class, from the description. An accessor is a method that returns an `Attribute`; Eloquent calls it when the page reads `$service->name`, `$service->per_kg` or `$service->unit` (the method `perKg` becomes the attribute `per_kg`, by convention).

Listing 7.10 `app/Models/Service.php`: a string key, casts, and three computed attributes.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Casts\Attribute;
use Illuminate\Database\Eloquent\Model;

#[Fillable(['id', 'description', 'price', 'is_active'])]
class Service extends Model
{
    // The key is a code such as "S0001", not a number the database counts.
    protected $keyType = 'string';

    public $incrementing = false;

    protected function casts(): array
    {
        return [
            'price' => 'integer',             // whole rupiah
            'is_active' => 'boolean',
        ];
    }

    // "Wash, dry & fold (per kg)" -> "Wash, dry & fold"
    protected function name(): Attribute
    {
        return Attribute::get(fn () => str_contains($this->description, ' (')
            ? substr($this->description, 0, strrpos($this->description, ' ('))
            : $this->description);
    }

    protected function perKg(): Attribute
    {
        return Attribute::get(
            fn () => str_ends_with($this->description, '(per kg)'));
    }
```

Listing 7.10 (Continued) *app/Models/Service.php: a string key, casts, and three computed attributes.*

```php
    protected function unit(): Attribute
    {
        return Attribute::get(fn () => match (true) {
            $this->per_kg => 'kg',
            str_ends_with($this->description, '(per pair)') => 'pair',
            default => 'item',
        });
    }
}
```

The order model (**Listing 7.11**) casts its columns to the right PHP types: the quantity to a decimal string with two digits, `pickup` to a boolean, the status to the `OrderStatus` enum, and the three timestamps to `Carbon` date objects that the views can format. `belongsTo` declares the other side of a one-to-many relationship. `paid()` and `balance()` compute the balance formula of Chapter 5 from the loaded payments.

Listing 7.11 `app/Models/Order.php`: casts, relationships, and the balance of an order.

```php
<?php

namespace App\Models;

use App\Enums\OrderStatus;
use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Eloquent\Relations\HasMany;

#[Fillable(['customer_id', 'service_id', 'quantity', 'pickup',
    'delivery_address', 'total', 'status', 'note', 'received_at',
    'finished_at', 'departed_at'])]
class Order extends Model
{
    protected function casts(): array
    {
        return [
            'quantity' => 'decimal:2',
            'pickup' => 'boolean',
            'total' => 'integer',
            'status' => OrderStatus::class,
            'received_at' => 'datetime',
            'finished_at' => 'datetime',
            'departed_at' => 'datetime',
        ];
    }

    public function customer(): BelongsTo
    {
        return $this->belongsTo(Customer::class);
    }

    public function service(): BelongsTo
    {
        return $this->belongsTo(Service::class);
    }

    public function payments(): HasMany
    {
        return $this->hasMany(Payment::class);
    }
```

Listing 7.11 (Continued) *app/Models/Order.php: casts, relationships, and the balance of an order.*

```php
    // Sums the loaded payments; load them with with('payments') first.
    public function paid(): int
    {
        return (int) $this->payments->sum('amount');
    }

    public function balance(): int
    {
        return $this->total - $this->paid();
    }
}
```

A payment (**Listing 7.12**) belongs to one order and casts its method to the `PaymentMethod` enum.

Listing 7.12 `app/Models/Payment.php`: a payment and its order.

```php
<?php

namespace App\Models;

use App\Enums\PaymentMethod;
use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

#[Fillable(['order_id', 'amount', 'method', 'paid_at'])]
class Payment extends Model
{
    protected function casts(): array
    {
        return [
            'amount' => 'integer',
            'method' => PaymentMethod::class,
            'paid_at' => 'datetime',
        ];
    }

    public function order(): BelongsTo
    {
        return $this->belongsTo(Order::class);
    }
}
```

The outlet model (**Listing 7.13**) completes the set; its `opened` column is cast to a date.

Listing 7.13 `app/Models/Outlet.php`: the three outlets of Chapter 3.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Model;

#[Fillable(['id', 'name', 'address', 'phone', 'opened'])]
class Outlet extends Model
{
    protected $keyType = 'string';

    public $incrementing = false;

    protected function casts(): array
    {
        return ['opened' => 'date'];
    }
}
```

Two things are missing compared with EF Core, and both on purpose. There is no list of columns: Eloquent reads whatever columns the table has, so `$customer->phone` works because the migration created `phone`. And there is no context object: `Customer::find(3)` asks the database directly through the default connection. The price of that convenience is that a typing mistake such as `$customer->phnoe` silently returns `null` instead of failing to compile.

7.6.3 Eloquent and EF Core side by side

Table 7.3 puts the everyday operations of the two ORMs next to each other, using the laundry's own queries. Reading it row by row is the quickest way to carry what you learned in Chapter 5 over to Laravel.

Table 7.3 Everyday operations in EF Core and in Eloquent.

Task	EF Core (Chapter 5)	Eloquent (this chapter)
one row by key	`await db.Customers.FindAsync(3)`	`Customer::find(3)`
one row or 404	`FirstOrDefaultAsync(...)` then `NotFound()`	route model binding, or `findOrFail(3)`
filter and sort	`db.Customers.Where(...).OrderBy(c => c.Name)`	`Customer::where(...)->orderBy('name')`
run the query	`await query.ToListAsync()`	`$query->get()`
related rows	`.Include(o => o.Customer)`	`->with('customer')`
related rows later	(explicit loading) `db.Entry(o).Collection(...).LoadAsync()`	`$order->load('payments')`
count, sum	`CountAsync()`, `SumAsync(p => p.Amount)`	`count()`, `sum('amount')`
insert	`db.Customers.Add(c); await db.SaveChangesAsync()`	`Customer::create($data)`
update	change a tracked entity, then `SaveChangesAsync()`	`$customer->update($data)` or change and `save()`
delete	`db.Customers.Remove(c); await db.SaveChangesAsync()`	`$customer->delete()`
exists	`AnyAsync(o => o.CustomerId == id)`	`$customer->orders()->exists()`

Three differences stand out. First, Eloquent has no separate "save all changes" step: `create`, `update`, `save` and `delete` each send their SQL at once, while EF Core collects changes in the change tracker and sends them together in one transaction when `SaveChangesAsync` is called. When several rows must change together in Laravel, wrap them in `DB::transaction(function () { … })`. Second, Eloquent calls are synchronous; PHP serves each request in its own process, so there is no thread to free while the database works, and no `async` or `await`. Third, EF Core checks property names at compile time, while Eloquent looks up column names at run time; the compiler catches `c.Phnoe`, but only a test, or a user, catches `$c->phnoe`.

7.7 Seeding from the Chapter 3 Files

The seeder (**Listing 7.14**) creates the same data as the `SeedData` class of Chapter 5: the twelve services and three outlets from the very CSV files of Chapter 3, copied to `database/data/`, and the three sample customers with their orders and payments. `$sari->orders()->create([...])` creates an order and fills `customer_id` from the relationship; `$order->payments()->create([...])` does the same for a payment.

Listing 7.14 `database/seeders/DatabaseSeeder.php`: services and outlets from the Chapter 3 CSV files, then sample customers, orders and payments.

```php
<?php

namespace Database\Seeders;

use App\Enums\OrderStatus;
use App\Enums\PaymentMethod;
use App\Models\Customer;
use App\Models\Outlet;
use App\Models\Service;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    // The same starting data as the ASP.NET Core version of Chapter 5.
    public function run(): void
    {
        foreach ($this->csv('Service.csv') as [$id, $description, $price]) {
            Service::create(['id' => $id, 'description' => $description,
                'price' => (int) $price]);
        }
        foreach ($this->csv('Outlet.csv') as [$id, $name, $address, $phone,
            $opened]) {
            Outlet::create(compact('id', 'name', 'address', 'phone', 'opened'));
        }

        $today = today();

        $sari = Customer::create(['name' => 'Sari Wulandari',
            'phone' => '081200000101',
            'address' => 'Jl. Kertajaya 12, Surabaya']);
        $order = $sari->orders()->create(['service_id' => 'S0001',
            'quantity' => 4.5, 'total' => 36000,
            'status' => OrderStatus::Finished,
            'received_at' => $today->copy()->subDay()->addHours(8),
            'finished_at' => $today->copy()->addHours(9)]);
        $order->payments()->create(['amount' => 36000,
            'method' => PaymentMethod::Qris,
            'paid_at' => $today->copy()->subDay()->addHours(8)]);
```

Listing 7.14 (Continued) *database/seeders/DatabaseSeeder.php: services and outlets from the Chapter 3 CSV files, then sample customers, orders and payments.*

```
        $budi = Customer::create(['name' => 'Budi Santoso',
            'phone' => '081200000102']);
        $order = $budi->orders()->create(['service_id' => 'S0002',
            'quantity' => 3, 'total' => 36000,
            'received_at' => $today->copy()->addMinutes(450)]);
        $order->payments()->create(['amount' => 20000,
            'method' => PaymentMethod::Cash,
            'paid_at' => $today->copy()->addMinutes(450)]);
        $budi->orders()->create(['service_id' => 'S0008', 'quantity' => 2,
            'pickup' => true, 'delivery_address' => 'Jl. Mulyosari 30, Surabaya',
            'total' => 80000, 'received_at' => $today->copy()->addHours(10)]);

        Customer::create(['name' => 'Dewi Lestari',
            'phone' => '081200000103', 'address' => 'Jl. Manyar 7, Surabaya']);
    }

    // The rows of database/data/<file>, without the header line.
    private function csv(string $file): array
    {
        $lines = file(database_path("data/{$file}"), FILE_IGNORE_NEW_LINES);
        $rows = array_map(fn ($line) => str_getcsv($line, ',', '"', ''), $lines);

        return array_slice($rows, 1);
    }
}
```

`str_getcsv($line, ',', '"', '')` passes all four arguments on purpose: PHP 8.4 deprecated relying on the default of the fourth argument, the escape character, and passing `''` switches that old, non-standard escaping off. `database_path()` is a Laravel helper that returns the absolute path of the `database` folder, so the seeder works from whatever folder Artisan is started in. `php artisan migrate:fresh --seed` builds the database from nothing in one command.

7.8 Routing

Listing 7.15 is the complete `routes/web.php`. Compare it with the routing of Chapter 3: there, conventional routes and attributes on controllers; here, every route of the application in one file.

Listing 7.15 `routes/web.php`: the public pages, the staff pages, and two extra actions on orders.

```php
<?php

use App\Http\Controllers\CounterController;
use App\Http\Controllers\CustomerController;
use App\Http\Controllers\HomeController;
use App\Http\Controllers\OrderController;
use App\Http\Controllers\OutletController;
use App\Http\Controllers\PaymentController;
use App\Http\Controllers\PriceController;
use App\Http\Controllers\ServiceController;
use Illuminate\Support\Facades\Route;

// The public pages (Chapter 3 built them in ASP.NET Core).
Route::get('/', HomeController::class)->name('home');
Route::get('services', [ServiceController::class, 'index'])
    ->name('services.index');
Route::get('services/{service}', [ServiceController::class, 'show'])
    ->name('services.show');
Route::get('outlets', [OutletController::class, 'index'])
    ->name('outlets.index');

// The staff pages (Chapter 5 built them in ASP.NET Core).
Route::get('counter', CounterController::class)->name('counter');
Route::resource('customers', CustomerController::class);
Route::resource('prices', PriceController::class)
    ->parameters(['prices' => 'service'])
    ->except(['show', 'destroy']);
Route::resource('orders', OrderController::class)
    ->only(['index', 'create', 'store', 'show']);
Route::post('orders/{order}/advance', [OrderController::class, 'advance'])
    ->name('orders.advance');
Route::post('orders/{order}/payments', [PaymentController::class, 'store'])
    ->name('orders.payments.store');
```

`Route::get('services/{service}', …)` has a route parameter named `service`, and the controller method declares `public function show(Service $service)`. Because the name and the type match, Laravel performs *route model binding*: it looks up the service whose key equals the value in the address and passes the model, or answers 404 if there is none [9]. It is the counterpart of `FirstOrDefaultAsync(...)` followed by `NotFound()` in every ASP.NET Core action, done once by the framework.

`Route::resource('customers', CustomerController::class)` declares seven routes at once; **Table 7.4** lists them [10]. Resource routes use HTTP methods the way they were meant: the same address `/customers/3` shows a customer with `GET`, updates it with `PUT` and deletes it with `DELETE`. Because HTML forms can only send `GET` and `POST`, Blade's `@method('PUT')` adds a hidden field `_method=PUT` that Laravel reads.

Table 7.4 The seven routes of `Route::resource('customers', …)`.

Method and address	Controller method	Route name	Purpose
`GET /customers`	`index`	`customers.index`	the list
`GET /customers/create`	`create`	`customers.create`	the empty form
`POST /customers`	`store`	`customers.store`	save a new customer
`GET /customers/{customer}`	`show`	`customers.show`	one customer
`GET /customers/{customer}/edit`	`edit`	`customers.edit`	the filled form
`PUT /customers/{customer}`	`update`	`customers.update`	save changes
`DELETE /customers/{customer}`	`destroy`	`customers.destroy`	delete

`->except([...])` and `->only([...])` trim the set: prices have no `show` or `destroy` (a service is switched off, never deleted, as in Chapter 5), and orders have no edit or delete pages. `->parameters(['prices' => 'service'])` renames the route parameter of the price routes from `{price}` to `{service}`, so that route model binding finds a `Service`. The two extra routes for advancing an order and recording a payment get names of their own with `->name(...)`. Every page builds its links with `route('customers.show', $customer)` rather than writing `/customers/3`, so an address can change in `web.php` without breaking a single link.

7.9 Controllers

7.9.1 Customers: validation and CRUD

Listing 7.16 is the customer controller. Read it next to `CustomersController` of Chapter 5: the eight C# actions are seven PHP methods, because Laravel's single `destroy` replaces the confirmation page plus the deletion.

Listing 7.16 `app/Http/Controllers/CustomerController.php`: complete CRUD for customers with Eloquent.

```php
<?php

namespace App\Http\Controllers;

use App\Models\Customer;
use Illuminate\Http\Request;
use Illuminate\Validation\Rule;

class CustomerController extends Controller
{
    // GET /customers?search=sari
    public function index(Request $request)
    {
        $search = trim((string) $request->query('search', ''));
        $customers = Customer::query()
            ->when($search !== '', fn ($query) => $query
                ->where('name', 'like', "%{$search}%")
                ->orWhere('phone', 'like', "%{$search}%"))
            ->orderBy('name')
            ->get();

        return view('customers.index', compact('customers', 'search'));
    }

    // GET /customers/3   (route model binding finds Customer 3, or 404)
    public function show(Customer $customer)
    {
        $customer->load(['orders' => fn ($q) => $q->latest('received_at'),
            'orders.service']);

        return view('customers.show', compact('customer'));
    }

    public function create()
    {
        return view('customers.create', ['customer' => new Customer()]);
    }

    public function store(Request $request)
    {
        $customer = Customer::create($request->validate($this->rules()));

        return redirect()->route('customers.show', $customer)
            ->with('message', "Customer {$customer->name} added.");
    }
```

Listing 7.16 (Continued) *app/Http/Controllers/CustomerController.php: complete CRUD for customers with Eloquent.*

```
    public function edit(Customer $customer)
    {
        return view('customers.edit', compact('customer'));
    }

    public function update(Request $request, Customer $customer)
    {
        $customer->update($request->validate($this->rules($customer)));

        return redirect()->route('customers.show', $customer)
            ->with('message', "Customer {$customer->name} saved.");
    }

    public function destroy(Customer $customer)
    {
        if ($customer->orders()->exists()) {
            return back()->with('error',
                'A customer with orders cannot be deleted.');
        }
        $customer->delete();

        return redirect()->route('customers.index')
            ->with('message', "Customer {$customer->name} deleted.");
    }

    private function rules(?Customer $customer = null): array
    {
        return [
            'name' => ['required', 'string', 'min:2', 'max:80'],
            'phone' => ['required', 'string', 'max:20',
                Rule::unique('customers', 'phone')->ignore($customer)],
            'address' => ['nullable', 'string', 'max:200'],
        ];
    }
}
```

- `index` builds a query with `Customer::query()`, adds the search with `when($search !== '', …)`, which applies the closure only when the condition is true, and ends with `get()`, which sends the SQL. `where('name', 'like', "%{$search}%")` is sent with the search text as a bound parameter, so it is as safe from SQL injection as EF Core's `Contains`. Unlike `instr`, SQLite's `LIKE` ignores the difference between upper and lower case for ASCII letters, so `sari` finds *Sari Wulandari*.
- `show` receives the customer through route model binding and loads the orders, newest first, with each order's service, using `load`, the counterpart of `Include` and `ThenInclude`.
- `store` and `update` call `$request->validate($rules)`. If a rule fails, Laravel stops the method, redirects back to the form, and puts the error messages and the submitted values into the session, where Blade finds them with `@error` and `old()`. If every rule passes, `validate` returns only the validated fields, which go straight into `create` or `update` [11].
- The phone rule `Rule::unique('customers', 'phone')->ignore($customer)` checks the database for another customer with the same number, ignoring the customer being edited: the whole `CheckPhoneIsFreeAsync` method of Chapter 5 in one line.
- `destroy` refuses to delete a customer with orders, exactly as in Chapter 5, and uses `back()->with('error', …)` to show why. `with` puts a *flash* message into the session for the next request only, the counterpart of `TempData`.

7.9.2 Orders, payments and prices

The order controller (**Listing 7.17**) follows `OrdersController` of Chapter 5 closely.

Listing 7.17 `app/Http/Controllers/OrderController.php`: the order list, order pages, new orders priced by the rules of Chapter 3, and the workflow.

```php
<?php

namespace App\Http\Controllers;

use App\Enums\OrderStatus;
use App\Models\Customer;
use App\Models\Order;
use App\Models\Service;
use App\Support\Money;
use Illuminate\Http\Request;
use Illuminate\Validation\Rule;

class OrderController extends Controller
{
    // GET /orders or /orders?status=queued
    public function index(Request $request)
    {
        $status = OrderStatus::tryFrom((string) $request->query('status'));
        $orders = Order::with(['customer', 'service', 'payments'])
            ->when($status, fn ($query) => $query->where('status', $status))
            ->latest('received_at')
            ->get();

        return view('orders.index', compact('orders', 'status'));
    }

    public function show(Order $order)
    {
        $order->load(['customer', 'service',
            'payments' => fn ($q) => $q->orderBy('paid_at')]);

        return view('orders.show', compact('order'));
    }

    public function create(Request $request)
    {
        return view('orders.create', [
            'customers' => Customer::orderBy('name')->get(),
            'services' => Service::where('is_active', true)
                ->orderBy('id')->get(),
            'customerId' => $request->integer('customer_id'),
        ]);
    }
```

Listing 7.17 (Continued) *app/Http/Controllers/OrderController.php: the order list, order pages, new orders priced by the rules of Chapter 3, and the workflow.*

```
    public function store(Request $request)
    {
        $data = $request->validate([
            'customer_id' => ['required', 'integer', 'exists:customers,id'],
            'service_id' => ['required', 'string',
                Rule::exists('services', 'id')->where('is_active', true)],
            'quantity' => ['required', 'numeric', 'min:0.1', 'max:100'],
            'delivery_address' => ['nullable', 'required_if_accepted:pickup',
                'string', 'max:200'],
            'note' => ['nullable', 'string', 'max:200'],
        ]);
        $service = Service::findOrFail($data['service_id']);
        $pickup = $request->boolean('pickup');

        // The price rules of Chapter 3: minimum 3 kg, whole items, pickup fee.
        $quantity = (float) $data['quantity'];
        $billed = $service->per_kg ? max($quantity, 3) : ceil($quantity);
        $subtotal = (int) round($billed * $service->price);
        $total = $subtotal + ($pickup && $subtotal < 100000 ? 10000 : 0);

        $order = Order::create([
            ...$data,
            'pickup' => $pickup,
            'total' => $total,
            'status' => OrderStatus::Queued,
            'received_at' => now(),
        ]);

        return redirect()->route('orders.show', $order)->with('message',
            "Order #{$order->id} created: " . Money::rp($total) . '.');
    }

    // POST /orders/5/advance: queued -> finished -> departed
    public function advance(Order $order)
    {
        $next = $order->status->next();
        if ($next === null) {
            return back()->with('error', 'This order has already left the shop.');
        }
        $order->status = $next;
        if ($next === OrderStatus::Finished) {
            $order->finished_at = now();
        } else {
            $order->departed_at = now();
        }
        $order->save();
```

Listing 7.17 (Continued) *app/Http/Controllers/OrderController.php: the order list, order pages, new orders priced by the rules of Chapter 3, and the workflow.*

```
        return back()->with('message',
            "Order #{$order->id} is now {$next->label()}.");
    }
}
```

- `index` uses *eager loading*: `Order::with(['customer', 'service', 'payments'])` loads the orders and then, in three more queries, all their customers, services and payments, and connects them in memory. Without `with`, every `$o->customer->name` in the view would send its own query: one for the list plus three per order, the *N+1 query problem* that Chapter 8 measures [8].

- `store` validates with rules that read the database: `exists:customers,id` and `Rule::exists('services', 'id')->where('is_active', true)` check that the customer exists and that the service exists *and* is offered. `required_if_accepted:pickup` makes the delivery address required only when the pickup box is ticked, the rule that took an `IValidatableObject` in Chapter 6.
- The price is computed with Chapter 3's rules in four lines: billed quantity (at least 3 for per-kg services, rounded up to whole items otherwise), subtotal, and the pickup fee below Rp 100.000. These lines use PHP floats; Problem 8.1 of Chapter 8 shows exactly when that is dangerous and replaces them with a class that gives the same result as the C# `PriceCalculator` in every case.
- `advance` asks the enum for the next status, which keeps the workflow in one place, stamps the right timestamp and calls `save()`, which sends an `UPDATE` for the changed columns only, just as EF Core's change tracker does.

The payment controller (**Listing 7.18**) validates the amount and the method, with `Rule::enum(PaymentMethod::class)` accepting only `cash`, `qris` or `transfer`, and creates the payment through the order's relationship.

Listing 7.18 `app/Http/Controllers/PaymentController.php`: recording a payment for an order.

```php
<?php

namespace App\Http\Controllers;

use App\Enums\PaymentMethod;
use App\Models\Order;
use App\Support\Money;
use Illuminate\Http\Request;
use Illuminate\Validation\Rule;

class PaymentController extends Controller
{
    // POST /orders/5/payments
    public function store(Request $request, Order $order)
    {
        $data = $request->validate([
            'amount' => ['required', 'integer', 'min:1'],
            'method' => ['required', Rule::enum(PaymentMethod::class)],
        ]);
        $order->payments()->create([...$data, 'paid_at' => now()]);

        return back()->with('message',
            'Payment of ' . Money::rp((int) $data['amount']) . ' recorded.');
    }
}
```

The price controller (**Listing 7.19**) mirrors `PricesController` of Chapter 5, including the next free service code, and the regular expression that requires `(per kg)`, `(per item)` or `(per pair)` at the end of a description, the rule of Chapter 6. A checkbox that is not ticked is not sent at all by the browser, so `$request->boolean('is_active')` turns "missing" into `false` explicitly.

Listing 7.19 `app/Http/Controllers/PriceController.php`: the owner's price list.

```php
<?php

namespace App\Http\Controllers;

use App\Models\Service;
use Illuminate\Http\Request;

// The owner's price list: the same Service rows that /services shows.
class PriceController extends Controller
{
    public function index()
    {
        return view('prices.index', ['services' => Service::orderBy('id')->get()]);
    }

    public function create()
    {
        return view('prices.create', ['service' => new Service([
            'is_active' => true])]);
    }

    public function store(Request $request)
    {
        $data = $this->validated($request);
        $last = Service::max('id') ?? 'S0000';          // S0012 -> S0013
        $data['id'] = sprintf('S%04d', (int) substr($last, 1) + 1);
        $service = Service::create($data);

        return redirect()->route('prices.index')
            ->with('message', "{$service->id} {$service->name} added.");
    }

    public function edit(Service $service)
    {
        return view('prices.edit', compact('service'));
    }

    // A new price counts for new orders only: every order keeps its total.
    public function update(Request $request, Service $service)
    {
        $service->update($this->validated($request));

        return redirect()->route('prices.index')
            ->with('message', "{$service->id} {$service->name} saved.");
    }
```

Listing 7.19 (Continued) *app/Http/Controllers/PriceController.php: the owner's price list.*

```php
    private function validated(Request $request): array
    {
        $data = $request->validate([
            'description' => ['required', 'string', 'max:60',
                'regex:/^.{3,} \(per (kg|item|pair)\)$/'],
            'price' => ['required', 'integer', 'between:1000,1000000'],
        ], [
            'description.regex' =>
                'End the description with (per kg), (per item) or (per pair).',
        ]);
        $data['is_active'] = $request->boolean('is_active');   // checkbox

        return $data;
    }
}
```

The remaining controllers are short. `CounterController` (**Listing 7.20**) computes the four numbers of the counter page, with `count()` and `sum('amount')`; it is an *invokable* controller with a single `__invoke` method, which the route names without a method. `HomeController`, `ServiceController` and `OutletController` serve the public pages of Chapter 3 and are in the code folder.

Listing 7.20 `app/Http/Controllers/CounterController.php`: four numbers for the counter.

```php
<?php

namespace App\Http\Controllers;

use App\Enums\OrderStatus;
use App\Models\Order;
use App\Models\Payment;

// GET /counter: what the cashier needs to know at a glance.
class CounterController extends Controller
{
    public function __invoke()
    {
        $today = today();
        $tomorrow = today()->addDay();

        return view('counter', [
            'queued' => Order::where('status', OrderStatus::Queued)->count(),
            'readyForPickup' => Order::where('status', OrderStatus::Finished)
                ->count(),
            'receivedToday' => Order::where('received_at', '>=', $today)
                ->where('received_at', '<', $tomorrow)->count(),
            'paidToday' => (int) Payment::where('paid_at', '>=', $today)
                ->where('paid_at', '<', $tomorrow)->sum('amount'),
        ]);
    }
}
```

7.9.3 Seeing the SQL

In Chapter 5 the terminal showed every SQL statement EF Core sent. Laravel can do the same in three ways. In `tinker`, `Customer::where('name', 'like', '%an%')->toRawSql()` returns the SQL with the values filled in, without running it. For a whole page, `DB::listen(fn ($query) => logger($query->sql, $query->bindings));` in `AppServiceProvider::boot()` writes every statement with its bindings to `storage/logs/laravel.log`. And the free Laravel Debugbar package, installed only in development with `composer require barryvdh/laravel-debugbar --dev`, shows a bar at the bottom of every page with the number of queries, their time, and which ones repeat.

Counting queries this way is worth a habit. Open the order list, look at the count, and compare it with what you expect: four for the list of this chapter (orders, then customers, services and payments). If the number grows with the number of orders, a `with` is missing. Chapter 8 turns exactly this count into a figure and compares it across both frameworks.

7.10 Blade Templates

7.10.1 The layout

Every page extends one layout (**Listing 7.21**). `@yield('content')` marks where a page's content goes, and `@yield('title')` where its title goes; a page fills them with `@section`. `{{`

`asset('css/site.css') }}` builds the address of the stylesheet in `public/css`, the same stylesheet as the ASP.NET Core version. `@class(['here' => request()->routeIs('orders.*')])` adds the class `here` to the menu link of the current section, like the `Here()` helper of Chapter 3's layout. `@session('message')` shows the flash message set by `with('message', …)`, with the value available as `$value`.

Listing 7.21 `resources/views/layouts/app.blade.php`: the layout shared by every page.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>@yield('title') - Profitina Laundry</title>
    <link rel="stylesheet" href="{{ asset('css/site.css') }}">
</head>
<body>
<header>
    <a class="brand" href="{{ route('home') }}">Profitina Laundry</a>
    <nav>
        <a href="{{ route('home') }}"
           @class(['here' => request()->routeIs('home')])>Home</a>
        <a href="{{ route('services.index') }}"
           @class(['here' => request()->routeIs('services.*')])>Services</a>
        <a href="{{ route('outlets.index') }}"
           @class(['here' => request()->routeIs('outlets.*')])>Outlets</a>
    </nav>
    <nav class="staff">
        <a href="{{ route('counter') }}"
           @class(['here' => request()->routeIs('counter')])>Counter</a>
        <a href="{{ route('orders.index') }}"
           @class(['here' => request()->routeIs('orders.*')])>Orders</a>
        <a href="{{ route('customers.index') }}"
           @class(['here' => request()->routeIs('customers.*')])>Customers</a>
        <a href="{{ route('prices.index') }}"
           @class(['here' => request()->routeIs('prices.*')])>Prices</a>
    </nav>
</header>
<main>
    @session('message')
        <p class="flash">{{ $value }}</p>
    @endsession
    @session('error')
        <p class="flash error">{{ $value }}</p>
    @endsession
    @yield('content')
</main>
<footer>
    <p>&copy; {{ date('Y') }} Profitina Laundry, Surabaya</p>
</footer>
</body>
</html>
```

7.10.2 Escaping, forms and old input

`{{ $customer->name }}` prints the name through PHP's `htmlspecialchars`, so a customer called `<script>alert(1)</script>` appears as text instead of running [12]. `{!! … !!}` prints without escaping and belongs only around HTML that the application itself produced. This is the same protection Razor gives with `@`, and forgetting it is the classic cross-site scripting mistake of hand-written PHP.

The customer form (**Listing 7.22**) is shared by the create and edit pages, which include it with `@include('customers._form')`. `old('name', $customer->name)` shows the value the user typed if the last submission failed validation, and otherwise the stored value; `@error('name') … @enderror` shows the message for that field, available as `$message`.

Listing 7.22 `resources/views/customers/_form.blade.php`: the fields shared by the create and edit pages.

```
<label for="name">Name</label>
<input id="name" name="name" value="{{ old('name', $customer->name) }}">
@error('name') <span class="error">{{ $message }}</span> @enderror

<label for="phone">Phone (WhatsApp)</label>
<input id="phone" name="phone" value="{{ old('phone', $customer->phone) }}">
@error('phone') <span class="error">{{ $message }}</span> @enderror

<label for="address">Address</label>
<input id="address" name="address"
       value="{{ old('address', $customer->address) }}">
@error('address') <span class="error">{{ $message }}</span> @enderror
```

The edit page (**Listing 7.23**) wraps the fields in a form. `@csrf` adds the hidden CSRF token that the `web` middleware group checks on every `POST`, `PUT` and `DELETE`; a form without it is rejected with status 419 [13]. `@method('PUT')` turns the `POST` into the `PUT` that the resource route expects.

Listing 7.23 `resources/views/customers/edit.blade.php`: the edit form with its CSRF token and method field.

```
@extends('layouts.app')

@section('title', 'Edit customer')

@section('content')
<h1>Edit {{ $customer->name }}</h1>
<form method="post" action="{{ route('customers.update', $customer) }}"
      class="form">
    @csrf
    @method('PUT')
    @include('customers._form')
    <button type="submit">Save</button>
    <a href="{{ route('customers.show', $customer) }}">Cancel</a>
</form>
@endsection
```

The customer page (**Listing 7.24**) lists the orders with `@forelse … @empty … @endforelse`, a loop with a branch for the empty list, and offers a delete button only when the customer has no orders. `@use('App\Support\Money')` imports the money helper into the template.

Listing 7.24 `resources/views/customers/show.blade.php`: one customer, the orders, and the delete button.

```
@extends('layouts.app')
@use('App\Support\Money')

@section('title', $customer->name)

@section('content')
<h1>{{ $customer->name }}</h1>
<p>{{ $customer->phone }} · {{ $customer->address ?? 'no address' }}
   · customer since {{ $customer->created_at->format('j M Y') }}</p>
<p>
    <a class="button"
       href="{{ route('orders.create', ['customer_id' => $customer->id]) }}">
        New order</a>
    <a href="{{ route('customers.edit', $customer) }}">Edit</a>
</p>
<h2>Orders</h2>
<table class="list">
    <thead>
        <tr><th>#</th><th>Received</th><th>Service</th><th class="num">Total</th>
            <th>Status</th></tr>
    </thead>
    <tbody>
    @forelse ($customer->orders as $o)
        <tr>
            <td><a href="{{ route('orders.show', $o) }}">{{ $o->id }}</a></td>
            <td>{{ $o->received_at->format('d/m H:i') }}</td>
            <td>{{ $o->service->name }}</td>
            <td class="num">{{ Money::rp($o->total) }}</td>
            <td>{{ $o->status->label() }}</td>
        </tr>
    @empty
        <tr><td colspan="5">No orders yet.</td></tr>
    @endforelse
    </tbody>
</table>
@if ($customer->orders->isEmpty())
    <form method="post" action="{{ route('customers.destroy', $customer) }}"
          onsubmit="return confirm('Delete this customer?')">
        @csrf
        @method('DELETE')
        <button type="submit" class="danger">Delete customer</button>
    </form>
@endif
@endsection
```

7.10.3 The order page

The order page (**Listing 7.25**) is the most complete view of the chapter. It shows the facts of the order, a button for the next step of the workflow, the payments, and a payment form while money is owed. The payment methods come from `PaymentMethod::cases()`, so a new method in the enum appears on the page without touching the HTML. `$order->finished_at?->format(...)` uses PHP's nullsafe operator: while the order is not finished, the value is `null` and nothing is printed.

Listing 7.25 `resources/views/orders/show.blade.php`: one order with its workflow button, payments and payment form.

```
@extends('layouts.app')
@use('App\Enums\OrderStatus')
@use('App\Enums\PaymentMethod')
@use('App\Support\Money')

@section('title', "Order #{$order->id}")

@section('content')
<h1>Order #{{ $order->id }} · {{ $order->status->label() }}</h1>
<dl class="facts">
    <dt>Customer</dt>
    <dd><a href="{{ route('customers.show', $order->customer) }}">
        {{ $order->customer->name }}</a> ({{ $order->customer->phone }})</dd>
    <dt>Service</dt>
    <dd>{{ $order->service->name }}, {{ $order->quantity }}
        {{ $order->service->unit }}</dd>
    <dt>Delivery</dt>
    <dd>{{ $order->pickup ? "to {$order->delivery_address}"
                          : 'collected at the shop' }}</dd>
    <dt>Received</dt><dd>{{ $order->received_at->format('d/m/Y H:i') }}</dd>
    <dt>Finished</dt><dd>{{ $order->finished_at?->format('d/m/Y H:i') }}</dd>
    <dt>Departed</dt><dd>{{ $order->departed_at?->format('d/m/Y H:i') }}</dd>
    <dt>Note</dt><dd>{{ $order->note }}</dd>
    <dt>Total</dt><dd>{{ Money::rp($order->total) }}</dd>
    <dt>Paid</dt><dd>{{ Money::rp($order->paid()) }}</dd>
    <dt>Balance</dt><dd><strong>{{ Money::rp($order->balance()) }}</strong></dd>
</dl>

@if ($order->status !== OrderStatus::Departed)
    <form method="post" action="{{ route('orders.advance', $order) }}">
        @csrf
        <button type="submit">
            {{ $order->status === OrderStatus::Queued ? 'Mark finished'
                                                      : 'Mark picked up' }}
        </button>
    </form>
@endif
```

Listing 7.25 (Continued) *resources/views/orders/show.blade.php: one order with its workflow button, payments and payment form.*

```
<h2>Payments</h2>
<table class="list">
    <thead><tr><th>Paid at</th><th>Method</th><th class="num">Amount</th></tr>
    </thead>
    <tbody>
    @foreach ($order->payments as $p)
        <tr><td>{{ $p->paid_at->format('d/m H:i') }}</td>
            <td>{{ $p->method->label() }}</td>
            <td class="num">{{ Money::rp($p->amount) }}</td></tr>
    @endforeach
    </tbody>
</table>

@if ($order->balance() > 0)
    <form method="post" action="{{ route('orders.payments.store', $order) }}"
          class="inline">
        @csrf
        <input name="amount" type="number" min="1"
               value="{{ old('amount', $order->balance()) }}">
        <select name="method">
            @foreach (PaymentMethod::cases() as $m)
                <option value="{{ $m->value }}">{{ $m->label() }}</option>
            @endforeach
        </select>
        <button type="submit">Record payment</button>
    </form>
    @error('amount') <p class="error">{{ $message }}</p> @enderror
    @error('method') <p class="error">{{ $message }}</p> @enderror
@endif
@endsection
```

The other views (the lists, the order form, the price pages, the public pages) follow the same patterns and are in the code folder. Placing any of them next to its Razor counterpart from Chapters 3 and 5 is a good exercise in reading both template languages: `@foreach` and `@if` exist in both, `asp-for` becomes `name` plus `old()`, `asp-validation-for` becomes `@error`, `<partial>` becomes `@include`, and `asp-route-id` becomes `route(..., $model)`.

7.10.4 Razor and Blade side by side

The two template languages solve the same problems with different punctuation. **Table 7.5** pairs the constructs used in the laundry's views.

Table 7.5 Razor and Blade constructs for the same tasks.

Task	Razor (Chapters 3 to 6)	Blade (this chapter)
print a value, escaped	`@Model.Name`	`{{ $customer->name }}`
print HTML unescaped	`@Html.Raw(html)`	`{!! $html !!}`
loop	`@foreach (var c in Model) { … }`	`@foreach ($customers as $c) … @endforeach`
condition	`@if (…) { … } else { … }`	`@if (…) … @else … @endif`
layout	`Layout = "_Layout",@RenderBody()`	`@extends('layouts.app'), @yield('content')`
partial view	`<partial name="_Form" model="Model" />`	`@include('customers._form')`
link to an action	`<a asp-action="Details" asp-route-id="@c.Id">`	`<a href="{{ route('customers.show', $c) }}">`
form field with value	`<input asp-for="Name" />`	`<input name="name" value="{{ old('name', $customer->name) }}">`
field error	`<span asp-validation-for="Name"></span>`	`@error('name') {{ $message }} @enderror`
anti-forgery token	automatic in `<form method="post">`	`@csrf`
flash message	`TempData["Message"]`	`session('message'),@session`

The deepest difference is in how the two know about the model. A Razor view declares `@model Customer`, and the compiler checks every `Model.Name` against the class; a mistake fails the build, which is why the type check described in Section 5.12 could cover the views. A Blade view receives loose variables from `view('customers.show', compact('customer'))`, and a misspelled `$customer->nmae` prints nothing until someone looks at the page. Tag helpers such as `asp-for` also write the field's name, value, type and validation attributes for you, while Blade leaves the HTML to you and gives you `old()` and `@error` to fill it. Neither is better in every way: Blade is closer to the HTML you already know, and Razor catches more mistakes before the page ever runs.

7.11 Running It

Where this book was written, Packagist (Composer's package registry) could not be reached, so no Laravel project could be created or run there. Every PHP file of the chapter was checked with `php -l` (PHP 8.4), the Blade files were checked by a script for balanced directives, and `tools/enum_demo.php` was run; its output in Section 7.6.1 is real. On your computer:

```
composer create-project laravel/laravel profitina-laundry "^13.0"
cd profitina-laundry
cp -R ../Chapter07/laravel/. .        # Windows: Copy-Item -Recurse -Force
php artisan migrate:fresh --seed
php artisan route:list --except-vendor
php artisan serve --port=4070
```

Set `'timezone' => 'Asia/Jakarta'` in `config/app.php` first, so that "today" on the counter page starts at midnight in Surabaya rather than at 07:00 (midnight UTC). `route:list` prints the routes of **Listing 7.15** with their names and controller methods; check that all resource routes are there before opening the browser. Then take the same tour as in Section 5.12:

1. `http://localhost:4070` shows the three popular services; `/services` and `/outlets` show the Chapter 3 data from the database.
2. **Counter** shows two orders in the queue, one ready for pickup and two received today.
3. **Customers**: search for `sari` in lower case; unlike the Chapter 5 version, *Sari Wulandari* is found, because `LIKE` ignores case.
4. Add a customer with an existing phone number; the form returns with *The phone has already been taken.*, Laravel's default message for `unique`, and the typed values are still in the fields.
5. **Orders → New order**: 2 kg of *Wash, dry & fold* for Dewi Lestari; the message says Rp 24.000. Tick pickup without an address; the message *The delivery address field is required when pickup is accepted.* appears.
6. On the order page, record a payment and advance the order twice.
7. **Prices**: change a price; `/services` shows the new price.

Also open `/up`: every Laravel 11 or newer application answers there with a small health page, the counterpart of the `/health` endpoint that Chapter 6 added by hand. The staff pages of this chapter have no sign-in yet; Laravel's authentication is the subject of Chapter 10. Until then, run the application only on your own computer.

7.12 Making It Better

- **Catch N+1 queries early.** `Model::preventLazyLoading(! app()->isProduction());` in `AppServiceProvider::boot()` makes Laravel throw an exception in development whenever a relationship is loaded lazily inside a loop. Forgetting a `with` then fails loudly on your laptop instead of slowly in the shop.
- **Move validation into form requests.** The rules in `store` and `update` grow with every requirement. A *form request* class (`php artisan make:request StoreOrderRequest`) holds the rules, the messages and even the authorization of one form, and the controller receives only validated input. Chapter 9 does this.
- **Write messages in the shop's language.** `php artisan lang:publish` copies Laravel's validation messages into `lang/en/validation.php`, where they can be rewritten; adding `lang/id/validation.php` and setting `APP_LOCALE=id` shows them in Indonesian.
- **Page long lists.** `->paginate(20)` instead of `->get()` returns one page and the links to the others; Chapter 8 (Problem 8.4) adds it in both frameworks.
- **Use Blade components for repeated markup.** A form field with its label and error message appears dozens of times; an anonymous component `<x-field name="phone" label="Phone (WhatsApp)" :value="$customer->phone" />` in `resources/views/components/field.blade.php` writes it once.

7.13 Using It for Real

- **MySQL instead of SQLite.** Change a few lines in `.env`: `DB_CONNECTION=mysql`, `DB_HOST`, `DB_DATABASE`, `DB_USERNAME` and `DB_PASSWORD`, then `php artisan migrate --seed`. The migrations are written in Laravel's schema builder, not in SQL, so they run unchanged on SQLite, MySQL, MariaDB, PostgreSQL and SQL Server. That is one advantage of writing migrations by hand in a portable language rather than generating them for one provider.
- **Shared PHP hosting.** Most Indonesian hosting plans support PHP 8.3 or newer and Composer. Upload the project, point the domain's document root at `public/` (never at the project folder,

which would expose `.env`), set `APP_ENV=production` and `APP_DEBUG=false`, and run `php artisan migrate --force` and `php artisan optimize`.

- **Your own services.** Replace the CSV files in `database/data/` before the first seeding, or use the Prices page afterwards, exactly as in Chapter 5.
- **Backups.** With SQLite the whole shop is `database/database.sqlite`. Laravel's scheduler can copy it every night: a scheduled closure in `routes/console.php` that runs `copy(database_path('database.sqlite'), storage_path('backups/laundry-'.date('Y-m-d').'.sqlite'))` daily, plus one cron entry on the server, `* * * * * php /path/to/artisan schedule:run`, which lets Laravel decide when each scheduled task is due. On MySQL, use `mysqldump` from cron instead, and in every case copy the backups to a second place.
- **Two frameworks, one shop.** Nothing forces a shop to choose. The ASP.NET Core version can run at the main outlet and the Laravel version at Wiyung, each with its own database, or both can share one MySQL database if their migrations are kept identical, which is harder than it sounds; the usual answer is one system of record and a small API between them, the subject of Chapters 10 and 14.
- **Personal data.** The same UU PDP duties as in Chapters 5 and 6 apply: collect only what the shop needs, protect the database file and its backups, and keep `.env`, which contains the application key, out of version control. The staff pages must not go online before Chapter 10's sign-in protects them.

7.14 Profitina in Practice

Profitina's own public site is built on PHP: its marketing site at profitina.com runs on WordPress with a custom bilingual (Indonesian and English) theme, and it publishes an `/llms.txt` file that tells AI answer engines what the site is about. The application at app.profitina.com, on the other hand, is written in Python with FastAPI. Running two stacks side by side is common, and for the same reason as in this chapter: each part uses the tool that suits it, while the ideas of routing, templates, an ORM and migrations stay the same across both. A developer who has built Profitina Laundry in ASP.NET Core and in Laravel can read either part of such a system on the first day.

7.15 Common Mistakes, and How to Avoid Them

- **Mass assignment without `#[Fillable]`.** `Customer::create($request->all())` on a model with no fillable list throws a `MassAssignmentException`; worse, making everything fillable with `$guarded = []` lets a request set any column. List the fillable attributes and pass only `$request->validate(...)`'s result.
- **Forgetting `@csrf`.** A form without the token is rejected with *419 Page Expired*. Every `POST`, `PUT` and `DELETE` form needs it.
- **Using `{!! !!}` for user data.** It prints without escaping and opens the page to cross-site scripting. Use `{{ }}` for everything that came from a user.
- **N+1 queries.** A loop that reads `$order->customer->name` without `with('customer')` sends one query per order. Load relationships with `with` or `load`, and switch on `preventLazyLoading` in development.
- **Running `migrate:fresh` on the server.** It drops every table. On the shop's server use `php artisan migrate --force`, which only runs new migrations.
- **Pointing the web server at the project folder.** Only `public/` may be reachable from the web; everything else, including `.env` and `database/database.sqlite`, must not.

- **Leaving `APP_DEBUG=true` in production.** Error pages then show code, settings and sometimes passwords. Production always runs with `APP_DEBUG=false`.
- **The wrong time zone.** Laravel's default time zone is UTC; without `Asia/Jakarta`, "today" starts at seven in the morning in Surabaya and orders received before seven count for yesterday.

7.16 Chapter Summary

- Laravel 13 (March 2026) runs on PHP 8.3 to 8.5; Composer installs it, Artisan creates files and runs migrations, and a new project uses SQLite by default.
- Migrations are written by hand with the schema builder and run in the order of their file names; `foreignId(...)->constrained()` creates foreign keys, and `restrictOnDelete` and `cascadeOnDelete` decide what deleting a parent does.
- Eloquent models are active records: each model knows its table, and objects save themselves. `#[Fillable]` limits mass assignment, `casts()` converts columns to PHP types and enums, accessors compute attributes, and methods such as `hasMany` and `belongsTo` declare relationships.
- `routes/web.php` holds every route; `Route::resource` declares the seven CRUD routes, and route model binding turns `{customer}` into a `Customer` or a 404.
- Controllers validate with `$request->validate(...)`, which redirects back with errors and old input when a rule fails, and use `with` for eager loading.
- Blade escapes everything printed with `{{ }}`, builds layouts with `@extends`, `@section` and `@yield`, shares fields with `@include`, and protects forms with `@csrf`.
- The Laravel version and the ASP.NET Core version share the database design, the pages and the rules; they differ in where things live and how much is done by convention.

7.17 Exercises

1. Write down the SQL that `Customer::query()->when($search !== '', …)->orderBy('name')->get()` sends for the search `an`, and list the customers of the sample data it returns. How does the result differ from the Chapter 5 version for the search `AN`?
2. Using **Table 7.4**, give the method, address and controller method of each of these actions: showing the edit form for customer 2, saving that form, deleting customer 4, and showing the list filtered by the search `budi`.
3. Add the public service page of Chapter 3's price quote to Laravel: a route `GET /services/{service}/quote?quantity=4.5&pickup=1` that shows the billed quantity, subtotal, fee and total for the given service. Write the route and the controller method.
4. The Blade view `orders/index` uses `$o->balance()`, which sums `$o->payments`. What happens to the number of queries if the controller forgets `'payments'` in `with([...])`, for 20 orders? What if `preventLazyLoading` is switched on?
5. Write a migration that adds a nullable `rack` column of at most 8 characters to `orders`, after `note`, with a `down()` that removes it. Which Artisan command creates the file, and which one runs it?
6. Compare the two ways of refusing to delete a customer with orders: the `destroy` method's check and the `restrictOnDelete()` of the migration. What does each protect against, and why keep both?

7.18 Solutions to the Exercises

1. `select * from "customers" where "name" like ? or "phone" like ? order by "name" asc`, with both bindings `%an%`. It returns Budi Santoso and Sari Wulandari (Dewi Lestari contains no `an`). For `AN`, SQLite's `LIKE` ignores case for ASCII letters, so the result is the same; the Chapter 5 version uses `instr`, which is case-sensitive, and finds nobody for `AN`.

2. `GET /customers/2/edit` → `edit`; `PUT /customers/2` (a `POST` with `_method=PUT`) → `update`; `DELETE /customers/4` (a `POST` with `_method=DELETE`) → `destroy`; `GET /customers?search=budi` → `index`.

3. The route and the method below (**Listing 7.26**); the view prints the five values. Route model binding supplies the service, and validation keeps the query string honest.

Listing 7.26 Exercise 7.3: a price-quote route and controller method (add `use Illuminate\Http\Request;` to the controller).

```
// routes/web.php
Route::get('services/{service}/quote', [ServiceController::class, 'quote'])
    ->name('services.quote');

// app/Http/Controllers/ServiceController.php
public function quote(Request $request, Service $service)
{
    $data = $request->validate(['quantity' => ['required', 'numeric',
        'min:0.1', 'max:100']]);
    $quantity = (float) $data['quantity'];
    $billed = $service->per_kg ? max($quantity, 3) : ceil($quantity);
    $subtotal = (int) round($billed * $service->price);
    $fee = $request->boolean('pickup') && $subtotal < 100000 ? 10000 : 0;

    return view('services.quote', compact('service', 'quantity', 'billed',
        'subtotal', 'fee') + ['total' => $subtotal + $fee]);
}
```

For `S0001` with 2 kg and pickup: billed 3, subtotal Rp 24.000, fee Rp 10.000, total Rp 34.000.

4. Without `'payments'`, every `$o->balance()` triggers a lazy load of that order's payments: 20 extra queries, 1 + 2 + 20 = 23 in all instead of 4. With `preventLazyLoading` on, the first `$o->payments` throws a `LazyLoadingViolationException` naming `App\Models\Order::payments`, and the mistake is found at once.

5. `php artisan make:migration add_rack_to_orders_table --table=orders` creates the file; `php artisan migrate` runs it. Its body:

Listing 7.27 Exercise 7.5: the `up` and `down` methods of the migration that adds a rack column.

```
public function up(): void
{
    Schema::table('orders', function (Blueprint $table) {
        $table->string('rack', 8)->nullable()->after('note');
    });
}

public function down(): void
{
    Schema::table('orders', function (Blueprint $table) {
        $table->dropColumn('rack');
    });
}
```

`after('note')` only affects MySQL and MariaDB, where columns have a visible order; SQLite adds the column at the end. Chapter 8 uses this migration in Problem 8.7 (**Listing 7.27**).

6. The check in `destroy` protects the user: it turns the situation into a sentence the cashier understands and keeps the page working. The foreign key protects the data: it holds even if a future piece of code, a `tinker` session or a hand-written SQL statement tries to delete the customer without the check. The first is good manners, the second is a guarantee, and a shop needs both.

References

[1] Laravel, "Release notes, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/releases

[2] Laravel, "Installation, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/installation

[3] The PHP Group, "PHP 8.5 released," Nov. 20, 2025. [Online]. Available: https://www.php.net/releases/8.5/en.php

[4] Laravel, "Database: migrations, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/migrations

[5] The PHP Group, "Enumerations," PHP Manual, accessed Sep. 2026. [Online]. Available: https://www.php.net/manual/en/language.enumerations.php

[6] Laravel, "Eloquent: getting started, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/eloquent

[7] Laravel Daily, "PHP attributes in Laravel 13: the ultimate guide," 2026. [Online]. Available: https://laraveldaily.com/post/php-attributes-in-laravel-13-the-ultimate-guide-36-new-attributes

[8] Laravel, "Eloquent: relationships, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/eloquent-relationships

[9] Laravel, "Routing: route model binding, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/routing#route-model-binding

[10] Laravel, "Controllers: resource controllers, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/controllers#resource-controllers

[11] Laravel, "Validation, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/validation

[12] Laravel, "Blade templates, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/blade

[13] Laravel, "CSRF protection, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/csrf

Chapter 8

Evaluation and Discussion II: The Same Feature Set in ASP.NET Core and Laravel

This chapter looks back over Chapters 5, 6 and 7 by solving seven real problems of Profitina Laundry twice, once in the ASP.NET Core application of Chapter 6 and once in the Laravel application of Chapter 7. Each problem is worked the same way: the problem as the shop states it, an understanding by hand with the shop's own numbers, a design on paper, the ASP.NET Core solution, the Laravel solution, and a tempting wrong solution with the reason it fails. The problems cover the four places where framework code lives: plain domain classes that know nothing about the web (the price rule and the order workflow), queries (the order list, the customer search and the revenue report), validation (a payment that is too large) and the schema (a new column). By the end of the chapter you will be able to decide for a new requirement which layer it belongs to, write it in either framework, check that both frameworks give the same answer, and recognize the classic mistakes before they reach a shop.

8.1 Where We Left Off

Chapter 5 connected the ASP.NET Core application of Chapter 3 to a SQLite database with Entity Framework Core: entity classes, a `DbContext`, migrations, seeding from the Chapter 3 CSV files, LINQ queries translated into SQL, and CRUD pages for customers, prices, orders and payments. Chapter 6 made that application trustworthy: validation attributes, form classes and a two-field rule; ASP.NET Core Identity with hashed passwords, lockout and the roles Owner and Cashier; a fallback policy that closes every page not explicitly opened; and a deployment to a Linux or Windows server. Chapter 7 built the same shop in Laravel 13: migrations written by hand, Eloquent models with casts, enums and relationships, resource routes with route model binding, validation in controllers, and Blade views.

The two applications now do the same things with the same data. That is exactly the situation in which differences show: a rule written slightly differently in two places, a query that is fast in one and slow in the other, a check present on one side only. The seven problems of this chapter come from that situation. They are the requests the owner made in the weeks after both versions went live.

8.2 How to Approach a Framework Problem

8.2.1 Six steps

Every problem in this chapter is solved in the same six steps, and they work for any feature in any framework.

1. **Say it in the shop's words.** "A customer must not be able to pay more than they owe" is a requirement; "add a max rule to the validator" is already a solution. Start with the requirement.

2. **Work an example by hand.** Take real numbers from the sample data and compute the expected answer on paper. That answer becomes the test.
3. **Decide the layer.** Does the rule need the web (a request, a form, a page)? The database? Neither? Rules that need neither go into a plain class that both frameworks can call and that can be run without a server. Rules about one form go into validation; questions about many rows go into a query; changes to what is stored go into a migration.
4. **Write it in one framework.** Usually the one you know best.
5. **Port it, then compare.** Write it in the other framework and run both on the example of step 2. They must agree to the rupiah.
6. **Try to break it.** Look for the tempting wrong solution: the version that works on the example and fails on the next one.

Figure 8.1 maps the seven problems onto the layers of step 3.

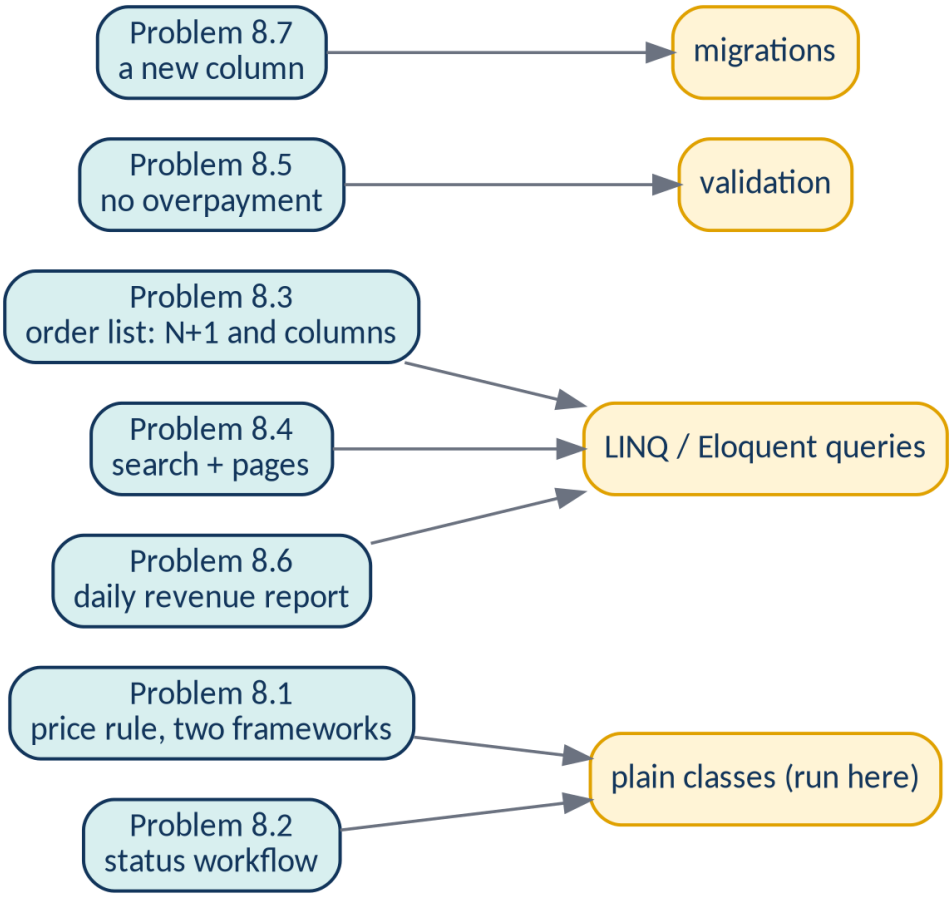


Figure 8.1 The seven problems of this chapter and the layer each one belongs to.

8.2.2 Where the code lives, and what was run

The solutions are complete applications: `Chapter08/aspnet/LaundryMvc` is the Chapter 6 application with every solution added, and `Chapter08/laravel` holds the Chapter 7 files with every solution added. Two small programs check the domain rules without a web server: `Chapter08/aspnet/CoreCheck` in C# and `Chapter08/laravel/tools/core_check.php` in plain PHP. Where this book was written, NuGet and Packagist could not be reached, so the two web applications could not be run there; the C# application was compiled against compile-only stand-ins for EF Core, and every PHP file passed `php -l`. The two check programs and the PHP

demonstration of the float mistake were run, and every output shown in this chapter is their real output.

8.3 Problem 8.1: One Price Rule, Two Frameworks

8.3.1 The problem

The price rules of Chapter 3 are: per-kg services are billed for at least 3 kg; services per item or per pair are billed in whole items, rounded up; pickup and delivery cost Rp 10.000, free from a subtotal of Rp 100.000. The C# application computes them in `PriceCalculator` with `decimal` arithmetic. The Laravel application of Chapter 7 computes them inline in `OrderController::store` with PHP floats. The owner asks for one thing: *the two systems must always charge the same price, for every quantity a cashier can type*. Write a PHP class that guarantees it, and show that the two agree.

8.3.2 Understanding it by hand

Take *Wash, dry & fold* at Rp 8.000 per kg. For 4.5 kg the bill is $4.5 \times 8000 = 36000$. For 2 kg the minimum applies: $3 \times 8000 = 24000$. For 4.02 kg: $4.02 \times 8000 = 32160$. *Shoes deep clean* at Rp 45.000 per pair for 1.5 pairs is billed as 2 pairs, Rp 90.000. *Bed cover* at Rp 35.000 for 2 items with pickup: $70000 + 10000 = 80000$. *Dry cleaning, shirt* at Rp 20.000 for 5 items with pickup: the subtotal is exactly Rp 100.000, so pickup is free and the total is Rp 100.000.

Every quantity the counter accepts has at most two decimals, so it is a whole number of hundredths: 4.5 kg is 450 hundredths, 4.02 kg is 402. Let h be the quantity in hundredths and P the price per unit in rupiah. The billed quantity b, also in hundredths, is

$$b = \begin{cases} \max(h,\ 300) & \text{per kg} \\ 100 \cdot \lceil h/100 \rceil & \text{per item or pair} \end{cases} \tag{8.1}$$

and the subtotal in rupiah, rounded half up to whole rupiah, is

$$S = \left\lfloor \frac{P \cdot b + 50}{100} \right\rfloor \tag{8.2}$$

For 4.02 kg: $b = \max(402,300) = 402$ by Equation (8.1), and $S = \lfloor (8000 \cdot 402 + 50)/100 \rfloor = \lfloor 3216050/100 \rfloor = 32160$ by Equation (8.2). Every step is integer arithmetic: multiplication, addition and whole-number division. The total is S plus Rp 10.000 when pickup is chosen and $S < 100000$.

8.3.3 Designing it on paper

The rule needs neither a request nor the database, so it belongs in a plain class: `App\Domain\PriceCalculator` in PHP, next to the existing `PriceCalculator` in C#. The PHP class receives the quantity as a *string*, exactly as it arrives from the form, and converts it to hundredths without ever becoming a float. A check program in each language runs the same nine cases and prints the same lines, so that comparing the two outputs is a single `diff`.

8.3.4 The ASP.NET Core solution

The C# side needs no new code. **Listing 8.1** is the `PriceCalculator` of Chapter 3, unchanged since then. `decimal` stores 4.02 exactly, so `billed * service.Price` is exact, and the application rounds the total half up with `MidpointRounding.AwayFromZero` when it stores it as whole rupiah.

Listing 8.1 `Models/PriceCalculator.cs`: the price rules of Chapter 3 in C#, with `decimal` arithmetic.

```
namespace LaundryMvc.Models;

public record Quote(Service Service, decimal Quantity, decimal Billed,
    decimal Subtotal, decimal PickupFee, decimal Total, List<string> Notes);

// The shop's pricing rules live here, not in a controller or a view.
public class PriceCalculator
{
    public const decimal MinimumKg = 3m;
    public const decimal PickupFee = 10_000m;
    public const decimal FreePickupFrom = 100_000m;

    public Quote Calculate(Service service, decimal quantity, bool pickup)
    {
        var notes = new List<string>();
        decimal billed = quantity;
        if (service.PerKg && quantity < MinimumKg)
        {
            billed = MinimumKg;
            notes.Add($"Minimum {MinimumKg} kg, so {quantity} kg is billed "
                + $"as {MinimumKg} kg.");
        }
        else if (!service.PerKg && quantity != Math.Ceiling(quantity))
        {
            billed = Math.Ceiling(quantity);
            notes.Add($"Items are counted whole: {quantity} becomes {billed}.");
        }
        decimal subtotal = billed * service.Price;
        decimal fee = 0m;
        if (pickup)
        {
            fee = subtotal >= FreePickupFrom ? 0m : PickupFee;
            notes.Add(fee == 0m
                ? $"Pickup and delivery are free from {Money.Rp(FreePickupFrom)}."
                : $"Pickup and delivery cost {Money.Rp(PickupFee)}.");
        }
        return new Quote(service, quantity, billed, subtotal, fee,
            subtotal + fee, notes);
    }
}
```

`CoreCheck` (**Listing 8.2**) compiles the application's model classes and the workflow class of Problem 8.2 and runs them on fixed cases. Its real output follows.

Listing 8.2 `CoreCheck/Program.cs`: the price rules and the workflow on fixed cases, in C#.

```
using LaundryMvc.Domain;
using LaundryMvc.Models;

// The same cases as core_check.php; both programs must print the same lines.
(string Description, decimal Price, decimal Qty, bool Pickup)[] cases =
[
    ("Wash, dry & fold (per kg)", 8000, 4.5m, false),
    ("Wash, dry & fold (per kg)", 8000, 2m, false),
    ("Wash, dry & fold (per kg)", 8000, 4.02m, false),
    ("Ironing only (per kg)", 6000, 3.35m, false),
    ("Curtain (per kg)", 15000, 4.4m, true),
    ("Bed cover, any size (per item)", 35000, 2m, true),
    ("Dry cleaning - suit 2 pieces (per item)", 65000, 2m, true),
    ("Dry cleaning - shirt/blouse (per item)", 20000, 5m, true),
    ("Shoes deep clean (per pair)", 45000, 1.5m, false),
];
var calculator = new PriceCalculator();
Console.WriteLine("Prices (Problem 8.1)");
foreach (var c in cases)
{
    var service = new Service { Description = c.Description, Price = c.Price };
    var quote = calculator.Calculate(service, c.Qty, c.Pickup);
    var total = (int)decimal.Round(quote.Total, MidpointRounding.AwayFromZero);
    Console.WriteLine($"  {service.Name,-28} {c.Qty,5:0.00} {service.Unit,-4} "
                      + $"{(c.Pickup ? "pickup" : ""),-6} {total,7}");
}

Console.WriteLine("Workflow (Problem 8.2)");
(OrderStatus Status, int Balance)[] steps =
[
    (OrderStatus.Queued, 25000), (OrderStatus.Finished, 25000),
    (OrderStatus.Finished, 0), (OrderStatus.Departed, 0),
];
foreach (var (status, balance) in steps)
{
    try
    {
        var next = OrderWorkflow.Advance(status, balance);
        Console.WriteLine($"  {status,-8} owing {balance,6} -> {next}");
    }
    catch (InvalidOperationException e)
    {
        Console.WriteLine($"  {status,-8} owing {balance,6} -> {e.Message}");
    }
}
```

```
Prices (Problem 8.1)
  Wash, dry & fold              4.50 kg            36000
  Wash, dry & fold              2.00 kg            24000
  Wash, dry & fold              4.02 kg            32160
  Ironing only                  3.35 kg            20100
  Curtain                       4.40 kg   pickup   76000
  Bed cover, any size           2.00 item pickup   80000
  Dry cleaning - suit 2 pieces  2.00 item pickup  130000
  Dry cleaning - shirt/blouse   5.00 item pickup  100000
  Shoes deep clean              1.50 pair          90000
Workflow (Problem 8.2)
  Queued   owing  25000 -> Finished
  Finished owing  25000 -> Collect Rp 25.000 before handing over the laundry.
  Finished owing      0 -> Departed
  Departed owing      0 -> This order has already left the shop.
```

8.3.5 The Laravel solution

Listing 8.3 implements Equation (8.1) and Equation (8.2) literally. `hundredths` reads the quantity with a regular expression, whole part and up to two decimals, and refuses anything else; `str_pad($m[2] ?? '', 2, '0')` turns `"5"` in `"4.5"` into 50 hundredths and `"02"` in `"4.02"` into 2. From there on, only `intdiv`, `+` and `*` on integers are used.

Listing 8.3 `app/Domain/PriceCalculator.php`: the same rules in PHP, with exact integer arithmetic.

```php
<?php

namespace App\Domain;

use InvalidArgumentException;

// Problem 8.1: the price rules of Chapter 3, with exact integer arithmetic,
// so that Laravel charges exactly what the ASP.NET Core version charges.
final class PriceCalculator
{
    public const MINIMUM_KG = 3;
    public const PICKUP_FEE = 10000;
    public const FREE_PICKUP_FROM = 100000;

    public static function total(bool $perKg, int $price, string $quantity,
        bool $pickup): int
    {
        $hundredths = self::hundredths($quantity);
        $billed = $perKg
            ? max($hundredths, self::MINIMUM_KG * 100)    // at least 3 kg
            : intdiv($hundredths + 99, 100) * 100;        // whole items
        $subtotal = intdiv($price * $billed + 50, 100);   // half up
        $fee = $pickup && $subtotal < self::FREE_PICKUP_FROM
            ? self::PICKUP_FEE : 0;

        return $subtotal + $fee;
    }

    // "4.5" -> 450, "3.35" -> 335: read as text, never as a float.
    public static function hundredths(string $quantity): int
    {
        if (! preg_match('/^(\d+)(?:\.(\d{1,2}))?$/', trim($quantity), $m)) {
            throw new InvalidArgumentException(
                "Not a quantity with at most two decimals: {$quantity}");
        }

        return (int) $m[1] * 100 + (int) str_pad($m[2] ?? '', 2, '0');
    }
}
```

The check program (**Listing 8.4**) loads the classes with `require`, since no Laravel is needed, and runs the same cases.

Listing 8.4 `tools/core_check.php`: the same cases as `CoreCheck`, in plain PHP.

```php
<?php
// Plain PHP, no Laravel needed: php tools/core_check.php
// The same cases as CoreCheck in C#; both must print the same lines.

foreach (['Enums/OrderStatus', 'Support/Money', 'Domain/PriceCalculator',
    'Domain/OrderWorkflow'] as $file) {
    require __DIR__ . "/../app/{$file}.php";
}

use App\Domain\OrderWorkflow;
use App\Domain\PriceCalculator;
use App\Enums\OrderStatus;

$cases = [
    ['Wash, dry & fold', 'kg', 8000, '4.5', false],
    ['Wash, dry & fold', 'kg', 8000, '2', false],
    ['Wash, dry & fold', 'kg', 8000, '4.02', false],
    ['Ironing only', 'kg', 6000, '3.35', false],
    ['Curtain', 'kg', 15000, '4.4', true],
    ['Bed cover, any size', 'item', 35000, '2', true],
    ['Dry cleaning - suit 2 pieces', 'item', 65000, '2', true],
    ['Dry cleaning - shirt/blouse', 'item', 20000, '5', true],
    ['Shoes deep clean', 'pair', 45000, '1.5', false],
];
echo "Prices (Problem 8.1)\n";
foreach ($cases as [$name, $unit, $price, $qty, $pickup]) {
    $total = PriceCalculator::total($unit === 'kg', $price, $qty, $pickup);
    printf("  %-28s %5.2f %-4s %-6s %7d\n", $name, $qty, $unit,
        $pickup ? 'pickup' : '', $total);
}

echo "Workflow (Problem 8.2)\n";
$steps = [[OrderStatus::Queued, 25000], [OrderStatus::Finished, 25000],
    [OrderStatus::Finished, 0], [OrderStatus::Departed, 0]];
foreach ($steps as [$status, $balance]) {
    try {
        $result = OrderWorkflow::advance($status, $balance)->label();
    } catch (DomainException $e) {
        $result = $e->getMessage();
    }
    printf("  %-8s owing %6d -> %s\n", $status->label(), $balance, $result);
}
```

Its real output is identical, line for line, to the C# output above; `diff` between the two prints nothing. The order controller of Problem 8.3 (**Listing 8.12**) calls `PriceCalculator::total($service->per_kg, $service->price, (string) $data['quantity'], $pickup)`, and the validation rule `decimal:0,2` guarantees that the quantity has at most two decimals before the calculator sees it. On the C# side, the same guarantee comes from a new rule in `OrderForm.Validate` (Problem 8.7, **Listing 8.23**).

8.3.6 A tempting wrong solution

The obvious PHP is `(int) ($service->price * $quantity)`, and it is right for most quantities. `tools/wrong_price.php` (**Listing 8.5**) prints what happens for three of them.

Listing 8.5 `tools/wrong_price.php`: the price computed with floats, three ways.

```php
<?php
// Tempting wrong solutions to Problem 8.1: arithmetic with floats.
// php tools/wrong_price.php

require __DIR__ . '/../app/Domain/PriceCalculator.php';

use App\Domain\PriceCalculator;

foreach (['4.02', '8.03', '4.5'] as $qty) {
    $float = 8000 * (float) $qty;                  // 8000 * 4.02 as a float
    printf("%s kg x Rp 8000: float %s, (int) %d, round %d, exact %d\n",
        $qty, var_export($float, true), (int) $float, (int) round($float),
        PriceCalculator::total(true, 8000, $qty, false));
}
```

```
4.02 kg x Rp 8000: float 32159.999999999996, (int) 32159, round 32160, exact 32160
8.03 kg x Rp 8000: float 64239.99999999999, (int) 64239, round 64240, exact 64240
4.5 kg x Rp 8000: float 36000.0, (int) 36000, round 36000, exact 36000
```

The number 4.02 has no exact binary representation, just as 1/3 has no exact decimal one; the float closest to it is a little smaller, and 8000 × 4.02 comes out as 32159.999999999996 [1], [2]. Casting to `int` cuts off the fraction and bills Rp 32.159: one rupiah less than the C# application for the same order, on an ordinary quantity. With `round` the result is correct here, which is why the Chapter 7 code worked on every example in that chapter; but a rule that is correct only because a rounding step happens to hide the error is fragile, and a later change (a discount of 10%, a surcharge of 7.5%) can make the hidden error visible again. Counting in integer hundredths removes the problem instead of hiding it.

8.4 Problem 8.2: The Order Workflow in One Place

8.4.1 The problem

After a customer left with her laundry without paying the Rp 16.000 she still owed, the owner set a new rule: *an order may not be marked as picked up while money is owed*. The rest of the workflow stays: queued, then finished, then departed, and nothing after departed.

8.4.2 Understanding it by hand

Budi Santoso's express order in the sample data costs Rp 36.000, and Rp 20.000 was paid. Marking it finished is allowed. Marking it picked up must be refused with a message that says how much to collect, Rp 16.000; after a payment of Rp 16.000 it must be allowed. Sari Wulandari's order is finished and fully paid, so marking it picked up is allowed. An order that has departed cannot move at all.

8.4.3 Designing it on paper

The decision needs two facts, the current status and the balance, and it produces either the next status or a reason. It needs no web and no database, so it becomes a plain class with one function, `Advance(status, balance)`, that returns the next status or throws an exception with the message. The controllers keep what does need the web and the database: loading the order with its payments, stamping the time, saving, and showing the message.

8.4.4 The ASP.NET Core solution

Listing 8.6 is the workflow class. The C# `switch` expression maps each status to the next, and `??` turns "no next status" into an exception in one line.

Listing 8.6 `Domain/OrderWorkflow.cs`: the workflow and the new rule, as a plain class.

```
using LaundryMvc.Models;

namespace LaundryMvc.Domain;

// Problem 8.2: which status may follow which, and when.
public static class OrderWorkflow
{
    public static OrderStatus? Next(OrderStatus status) => status switch
    {
        OrderStatus.Queued => OrderStatus.Finished,
        OrderStatus.Finished => OrderStatus.Departed,
        _ => null
    };

    // The next status, or an exception that explains why there is none.
    public static OrderStatus Advance(OrderStatus status, int balance)
    {
        var next = Next(status)
            ?? throw new InvalidOperationException(
                "This order has already left the shop.");
        if (next == OrderStatus.Departed && balance > 0)
        {
            throw new InvalidOperationException(
                $"Collect {Money.Rp(balance)} before handing over the laundry.");
        }
        return next;
    }
}
```

The `Advance` action of `OrdersController` (**Listing 8.10**, shown in full in Problem 8.3) loads the order *with its payments*, calls `OrderWorkflow.Advance(order.Status, order.Balance)`, catches the exception and shows its message, and otherwise stamps `FinishedAt` or `DepartedAt`. The second half of the `CoreCheck` output above shows the four cases of Section 8.4.2 giving the expected answers.

8.4.5 The Laravel solution

Listing 8.7 is the same class in PHP. The status enum of Chapter 7 already knows the next status, so the class only adds the balance rule; `?? throw` works in PHP 8 exactly as in C#.

Listing 8.7 `app/Domain/OrderWorkflow.php`: the workflow and the new rule in PHP.

```php
<?php

namespace App\Domain;

use App\Enums\OrderStatus;
use App\Support\Money;
use DomainException;

// Problem 8.2: which status may follow which, and when.
final class OrderWorkflow
{
    // The next status, or an exception that explains why there is none.
    public static function advance(OrderStatus $status, int $balance): OrderStatus
    {
        $next = $status->next()
            ?? throw new DomainException('This order has already left the shop.');
        if ($next === OrderStatus::Departed && $balance > 0) {
            throw new DomainException('Collect ' . Money::rp($balance)
                . ' before handing over the laundry.');
        }

        return $next;
    }
}
```

In `OrderController::advance` (**Listing 8.12**) the balance is computed in the database with `$order->total - (int) $order->payments()->sum('amount')`: one `SELECT SUM(...)`, no payment models loaded. The last lines of `core_check.php`'s output match the C# lines exactly, including the formatted amount in the message.

8.4.6 A tempting wrong solution

In C#, the natural first version loads the order with `await db.Orders.FindAsync(id)`, as the Chapter 5 `Advance` did, and then asks for `order.Balance`. It compiles, and it is wrong in a way no test with an unpaid order will catch: `FindAsync` loads the order row only, `Payments` is an empty list, `Paid` is 0 and `Balance` equals `Total`. Every order, paid or not, is now refused at pickup. The fix is the `Include(o => o.Payments)` in the solution, or computing the balance in the database as the Laravel version does. The lesson is the one from Section 5.6.2: a navigation property that was not loaded is not an error, it is an empty list, and anything computed from it is silently wrong.

8.5 Problem 8.3: The Order List, Without N+1 and Without Waste

8.5.1 The problem

After three months the shop has about 750 orders, and the order list, which shows all of them with every payment loaded, has become slow to open on the counter's old laptop. The owner asks for the list to open instantly, with 20 orders per page, newest first.

8.5.2 Understanding it by hand

Count the SQL statements for one page of 20 orders, which is what the database has to answer. In Laravel without eager loading, the view touches three relationships per order: 1 query for the orders plus $3 \times 20 = 60$, 61 in all. With `with(['customer', 'service', 'payments'])`, as in Chapter 7: 4. In EF Core with three `Include`s, as in Chapter 5: 1 statement with joins, but it returns every column of four tables and every payment row. A page needs only seven values per

order: number, date, customer name, service, total, amount paid and status. **Figure 8.2** summarizes the counts.

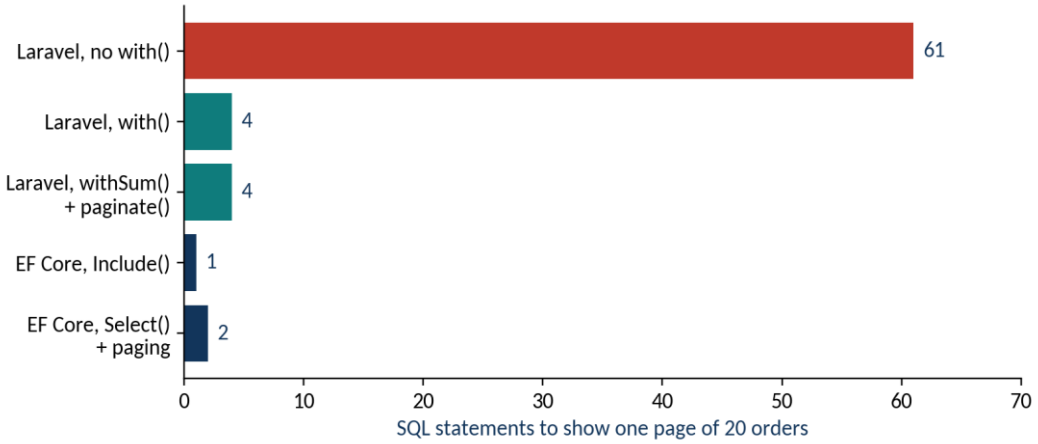


Figure 8.2 SQL statements needed to show one page of 20 orders, by the way the list is loaded; the counts follow from how each call is documented to load data.

8.5.3 Designing it on paper

Two ideas solve it in both frameworks. *Paging*: ask the database for one page only, and for the total count to draw the pager. *Projection*: ask only for the columns the page shows, and let the database add up the payments instead of shipping them. In EF Core, `Select` into a small record does both the choosing of columns and the summing; in Eloquent, `select([...])` chooses the order columns, `with(['customer:id,name', ...])` limits the related columns, and `withSum('payments', 'amount')` adds the sum as a computed column [3], [4].

8.5.4 The ASP.NET Core solution

The page shows `OrderRow` records (**Listing 8.8**), not entities.

Listing 8.8 `Models/OrderRow.cs`: exactly the columns of the order list.

```
namespace LaundryMvc.Models;

// Problem 8.3: exactly the columns the order list shows, nothing more.
public record OrderRow(int Id, DateTime ReceivedAt, string Customer,
                       string Service, int Total, int Paid, OrderStatus Status)
{
    public int Balance => Total - Paid;
}
```

`PagedList<T>` (**Listing 8.9**) runs two statements: `CountAsync()` for the total, and the query with `Skip` and `Take` for one page [5]. It clamps the page number, so `?page=99` shows the last page instead of an empty one.

Listing 8.9 `Domain/PagedList.cs`: one page of an ordered query and what the pager needs.

```
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Domain;

public interface IPaged
{
    int Page { get; }
    int PageCount { get; }
    int TotalCount { get; }
}

// Problem 8.4: one page of a query, plus what the pager needs to know.
public class PagedList<T>(List<T> items, int page, int pageCount,
                          int totalCount) : IPaged
{
    public List<T> Items { get; } = items;
    public int Page { get; } = page;
    public int PageCount { get; } = pageCount;
    public int TotalCount { get; } = totalCount;

    // The query must already be ordered, or pages may overlap or skip rows.
    public static async Task<PagedList<T>> CreateAsync(IQueryable<T> query,
                                                       int page, int pageSize)
    {
        var total = await query.CountAsync();
        var pageCount = Math.Max(1, (total + pageSize - 1) / pageSize);
        page = Math.Clamp(page, 1, pageCount);
        var items = await query.Skip((page - 1) * pageSize).Take(pageSize)
                               .ToListAsync();
        return new PagedList<T>(items, page, pageCount, total);
    }
}
```

Listing 8.10 is the whole `OrdersController` of this chapter; `Index` is at the top. The `Select` builds an `OrderRow` from `o.Customer!.Name`, `o.Service!.Description` and `o.Payments.Sum(p => p.Amount)`; EF Core translates the navigations into joins and the `Sum` into a correlated `SELECT SUM(...)` inside the same statement, and no entity is tracked, because the result is not an entity. The query is ordered by time and then by number, so that two orders received in the same second never swap places between pages.

Listing 8.10 `Controllers/OrdersController.cs`: the order list as a projection with pages, the workflow of Problem 8.2, and the payment check of Problem 8.5.

```
using LaundryMvc.Data;
using LaundryMvc.Domain;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Controllers;

public class OrdersController(LaundryDbContext db, PriceCalculator calculator)
    : Controller
{
    // Problem 8.3: one query with only the needed columns, 20 rows a page.
    public async Task<IActionResult> Index(OrderStatus? status, int page = 1)
    {
        var query = db.Orders.AsQueryable();
        if (status is not null)
        {
            query = query.Where(o => o.Status == status);
        }
        var rows = query
            .OrderByDescending(o => o.ReceivedAt).ThenByDescending(o => o.Id)
            .Select(o => new OrderRow(o.Id, o.ReceivedAt, o.Customer!.Name,
                o.Service!.Description, o.Total,
                o.Payments.Sum(p => p.Amount), o.Status));
        ViewData["Status"] = status;
        return View(await PagedList<OrderRow>.CreateAsync(rows, page, 20));
    }

    public async Task<IActionResult> Details(int id)
    {
        var order = await db.Orders
            .Include(o => o.Customer)
            .Include(o => o.Service)
            .Include(o => o.Payments)
            .AsNoTracking()
            .FirstOrDefaultAsync(o => o.Id == id);
        return order is null ? NotFound() : View(order);
    }

    public async Task<IActionResult> Create(int? customerId)
    {
        await FillListsAsync();
        return View(new OrderForm { CustomerId = customerId ?? 0 });
    }
```

Listing 8.10 (Continued) *Controllers/OrdersController.cs: the order list as a projection with pages, the workflow of Problem 8.2, and the payment check of Problem 8.5.*

```
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(OrderForm form)
    {
        var service = await db.Services.FindAsync(form.ServiceId ?? "");
        if (form.ServiceId is not null && (service is null || !service.IsActive))
        {
            ModelState.AddModelError(nameof(form.ServiceId),
                "This service is not offered any more.");
        }
        if (form.CustomerId > 0
            && !await db.Customers.AnyAsync(c => c.Id == form.CustomerId))
        {
            ModelState.AddModelError(nameof(form.CustomerId),
                "This customer does not exist.");
        }
        if (!ModelState.IsValid || service is null)
        {
            await FillListsAsync();
            return View(form);
        }

        var quote = calculator.Calculate(service, form.Quantity, form.Pickup);
        var order = new Order
        {
            CustomerId = form.CustomerId,
            ServiceId = service.Id,
            Quantity = form.Quantity,
            Pickup = form.Pickup,
            DeliveryAddress = form.Pickup ? form.DeliveryAddress : null,
            Note = form.Note,
            Rack = form.Rack,
            Total = (int)decimal.Round(quote.Total,
                                       MidpointRounding.AwayFromZero),
            Status = OrderStatus.Queued,
            ReceivedAt = DateTime.Now
        };
        db.Orders.Add(order);
        await db.SaveChangesAsync();
        TempData["Message"] = $"Order #{order.Id} created: {Money.Rp(order.Total)}."
            + (quote.Notes.Count > 0 ? " " + string.Join(" ", quote.Notes) : "");
        return RedirectToAction(nameof(Details), new { id = order.Id });
    }
```

Listing 8.10 (Continued) *Controllers/OrdersController.cs: the order list as a projection with pages, the workflow of Problem 8.2, and the payment check of Problem 8.5.*

```
    // Problem 8.2: the workflow decides; the controller stamps the time.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Advance(int id)
    {
        var order = await db.Orders.Include(o => o.Payments)
                                   .FirstOrDefaultAsync(o => o.Id == id);
        if (order is null) return NotFound();
        try
        {
            order.Status = OrderWorkflow.Advance(order.Status, order.Balance);
        }
        catch (InvalidOperationException e)
        {
            TempData["Error"] = e.Message;
            return RedirectToAction(nameof(Details), new { id });
        }
        if (order.Status == OrderStatus.Finished)
        {
            order.FinishedAt = DateTime.Now;
        }
        else
        {
            order.DepartedAt = DateTime.Now;
        }
        await db.SaveChangesAsync();
        TempData["Message"] = $"Order #{id} is now {order.Status}.";
        return RedirectToAction(nameof(Details), new { id });
    }

    // Problem 8.5: a payment may not be more than what is still owed.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> AddPayment(int id, PaymentForm payment)
    {
        var balance = await db.Orders.Where(o => o.Id == id)
            .Select(o => (int?)(o.Total - o.Payments.Sum(p => p.Amount)))
            .FirstOrDefaultAsync();
        if (balance is null) return NotFound();
        if (ModelState.IsValid && payment.Amount > balance)
        {
            ModelState.AddModelError(nameof(payment.Amount),
                $"Only {Money.Rp(balance.Value)} is still owed.");
        }
        if (!ModelState.IsValid)
        {
```

Listing 8.10 (Continued) *Controllers/OrdersController.cs: the order list as a projection with pages, the workflow of Problem 8.2, and the payment check of Problem 8.5.*

```
            TempData["Error"] = ModelState.Values.SelectMany(v => v.Errors)
                                             .First().ErrorMessage;
            return RedirectToAction(nameof(Details), new { id });
        }
        db.Payments.Add(new Payment
        {
            OrderId = id, Amount = payment.Amount, Method = payment.Method,
            PaidAt = DateTime.Now
        });
        await db.SaveChangesAsync();
        TempData["Message"] = $"Payment of {Money.Rp(payment.Amount)} recorded.";
        return RedirectToAction(nameof(Details), new { id });
    }

    async Task FillListsAsync()
    {
        ViewBag.Customers = new SelectList(
            await db.Customers.OrderBy(c => c.Name).ToListAsync(),
            nameof(Customer.Id), nameof(Customer.Name));
        ViewBag.Services = new SelectList(
            await db.Services.Where(s => s.IsActive).OrderBy(s => s.Id)
                             .ToListAsync(),
            nameof(Service.Id), nameof(Service.Description));
    }
}
```

The view (**Listing 8.11**) receives a `PagedList<OrderRow>` and ends with the shared pager partial of Problem 8.4.

Listing 8.11 `Views/Orders/Index.cshtml`: the order list from `OrderRow` records, with the pager.

```
@model PagedList<OrderRow>
@{
    ViewData["Title"] = "Orders";
    var current = ViewData["Status"] as OrderStatus?;
}
<h1>Orders</h1>
<p class="filters">
    <a asp-action="Index" class="@(current is null ? "on" : "")">All</a>
    @foreach (var s in Enum.GetValues<OrderStatus>())
    {
        <a asp-action="Index" asp-route-status="@s"
           class="@(current == s ? "on" : "")">@s</a>
    }
    <a class="button" asp-action="Create">New order</a>
</p>
<table class="list">
    <thead>
        <tr><th>#</th><th>Received</th><th>Customer</th><th>Service</th>
            <th class="num">Total</th><th class="num">Balance</th>
            <th>Status</th></tr>
    </thead>
    <tbody>
    @foreach (var o in Model.Items)
    {
        <tr>
            <td><a asp-action="Details" asp-route-id="@o.Id">@o.Id</a></td>
            <td>@o.ReceivedAt.ToString("dd/MM HH:mm")</td>
            <td>@o.Customer</td>
            <td>@o.Service</td>
            <td class="num">@Money.Rp(o.Total)</td>
            <td class="num">@Money.Rp(o.Balance)</td>
            <td>@o.Status</td>
        </tr>
    }
    </tbody>
</table>
<partial name="_Pager" model="Model" />
```

8.5.5 The Laravel solution

Listing 8.12 is the whole `OrderController` of this chapter. `index` selects six order columns, loads only `id` and `name` of the customers and `id` and `description` of the services, adds `payments_sum_amount` with `withSum`, and pages with `paginate(20)`. The eager-loaded relationships need their key columns (`customer_id`, `service_id` on the order, `id` on the related model), or Eloquent cannot connect the rows; that is why the select lists them.

Listing 8.12 `app/Http/Controllers/OrderController.php`: the order list with pages and sums, prices from Problem 8.1, and the workflow of Problem 8.2.

```php
<?php

namespace App\Http\Controllers;

use App\Domain\OrderWorkflow;
use App\Domain\PriceCalculator;
use App\Enums\OrderStatus;
use App\Models\Customer;
use App\Models\Order;
use App\Models\Service;
use App\Support\Money;
use DomainException;
use Illuminate\Http\Request;
use Illuminate\Validation\Rule;

class OrderController extends Controller
{
    // Problem 8.3: four queries in all, only the needed columns, 20 a page.
    public function index(Request $request)
    {
        $status = OrderStatus::tryFrom((string) $request->query('status'));
        $orders = Order::query()
            ->select(['id', 'customer_id', 'service_id', 'total', 'status',
                'received_at'])
            ->with(['customer:id,name', 'service:id,description'])
            ->withSum('payments', 'amount')     // adds payments_sum_amount
            ->when($status, fn ($query) => $query->where('status', $status))
            ->orderByDesc('received_at')->orderByDesc('id')
            ->paginate(20)
            ->withQueryString();

        return view('orders.index', compact('orders', 'status'));
    }

    public function show(Order $order)
    {
        $order->load(['customer', 'service',
            'payments' => fn ($q) => $q->orderBy('paid_at')]);

        return view('orders.show', compact('order'));
    }
```

Listing 8.12 (Continued) *app/Http/Controllers/OrderController.php: the order list with pages and sums, prices from Problem 8.1, and the workflow of Problem 8.2.*

```
    public function create(Request $request)
    {
        return view('orders.create', [
            'customers' => Customer::orderBy('name')->get(),
            'services' => Service::where('is_active', true)
                ->orderBy('id')->get(),
            'customerId' => $request->integer('customer_id'),
        ]);
    }

    public function store(Request $request)
    {
        $data = $request->validate([
            'customer_id' => ['required', 'integer', 'exists:customers,id'],
            'service_id' => ['required', 'string',
                Rule::exists('services', 'id')->where('is_active', true)],
            'quantity' => ['required', 'numeric', 'decimal:0,2', 'min:0.1',
                'max:100'],
            'delivery_address' => ['nullable', 'required_if_accepted:pickup',
                'string', 'max:200'],
            'rack' => ['nullable', 'regex:/^[A-Z]-[0-9]{2}$/'],
            'note' => ['nullable', 'string', 'max:200'],
        ], [
            'rack.regex' => 'A rack looks like B-03: a letter, a dash, two digits.',
        ]);
        $service = Service::findOrFail($data['service_id']);
        $pickup = $request->boolean('pickup');

        $order = Order::create([
            ...$data,
            'pickup' => $pickup,
            'total' => PriceCalculator::total($service->per_kg, $service->price,
                (string) $data['quantity'], $pickup),
            'status' => OrderStatus::Queued,
            'received_at' => now(),
        ]);

        return redirect()->route('orders.show', $order)->with('message',
            "Order #{$order->id} created: " . Money::rp($order->total) . '.');
    }
```

Listing 8.12 (Continued) *app/Http/Controllers/OrderController.php: the order list with pages and sums, prices from Problem 8.1, and the workflow of Problem 8.2.*

```
    // Problem 8.2: the workflow decides; the controller stamps the time.
    public function advance(Order $order)
    {
        $balance = $order->total - (int) $order->payments()->sum('amount');
        try {
            $next = OrderWorkflow::advance($order->status, $balance);
        } catch (DomainException $e) {
            return back()->with('error', $e->getMessage());
        }
        $order->status = $next;
        if ($next === OrderStatus::Finished) {
            $order->finished_at = now();
        } else {
            $order->departed_at = now();
        }
        $order->save();

        return back()->with('message',
            "Order #{$order->id} is now {$next->label()}.");
    }
}
```

In the view, the balance becomes `$o->total - (int) $o->payments_sum_amount`; the cast matters, because the sum of no payments is `null`, not 0. `paginate` runs a `COUNT` and the page query, and the two eager loads add one query each: 4 in all, whatever the number of orders.

8.5.6 A tempting wrong solution

In EF Core the tempting mistake is to move the work into memory: `var all = await db.Orders.Include(...).ToListAsync();` followed by `all.Where(...).Skip(...).Take(20)`. It returns the right page, and it reads every order and every payment of the shop on every click, which is exactly the slowness the owner complained about. Everything before `ToListAsync()` runs in the database; everything after runs in C#. Filtering, sorting, paging and summing belong before it.

In Laravel the tempting mistake is to drop `with(...)` because the page "works without it". It does work, with 61 statements per page instead of 4; Chapter 7's `preventLazyLoading` turns that mistake into an exception during development.

8.6 Problem 8.4: A Customer Search That Finds Sari

8.6.1 The problem

The cashier types `sari` and the Chapter 5 application finds nobody, while the Laravel application finds Sari Wulandari. With 250 customers the list also needs pages. The owner asks for the same search in both: ignore upper and lower case, match name or phone, 15 customers per page, and keep the search when moving between pages.

8.6.2 Understanding it by hand

`Contains` in EF Core's SQLite provider becomes `instr(Name, 'sari') > 0`, and `instr` compares bytes, so `S` and `s` differ. SQLite's `LIKE` compares ASCII letters without regard to case, so `Name LIKE '%sari%'` finds *Sari Wulandari*. For the pages: 250 customers at 15 per page are $\lceil 250/15 \rceil = 17$

pages, the last with 10 customers. Page 2 of a search for `an` must still search for `an`, so its link must be `?search=an&page=2`, not `?page=2`.

8.6.3 Designing it on paper

In EF Core, `EF.Functions.Like(c.Name, pattern)` asks explicitly for SQL `LIKE` [6]. In Laravel, the Chapter 7 search already uses `like`. Both get paging: `PagedList` in C#, `paginate(15)` in Laravel. Both get a pager that copies the current query string and changes only `page`.

8.6.4 The ASP.NET Core solution

Listing 8.13 is the new `CustomersController`. `Index` builds the pattern `%sari%` once and passes it as a parameter; it orders by name and then by `Id`, because two customers may share a name and pages need a complete order.

Listing 8.13 `Controllers/CustomersController.cs`: a case-insensitive search with pages.

```
using LaundryMvc.Data;
using LaundryMvc.Domain;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Controllers;

public class CustomersController(LaundryDbContext db) : Controller
{
    // Problem 8.4. GET /Customers?search=sari&page=2
    public async Task<IActionResult> Index(string? search, int page = 1)
    {
        var query = db.Customers.AsNoTracking();
        if (!string.IsNullOrWhiteSpace(search))
        {
            // LIKE ignores upper/lower case for ASCII letters in SQLite and
            // in MySQL's usual collations; Contains() would not in SQLite.
            var pattern = $"%{search.Trim()}%";
            query = query.Where(c => EF.Functions.Like(c.Name, pattern)
                                  || EF.Functions.Like(c.Phone, pattern));
        }
        ViewData["Search"] = search;
        return View(await PagedList<Customer>.CreateAsync(
            query.OrderBy(c => c.Name).ThenBy(c => c.Id), page, pageSize: 15));
    }

    // GET /Customers/Details/3
    public async Task<IActionResult> Details(int id)
    {
        var customer = await db.Customers
            .Include(c => c.Orders).ThenInclude(o => o.Service)
            .AsNoTracking()
            .FirstOrDefaultAsync(c => c.Id == id);
        return customer is null ? NotFound() : View(customer);
    }

    public IActionResult Create() => View(new Customer());
```

Listing 8.13 (Continued) *Controllers/CustomersController.cs: a case-insensitive search with pages.*

```
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(
        [Bind("Name,Phone,Address")] Customer customer)
    {
        customer.Phone = IndonesianMobileAttribute.Normalize(customer.Phone);
        await CheckPhoneIsFreeAsync(customer.Phone, customer.Id);
        if (!ModelState.IsValid) return View(customer);

        db.Customers.Add(customer);
        await db.SaveChangesAsync();          // INSERT, then customer.Id is set
        TempData["Message"] = $"Customer {customer.Name} added.";
        return RedirectToAction(nameof(Details), new { id = customer.Id });
    }

    public async Task<IActionResult> Edit(int id)
    {
        var customer = await db.Customers.FindAsync(id);
        return customer is null ? NotFound() : View(customer);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Edit(int id,
        [Bind("Id,Name,Phone,Address")] Customer input)
    {
        if (id != input.Id) return BadRequest();
        var customer = await db.Customers.FindAsync(id);
        if (customer is null) return NotFound();

        input.Phone = IndonesianMobileAttribute.Normalize(input.Phone);
        await CheckPhoneIsFreeAsync(input.Phone, id);
        if (!ModelState.IsValid) return View(input);

        customer.Name = input.Name;           // the tracked entity is changed,
        customer.Phone = input.Phone;         // so SaveChanges sends an UPDATE
        customer.Address = input.Address;     // for exactly these columns
        await db.SaveChangesAsync();
        TempData["Message"] = $"Customer {customer.Name} saved.";
        return RedirectToAction(nameof(Details), new { id });
    }
```

Listing 8.13 (Continued) *Controllers/CustomersController.cs: a case-insensitive search with pages.*

```
    [Authorize(Roles = AppRoles.Owner)]
    public async Task<IActionResult> Delete(int id)
    {
        var customer = await db.Customers.Include(c => c.Orders)
            .AsNoTracking().FirstOrDefaultAsync(c => c.Id == id);
        return customer is null ? NotFound() : View(customer);
    }

    [HttpPost, ActionName("Delete")]
    [Authorize(Roles = AppRoles.Owner)]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> DeleteConfirmed(int id)
    {
        var customer = await db.Customers.FindAsync(id);
        if (customer is null) return NotFound();
        if (await db.Orders.AnyAsync(o => o.CustomerId == id))
        {
            TempData["Error"] = "A customer with orders cannot be deleted.";
            return RedirectToAction(nameof(Details), new { id });
        }
        db.Customers.Remove(customer);
        await db.SaveChangesAsync();          // DELETE
        TempData["Message"] = $"Customer {customer.Name} deleted.";
        return RedirectToAction(nameof(Index));
    }

    async Task CheckPhoneIsFreeAsync(string phone, int ownId)
    {
        if (await db.Customers.AnyAsync(c => c.Phone == phone && c.Id != ownId))
        {
            ModelState.AddModelError(nameof(Customer.Phone),
                "Another customer already uses this phone number.");
        }
    }
}
```

The pager (**Listing 8.14**) is a partial view that works for any `IPaged` model. Its local function `PageLink` copies every value of the current query string into a dictionary and replaces `page`, and `asp-all-route-data` turns the dictionary into the link. The search, and on the order list the status filter, survive every click.

Listing 8.14 `Views/Shared/_Pager.cshtml`: one pager for every list, keeping the query string.

```
@model IPaged
@functions {
    // The current query string (search, status...) with another page number.
    Dictionary<string, string> PageLink(int page)
    {
        var values = Context.Request.Query
            .ToDictionary(q => q.Key, q => q.Value.ToString());
        values["page"] = page.ToString();
        return values;
    }
}
<nav class="pager">
    @if (Model.Page > 1)
    {
        <a asp-all-route-data="PageLink(Model.Page - 1)">&laquo; Previous</a>
    }
    <span>Page @Model.Page of @Model.PageCount · @Model.TotalCount in all</span>
    @if (Model.Page < Model.PageCount)
    {
        <a asp-all-route-data="PageLink(Model.Page + 1)">Next &raquo;</a>
    }
</nav>
```

8.6.5 The Laravel solution

The Laravel `index` (**Listing 8.15**) changes in two places: the search is wrapped in `where(fn ($q) => ...)`, and `get()` becomes `paginate(15)->withQueryString()`.

Listing 8.15 `app/Http/Controllers/CustomerController.php`: the same search with pages in Laravel.

```php
<?php

namespace App\Http\Controllers;

use App\Models\Customer;
use Illuminate\Http\Request;
use Illuminate\Validation\Rule;

class CustomerController extends Controller
{
    // Problem 8.4. GET /customers?search=sari&page=2
    public function index(Request $request)
    {
        $search = trim((string) $request->query('search', ''));
        $customers = Customer::query()
            ->when($search !== '', fn ($query) => $query->where(fn ($q) => $q
                ->where('name', 'like', "%{$search}%")
                ->orWhere('phone', 'like', "%{$search}%")))
            ->orderBy('name')->orderBy('id')
            ->paginate(15)
            ->withQueryString();          // keep ?search= in the page links

        return view('customers.index', compact('customers', 'search'));
    }

    // GET /customers/3   (route model binding finds Customer 3, or 404)
    public function show(Customer $customer)
    {
        $customer->load(['orders' => fn ($q) => $q->latest('received_at'),
            'orders.service']);

        return view('customers.show', compact('customer'));
    }

    public function create()
    {
        return view('customers.create', ['customer' => new Customer()]);
    }

    public function store(Request $request)
    {
        $customer = Customer::create($request->validate($this->rules()));
```

Listing 8.15 (Continued) *app/Http/Controllers/CustomerController.php: the same search with pages in Laravel.*

```
        return redirect()->route('customers.show', $customer)
            ->with('message', "Customer {$customer->name} added.");
    }

    public function edit(Customer $customer)
    {
        return view('customers.edit', compact('customer'));
    }

    public function update(Request $request, Customer $customer)
    {
        $customer->update($request->validate($this->rules($customer)));

        return redirect()->route('customers.show', $customer)
            ->with('message', "Customer {$customer->name} saved.");
    }

    public function destroy(Customer $customer)
    {
        if ($customer->orders()->exists()) {
            return back()->with('error',
                'A customer with orders cannot be deleted.');
        }
        $customer->delete();

        return redirect()->route('customers.index')
            ->with('message', "Customer {$customer->name} deleted.");
    }

    private function rules(?Customer $customer = null): array
    {
        return [
            'name' => ['required', 'string', 'min:2', 'max:80'],
            'phone' => ['required', 'string', 'max:20',
                Rule::unique('customers', 'phone')->ignore($customer)],
            'address' => ['nullable', 'string', 'max:200'],
        ];
    }
}
```

`withQueryString()` makes every page link carry the current query string, the job `PageLink` does in C#. The pager partial (**Listing 8.16**) uses the paginator's own methods, `previousPageUrl()`, `nextPageUrl()`, `currentPage()`, `lastPage()` and `total()`, and prints the same line as the C# pager [7].

Listing 8.16 `resources/views/_pager.blade.php`: the pager in Blade.

```
{{-- One pager for every list: $page is a LengthAwarePaginator. --}}
<nav class="pager">
    @if ($page->previousPageUrl())
        <a href="{{ $page->previousPageUrl() }}">&laquo; Previous</a>
    @endif
    <span>Page {{ $page->currentPage() }} of {{ $page->lastPage() }}
        · {{ $page->total() }} in all</span>
    @if ($page->nextPageUrl())
        <a href="{{ $page->nextPageUrl() }}">Next &raquo;</a>
    @endif
</nav>
```

8.6.6 A tempting wrong solution

Why the extra `where(fn ($q) => ...)` around the two conditions? Without it the SQL is `where name like ? or phone like ?`, which is correct as long as nothing else is added. The day someone adds "only customers of this outlet", the query becomes `where name like ? or phone like ? and outlet_id = ?`, and because `and` binds more tightly than `or`, every customer whose *name* matches is shown, whatever their outlet. The closure produces parentheses: `where (name like ? or phone like ?) and outlet_id = ?`.

In EF Core, the tempting mistake is paging without `OrderBy`. SQLite then returns rows in whatever order it finds convenient, which can differ between the query for page 1 and the query for page 2, so a customer can appear on both pages or on neither. EF Core warns about it in the log: *the query uses a row limiting operator ('Skip'/'Take') without an 'OrderBy' operator*. `PagedList` documents the same requirement in its comment.

8.7 Problem 8.5: No Overpayment

8.7.1 The problem

A cashier typed Rp 60.000 instead of Rp 16.000 for Budi Santoso's balance, and the order page showed a balance of −Rp 44.000. The owner asks: *a payment may not be more than what is still owed, and the message must say how much that is.*

8.7.2 Understanding it by hand

Budi's order: total Rp 36.000, paid Rp 20.000, balance Rp 16.000 by the balance formula of Section 5.3. A payment of Rp 16.000 is accepted and leaves 0. A payment of Rp 20.000 is refused with "Only Rp 16.000 is still owed." A payment of Rp 500 is refused by the minimum of Rp 1.000 from Chapter 6. The check must use the balance at the moment of the payment, read from the database, never a number sent by the form.

8.7.3 Designing it on paper

This is validation that needs the database, so in ASP.NET Core it goes into the controller, after the attribute checks of `PaymentForm`; in Laravel it becomes a validation rule whose limit is computed before `validate` is called.

8.7.4 The ASP.NET Core solution

`AddPayment` in **Listing 8.10** asks the database for the balance in one statement: `Select(o => (int?)(o.Total - o.Payments.Sum(p => p.Amount)))`. The cast to `int?` lets `FirstOrDefaultAsync` return `null` for an order that does not exist, which becomes a 404. Only if the attributes passed (`ModelState.IsValid`) is the amount compared with the balance, so that the cashier sees one message at a time, the most basic one first.

8.7.5 The Laravel solution

Listing 8.17 computes the balance with `sum('amount')` and puts it into the rule itself: `"max:{$balance}"`, where `max` compares numbers because the field also has the `integer` rule [8]. The custom message for `amount.max` uses `Money::rp`, so both applications print the same sentence.

Listing 8.17 `app/Http/Controllers/PaymentController.php`: a payment limited to the balance.

```php
<?php

namespace App\Http\Controllers;

use App\Enums\PaymentMethod;
use App\Models\Order;
use App\Support\Money;
use Illuminate\Http\Request;
use Illuminate\Validation\Rule;

class PaymentController extends Controller
{
    // Problem 8.5: a payment may not be more than what is still owed.
    public function store(Request $request, Order $order)
    {
        $balance = $order->total - (int) $order->payments()->sum('amount');
        $data = $request->validate([
            'amount' => ['required', 'integer', 'min:1000', "max:{$balance}"],
            'method' => ['required', Rule::enum(PaymentMethod::class)],
        ], [
            'amount.max' => 'Only ' . Money::rp($balance) . ' is still owed.',
        ]);
        $order->payments()->create([...$data, 'paid_at' => now()]);

        return back()->with('message',
            'Payment of ' . Money::rp((int) $data['amount']) . ' recorded.');
    }
}
```

8.7.6 A tempting wrong solution

Comparing the amount with the order's *total* instead of its balance accepts Rp 36.000 on top of the Rp 20.000 already paid. Comparing with a balance sent as a hidden field of the form lets anyone who edits the field pay anything. And a subtler problem remains in both solutions: two cashiers who record the last payment of the same order at the same second could both pass the check. SQLite serializes writes, and a laundry counter rarely has two cashiers on one order, so the solutions accept the risk; a busy system would run the check and the insert in one transaction with a lock, or recheck the balance after saving.

8.8 Problem 8.6: The Daily Revenue Report

8.8.1 The problem

Every evening the owner compares the cash in the drawer, the QRIS statement and the bank's transfer list with what the system recorded. She asks for a report: *money received per day, split by payment method, for a chosen week*.

8.8.2 Understanding it by hand

In the sample data there are two payments: Rp 36.000 by QRIS yesterday at 08:00 (Sari Wulandari) and Rp 20.000 in cash today at 07:30 (Budi Santoso). For the default range, the last seven days up to today, the report has two lines, yesterday with QRIS Rp 36.000 and today with cash Rp 20.000, one payment each, and a total line of Rp 56.000 from 2 payments. The day of a payment is its date in local time; a payment at 00:30 belongs to the new day.

8.8.3 Designing it on paper

Let the database group and sum: one row per day and method. Then turn those rows, a few dozen at most, into one line per day with a column per method, in memory. The date range is *from the first day at midnight, up to but not including the day after the last day*, which includes every moment of the last day without having to write 23:59:59.

8.8.4 The ASP.NET Core solution

A report line is a small record (**Listing 8.18**).

Listing 8.18 `Models/DailyRevenueRow.cs`: one line of the report.

```
namespace LaundryMvc.Models;

// Problem 8.6: one line of the daily revenue report.
public record DailyRevenueRow(DateTime Day, int Cash, int Qris, int Transfer,
                              int Payments)
{
    public int Total => Cash + Qris + Transfer;
}
```

`ReportsController` (**Listing 8.19**) is for owners only. It receives the range as `DateOnly?` values, which model binding reads from `?from=2026-09-01&to=2026-09-07`, defaults to the last seven days and swaps the two if they arrive in the wrong order. `GroupBy(p => new { Day = p.PaidAt.Date, p.Method })` is translated into a `GROUP BY` on the date part of the timestamp and the method column; `Count()` and `Sum()` become `COUNT(*)` and `SUM("Amount")` [9]. The second, in-memory `GroupBy` builds the report lines.

Listing 8.19 `Controllers/ReportsController.cs`: money received per day and method, grouped in the database.

```
using LaundryMvc.Data;
using LaundryMvc.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace LaundryMvc.Controllers;

// Problem 8.6: money received per day and per payment method.
[Authorize(Roles = AppRoles.Owner)]
public class ReportsController(LaundryDbContext db) : Controller
{
    // GET /Reports/Daily?from=2026-09-01&to=2026-09-07
    public async Task<IActionResult> Daily(DateOnly? from, DateOnly? to)
    {
        var last = to ?? DateOnly.FromDateTime(DateTime.Today);
        var first = from ?? last.AddDays(-6);
        if (first > last) (first, last) = (last, first);
        var start = first.ToDateTime(TimeOnly.MinValue);
        var end = last.AddDays(1).ToDateTime(TimeOnly.MinValue);  // exclusive

        // One row per day and method, computed by the database.
        var groups = await db.Payments
            .Where(p => p.PaidAt >= start && p.PaidAt < end)
            .GroupBy(p => new { Day = p.PaidAt.Date, p.Method })
            .Select(g => new
            {
                g.Key.Day, g.Key.Method,
                Count = g.Count(), Amount = g.Sum(p => p.Amount)
            })
            .ToListAsync();

        // Turned into one line per day, with a column per method.
        var rows = groups
            .GroupBy(g => g.Day)
            .OrderBy(d => d.Key)
            .Select(d => new DailyRevenueRow(d.Key,
                Cash: d.Where(g => g.Method == PaymentMethod.Cash)
                       .Sum(g => g.Amount),
                Qris: d.Where(g => g.Method == PaymentMethod.Qris)
                       .Sum(g => g.Amount),
                Transfer: d.Where(g => g.Method == PaymentMethod.Transfer)
                           .Sum(g => g.Amount),
                Payments: d.Sum(g => g.Count)))
            .ToList();
```

Listing 8.19 (Continued) *Controllers/ReportsController.cs: money received per day and method, grouped in the database.*

```
        ViewData["From"] = first.ToString("yyyy-MM-dd");
        ViewData["To"] = last.ToString("yyyy-MM-dd");
        return View(rows);
    }
}
```

The view (**Listing 8.20**) prints the lines and a total line.

Listing 8.20 `Views/Reports/Daily.cshtml`: the daily revenue report.

```
@model List<DailyRevenueRow>
@{
    ViewData["Title"] = "Daily revenue";
}
<h1>Daily revenue</h1>
<form method="get" class="search">
    <input type="date" name="from" value="@ViewData["From"]" />
    <input type="date" name="to" value="@ViewData["To"]" />
    <button type="submit">Show</button>
</form>
<table class="list">
    <thead>
        <tr><th>Day</th><th class="num">Cash</th><th class="num">QRIS</th>
            <th class="num">Transfer</th><th class="num">Total</th>
            <th class="num">Payments</th></tr>
    </thead>
    <tbody>
    @foreach (var r in Model)
    {
        <tr>
            <td>@r.Day.ToString("ddd dd/MM/yyyy")</td>
            <td class="num">@Money.Rp(r.Cash)</td>
            <td class="num">@Money.Rp(r.Qris)</td>
            <td class="num">@Money.Rp(r.Transfer)</td>
            <td class="num"><strong>@Money.Rp(r.Total)</strong></td>
            <td class="num">@r.Payments</td>
        </tr>
    }
    <tr class="total"><th>All days</th>
        <td class="num">@Money.Rp(Model.Sum(r => r.Cash))</td>
        <td class="num">@Money.Rp(Model.Sum(r => r.Qris))</td>
        <td class="num">@Money.Rp(Model.Sum(r => r.Transfer))</td>
        <td class="num">@Money.Rp(Model.Sum(r => r.Total))</td>
        <td class="num">@Model.Sum(r => r.Payments)</td></tr>
    </tbody>
</table>
```

8.8.5 The Laravel solution

Listing 8.21 writes the grouping with `selectRaw('date(paid_at) as day, method, count(*) as payments, sum(amount) as amount')` and `groupByRaw('date(paid_at), method')` [10]. `date(...)` exists in SQLite and in MySQL with the same meaning, so the report runs on both. `toBase()` returns plain rows instead of `Payment` models, because the rows are not payments and must not go through the model's casts. The collection methods `groupBy('day')`, `sortKeys()` and `where('method', 'cash')->sum('amount')` then build the lines, as LINQ does in C#.

Listing 8.21 `app/Http/Controllers/ReportController.php`: the same report with the query builder and collections.

```php
<?php

namespace App\Http\Controllers;

use App\Models\Payment;
use Illuminate\Http\Request;
use Illuminate\Support\Carbon;

// Problem 8.6: money received per day and per payment method.
class ReportController extends Controller
{
    // GET /reports/daily?from=2026-09-01&to=2026-09-07
    public function daily(Request $request)
    {
        $request->validate(['from' => ['nullable', 'date'],
            'to' => ['nullable', 'date']]);
        $last = ($request->date('to') ?? today())->startOfDay();
        $first = ($request->date('from') ?? $last->copy()->subDays(6))
            ->startOfDay();
        if ($first->gt($last)) {
            [$first, $last] = [$last, $first];
        }

        // One row per day and method, computed by the database.
        $groups = Payment::query()
            ->where('paid_at', '>=', $first)
            ->where('paid_at', '<', $last->copy()->addDay())
            ->selectRaw('date(paid_at) as day, method, count(*) as payments,'
                . ' sum(amount) as amount')
            ->groupByRaw('date(paid_at), method')
            ->toBase()                    // plain rows, not Payment models
            ->get();

        // Turned into one line per day, with a column per method.
        $rows = $groups->groupBy('day')->sortKeys()
            ->map(function ($day, $date) {
                $cash = (int) $day->where('method', 'cash')->sum('amount');
                $qris = (int) $day->where('method', 'qris')->sum('amount');
                $transfer = (int) $day->where('method', 'transfer')
                    ->sum('amount');
```

Listing 8.21 (Continued) *app/Http/Controllers/ReportController.php: the same report with the query builder and collections.*

```php
                return (object) ['day' => Carbon::parse($date),
                    'cash' => $cash, 'qris' => $qris, 'transfer' => $transfer,
                    'total' => $cash + $qris + $transfer,
                    'payments' => (int) $day->sum('payments')];
            })->values();

        return view('reports.daily', ['rows' => $rows,
            'from' => $first->toDateString(), 'to' => $last->toDateString()]);
    }
}
```

Listing 8.22 `resources/views/reports/daily.blade.php`: the report in Blade.

```
@extends('layouts.app')
@use('App\Support\Money')

@section('title', 'Daily revenue')

@section('content')
<h1>Daily revenue</h1>
<form method="get" class="search">
    <input type="date" name="from" value="{{ $from }}">
    <input type="date" name="to" value="{{ $to }}">
    <button type="submit">Show</button>
</form>
<table class="list">
    <thead>
        <tr><th>Day</th><th class="num">Cash</th><th class="num">QRIS</th>
            <th class="num">Transfer</th><th class="num">Total</th>
            <th class="num">Payments</th></tr>
    </thead>
    <tbody>
    @foreach ($rows as $r)
        <tr>
            <td>{{ $r->day->format('D d/m/Y') }}</td>
            <td class="num">{{ Money::rp($r->cash) }}</td>
            <td class="num">{{ Money::rp($r->qris) }}</td>
            <td class="num">{{ Money::rp($r->transfer) }}</td>
            <td class="num"><strong>{{ Money::rp($r->total) }}</strong></td>
            <td class="num">{{ $r->payments }}</td>
        </tr>
    @endforeach
    <tr class="total"><th>All days</th>
        <td class="num">{{ Money::rp($rows->sum('cash')) }}</td>
        <td class="num">{{ Money::rp($rows->sum('qris')) }}</td>
        <td class="num">{{ Money::rp($rows->sum('transfer')) }}</td>
        <td class="num">{{ Money::rp($rows->sum('total')) }}</td>
        <td class="num">{{ $rows->sum('payments') }}</td></tr>
    </tbody>
</table>
@endsection
```

The Blade view (**Listing 8.22**) is line for line the Razor view of **Listing 8.20**. Until Chapter 10 adds sign-in to the Laravel application, its report is as open as its other staff pages, which is one more reason to keep that version on the local machine for now.

8.8.6 A tempting wrong solution

Grouping by `PaidAt` instead of `PaidAt.Date` produces one line per payment, because no two payments share the same second. Storing amounts as `decimal`, as the service price is, would make `Sum` fail on SQLite at run time with EF Core, which cannot translate a decimal sum for that provider (Section 5.3); the report works because Chapter 5 stored amounts as whole rupiah. And in Laravel, forgetting to set the application's time zone to `Asia/Jakarta` shifts every day boundary to 07:00 local time, so the morning's payments before seven appear under the previous day.

8.9 Problem 8.7: A Rack for Every Finished Bag

8.9.1 The problem

Finished bags wait on numbered racks, and customers wait while the washer searches for their bag. The owner asks for a *rack* field on each order, such as `B-03`: a capital letter, a dash and two digits, optional, entered at the counter and shown on the order page.

8.9.2 Understanding it by hand

`B-03` and `A-10` are valid; `b-03`, `B3` and `B-003` are not. Existing orders have no rack, so the new column must allow empty values; otherwise the migration would have to invent a rack for 750 old orders.

8.9.3 Designing it on paper

A schema change is a migration in both frameworks, plus a model property, a form field with a validation rule, and a line on the order page. The order of work matters: model and migration first, then the form, then the page.

8.9.4 The ASP.NET Core solution

`Order` gets `public string? Rack { get; set; }` and `LaundryDbContext` gets `e.Property(o => o.Rack).HasMaxLength(8)`. `OrderForm` (**Listing 8.23**) gets the property with a regular expression, and its `Validate` method the two-decimal rule of Problem 8.1.

Listing 8.23 `Models/OrderForm.cs` with the rack and the two-decimal rule.

```
using System.ComponentModel.DataAnnotations;

namespace LaundryMvc.Models;

// What the counter form may send: nothing about Total, Status or dates.
public class OrderForm : IValidatableObject
{
    [Range(1, int.MaxValue, ErrorMessage = "Choose a customer.")]
    [Display(Name = "Customer")]
    public int CustomerId { get; set; }

    [Required(ErrorMessage = "Choose a service.")]
    [Display(Name = "Service")]
    public string? ServiceId { get; set; }

    [Range(typeof(decimal), "0.1", "100",
        ParseLimitsInInvariantCulture = true,
        ErrorMessage = "The quantity must be between 0.1 and 100.")]
    [Display(Name = "Quantity (kg, items or pairs)")]
    public decimal Quantity { get; set; } = 1;

    [Display(Name = "Pickup and delivery")]
    public bool Pickup { get; set; }

    [StringLength(200)]
    [Display(Name = "Delivery address")]
    public string? DeliveryAddress { get; set; }

    [StringLength(200)]
    public string? Note { get; set; }

    [RegularExpression("^[A-Z]-[0-9]{2}$",
        ErrorMessage = "A rack looks like B-03: a letter, a dash, two digits.")]
    public string? Rack { get; set; }

    // A rule about two fields together: delivery needs an address.
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (decimal.Round(Quantity, 2) != Quantity)
        {
            yield return new ValidationResult(
                "Use at most two decimals, for example 4.25.",
                [nameof(Quantity)]);
        }
        if (Pickup && string.IsNullOrWhiteSpace(DeliveryAddress))
        {
```

Listing 8.23 (Continued) *Models/OrderForm.cs with the rack and the two-decimal rule.*

```
            yield return new ValidationResult(
                "Pickup and delivery needs a delivery address.",
                [nameof(DeliveryAddress)]);
        }
    }
}
```

Then `dotnet ef migrations add AddRackToOrders` compares the model with the snapshot and writes a migration whose `Up` calls `migrationBuilder.AddColumn<string>(name: "Rack", table: "Orders", type: "TEXT", maxLength: 8, nullable: true)` and whose `Down` calls `DropColumn` [11]. `dotnet ef database update`, or the `MigrateAsync()` at start-up of Chapter 6, applies it. `Create` copies `form.Rack` into the order, and the details page prints it.

8.9.5 The Laravel solution

Listing 8.24 is the migration, written by hand; `Schema::table` changes an existing table where `Schema::create` made a new one [12]. `'rack'` is added to the model's `#[Fillable]` list, the store rules get `'rack' => ['nullable', 'regex:/^[A-Z]-[0-9]{2}$/']` with the same message as C#, and the form and the order page get one field and one line.

Listing 8.24 `2026_09_20_000001_add_rack_to_orders_table.php`: the rack column, added and removed.

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

// Problem 8.7: where the finished bag waits, for example rack "B-03".
return new class extends Migration
{
    public function up(): void
    {
        Schema::table('orders', function (Blueprint $table) {
            $table->string('rack', 8)->nullable()->after('note');
        });
    }

    public function down(): void
    {
        Schema::table('orders', function (Blueprint $table) {
            $table->dropColumn('rack');
        });
    }
};
```

8.9.6 A tempting wrong solution

The tempting shortcut is to edit the migration that created the orders table, add the column there, and run the migrations again. On your own laptop, `migrate:fresh` or deleting `laundry.db` makes it look fine. On the shop's server, where the original migration has already run, nothing happens: Laravel and EF Core both skip migrations recorded as applied, so the column never appears and the first order with a rack fails. A migration that has left your computer is history; changes are always new migrations.

8.10 The Problems Side by Side

Table 8.1 collects the seven solutions.

Table 8.1 The seven problems, their solutions in both frameworks, and what each teaches.

Problem	ASP.NET Core	Laravel	Lesson
8.1 price	`PriceCalculator`, `decimal`	`PriceCalculator`, integer hundredths	same rule, proven equal on fixed cases
8.2 workflow	`OrderWorkflow.Advance` + `Include(Payments)`	`OrderWorkflow::advance` + `sum('amount')`	plain classes for rules; load what you compute from
8.3 order list	`Select` into `OrderRow`, `PagedList`	`select`, `with` on columns, `withSum`, `paginate`	page and project in the database
8.4 search	`EF.Functions.Like`, `OrderBy` + `ThenBy`	grouped `where`, `withQueryString`	know what your provider's string functions do
8.5 overpayment	balance in one query, `ModelState` error	`"max:{$balance}"` rule	limits from the database, never from the form
8.6 report	`GroupBy` date and method, then pivot	`selectRaw` + `groupByRaw`, collections	aggregate in SQL, shape in memory
8.7 rack	model + `migrations add`	hand-written `Schema::table` migration	schema changes are always new migrations

Read down the last column and the chapter's theme is plain. The frameworks differ in vocabulary, in how much they generate and in how much they check at compile time; the decisions that make an application correct are the same in both.

8.11 Profitina in Practice

Two of this chapter's habits are everyday practice at Profitina. The first is proving that two implementations agree. Profitina's historical dashboard shows a weekly score per brand computed from a growing table, and its figures are checked against an independent computation over the same rows before a change is released, the way `CoreCheck` and `core_check.php` must print identical lines. The second is the release gate: the whole automated test suite, well over a thousand tests in Python plus JavaScript guards for the interface, must pass before a release, and a staged release is only reported successful after the deployed version answers its health endpoint. The seven problems of this chapter would each become tests in such a suite: the nine price cases, the four workflow cases, the query count of the order list, the search for `sari`, the refused overpayment, the two report lines, and a migration that runs on a copy of the production database.

8.12 Common Mistakes, and How to Avoid Them

- **Porting a rule by reading the code instead of the requirement.** The Chapter 7 price code was a faithful port with floats; it matched only because of `round`. Port the rule, then prove equality on fixed cases.
- **Computing from relationships that were not loaded.** In EF Core an unloaded collection is empty; in Laravel it is loaded lazily, one query per model. Either way, know what is loaded before computing from it.
- **Moving work to memory by accident.** In EF Core, a `ToList` before `Where`, `Skip` or `Sum` reads the whole table. Keep the query an `IQueryable` until the last step.
- **Paging without a complete order.** Order by a unique column last, such as `Id`, or rows can repeat or vanish between pages.

- **Ungrouped `orWhere`.** Wrap alternatives in a closure so that they become one parenthesized condition.
- **Trusting numbers from the form.** Balances, prices and totals are read from the database on the server, never taken from hidden fields.
- **Grouping by a timestamp instead of its date,** and forgetting the time zone of the application.
- **Editing a migration that has already run.** Add a new migration instead.

8.13 Chapter Summary

- A framework problem is solved in six steps: state it in the shop's words, work an example by hand, decide the layer, write it in one framework, port and compare, and try to break it.
- Rules that need neither the web nor the database belong in plain classes that both frameworks can call and that a console program can check; the price rule and the order workflow were proven identical in C# and PHP by two programs that print the same lines.
- Money and quantities are safest as integers: whole rupiah in the database and integer hundredths in PHP's arithmetic; floats can lose a rupiah on ordinary quantities such as 4.02 kg.
- Lists are paged and projected in the database: two statements in EF Core, four in Eloquent, instead of 61 for a page loaded lazily.
- String functions differ between providers: `Contains` is case-sensitive on SQLite, `LIKE` is not.
- Limits that depend on stored data, such as the balance of an order, are computed on the server at the moment of the request.
- Reports group and sum in SQL and are shaped in memory; days are ranges from midnight to midnight in the shop's time zone.
- Schema changes are new migrations, generated by `dotnet ef` or written for Artisan, never edits to old ones.

8.14 Exercises

1. By hand, compute with Equation (8.1) and Equation (8.2) the total for *Curtain* (Rp 15.000 per kg) at 2.5 kg with pickup, and for *Helmet cleaning* (Rp 30.000 per item) at 3.2 items without pickup. Add both cases to `CoreCheck` and `core_check.php` and check that the outputs still agree.
2. The owner introduces a 10% discount for orders of 10 kg or more of per-kg services, rounded down to whole rupiah. Extend both `PriceCalculator` classes and state the discounted total for 12 kg of *Wash, dry & fold*.
3. The workflow gets a fourth status, *Cancelled*, allowed only from *Queued*. Change both workflow classes. What must also change in the database, in EF Core and in Laravel?
4. For a page of 20 orders, how many SQL statements does the Laravel order list send if `withSum` is replaced by `with('payments')` and the view sums `$o->payments`? And if the view uses `$o->payments()->sum('amount')` instead?
5. Extend the daily report with a parameter `method` that restricts it to one payment method, in both frameworks. Validate the parameter.
6. Write, in both frameworks, the query for a new report: the ten customers with the largest total balance still owed, with name, phone and amount. Which part runs in the database?

8.15 Solutions to the Exercises

1. Curtain: $h = 250$, per kg, so $b = \max(250,300) = 300$ and $S = \lfloor(15000 \cdot 300 + 50)/100\rfloor = 45000$; pickup adds Rp 10.000 because $45000 < 100000$: total Rp 55.000. Helmet: $h = 320$, per item, so $b = 100 \cdot \lceil 3.2 \rceil = 400$ and $S = \lfloor(30000 \cdot 400 + 50)/100\rfloor = 120000$; no pickup: total Rp 120.000. In `CoreCheck` add `("Curtain (per kg)", 15000, 2.5m, true)` and `("Helmet cleaning (per item)", 30000, 3.2m, false)`; in `core_check.php` add `['Curtain', 'kg', 15000, '2.5', true]` and `['Helmet cleaning', 'item', 30000, '3.2', false]`. Both print 55000 and 120000, and `diff` stays silent.

2. After the subtotal, subtract `subtotal / 10` for per-kg services when the billed quantity is at least 10 kg. In C#, `if (service.PerKg && billed >= 10) subtotal -= decimal.Floor(subtotal / 10);`. In PHP, `if ($perKg && $billed >= 1000) { $subtotal -= intdiv($subtotal, 10); }` (1000 hundredths = 10 kg). For 12 kg of *Wash, dry & fold*: $S = 12 \times 8000 = 96000$, discount $\lfloor 96000/10 \rfloor = 9600$, total Rp 86.400. Add the case to both check programs; a discount is exactly the kind of change that Section 8.3.6 warned could expose a float error, and with integer arithmetic it cannot.

3. Add `Cancelled` to the C# enum and `case Cancelled = 'cancelled';` to the PHP enum. In C#, `OrderWorkflow` gets a method `Cancel(OrderStatus status)` that returns `Cancelled` from `Queued` and throws otherwise; `Next` stays unchanged, so the normal workflow never reaches `Cancelled`. In PHP, a static `cancel(OrderStatus $status)` does the same, and `next()` in the enum returns `null` for `Cancelled`. The database needs no structural change, because both store the status as text, with room for 12 characters; but the C# `HasConversion<string>()` means old code reading a new value would fail, so deploy the code before cancelling the first order. A new `CancelledAt` timestamp, if wanted, is a migration in both.

4. With `with('payments')`: the `COUNT`, the orders, the customers, the services and the payments of the 20 orders, 5 statements, but every payment row is transferred. With `$o->payments()->sum('amount')` in the view: the relationship *method* always builds a new query, so 4 statements plus one `SUM` per order, 24 in all; `preventLazyLoading` does not catch this, because nothing is lazy-loaded. `withSum` does the same work inside the order query.

5. In C#, add `PaymentMethod? method` to `Daily` and `if (method is not null) query = query.Where(p => p.Method == method);` before the `GroupBy`; model binding refuses unknown names, leaving `method` null, so validate by checking `ModelState.IsValid` and showing a message. In Laravel, add `'method' => ['nullable', Rule::enum(PaymentMethod::class)]` to the validation and `->when($request->filled('method'), fn ($q) => $q->where('method', $request->input('method')))` to the query. The pager is unaffected, since the report has no pages.

6. The balance per customer is the sum of totals minus the sum of payments over that customer's orders. In C#: `db.Customers.Select(c => new { c.Name, c.Phone, Owed = c.Orders.Sum(o => o.Total) - c.Orders.SelectMany(o => o.Payments).Sum(p => p.Amount) }).Where(r => r.Owed > 0).OrderByDescending(r => r.Owed).Take(10)`; EF Core translates all of it, including both sums, the filter, the sort and the limit, into one SQL statement. In Laravel, `withSum('orders', 'total')` handles the first sum, but the payments are two relationships away, so both sums are written as subqueries: `$sums = Customer::select('name', 'phone')->selectSub(Order::selectRaw('coalesce(sum(total), 0)')->whereColumn('customer_id', 'customers.id'), 'billed')->selectSub(Payment::join('orders', 'orders.id', '=', 'payments.order_id')->selectRaw('coalesce(sum(amount), 0)')-`

`>whereColumn('orders.customer_id', 'customers.id'), 'paid');` and then `DB::query()->fromSub($sums, 'c')->selectRaw('name, phone, billed - paid as owed')->whereRaw('billed - paid > 0')->orderByDesc('owed')->limit(10)->get()`. Wrapping the first query with `fromSub` lets the outer query filter and sort on the computed columns. In both frameworks everything runs in the database and only ten rows come back.

References

[1] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[2] The PHP Group, "Floating point numbers," PHP Manual, accessed Sep. 2026. [Online]. Available: https://www.php.net/manual/en/language.types.float.php

[3] Microsoft, "Efficient querying," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/performance/efficient-querying

[4] Laravel, "Eloquent: relationships (eager loading and aggregating related models), Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/eloquent-relationships

[5] Microsoft, "Pagination," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/querying/pagination

[6] Microsoft, "Function mappings of the SQLite EF Core provider," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/providers/sqlite/functions

[7] Laravel, "Database: pagination, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/pagination

[8] Laravel, "Validation: available validation rules, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/validation#available-validation-rules

[9] Microsoft, "Complex query operators," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/querying/complex-query-operators

[10] Laravel, "Database: query builder, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/queries

[11] Microsoft, "Migrations overview," Microsoft Learn, accessed Sep. 2026. [Online]. Available: https://learn.microsoft.com/ef/core/managing-schemas/migrations/

[12] Laravel, "Database: migrations, Laravel 13.x," accessed Sep. 2026. [Online]. Available: https://laravel.com/docs/13.x/migrations

Chapter 9

Laravel II: Migrations, Seeders, Middleware and Form Requests

Chapter 7 built the Profitina Laundry counter in Laravel, and Chapter 8 hardened it with exact prices, a workflow and paging. The application works, but it is still a program for one person on one laptop. This chapter prepares it for a real shop with several outlets, several staff members and a database that already holds months of orders. By the end of the chapter you will be able to change a table that already contains data without losing a single row, and explain what the `migrations` table and its batches record; fill a database with a believable month of made-up orders using model factories and seeders, while keeping the real price list apart from the made-up data; explain what middleware is, write a "before" and an "after" middleware, register them and attach them to routes; move validation out of the controllers into form request classes, including rules that need the database; and read the whole path of one request through the router, the middleware, the form request and the controller.

9.1 Bu Wati's Third Outlet

Profitina Laundry began in 2019 as one shop on Jl. Kenari Raya in Rungkut, Surabaya. Bu Wati, the owner, opened a second outlet near the campus in Keputih in 2022 and a third one in Wiyung in January 2026. The price list is the same everywhere, and the customers move between the outlets: a student drops a bag of laundry in Keputih on Monday and asks on Wednesday, by phone, whether it is ready. Until now the application of Chapters 7 and 8 has not known where an order was taken in. Every order looks as if it came from nowhere.

Three other things have started to hurt at the same time. The first is the database itself. Dimas, the cashier who also looks after the computer, keeps a copy of the application on his laptop to try changes before they go to the shop's computer. When he added the `rack` column in Chapter 8, he ran the change on his laptop, then on the shop's computer, and a week later nobody could remember which computer had which columns. The second is test data. The counter page, the order list with twenty orders a page and the daily report of Chapter 8 all look empty on a fresh database, so every change has to be tried by typing orders by hand. The third is a real mistake that cost money: a finished order was paid twice, once in cash at the counter and once by QRIS on the phone, because the second cashier did not see the first payment, and nothing in the application stopped a payment on an order that the customer had already taken home.

Each of these has a standard answer in Laravel [1]. A change to the database is written as a *migration*, a small class that knows how to make the change and how to undo it, and Laravel records which migrations each database has already run. Made-up data comes from *factories*, which describe how to make one plausible customer or order, and *seeders*, which decide how many to make and in which order. Checks that must happen before a request reaches a controller, such as "nothing may change an order that has left the shop", are written once as *middleware*. And the growing lists of validation rules move out of the controllers into *form request* classes, where a rule that needs the database, such as "you cannot pay more than is still owed", has a proper home.

None of these ideas is new to you. Chapter 2 introduced migrations and dependency injection, Chapter 6 used ASP.NET Core's middleware pipeline and model validation, and Chapter 8 already wrote the price rule and the workflow as plain PHP classes. This chapter shows how Laravel expresses each idea, why it expresses it that way, and how the pieces fit together on the same Profitina Laundry application.

9.2 The Idea in Plain Words

9.2.1 Migrations: a logbook of changes to the database

Think of the shop's building. When Bu Wati adds a shelf in the back room, she does not draw a new plan of the whole building; she writes in a logbook "September 20: shelf B added along the east wall", and a builder in any of her outlets can repeat exactly that change. If the shelf was a mistake, the logbook entry also says how to take it down.

A migration is one such entry for the database. It has two methods: `up()` makes the change and `down()` undoes it. The file name starts with a date and time, so that the entries always run in the same order on every computer. Laravel keeps a table called `migrations` in each database, with one row per entry that has already run on that database, and the number of the *batch* in which it ran. When Dimas types `php artisan migrate`, Laravel compares the files in `database/migrations` with the rows in that table and runs only the files that are missing, all of them in one new batch [2]. When he types `php artisan migrate:rollback`, Laravel undoes the last batch, running the `down()` methods newest first.

The consequence is important: the logbook of files, not anybody's memory, decides what the database looks like. Dimas's laptop and the shop's computer can be brought to the same state by the same command, and a new outlet's computer can be built from nothing by running every entry in order. The practice has a name, *evolutionary database design*: the schema changes in small, recorded, reversible steps, together with the code that uses it [3].

9.2.2 Factories and seeders: a believable month, made up on purpose

To try a page that lists twenty orders a page, you need more than twenty orders. Typing them by hand is slow, and the orders you type are all alike. A *factory* describes how to make one plausible record: a customer has a name, a phone number that starts with 0812, and sometimes an address; an order belongs to a customer, an outlet and an active service, has a weight between one and nine kilograms, and is sometimes picked up and delivered. The factory uses a library called Faker to invent the names and numbers [4], and it can be asked for one record or four hundred [5].

A *seeder* is a small program that fills the database. It decides what to create and in which order: services and outlets first, because orders refer to them; customers next; orders and payments last. Laravel runs the seeders with `php artisan db:seed`, or together with a fresh set of tables with `php artisan migrate:fresh --seed` [6].

There are two kinds of seed data, and mixing them up is a classic mistake. The price list and the three outlets are *real* data: every database needs them, including the shop's own. The forty invented customers and their orders are *made-up* data: useful on a laptop, dangerous in the shop, where a cashier might phone an invented customer or count invented money in the daily report. This chapter keeps them in separate seeders and runs the made-up ones only on a developer's computer.

9.2.3 Middleware: the checkpoints every request passes

Before a request reaches a controller, it passes through a row of checkpoints, and on the way back the response passes through the same checkpoints in reverse order. Each checkpoint is a *middleware*: a class with one method, `handle`, that receives the request and a function called `$next` that stands for "everything behind me". A middleware may look at the request and then call `$next` to let it continue; it may refuse to call `$next` and answer the request itself, for example with a redirect or an error; and it may take the response that `$next` returned and change it before handing it back [7]. You met the same design in ASP.NET Core in Chapter 6, where it is also called the middleware pipeline.

The picture that helps most is an onion: the controller is the core, each middleware is a layer around it, and a request travels in through every layer and out again. Code that runs before `$next` is a "before" middleware; code that runs after it is an "after" middleware; many middleware classes do both. Laravel's own sessions, cookies, protection against forged form submissions, sign-in checks and rate limits are all middleware, so a new rule of the shop can join them as an equal.

9.2.4 Form requests: the counter checklist, in one place

When a customer hands over laundry, the cashier runs down a checklist: who is the customer, which service, how heavy, delivered or collected, and if delivered, where. In Chapters 7 and 8 that checklist was written inside the controller, as an array passed to `$request->validate()`. That works for three fields; it becomes hard to read at eight, and it cannot easily express a check that needs the order itself, such as "no more than the balance may be paid".

A *form request* is a class that holds the whole checklist for one kind of request: whether this user may make it at all (`authorize`), how to tidy the raw input first (`prepareForValidation`), the rules themselves (`rules`), friendlier messages (`messages` and `attributes`), and checks that run after the rules pass and may use the database (`after`) [8]. When a controller method asks for a form request as a parameter, Laravel builds it and validates it *before* the method runs. If anything fails, the controller never runs: a person is sent back to the form with the messages, and a program receives a JSON answer with status 422 and the list of errors.

Figure 9.1 puts the four ideas of this section on one request: a cashier records a payment on order 7. The request passes the `web` group of middleware, our timing middleware, the router, which turns the `7` in the URL into an `Order` object, our `order.open` middleware, and the form request, before the controller runs. Every red box is a place where the request may stop, and the controller only ever sees a request that has passed them all.

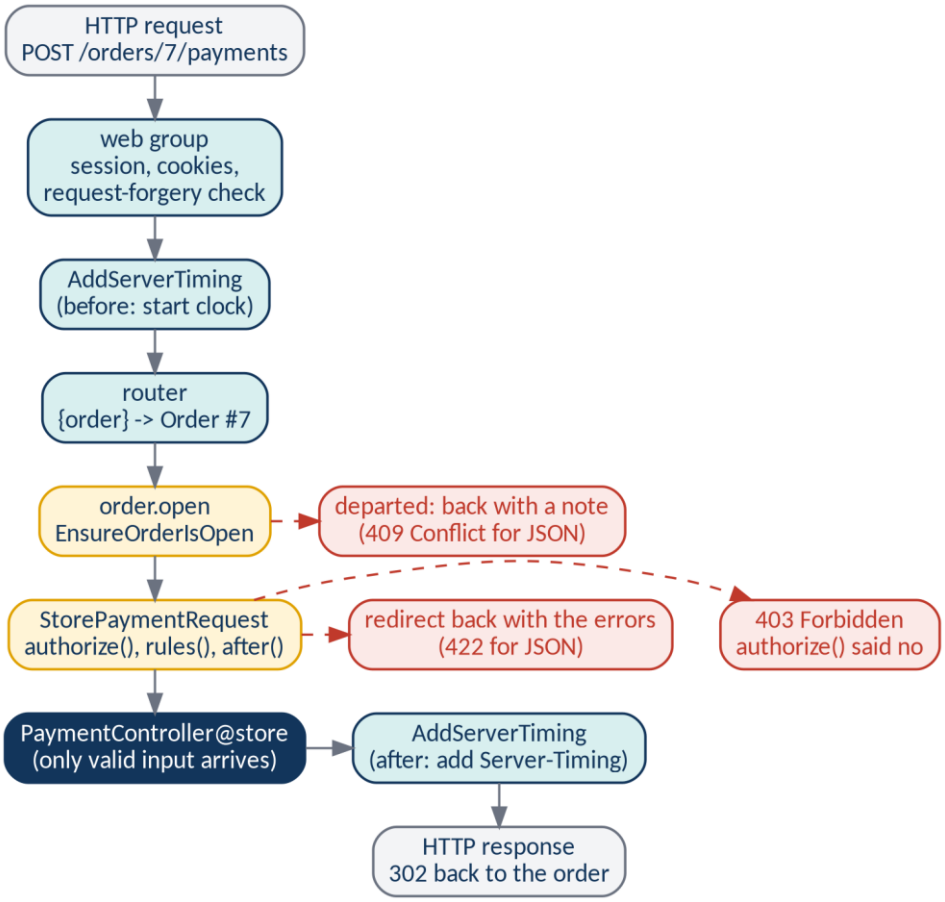


Figure 9.1 The path of one request, a payment on order 7, through middleware, route model binding and a form request; the red boxes are the places where the request stops before the controller.

9.3 Where Chapter 8 Left the Project

This chapter changes the project of Chapter 8, which is the Chapter 7 project plus the Chapter 8 solutions. It adds files and replaces a few, and the chapter's `laravel` folder contains exactly those files. **Table 9.1** lists them, so that you always know what is new and what has changed.

Table 9.1 The files that this chapter adds to or changes in the Chapter 8 project.

File	New or changed	Why
`database/migrations/2026_09_27_000001_add_outlet_id_to_orders_table.php`	new	orders remember their outlet
`app/Models/Customer.php`, `Order.php`, `Payment.php`	changed	`HasFactory`; `Order` gets `outlet()`
`database/factories/*Factory.php` (three)	new	how to invent one record
`database/seeders/*.php` (four)	new or changed	real data and made-up data apart
`app/Http/Middleware/EnsureOrderIsOpen.php`	new	departed orders are closed
`app/Http/Middleware/AddServerTiming.php`	new	an "after" middleware
`bootstrap/app.php`, `routes/web.php`	changed	register and use the middleware
`app/Http/Requests/StoreOrderRequest.php`, `StorePaymentRequest.php`	new	the checklists
`app/Http/Controllers/OrderController.php`, `PaymentController.php`	changed	use the form requests
`resources/views/orders/create.blade.php`	changed	a list of outlets
`tools/pipeline_demo.php`	new	the onion in plain PHP

The domain classes of Chapter 8 stay exactly as they were: `App\Domain\PriceCalculator` still computes every total with whole numbers, and `App\Domain\OrderWorkflow` still decides that an order moves from queued to finished to departed, and may depart only when it is fully paid. This chapter uses both, and changes neither. That is the reward for having put the shop's rules into plain classes: new framework features wrap around them without touching them.

9.4 Migrations in Depth

9.4.1 What the database remembers

A fresh copy of the project that has run the migrations of Chapters 7 and 8 has, besides its own tables, a table named `migrations` with two interesting columns: `migration`, the file name without `.php`, and `batch`, a number. The four `create_*` files of Chapter 7 ran together the first time `php artisan migrate` was typed, so they share batch 1. The `add_rack_to_orders` file of Chapter 8 ran later, on its own, in batch 2. **Figure 9.2** shows that history with this chapter's new file added as batch 3.

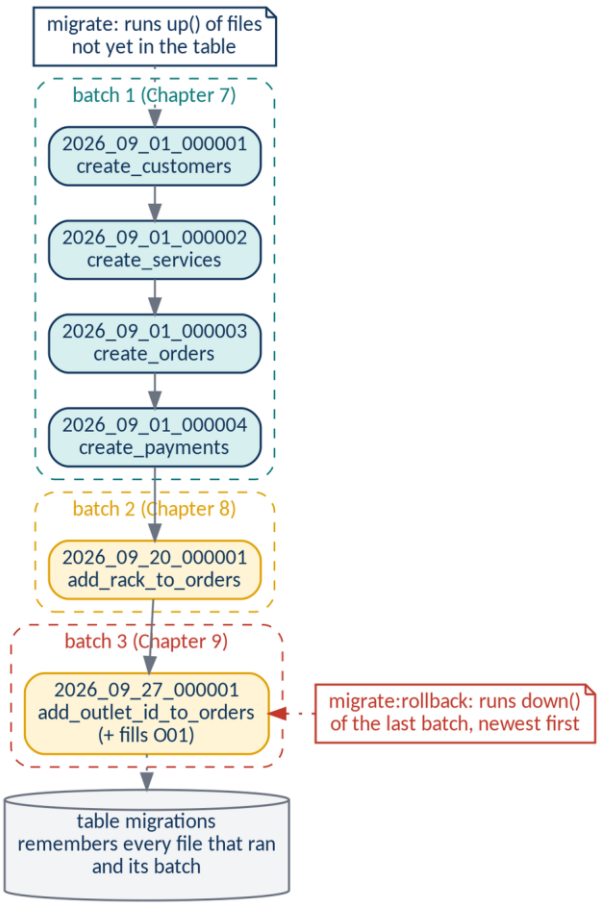


Figure 9.2 The migration history of the project: each run of `migrate` forms a batch, the `migrations` table records it, and `migrate:rollback` undoes only the last batch.

Two consequences follow from this bookkeeping. The first is that a migration file which has already run is never run again on that database, whatever you change in it. If Dimas had edited the Chapter 8 file to make the rack column longer, his laptop, which had already run the file, would keep the short column, while a fresh database would get the long one: two databases built from "the same" files, now different. A migration that has left your own computer is therefore history; a change of mind becomes a *new* migration. The second consequence is that `migrate:rollback` is precise: it undoes only the batch recorded last, so the rack column of batch 2 is safe when batch 3 is rolled back.

9.4.2 Changing a table that already holds orders

The shop wants every order to remember its outlet. In the database this is a new column `outlet_id` on `orders`, holding one of the codes `O01`, `O02` or `O03` from the `outlets` table of Chapter 7, with a foreign key so that no order can point at an outlet that does not exist. The difficulty is that the table already holds orders, perhaps thousands of them, and each of them needs a sensible value.

Listing 9.1 shows the migration. It is created with `php artisan make:migration add_outlet_id_to_orders_table --table=orders`, which writes the skeleton with `Schema::table` instead of `Schema::create`, because the table already exists [2].

Listing 9.1 `2026_09_27_000001_add_outlet_id_to_orders_table.php`: a new column with a foreign key, and the old orders filled in.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Schema;

// Which of the three outlets took the order in.
return new class extends Migration
{
    public function up(): void
    {
        Schema::table('orders', function (Blueprint $table) {
            $table->string('outlet_id', 3)->nullable()->after('customer_id');
            $table->foreign('outlet_id')->references('id')->on('outlets')
                ->restrictOnDelete();
        });

        // Every order so far was taken in at the first shop, Rungkut.
        DB::table('orders')->whereNull('outlet_id')
            ->update(['outlet_id' => 'O01']);
    }

    public function down(): void
    {
        Schema::table('orders', function (Blueprint $table) {
            $table->dropForeign(['outlet_id']);
            $table->dropColumn('outlet_id');
        });
    }
};
```

Read `up()` in three steps. First, the column is added as `nullable()`. That is not laziness: a database cannot add a `NOT NULL` column without a default to a table that has rows, because it would not know what to put in the existing rows, and a default of `'O01'` for every *future* order would be wrong too. Second, the foreign key: `references('id')->on('outlets')` ties the column to the primary key of `outlets`, and `restrictOnDelete()` forbids deleting an outlet that still has orders, the same choice Chapter 7 made for customers and services. Third, the data: every order that existed before this migration was taken in at the first shop, so the migration fills `O01` into all rows whose `outlet_id` is still empty. This is a *data migration* inside a schema migration, and it is written with the query builder (`DB::table`) instead of the `Order` model on purpose: a migration must still work a year from now, when the `Order` class may have changed shape, and the query builder only needs the table.

`after('customer_id')` only decides where the column appears when you look at the table in a database tool; it has no effect on the application and is ignored by databases that cannot place columns, but it keeps the table readable. `down()` undoes the change in the opposite order: first the foreign key, then the column. On SQLite, which the new Laravel projects use by default, Laravel rebuilds the table behind the scenes to drop a column with a foreign key; on MySQL and PostgreSQL it issues `ALTER TABLE` statements [2]. The migration is the same for all of them, which is exactly the point of writing it in PHP instead of in one database's SQL.

9.4.3 The commands you need

Table 9.2 lists the commands that work with migrations. Each one is run in the project folder, in a terminal or in VS Code's terminal panel, on Windows, macOS and Linux alike.

Table 9.2 The migration commands of Artisan and when to use each.

Command	What it does	Use it when
`php artisan make:migration add_x_to_orders_table --table=orders`	writes a new migration file with `Schema::table`	you need a change
`php artisan migrate:status`	lists every file with "Ran" or "Pending" and its batch	before and after every change
`php artisan migrate`	runs the pending files as one new batch	to apply your changes
`php artisan migrate --pretend`	prints the SQL without running it	to review a change on a real database
`php artisan migrate:rollback`	runs `down()` of the last batch, newest first	to undo what you just did
`php artisan migrate:rollback --step=1`	undoes only the newest file	to undo one file of a larger batch
`php artisan migrate:fresh --seed`	drops every table, runs every migration, then the seeders	on your own computer only

The last line deserves its warning. `migrate:fresh` drops every table, with every row in it, before it rebuilds them. On a laptop full of made-up data that is exactly what you want; on the shop's computer it would erase the business. Laravel asks for confirmation when the application's environment is `production`, but the safer habit is to never type it anywhere except your own computer.

9.4.4 Four rules for a database that other people share

The practice of migrations comes down to four rules, and they hold for Entity Framework Core's migrations in Chapter 5 and for any other framework as well.

- **Never edit a migration that has run anywhere but on your own computer.** Write a new one.
- **Make every `down()` a true undo.** If `up()` adds two columns, `down()` drops both; test it once with `migrate:rollback` followed by `migrate`.
- **Add columns as nullable or with a default, then fill them.** A column that must eventually be required can be tightened by a later migration, after the data is in place.
- **Keep data changes inside migrations small and written with the query builder.** Large data changes belong in a command that can be run, watched and repeated separately.

9.5 Factories and Seeders

9.5.1 Models that can be invented

A model can only be made by a factory if it says so, with the `HasFactory` trait. **Listing 9.2** shows the `Order` model of this chapter: compared with Chapter 8 it has the trait, `outlet_id` in its list of fillable fields, and a new `outlet()` relationship. `Customer` and `Payment` gain the trait in the same way. The `#[Fillable]` attribute above the class is the Laravel 13 way of listing the fields that may be filled from an array, which Chapter 7 introduced [1].

Listing 9.2 `app/Models/Order.php`: the Chapter 8 model with `HasFactory` and the new relationship to the outlet.

```php
<?php

namespace App\Models;

use App\Enums\OrderStatus;
use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Eloquent\Relations\HasMany;

#[Fillable(['customer_id', 'outlet_id', 'service_id', 'quantity', 'pickup',
    'delivery_address', 'total', 'status', 'note', 'rack', 'received_at',
    'finished_at', 'departed_at'])]
class Order extends Model
{
    use HasFactory;

    protected function casts(): array
    {
        return [
            'quantity' => 'decimal:2',
            'pickup' => 'boolean',
            'total' => 'integer',
            'status' => OrderStatus::class,
            'received_at' => 'datetime',
            'finished_at' => 'datetime',
            'departed_at' => 'datetime',
        ];
    }

    public function customer(): BelongsTo
    {
        return $this->belongsTo(Customer::class);
    }

    public function outlet(): BelongsTo
    {
        return $this->belongsTo(Outlet::class);
    }

    public function service(): BelongsTo
    {
        return $this->belongsTo(Service::class);
    }
```

Listing 9.2 (Continued) *app/Models/Order.php: the Chapter 8 model with HasFactory and the new relationship to the outlet.*

```php
    public function payments(): HasMany
    {
        return $this->hasMany(Payment::class);
    }

    // Sums the loaded payments; load them with with('payments') first.
    public function paid(): int
    {
        return (int) $this->payments->sum('amount');
    }

    public function balance(): int
    {
        return $this->total - $this->paid();
    }
}
```

By convention, `Order::factory()` looks for the class `Database\Factories\OrderFactory`; the file lives in `database/factories` and is created with `php artisan make:factory OrderFactory --model=Order` [5].

9.5.2 Writing the factories

The customer factory in **Listing 9.3** is the simplest. Its only method, `definition()`, returns the fields of one customer. `fake()` returns a Faker generator: `name()` invents a full name, `numerify('0812########')` replaces every `#` with a random digit, and `optional(0.6)` returns a street address six times out of ten and `null` otherwise, because real customers do not always leave an address [4]. `unique()` makes the generator remember the phone numbers it has already produced in this run, since the `phone` column of `customers` is unique and a duplicate would stop the seeder with an error.

Listing 9.3 `database/factories/CustomerFactory.php`: one invented customer.

```php
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends Factory<\App\Models\Customer>
 */
class CustomerFactory extends Factory
{
    public function definition(): array
    {
        return [
            'name' => fake()->name(),
            'phone' => fake()->unique()->numerify('0812########'),
            'address' => fake()->optional(0.6)->streetAddress(),
        ];
    }
}
```

Faker speaks many languages. With `APP_FAKER_LOCALE=id_ID` in the project's `.env` file, `fake()->name()` produces Indonesian names such as the ones the shop actually serves, and the demo pages look far more like the real counter.

The order factory in **Listing 9.4** has more to teach. Three of its fields are not values but *instructions*. `'customer_id' => Customer::factory()` means "make a new customer for this order, unless I am told otherwise". `'outlet_id'` and `'service_id'` are closures, run only when a record is actually made, that pick a random existing outlet and a random active service. The difference matters: a closure is evaluated once per order, so the orders spread over all outlets and services, while a plain value would be the same for every order.

Listing 9.4 `database/factories/OrderFactory.php`: one invented order, priced by the real rule, with states for finished and departed orders.

```php
<?php

namespace Database\Factories;

use App\Domain\PriceCalculator;
use App\Enums\OrderStatus;
use App\Models\Customer;
use App\Models\Order;
use App\Models\Outlet;
use App\Models\Service;
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends Factory<Order>
 */
class OrderFactory extends Factory
{
    public function definition(): array
    {
        $pickup = fake()->boolean(25);

        return [
            'customer_id' => Customer::factory(),
            'outlet_id' => fn () => Outlet::inRandomOrder()->value('id'),
            'service_id' => fn () => Service::where('is_active', true)
                ->inRandomOrder()->value('id'),
            'quantity' => fake()->randomFloat(1, 1, 9),
            'pickup' => $pickup,
            'delivery_address' => $pickup ? fake()->streetAddress() : null,
            'total' => 0,                       // worked out in configure()
            'status' => OrderStatus::Queued,
            'received_at' => fake()->dateTimeBetween('-30 days', 'now'),
        ];
    }

    // The price always follows the shop's rule; it is never made up.
    public function configure(): static
    {
        return $this->afterMaking(function (Order $order) {
            $service = Service::findOrFail($order->service_id);
            $order->total = PriceCalculator::total($service->per_kg,
                $service->price, (string) $order->quantity, $order->pickup);
        });
    }
```

Listing 9.4 (Continued) *database/factories/OrderFactory.php: one invented order, priced by the real rule, with states for finished and departed orders.*

```php
    public function finished(): static
    {
        return $this->state(fn (array $attributes) => [
            'status' => OrderStatus::Finished,
            'finished_at' => now(),
        ]);
    }

    public function departed(): static
    {
        return $this->state(fn (array $attributes) => [
            'status' => OrderStatus::Departed,
            'finished_at' => now(),
            'departed_at' => now(),
        ]);
    }
}
```

The most important line is the one that is *not* random: the total. A made-up order with a made-up total would teach the report of Chapter 8 to show numbers that no real order could produce, and a bug in the price rule would stay hidden behind them. So the definition sets `total` to 0, and `configure()` registers an `afterMaking` callback that computes the real total with `PriceCalculator::total`, the same method that the counter uses. `afterMaking` runs for every record the factory makes, whether it is only made in memory or also saved, so the total is always right.

The two methods at the end are *states*: named variations of the definition. `Order::factory()->finished()` makes an order that is already washed, and `Order::factory()->departed()` one that has left the shop. States keep tests and seeders readable; the reader of `Order::factory()->departed()->create()` knows at once what kind of order is needed, without reading the list of fields.

9.5.3 Writing the seeders

Chapter 8's `DatabaseSeeder` did everything in one method: it read the price list and the outlets from CSV files and created three customers with their orders. This chapter splits it into three seeders with one job each, and a `DatabaseSeeder` that decides which ones run.

Listing 9.5 is the seeder for the real data. It reads the same `Service.csv` and `Outlet.csv` files as Chapters 3, 5 and 7, so the ASP.NET Core and Laravel versions of the shop still charge exactly the same prices. The one change from Chapter 8 is `updateOrCreate` in place of `create`: the first argument says how to find an existing row (by its code, such as `S0001`) and the second what it should contain. Running the seeder twice therefore changes nothing the second time, and running it after the owner corrects a price in the CSV file updates that price instead of failing on a duplicate key. A seeder that can safely run again is called *idempotent*, and real-data seeders should always be.

Listing 9.5 `database/seeders/PriceListSeeder.php`: the real price list and outlets, safe to run again.

```php
<?php

namespace Database\Seeders;

use App\Models\Outlet;
use App\Models\Service;
use Illuminate\Database\Seeder;

// The real price list and the three outlets: needed in every database.
class PriceListSeeder extends Seeder
{
    public function run(): void
    {
        foreach ($this->csv('Service.csv') as [$id, $description, $price]) {
            Service::updateOrCreate(['id' => $id], [
                'description' => $description, 'price' => (int) $price]);
        }
        foreach ($this->csv('Outlet.csv') as [$id, $name, $address, $phone,
            $opened]) {
            Outlet::updateOrCreate(['id' => $id],
                compact('name', 'address', 'phone', 'opened'));
        }
    }

    // The rows of database/data/<file>, without the header line.
    private function csv(string $file): array
    {
        $lines = file(database_path("data/{$file}"), FILE_IGNORE_NEW_LINES);
        $rows = array_map(fn ($line) => str_getcsv($line, ',', '"', ''), $lines);

        return array_slice($rows, 1);
    }
}
```

The three hand-written customers of Chapter 7, Sari, Budi and Dewi, move unchanged into `DemoSeeder`, with an outlet added to each of their orders. They are useful because the book refers to them by name: Sari's order of 4.5 kg is always there and always costs Rp 36.000. **Listing 9.6** adds the crowd around them.

Listing 9.6 `database/seeders/BusyMonthSeeder.php`: forty invented customers and a month of orders that move through the workflow as real ones do.

```php
<?php

namespace Database\Seeders;

use App\Enums\OrderStatus;
use App\Models\Customer;
use App\Models\Order;
use App\Models\Payment;
use Illuminate\Database\Seeder;

// A made-up month of work, so that lists, paging and reports have data.
class BusyMonthSeeder extends Seeder
{
    public function run(): void
    {
        Customer::factory()->count(40)->create()->each(function ($customer) {
            Order::factory()->count(fake()->numberBetween(0, 5))
                ->for($customer)->create()
                ->each(fn (Order $order) => $this->liveThrough($order));
        });
    }

    // Older orders are further along, as they would be in a real shop.
    private function liveThrough(Order $order): void
    {
        if (fake()->boolean(70)) {                  // most pay when they drop
            Payment::factory()->for($order)->create([
                'amount' => $order->total, 'paid_at' => $order->received_at]);
        }
        $age = $order->received_at->diffInDays(now());
        if ($age >= 2) {
            $order->status = OrderStatus::Finished;
            $order->finished_at = $order->received_at->copy()->addDays(2);
        }
        if ($age >= 5 && $order->payments()->sum('amount') >= $order->total) {
            $order->status = OrderStatus::Departed;
            $order->departed_at = $order->received_at->copy()->addDays(5);
        }
        $order->save();
    }
}
```

`Customer::factory()->count(40)->create()` makes and saves forty customers and returns them as a collection. For each customer, `Order::factory()->count(...)->for($customer)->create()` makes between zero and five orders that belong to that customer; `for()` replaces the "make a new customer" instruction of the factory. The private method `liveThrough` then gives each order a plausible history. Seven orders in ten are paid in full when they are dropped off. An order that is at least two days old is finished, and one that is at least five days old and fully paid has departed. This follows the rule of Chapter 8's `OrderWorkflow`, that laundry leaves only when it is paid, so the made-up data never contains a situation the real application would refuse to create.

Listing 9.7 ties the three together.

Listing 9.7 `database/seeders/DatabaseSeeder.php`: real data always, made-up data only on a developer's or a test computer.

```php
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    public function run(): void
    {
        $this->call(PriceListSeeder::class);      // real data: always

        if (app()->environment('local', 'testing')) {
            $this->call([DemoSeeder::class,       // made-up data: never in
                BusyMonthSeeder::class]);         // the shop's real database
        }
    }
}
```

`app()->environment('local', 'testing')` is true when the `APP_ENV` setting in `.env` is `local`, as it is in every new project, or `testing`, as it is when the tests of Chapter 10 run. On the shop's computer `APP_ENV` is `production`, so `php artisan db:seed` there installs or updates the price list and the outlets and nothing else. The order of the calls in `$this->call([...])` is the order in which the seeders run, and it must follow the foreign keys: services and outlets before orders.

9.5.4 Running them

With the files in place, one command rebuilds a laptop's database from nothing:

```
php artisan migrate:fresh --seed
```

It drops every table, runs all six migration files, from Chapter 7's four through Chapter 8's rack column to this chapter's outlet column, and then runs `DatabaseSeeder`. Open the order list afterwards and it has several pages, with orders in every status and at every outlet; the daily report of Chapter 8 has a column of numbers for each of the last thirty days. Because the data is random, your list will not look like anybody else's, but Sari's, Budi's and Dewi's orders are always there. To look at the data from the terminal, `php artisan tinker` opens a PHP prompt inside the application where `Order::where('status', 'departed')->count()` or `Order::first()->outlet->name` can be typed and answered at once.

9.6 Middleware

9.6.1 The onion, without Laravel

Middleware becomes clear the moment you see how little code it needs. **Listing 9.8** builds a two-layer pipeline around a pretend controller in plain PHP, with nothing but closures and `array_reduce`. It runs anywhere PHP runs, with no framework installed.

Listing 9.8 `tools/pipeline_demo.php`: a middleware pipeline in plain PHP, the same idea that Laravel's pipeline is built on.

```php
<?php

// How a middleware pipeline works, in plain PHP (no Laravel needed):
//   php tools/pipeline_demo.php
// Each layer gets the request and $next, the rest of the pipeline.

$log = fn (string $line) => print("  $line\n");

$timing = function (array $request, Closure $next) use ($log) {
    $log('timing: start the clock');
    $response = $next($request);
    $log('timing: add the Server-Timing header');
    return $response + ['Server-Timing' => 'app;dur=1.2'];
};

$orderOpen = function (array $request, Closure $next) use ($log) {
    if ($request['status'] === 'departed') {
        $log('order.open: departed, stop here');
        return ['status' => 409];               // $next is never called
    }
    $log('order.open: still open, go on');
    return $next($request);
};

$controller = function (array $request) use ($log) {
    $log("controller: record the payment on order #{$request['order']}");
    return ['status' => 302];
};

// Wrap the controller in the layers, the last layer innermost.
$pipeline = array_reduce(array_reverse([$timing, $orderOpen]),
    fn (Closure $next, Closure $layer) =>
        fn (array $request) => $layer($request, $next),
    $controller);

foreach (['finished', 'departed'] as $status) {
    echo "Payment on an order that is $status:\n";
    $response = $pipeline(['order' => 7, 'status' => $status]);
    echo '  -> ', json_encode($response), "\n";
}
```

Each layer is a function of two arguments, the request and `$next`. The interesting line is the `array_reduce`: it starts with the controller and wraps it, layer by layer, in a new function that calls the layer with the request and with everything inside it as `$next`. The layers are reversed first, so that the first layer in the list ends up outermost, which is how Laravel orders them too. The timing layer's "Server-Timing" value is a fixed number here, because the demo is about the order of events, not about measuring. Running it prints:

```
php tools/pipeline_demo.php
```

```
Payment on an order that is finished:
  timing: start the clock
  order.open: still open, go on
  controller: record the payment on order #7
  timing: add the Server-Timing header
  -> {"status":302,"Server-Timing":"app;dur=1.2"}
Payment on an order that is departed:
  timing: start the clock
  order.open: departed, stop here
  timing: add the Server-Timing header
  -> {"status":409,"Server-Timing":"app;dur=1.2"}
```

Read the second run carefully. The `order.open` layer did not call `$next`, so the controller never ran, and nothing was recorded. Yet the timing layer, which is *outside* it, still ran its "after" part and

added its header to the 409 answer. A middleware that stops a request stops everything *inside* it, never anything outside it. Laravel's pipeline adds services and conveniences around this idea, but the idea is exactly these twenty lines.

9.6.2 A "before" middleware for a rule of the shop

Listing 9.9 is the rule that would have saved Bu Wati the double payment: an order that has left the shop is closed. The class is created with `php artisan make:middleware EnsureOrderIsOpen` and lives in `app/Http/Middleware`.

Listing 9.9 `app/Http/Middleware/EnsureOrderIsOpen.php`: nothing may change an order that has left the shop.

```php
<?php

namespace App\Http\Middleware;

use App\Enums\OrderStatus;
use App\Models\Order;
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

// An order that has left the shop is history: nothing may change it.
class EnsureOrderIsOpen
{
    public function handle(Request $request, Closure $next): Response
    {
        $order = $request->route('order');      // already bound by the router

        if ($order instanceof Order
            && $order->status === OrderStatus::Departed) {
            $message = "Order #{$order->id} has left the shop and is closed.";
            if ($request->expectsJson()) {
                abort(409, $message);           // for programs: 409 Conflict
            }
            return back()->with('error', $message);   // for people: a note
        }

        return $next($request);
    }
}
```

`$request->route('order')` asks the router for the route parameter named `order`. Because the controllers of Chapter 7 declare `Order $order` as a parameter, Laravel's *route model binding* has already turned the `7` in the URL into the `Order` with id 7, or answered 404 if there is none, before our middleware runs; we receive the model, not the text [9]. The `instanceof` test is a guard: if the middleware is ever attached to a route without an `{order}` parameter, it simply lets the request through.

The answer to a closed order depends on who is asking. A person at the counter clicked a button on the order page, so the kind answer is to go back to that page with a note, using the same `error` flash message that Chapter 7's layout already shows in red. A program, such as the phone app of Chapter 10, cannot read a red note; it sends `Accept: application/json` and needs a status code. `expectsJson()` tells the two apart, and `abort(409, ...)` answers the program with *409 Conflict*, the HTTP status for a request that conflicts with the current state of the resource [10]. Laravel turns the `abort` into a JSON body `{"message": "..."}` for such clients.

9.6.3 An “after” middleware

Listing 9.10 shows the other half of the onion. `AddServerTiming` starts a clock, lets the rest of the pipeline run by calling `$next`, and then adds a header to the response it got back.

Listing 9.10 `app/Http/Middleware/AddServerTiming.php`: an “after” middleware that measures the rest of the pipeline.

```php
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

// Measures how long the rest of the pipeline took, for the browser's tools.
class AddServerTiming
{
    public function handle(Request $request, Closure $next): Response
    {
        $start = hrtime(true);                  // before: start the clock

        $response = $next($request);

        $ms = (hrtime(true) - $start) / 1e6;    // after: stop it
        $response->headers->set('Server-Timing',
            sprintf('app;dur=%.1f', $ms));

        return $response;
    }
}
```

`hrtime(true)` returns a clock reading in nanoseconds; the difference divided by 10^6 is milliseconds. The `Server-Timing` header is a standard that browsers understand: open the developer tools of Chrome, Edge or Firefox, choose the Network tab, click a request and look at “Timing”, and the value appears as a bar named `app` [11]. When the order list feels slow at the Keputih outlet, this header tells you at once whether the time is spent in the application or on the network.

9.6.4 Registering the middleware

A middleware class does nothing until it is registered. Since Laravel 11 this happens in `bootstrap/app.php`, which replaced the older `app/Http/Kernel.php` [7]. **Listing 9.11** shows the file of this chapter.

Listing 9.11 `bootstrap/app.php`: `AddServerTiming` joins the `web` group, and `EnsureOrderIsOpen` gets the short name `order.open`.

```php
<?php

use App\Http\Middleware\AddServerTiming;
use App\Http\Middleware\EnsureOrderIsOpen;
use Illuminate\Foundation\Application;
use Illuminate\Foundation\Configuration\Exceptions;
use Illuminate\Foundation\Configuration\Middleware;

return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )
    ->withMiddleware(function (Middleware $middleware): void {
        $middleware->web(append: [AddServerTiming::class]);
        $middleware->alias([
            'order.open' => EnsureOrderIsOpen::class,
        ]);
    })
    ->withExceptions(function (Exceptions $exceptions): void {
        //
    })->create();
```

There are three ways to register a middleware, and this file uses two of them. `$middleware->web(append: [...])` adds a class to the end of the `web` *group*, the list that runs for every route in `routes/web.php`; that is right for timing, which should measure every page. `$middleware->alias([...])` gives a class a short name that routes can ask for; that is right for `order.open`, which only some routes need. The third way, `$middleware->append(...)`, would add a class to the *global* middleware that runs for every request of every kind, including the ones that match no route; neither of our classes needs that. **Table 9.3** lists the middleware that Laravel already provides under short names and that the following chapters use.

Table 9.3 Built-in route middleware of Laravel that this book uses, with their short names.

Name	What it checks or does	Typical use
`auth`	the user is signed in, else redirect to sign-in (or 401 for JSON)	staff pages (Chapter 10)
`guest`	the user is *not* signed in	the sign-in page (Chapter 10)
`throttle:5,1`	at most 5 requests a minute per user or address, else 429	the sign-in form
`can:manage-prices`	the user passes an authorization check, else 403	the price list (Chapter 10)
`verified`	the user has confirmed an e-mail address	optional sign-up flows
`signed`	the URL carries a valid signature	links sent by e-mail or WhatsApp

9.6.5 Attaching middleware to routes

Listing 9.12 is the routes file of Chapter 8 with one change at the bottom: the two routes that change an order, “advance” and “record a payment”, are now inside a group that uses `order.open`.

Listing 9.12 `routes/web.php`: the routes of Chapter 8, with the routes that change an order grouped behind `order.open`.

```php
<?php

use App\Http\Controllers\CounterController;
use App\Http\Controllers\CustomerController;
use App\Http\Controllers\HomeController;
use App\Http\Controllers\OrderController;
use App\Http\Controllers\OutletController;
use App\Http\Controllers\PaymentController;
use App\Http\Controllers\PriceController;
use App\Http\Controllers\ReportController;
use App\Http\Controllers\ServiceController;
use Illuminate\Support\Facades\Route;

// The public pages (Chapter 3 built them in ASP.NET Core).
Route::get('/', HomeController::class)->name('home');
Route::get('services', [ServiceController::class, 'index'])
    ->name('services.index');
Route::get('services/{service}', [ServiceController::class, 'show'])
    ->name('services.show');
Route::get('outlets', [OutletController::class, 'index'])
    ->name('outlets.index');

// The staff pages (Chapter 5 built them in ASP.NET Core).
Route::get('counter', CounterController::class)->name('counter');
Route::resource('customers', CustomerController::class);
Route::resource('prices', PriceController::class)
    ->parameters(['prices' => 'service'])
    ->except(['show', 'destroy']);
Route::resource('orders', OrderController::class)
    ->only(['index', 'create', 'store', 'show']);
Route::get('reports/daily', [ReportController::class, 'daily'])
    ->name('reports.daily');

// Chapter 9: nothing may change an order that has left the shop.
Route::middleware('order.open')->group(function () {
    Route::post('orders/{order}/advance', [OrderController::class, 'advance'])
        ->name('orders.advance');
    Route::post('orders/{order}/payments', [PaymentController::class, 'store'])
        ->name('orders.payments.store');
});
```

`Route::middleware('order.open')->group(function () { ... })` applies the middleware to every route defined inside the closure. The same effect for a single route is `->middleware('order.open')` at the end of its definition. Grouping has an advantage that is easy to miss: the rule is now visible *in the routes file*. Anyone who opens `routes/web.php` sees which pages are guarded by which rules, without opening a single controller. Showing the order and listing the orders stay outside the group, of course; a departed order must still be visible.

Chapter 8's `OrderController::advance` also refused to move a departed order, through `OrderWorkflow`, and it still does. The two checks are not a duplicate but two layers of defense: the workflow protects the rule wherever it is called from, even from a command or a test, and the middleware stops a request early, for every route in the group, including routes that do not use the workflow at all, such as payments.

9.7 Form Requests

9.7.1 The order checklist

Listing 9.13 moves Chapter 8's validation of a new order out of the controller, adds the outlet, and adds the tidying that the counter needs. It is created with `php artisan make:request StoreOrderRequest` and lives in `app/Http/Requests`.

Listing 9.13 `app/Http/Requests/StoreOrderRequest.php`: the checklist for a new order, with tidying, rules, messages and field names.

```php
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rule;

class StoreOrderRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;               // Chapter 10 decides who may do this
    }

    // Tidy the raw input before any rule looks at it.
    protected function prepareForValidation(): void
    {
        $this->merge([
            'quantity' => str_replace(',', '.', (string) $this->quantity),
            'pickup' => $this->boolean('pickup'),
            'rack' => strtoupper(trim((string) $this->rack)) ?: null,
        ]);
    }

    public function rules(): array
    {
        return [
            'customer_id' => ['required', 'integer', 'exists:customers,id'],
            'outlet_id' => ['required', 'string', 'exists:outlets,id'],
            'service_id' => ['required', 'string',
                Rule::exists('services', 'id')->where('is_active', true)],
            'quantity' => ['required', 'numeric', 'decimal:0,2', 'min:0.1',
                'max:100'],
            'pickup' => ['boolean'],
            'delivery_address' => ['nullable', 'required_if_accepted:pickup',
                'string', 'max:200'],
            'rack' => ['nullable', 'regex:/^[A-Z]-[0-9]{2}$/'],
            'note' => ['nullable', 'string', 'max:200'],
        ];
    }
```

Listing 9.13 (Continued) *app/Http/Requests/StoreOrderRequest.php: the checklist for a new order, with tidying, rules, messages and field names.*

```
    public function messages(): array
    {
        return [
            'rack.regex' => 'A rack looks like B-03: a letter, a dash, two digits.',
            'delivery_address.required_if_accepted' =>
                'Where should the laundry be delivered?',
        ];
    }

    public function attributes(): array
    {
        return ['outlet_id' => 'outlet', 'service_id' => 'service',
            'customer_id' => 'customer'];
    }
}
```

The methods run in a fixed order. `authorize()` comes first; returning `false` would answer 403 before any rule is checked, and Chapter 10 replaces the `true` with a real check. `prepareForValidation()` then tidies the raw input. Cashiers in Indonesia type "4,5" as often as "4.5", because the comma is the Indonesian decimal separator, so the comma becomes a dot. An unticked checkbox sends nothing at all, so `boolean('pickup')` turns "missing" into `false` and "1" into `true`. A rack typed as "b-03" becomes "B-03", and an empty rack becomes `null`. The rules then check the tidied values.

Most rules are those of Chapter 8. `exists:outlets,id` is new and checks that the outlet code is a row of the `outlets` table; `Rule::exists('services', 'id')->where('is_active', true)` checks more precisely that the service exists *and* is still offered. `decimal:0,2` allows at most two decimals, which is what the `quantity` column can store, and `required_if_accepted:pickup` makes the delivery address required only when the pickup box is ticked. `messages()` replaces two of Laravel's generic messages with the words a cashier needs, and `attributes()` makes the other messages say "The outlet field is required" instead of "The outlet id field is required" [8].

9.7.2 A rule that needs the order

Listing 9.14 is the checklist for a payment, and it shows the method that the controller version could not express well: `after()`.

Listing 9.14 `app/Http/Requests/StorePaymentRequest.php`: simple rules first, then a check against what is still owed.

```php
<?php

namespace App\Http\Requests;

use App\Enums\PaymentMethod;
use App\Support\Money;
use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rule;
use Illuminate\Validation\Validator;

class StorePaymentRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return [
            'amount' => ['required', 'integer', 'min:1000'],
            'method' => ['required', Rule::enum(PaymentMethod::class)],
        ];
    }

    // A check that needs the order itself, run after the rules above pass.
    public function after(): array
    {
        return [
            function (Validator $validator) {
                if ($validator->errors()->isNotEmpty()) {
                    return;
                }
                $order = $this->route('order')->load('payments');
                if ((int) $this->input('amount') > $order->balance()) {
                    $validator->errors()->add('amount', 'Only '
                        . Money::rp($order->balance()) . ' is still owed.');
                }
            },
        ];
    }
}
```

`after()` returns a list of functions that Laravel calls after the rules have been checked, each with the validator. Ours first returns at once if the simple rules already failed: there is no point in comparing an amount that is not a number with the balance. Otherwise it takes the order from the route, as the middleware did, loads its payments, and adds an error to `amount` if the amount is larger than the balance. The error joins the others: the cashier sees “Only Rp 16.000 is still owed.” under the amount field, exactly like a message from an ordinary rule, and the controller is never called. Chapter 8 solved the same problem with a `max:` rule built from the balance inside the controller; the form request keeps that knowledge next to the other payment rules, where the next programmer will look for it.

9.7.3 Controllers that trust their input

With the checklists moved out, the controllers become short. **Listing 9.15** is the whole payment controller.

Listing 9.15 `app/Http/Controllers/PaymentController.php`: by the time `store` runs, the request has been checked.

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests\StorePaymentRequest;
use App\Models\Order;
use App\Support\Money;

class PaymentController extends Controller
{
    // POST /orders/5/payments: the request object has checked everything.
    public function store(StorePaymentRequest $request, Order $order)
    {
        $data = $request->validated();
        $order->payments()->create([...$data, 'paid_at' => now()]);

        return back()->with('message',
            'Payment of ' . Money::rp((int) $data['amount']) . ' recorded.');
    }
}
```

The only change in the method's signature is the type of its first parameter: `StorePaymentRequest` instead of `Request`. That is all it takes. Laravel's service container sees the type, builds the form request, validates it and only then calls `store`. `$request->validated()` returns *only* the fields that have rules, here `amount` and `method`; any other field a malicious form might add, such as `order_id` or `paid_at`, is left out, so it can never reach `create()`. **Listing 9.16** shows the order controller, where `store` now receives a `StoreOrderRequest` and `create` also sends the list of outlets to the form.

Listing 9.16 `app/Http/Controllers/OrderController.php`: the Chapter 8 controller with `StoreOrderRequest` in `store` and the outlets in `create`.

```php
<?php

namespace App\Http\Controllers;

use App\Domain\OrderWorkflow;
use App\Domain\PriceCalculator;
use App\Enums\OrderStatus;
use App\Http\Requests\StoreOrderRequest;
use App\Models\Customer;
use App\Models\Order;
use App\Models\Outlet;
use App\Models\Service;
use App\Support\Money;
use DomainException;
use Illuminate\Http\Request;

class OrderController extends Controller
{
    public function index(Request $request)
    {
        $status = OrderStatus::tryFrom((string) $request->query('status'));
        $orders = Order::query()
            ->select(['id', 'customer_id', 'service_id', 'total', 'status',
                'received_at'])
            ->with(['customer:id,name', 'service:id,description'])
            ->withSum('payments', 'amount')     // adds payments_sum_amount
            ->when($status, fn ($query) => $query->where('status', $status))
            ->orderByDesc('received_at')->orderByDesc('id')
            ->paginate(20)
            ->withQueryString();

        return view('orders.index', compact('orders', 'status'));
    }

    public function show(Order $order)
    {
        $order->load(['customer', 'service',
            'payments' => fn ($q) => $q->orderBy('paid_at')]);

        return view('orders.show', compact('order'));
    }
```

Listing 9.16 (Continued) *app/Http/Controllers/OrderController.php: the Chapter 8 controller with StoreOrderRequest in store and the outlets in create.*

```
    public function create(Request $request)
    {
        return view('orders.create', [
            'customers' => Customer::orderBy('name')->get(),
            'outlets' => Outlet::orderBy('id')->get(),
            'services' => Service::where('is_active', true)
                ->orderBy('id')->get(),
            'customerId' => $request->integer('customer_id'),
        ]);
    }

    // Only checked input arrives here; see StoreOrderRequest.
    public function store(StoreOrderRequest $request)
    {
        $data = $request->validated();
        $service = Service::findOrFail($data['service_id']);

        $order = Order::create([
            ...$data,
            'total' => PriceCalculator::total($service->per_kg,
                $service->price, (string) $data['quantity'], $data['pickup']),
            'status' => OrderStatus::Queued,
            'received_at' => now(),
        ]);

        return redirect()->route('orders.show', $order)->with('message',
            "Order #{$order->id} created: " . Money::rp($order->total) . '.');
    }

    public function advance(Order $order)
    {
        $balance = $order->total - (int) $order->payments()->sum('amount');
        try {
            $next = OrderWorkflow::advance($order->status, $balance);
        } catch (DomainException $e) {
            return back()->with('error', $e->getMessage());
        }
        $order->status = $next;
        if ($next === OrderStatus::Finished) {
            $order->finished_at = now();
        } else {
            $order->departed_at = now();
        }
        $order->save();
```

Listing 9.16 (Continued) *app/Http/Controllers/OrderController.php: the Chapter 8 controller with StoreOrderRequest in store and the outlets in create.*

```
        return back()->with('message',
            "Order #{$order->id} is now {$next->label()}.");
    }
}
```

`...$data` spreads the validated fields into the array for `Order::create`, and the controller adds the three fields that the customer never chooses: the total, from `PriceCalculator`; the status, always queued; and the time it was received. The form of Chapter 8 needs one more field, a list of outlets; `resources/views/orders/create.blade.php` in the chapter's folder adds it above the list of services.

9.8 Running It

After copying the chapter's files over the Chapter 8 project, run the steps in the chapter's `README.md`. Laravel 13 needs PHP 8.3 or newer; PHP 8.5, released in November 2025, is the current version and the one this book recommends [12]. On a database that already holds your Chapter 8 orders, first run `php artisan migrate:status`: the six earlier files show "Ran" and the new outlet migration shows "Pending". `php artisan migrate` then runs only that file, as batch 3, and every existing order now belongs to Rungkut. `php artisan migrate:status` confirms it, and `php artisan migrate:rollback` followed by `php artisan migrate` is a good way to prove to yourself that `down()` really undoes `up()`.

To see the new features at work, rebuild with demo data (`php artisan migrate:fresh --seed`), start the server (`php artisan serve --port=4090`) and try four things.

1. Open **Orders**, then **New order**, choose Budi, the Wiyung outlet, "Wash, dry & fold", type `2,5` as the quantity, tick pickup and leave the address empty. The form comes back with "Where should the laundry be delivered?" and keeps everything you typed. Add an address and save: the order is created for 3 kg, the minimum, plus the pickup fee, Rp 34.000.
2. On that order, record a payment of Rp 40.000. The page comes back with "Only Rp 34.000 is still owed."
3. Pay Rp 34.000, then press **Mark finished** and **Mark picked up**. The order is now departed. Open the page again in a second browser tab and try to record a payment there: you return to the order with the red note "Order #… has left the shop and is closed."
4. Open the browser's developer tools on any page, choose the Network tab and click the page's request: the Timing section shows the `app` value from `AddServerTiming`.

These steps were written from the code and the Laravel 13 documentation; the environment in which this book was written could not install Laravel, as the chapter's `README.md` explains, so no screenshots of this application are shown. The plain-PHP pipeline demo of Section 9.6.1 was run, and its output is the real one.

9.9 Making It Better

Middleware parameters. A route middleware can receive arguments after a colon: `->middleware('throttle:5,1')` passes "5" and "1" to Laravel's rate limiter. Your own middleware gets its parameters as extra arguments of `handle` after `$next`, always as strings. Exercise 9.3 writes a middleware that logs slow requests, with the limit in milliseconds as a parameter.

Middleware on the controller. Laravel 13 lets a controller declare its middleware with PHP attributes, such as a `#[Middleware(...)]` attribute on the class or on a method [1]. That keeps the rule next to the code it guards; the routes file keeps it next to the URL. Either is fine, but pick one per project, so that nobody has to look in two places.

Safer validated data. `$request->safe()->only(['amount', 'method'])` and `$request->safe()->except([...])` return a chosen part of the validated data, which is handy when a controller needs to treat some fields differently from the others [8].

Big data migrations. The outlet migration fills every old order with one `UPDATE` statement, which is fine for thousands of rows. When each row needs its own value, the query builder's `lazyById()` walks the table in chunks of a thousand rows without loading it all into memory; Exercise 9.2 uses it.

Factories in tests. The states `finished()` and `departed()` are written for tests as much as for seeders. Chapter 10 creates a departed order in one line, `Order::factory()->departed()->create()`, and checks that the API refuses a payment on it.

9.10 Using It for Real

The four tools of this chapter carry over to any small business with a counter, whether a tailor, a print shop, a motorbike workshop or a pharmacy's prescription desk.

- **Start the migration history on the first day.** Even a one-table prototype should create its table with a migration. The day a second computer, a second developer or a hosting service appears, the history is already there. When a business runs on a database that was built by hand, write one migration that describes the current tables, mark it as run on that database, and continue from there.
- **Seed the real reference data from files the owner can read.** Prices, branches, product categories and tax rates change, and the owner is the one who knows them. A CSV file in `database/data`, read by an idempotent seeder, lets the owner correct a price and a developer apply the correction with one command, as Profitina Laundry does with `Service.csv`.
- **Invent realistic data, never realistic people.** Made-up customers must be obviously made up and must never reach the real database. The `environment('local', 'testing')` guard of **Listing 9.7** is the minimum. When real data is copied to a laptop to reproduce a problem, it contains personal data, names, phone numbers and addresses, which Indonesia's personal data protection law, UU PDP 27/2022, protects; mask or delete it before it leaves the shop's computer.
- **Turn every "this must never happen again" into middleware or a form request.** A workshop that must not hand over a motorbike before the bill is paid, a print shop that must not reprint a job marked "delivered", a pharmacy that must not change a dispensed prescription: each is the same `EnsureOrderIsOpen` with different words. Put the rule where every route that could break it passes.
- **Validate with the words of the counter.** Custom messages in the staff's language, and tidying such as the comma-to-dot rule, cost a few lines and save many mistakes at the counter [13].

9.11 Common Mistakes, and How to Avoid Them

- **Editing a migration that has already run.** The change silently does nothing on the databases that ran the old version. Write a new migration instead.
- **Adding a required column to a table with rows.** `$table->string('outlet_id', 3)` without `nullable()` or a default fails on a table that has orders. Add it nullable, fill it, and tighten it later if needed.
- **Using models inside migrations.** `Order::query()->update(...)` in a migration works today and breaks the day the model gains a cast or a global scope that the old table cannot satisfy. Use `DB::table()`.
- **A `down()` that does not undo `up()`.** Dropping the column but not the foreign key, or forgetting one of two columns, makes `migrate:rollback` fail halfway and leaves the database between two states. Test the undo once.
- **Made-up data in the real database.** A `DatabaseSeeder` that calls the demo seeders unconditionally puts invented customers into the shop's system the first time somebody runs `db:seed` there. Guard them by environment.

- **Random values where the rules decide.** A factory that invents the total, or a departed order that is not paid, creates data the application itself could never produce, and reports and tests built on it lie. Compute derived fields with the real rules, as `OrderFactory::configure` does.
- **Forgetting `return $next($request)`.** A middleware that checks something and then forgets to call `$next` answers every request with an empty response. Every path through `handle` must return a response: either its own or the one from `$next`.
- **Registering a middleware globally when it needs a route parameter.** Global middleware runs before routing, so `$request->route('order')` is `null` there. `EnsureOrderIsOpen` belongs on routes, as an alias.
- **Using `$request->all()` after validation.** It returns every field that was sent, including ones without rules. Use `$request->validated()` or `$request->safe()`.
- **Checking the database in `rules()` without guarding.** A rule that calls `$this->route('order')->balance()` works until a request arrives with an invalid amount; put checks that depend on other fields in `after()`, behind the "simple rules passed" guard.

9.12 Chapter Summary

- A migration is a recorded, reversible change to the database. The `migrations` table remembers which files ran and in which batch; `migrate` runs the missing files as a new batch and `migrate:rollback` undoes the last batch. A migration that has run elsewhere is never edited, only followed by a new one.
- Changing a table that holds data means adding the column as nullable or with a default, filling it (here: all old orders belong to outlet O01) with the query builder, and writing a `down()` that truly undoes the change.
- A factory describes one plausible record, with Faker for invented values, closures and related factories for foreign keys, `afterMaking` for fields that the real rules compute, and states for named variations. A seeder decides what to create and in which order.
- Real reference data (the price list, the outlets) is seeded idempotently with `updateOrCreate` in every database; made-up data only on developers' and test computers.
- Middleware wraps the controller like the layers of an onion. A "before" middleware may stop a request by not calling `$next`; an "after" middleware changes the response that `$next` returns. Middleware is registered in `bootstrap/app.php`, in a group or under an alias, and attached to routes one by one or in groups.
- `EnsureOrderIsOpen` closes departed orders for every route that changes an order, answering people with a note and programs with 409; `AddServerTiming` measures every page.
- A form request holds the checklist of one kind of request: `authorize`, `prepareForValidation`, `rules`, `messages`, `attributes` and `after`. The controller receives it as a typed parameter and only runs when everything passed, and `validated()` gives it only the checked fields.

9.13 Exercises

1. A new laptop has run no migrations. Dimas runs `php artisan migrate` once with all six files of this project present, then `php artisan migrate:rollback`. Which tables and columns exist afterwards? Answer the same question for the shop's computer, which ran Chapter 7's files, then Chapter 8's, then this chapter's, each on a different day.
2. The shop wants to record when each order was *promised* to the customer. Write a migration that adds a nullable `promised_at` timestamp to `orders`, fills it for the existing orders with the

time received plus two days, and can be undone. Walk the table in chunks rather than loading it at once.

3. Write a middleware `LogSlowRequests` that takes a limit in milliseconds as a parameter (default 500) and writes a warning to Laravel's log for every request slower than that. Show how to register it under the alias `slow` and attach it with a limit of 300 ms to the order list.
4. A cashier types the rack as "b-3". Follow it through `StoreOrderRequest`: what does `prepareForValidation` make of it, which rule fails, and which message does the cashier see? Would "B-03" (with a space at the end) pass?
5. Somebody runs `php artisan db:seed` twice on a laptop, without `migrate:fresh` in between. What happens with `PriceListSeeder`, with `DemoSeeder` and with `BusyMonthSeeder`? Which of them should be changed if repeated runs must work, and how?
6. In the pipeline of **Listing 9.8**, swap the two layers so that `$orderOpen` is outermost. What does the program print for the departed order, and why is the new order of layers worse for `AddServerTiming` in the real application?

9.14 Solutions to the Exercises

1. The single `migrate` puts all six files into batch 1, because they ran together. `migrate:rollback` undoes the last batch, which is all of batch 1, newest first: the outlet column, the rack column, then payments, orders, services with outlets, and customers. The laptop is left with only Laravel's own tables and an empty `migrations` table. On the shop's computer the six files are spread over three batches (the four files of Chapter 7 in batch 1, Chapter 8's in batch 2, this chapter's in batch 3), so the same `migrate:rollback` undoes only batch 3: the `outlet_id` column and its foreign key are dropped, and everything else, including the rack column and every order, stays. The same command therefore does very different things on two computers, which is why `migrate:status` should always come first.

2. **Listing 9.17** adds the column as nullable, directly after `received_at`, and fills it in a second step. `lazyById()` reads the table in chunks of 1,000 rows, ordered by id, so even a table of a million orders never has to fit in memory at once; each row is updated with its own value. The date arithmetic uses PHP's `strtotime` on the stored text instead of Carbon and the `Order` model, following the rule that migrations use only the query builder. `down()` drops the column; no foreign key is involved.

Listing 9.17 `2026_10_01_000001_add_promised_at_to_orders_table.php`: a new column filled row by row, in chunks.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Schema;

// Exercise 9.2: when the laundry was promised to the customer.
return new class extends Migration
{
    public function up(): void
    {
        Schema::table('orders', function (Blueprint $table) {
            $table->timestamp('promised_at')->nullable()->after('received_at');
        });

        // Old orders: the usual promise, two days after they came in.
        DB::table('orders')->whereNull('promised_at')->lazyById()
            ->each(function (object $order) {
                DB::table('orders')->where('id', $order->id)->update([
                    'promised_at' => date('Y-m-d H:i:s',
                        strtotime($order->received_at . ' +2 days')),
                ]);
            });
    }

    public function down(): void
    {
        Schema::table('orders', function (Blueprint $table) {
            $table->dropColumn('promised_at');
        });
    }
};
```

3. Listing 9.18 is the middleware. The parameter arrives as the extra argument `$limitMs`, as a string, after `$next`; the default “500” is used when the route gives none. It is an “after” middleware: it measures around `$next` and only then decides whether to write to the log with `Log::warning`, whose entries appear in `storage/logs/laravel.log`.

Listing 9.18 `app/Http/Middleware/LogSlowRequests.php`: a route middleware with a parameter.

```php
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Log;
use Symfony\Component\HttpFoundation\Response;

// Exercise 9.3: route middleware with a parameter, e.g. 'slow:300'.
class LogSlowRequests
{
    public function handle(Request $request, Closure $next,
        string $limitMs = '500'): Response
    {
        $start = hrtime(true);
        $response = $next($request);
        $ms = (hrtime(true) - $start) / 1e6;

        if ($ms > (float) $limitMs) {
            Log::warning('Slow request', [
                'url' => $request->fullUrl(),
                'ms' => round($ms, 1),
                'limit' => (int) $limitMs,
            ]);
        }

        return $response;
    }
}
```

Register it in `bootstrap/app.php` inside `withMiddleware` by adding `'slow' => LogSlowRequests::class` to the array given to `alias`, with `use App\Http\Middleware\LogSlowRequests;` at the top of the file. Then attach it in `routes/web.php` by adding `->middleware('slow:300')` to the definition of the order list; because the list is defined by `Route::resource`, write `Route::resource('orders', OrderController::class)->only([...])->middleware('slow:300')`, which applies it to the four resource routes, or define the index route separately if only the list should be watched.

4. `prepareForValidation` trims and upper-cases the text, so “b-3” becomes “B-3”. The rule `regex:/^[A-Z]-[0-9]{2}$/` requires exactly two digits after the dash, so “B-3” fails, and the cashier sees the custom message “A rack looks like B-03: a letter, a dash, two digits.” “B-03” passes: `trim` removes the space and the result “B-03” matches.

5. `PriceListSeeder` uses `updateOrCreate` keyed on the code, so the second run finds every service and outlet and updates them to the same values: nothing changes, nothing fails. `DemoSeeder` uses `Customer::create` with fixed phone numbers; the second run tries to insert Sari’s phone number again and stops with a unique-constraint error, so the run fails at that point. `BusyMonthSeeder` would add forty more customers and their orders (Faker’s `unique()` only remembers numbers within one run, so a clash with the first run’s numbers is possible but unlikely). To make repeated runs work, change `DemoSeeder` to `Customer::firstOrCreate(['phone' => ...], [...])` and skip creating the orders when the customer already existed, and accept that `BusyMonthSeeder` adds data each time, or guard it with a check such as `if (Customer::count() > 10) return;`.

6. With `$orderOpen` outermost, the departed order stops in the outer layer, which never calls `$next`; the timing layer is inside it and never runs. The program prints “order.open: departed, stop here” and the answer `{"status":409}` without a Server-Timing value. In the real application

this means that refused requests are not measured at all, and the time spent by the outer layers is never part of the measurement. A layer that measures or logs should be as far out as possible, which is why `AddServerTiming` joins the `web` group, and a layer that refuses should be as far in as it needs to be to have the information it checks, here after routing.

References

[1] Laravel, "Release Notes," *Laravel 13.x Documentation*, Mar. 17, 2026. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/releases

[2] Laravel, "Database: Migrations," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/migrations

[3] P. Sadalage and M. Fowler, "Evolutionary Database Design," martinfowler.com, May 2016. Accessed: Sep. 25, 2026. [Online]. Available: https://martinfowler.com/articles/evodb.html

[4] FakerPHP, "FakerPHP / Faker documentation." Accessed: Sep. 25, 2026. [Online]. Available: https://fakerphp.org/

[5] Laravel, "Eloquent: Factories," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/eloquent-factories

[6] Laravel, "Database: Seeding," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/seeding

[7] Laravel, "Middleware," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/middleware

[8] Laravel, "Validation," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/validation

[9] Laravel, "Routing: Route Model Binding," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/routing#route-model-binding

[10] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[11] MDN Web Docs, "Server-Timing header," Mozilla. Accessed: Sep. 25, 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Server-Timing

[12] The PHP Group, "PHP 8.5 Release Announcement," php.net, Nov. 20, 2025. Accessed: Sep. 25, 2026. [Online]. Available: https://www.php.net/releases/8.5/en.php

[13] OWASP Foundation, "Input Validation Cheat Sheet," *OWASP Cheat Sheet Series*. Accessed: Sep. 25, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

Chapter 10

Laravel III: Authentication, API Resources and Testing

At the end of Chapter 9 the Profitina Laundry application knew its outlets, filled itself with a believable month of orders and refused to change an order that had left the shop. It still had three gaps. Anyone who could reach the counter computer could open every page, including the price list. Only people with a browser could use it, while the drivers who pick up and deliver laundry wanted to record their work on a phone. And every check that the application worked was done by hand, by clicking. This chapter closes the three gaps. By the end of it you will be able to explain the difference between authentication and authorization; add sign-in and sign-out for staff with Laravel's session guard, and give the owner rights that a cashier does not have; issue, use and revoke API tokens with Laravel Sanctum so that other programs can use the application safely; shape every JSON answer with API resources instead of sending database rows as they are; and write unit and feature tests that check the price rule, the sign-in, the permissions and the API automatically, in seconds, every time something changes.

10.1 A Price List Anyone Could Change

One Saturday afternoon in September, the Keputih outlet charged a customer Rp 6.000 per kilogram for "Wash, dry & fold" instead of Rp 8.000. Nobody had meant to change the price. A new part-time cashier had opened the **Prices** page to *look* at the prices, as the owner had told him to, clicked **Edit** on the wrong line and saved. The application of Chapter 9 did not ask who he was, so it did not know that he should not be able to do that, and it could not say afterwards who had done it.

The same week Bu Wati hired two drivers for the pickup and delivery service that Chapter 3 priced at Rp 10.000. The drivers are on the road all day. They need to see the orders to collect, mark a bag as delivered and record a payment made at the customer's door, and they will do it with a small phone app that a nephew of Dimas is writing. That app is not a browser: it cannot fill in the Blade forms or click the buttons of Chapter 7, and it would be dangerous to let it try. It needs an *API*, a set of URLs that accept and return JSON, and a way to prove that the phone belongs to a driver of the shop.

And there is Dimas. Every change he makes, whether a new rule, a new column or a new page, he checks by clicking through the application: creating an order, paying it, advancing it. It takes twenty minutes, so he does it less and less often, and in Chapter 9 a change to the payment form would have broken the pickup rule if Bu Wati had not noticed a wrong total the next morning.

Laravel has a standard answer for each problem [1]. *Authentication*, finding out who is making a request, comes built in: a guard that checks an e-mail address and a password, remembers the person in a session and lets middleware turn guests away [2]. *Authorization*, deciding what that person may do, is done by *gates* and *policies* [3]. For programs, *Laravel Sanctum* issues API tokens, long random strings that a phone keeps and sends with every request [4]. *API resources* decide exactly what each JSON answer contains [5]. And Laravel's testing tools, built on PHPUnit,

send pretend requests through the whole application and check the answers without a browser [6].

10.2 The Idea in Plain Words

10.2.1 Who are you, and what may you do?

Two questions are easy to confuse, and much of security depends on keeping them apart. *Authentication* asks "who are you?", and the answer is a person: Dimas, Bu Wati, a driver. *Authorization* asks "may you do this?", and the answer is yes or no for one action on one thing: may Dimas edit the price of S0001? The first question is answered once, when someone signs in; the second is asked again and again, for every action. HTTP has a separate status code for each failure: *401 Unauthorized* means "I do not know who you are" (despite its name, it is about authentication), and *403 Forbidden* means "I know who you are, and you may not do this" [7].

Profitina Laundry needs only two roles for now. The *owner* may do everything, including changing prices. A *cashier* may register customers, take orders, record payments and move orders along, but may not touch the price list. The drivers will sign in as cashiers from their phones.

10.2.2 Sessions for browsers, tokens for programs

A browser and a phone app prove who they are in different ways, shown side by side in **Figure 10.1**.

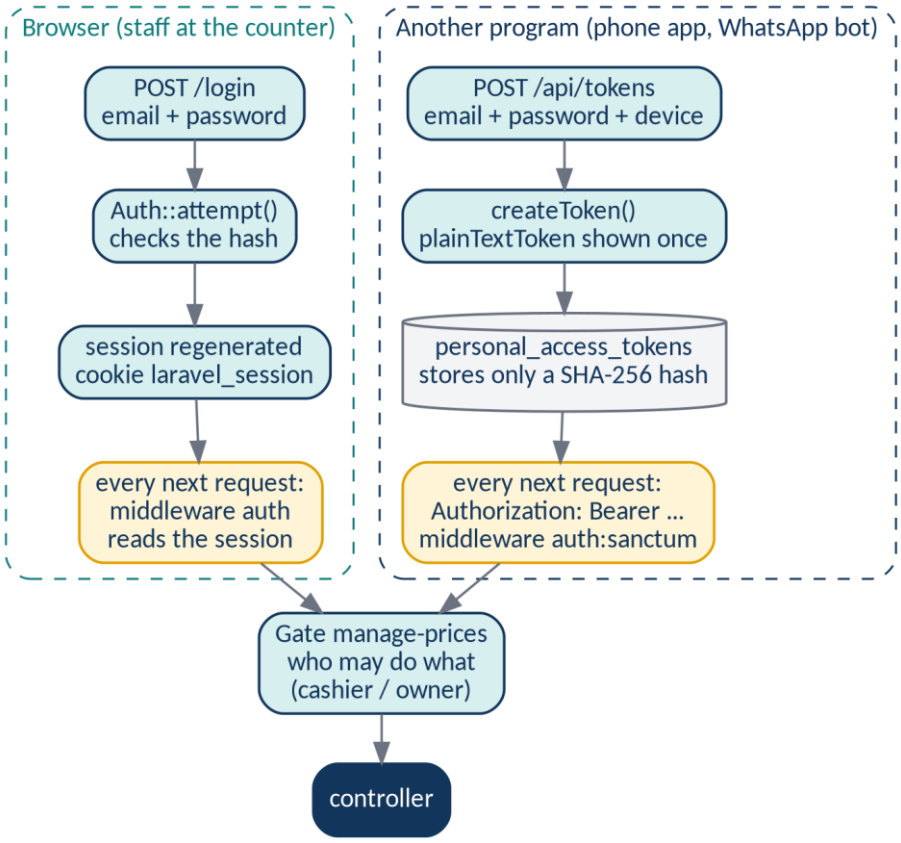


Figure 10.1 Two ways to prove who is asking: a browser signs in once and carries a session cookie; a program exchanges the password for a token and sends it with every request; both end at the same authorization check.

A browser signs in once, with a form. Laravel checks the password against the stored hash, remembers in the *session*, on the server, that this visitor is Dimas, and gives the browser a cookie that names the session. Every later request carries the cookie automatically, and the `auth` middleware finds Dimas again from it. This is comfortable for people, and Chapter 6 did the same in ASP.NET Core with Identity's cookie.

A program cannot be trusted with a cookie that a browser manages for it, and it should not keep the driver's password either. Instead it sends the e-mail address and the password once, to a special URL, and receives a *token*: a long random string. The server stores only a SHA-256 hash of the token in the table `personal_access_tokens`, so even someone who steals a copy of the database cannot use the tokens in it [4]. The program sends the token in an `Authorization: Bearer ...` header with every request, the standard way of carrying such tokens in HTTP [8], and the `auth:sanctum` middleware finds the user from it. A token belongs to one device: when a driver loses her phone, the owner deletes that one token and the phone is locked out, while the driver's password and her other devices keep working.

10.2.3 Resources: deciding what an answer contains

Returning an Eloquent model from a controller does produce JSON, because models know how to turn themselves into arrays. It is still a mistake for an API. The JSON then contains every column, including ones a phone has no business seeing, in the database's own shape: `quantity` as the text `"4.50"`, the status as whatever the database stores, foreign keys instead of names. And the day a column is renamed, every app that reads the JSON breaks.

An *API resource* is a class that states the answer explicitly [5]. Its `toArray` method lists the fields to send, their names, and how to compute the ones that are not columns, such as `paid` and `balance`. It can nest other resources, so an order carries its customer's name and phone, and include a relation only when the controller has loaded it. The database can change behind the resource while the JSON stays the same.

10.2.4 Tests: a tireless second pair of hands

A *test* is a small program that uses the application the way a person would and checks the result. A *unit test* checks one class alone, such as `PriceCalculator`, with no database and no HTTP; it runs in milliseconds. A *feature test* sends a pretend request through the whole application, routing, middleware, form requests, controllers and a real database, and checks the response and the database afterwards [6]. Laravel gives each feature test a fresh, empty database: the `RefreshDatabase` trait runs the migrations once and wraps each test in a transaction that is rolled back at the end, so tests never see each other's data [9]. What took Dimas twenty minutes of clicking takes a few seconds, and it is done the same way every time.

10.3 Users and Roles

A new Laravel project already has a `users` table, a `User` model and a `UserFactory`. This chapter gives each user a role. **Listing 10.1** lists the two roles as a backed enum, like `OrderStatus` in Chapter 7.

Listing 10.1 `app/Enums/Role.php`: the two roles of the shop.

```php
<?php

namespace App\Enums;

enum Role: string
{
    case Owner = 'owner';        // Bu Wati: everything, including prices
    case Cashier = 'cashier';    // the counter staff: orders and payments
}
```

Listing 10.2 adds the column. Every existing user becomes a cashier by default, which is the safe choice: an account that should have more rights is easy to promote, while an account that has too many rights may do damage before anyone notices.

Listing 10.2 `2026_10_04_000001_add_role_to_users_table.php`: every user gets a role, cashier unless told otherwise.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('role', 10)->default('cashier')->after('email');
        });
    }

    public function down(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('role');
        });
    }
};
```

Listing 10.3 is the `User` model. It uses the same `#[Fillable]` attribute as the other models of the book, and `#[Hidden]` to keep the password hash and the "remember me" token out of every array and JSON made from a user. Three casts do important work. `'password' => 'hashed'` hashes any plain password assigned to the model before it is saved, with the algorithm configured in `config/hashing.php`, bcrypt unless changed [10]. `'role' => Role::class` turns the stored text into the enum. And the `HasApiTokens` trait from Sanctum gives each user the `createToken` method and the `tokens` relation used in Section 10.5.

Listing 10.3 `app/Models/User.php`: a user with a role, a hashed password and API tokens.

```php
<?php

namespace App\Models;

use App\Enums\Role;
use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Attributes\Hidden;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Sanctum\HasApiTokens;

#[Fillable(['name', 'email', 'password', 'role'])]
#[Hidden(['password', 'remember_token'])]
class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;

    protected function casts(): array
    {
        return [
            'email_verified_at' => 'datetime',
            'password' => 'hashed',          // stored as a hash, never plain
            'role' => Role::class,
        ];
    }

    public function isOwner(): bool
    {
        return $this->role === Role::Owner;
    }
}
```

The `UserFactory` of the new project gets a role and an `owner()` state, in **Listing 10.4**. The plain password “password” is hashed by the model’s cast, so every factory user can sign in with it in a test.

Listing 10.4 `database/factories/UserFactory.php`: cashiers by default, and an `owner()` state.

```php
<?php

namespace Database\Factories;

use App\Enums\Role;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

/**
 * @extends Factory<\App\Models\User>
 */
class UserFactory extends Factory
{
    public function definition(): array
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' => 'password',           // hashed by the model's cast
            'role' => Role::Cashier,
            'remember_token' => Str::random(10),
        ];
    }

    public function owner(): static
    {
        return $this->state(fn (array $attributes) => ['role' => Role::Owner]);
    }
}
```

Finally, `DatabaseSeeder` creates one owner and one cashier on developers' computers, next to the demo data of Chapter 9 (**Listing 10.5**). These two accounts have well-known passwords and exist only for trying the application; the shop's real accounts are created on the shop's computer, one per person, with passwords that nobody else knows.

Listing 10.5 `database/seeders/DatabaseSeeder.php`: two demo accounts, on developers' computers only.

```php
<?php

namespace Database\Seeders;

use App\Models\User;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    public function run(): void
    {
        $this->call(PriceListSeeder::class);      // real data: always

        if (app()->environment('local', 'testing')) {
            User::factory()->owner()->create(['name' => 'Bu Wati',
                'email' => 'owner@laundry.test']);
            User::factory()->create(['name' => 'Dimas',
                'email' => 'cashier@laundry.test']);
            $this->call([DemoSeeder::class, BusyMonthSeeder::class]);
        }
    }
}
```

10.4 Signing In to the Staff Pages

10.4.1 The sign-in controller

Laravel offers ready-made starter kits that generate complete sign-in, registration and password-reset screens [2]. They are excellent for a new product, but they also generate a great deal of code at once, and a shop whose staff are created by the owner needs no public registration. This chapter writes the three actions by hand instead, in **Listing 10.6**, which also shows exactly what a starter kit does inside.

Listing 10.6 `app/Http/Controllers/Auth/LoginController.php`: show the form, sign in, sign out.

```php
<?php

namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Validation\ValidationException;

class LoginController extends Controller
{
    public function create()
    {
        return view('auth.login');
    }

    public function store(Request $request)
    {
        $credentials = $request->validate([
            'email' => ['required', 'email'],
            'password' => ['required', 'string'],
        ]);

        if (! Auth::attempt($credentials, $request->boolean('remember'))) {
            throw ValidationException::withMessages([
                'email' => 'These credentials do not match our records.',
            ]);
        }

        $request->session()->regenerate();      // a new session id after login

        return redirect()->intended(route('counter'));
    }

    public function destroy(Request $request)
    {
        Auth::logout();
        $request->session()->invalidate();
        $request->session()->regenerateToken();

        return redirect()->route('home');
    }
}
```

`store` first validates the two fields, then calls `Auth::attempt`. That one call finds the user by e-mail, checks the typed password against the stored hash with a comparison that takes the same time whether the password is nearly right or completely wrong, and, if it matches, remembers the user in the session; with `true` as the second argument it also sets a long-lived “remember me” cookie. When the attempt fails, the method throws a `ValidationException` with a message on the

`email` field, so the form comes back exactly as for any other validation error. The message deliberately does not say whether the e-mail or the password was wrong: telling a stranger "this e-mail exists" helps them guess [11].

After a successful sign-in, `$request->session()->regenerate()` gives the session a new identifier. This defends against *session fixation*, an attack in which someone plants a session id in a victim's browser before the victim signs in, and then uses the same id afterwards. `redirect()->intended(...)` returns the user to the page they originally wanted, if the `auth` middleware sent them to sign in, or to the counter otherwise. `destroy` does the reverse: it forgets the user, throws the whole session away and makes a new token for protecting forms, so nothing of the old session can be reused. The view `auth/login.blade.php` in the chapter's folder is an ordinary Blade form with the two fields, a "remember me" box and `@csrf`.

10.4.2 Guarding the routes

Listing 10.7 is the routes file of this chapter. It sorts the routes into four groups: public pages that anyone may see, the sign-in routes, which only guests may use, the staff pages behind `auth`, and inside them the price list behind the extra rule `can:manage-prices`.

Listing 10.7 `routes/web.php`: public pages, guest-only sign-in, staff pages behind `auth`, and the price list for the owner.

```php
<?php

use App\Http\Controllers\Auth\LoginController;
use App\Http\Controllers\CounterController;
use App\Http\Controllers\CustomerController;
use App\Http\Controllers\HomeController;
use App\Http\Controllers\OrderController;
use App\Http\Controllers\OutletController;
use App\Http\Controllers\PaymentController;
use App\Http\Controllers\PriceController;
use App\Http\Controllers\ReportController;
use App\Http\Controllers\ServiceController;
use Illuminate\Support\Facades\Route;

// The public pages: anyone may see them.
Route::get('/', HomeController::class)->name('home');
Route::get('services', [ServiceController::class, 'index'])
    ->name('services.index');
Route::get('services/{service}', [ServiceController::class, 'show'])
    ->name('services.show');
Route::get('outlets', [OutletController::class, 'index'])
    ->name('outlets.index');

// Signing in and out.
Route::middleware('guest')->group(function () {
    Route::get('login', [LoginController::class, 'create'])->name('login');
    Route::post('login', [LoginController::class, 'store'])
        ->middleware('throttle:5,1');           // five tries a minute
});
Route::post('logout', [LoginController::class, 'destroy'])
    ->middleware('auth')->name('logout');

// The staff pages: only for signed-in staff.
Route::middleware('auth')->group(function () {
    Route::get('counter', CounterController::class)->name('counter');
    Route::resource('customers', CustomerController::class);
    Route::resource('orders', OrderController::class)
        ->only(['index', 'create', 'store', 'show']);
    Route::get('reports/daily', [ReportController::class, 'daily'])
        ->name('reports.daily');
```

Listing 10.7 (Continued) *routes/web.php: public pages, guest-only sign-in, staff pages behind auth, and the price list for the owner.*

```php
    Route::middleware('order.open')->group(function () {
        Route::post('orders/{order}/advance',
            [OrderController::class, 'advance'])->name('orders.advance');
        Route::post('orders/{order}/payments',
            [PaymentController::class, 'store'])
            ->name('orders.payments.store');
    });

    // The price list: the owner only.
    Route::resource('prices', PriceController::class)
        ->parameters(['prices' => 'service'])
        ->except(['show', 'destroy'])
        ->middleware('can:manage-prices');
});
```

The `guest` middleware sends a user who is already signed in away from the sign-in form. `throttle:5,1` on the POST route allows five sign-in attempts per minute from one user or address, and answers the sixth with 429 Too Many Requests; that turns guessing a password from thousands of tries a minute into five. Sign-out is a POST route, never a GET: a link that signs people out could be triggered by any web page that embeds it as an image.

The `auth` middleware needs to know where to send guests. **Listing 10.8** tells it with `redirectGuestsTo('/login')`, and also adds the API routes file that Section 10.5 writes, with the same timing middleware as the web pages.

Listing 10.8 `bootstrap/app.php`: the API routes, timing for both groups, and the sign-in page for guests.

```php
<?php

use App\Http\Middleware\AddServerTiming;
use App\Http\Middleware\EnsureOrderIsOpen;
use Illuminate\Foundation\Application;
use Illuminate\Foundation\Configuration\Exceptions;
use Illuminate\Foundation\Configuration\Middleware;

return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        api: __DIR__.'/../routes/api.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )
    ->withMiddleware(function (Middleware $middleware): void {
        $middleware->web(append: [AddServerTiming::class]);
        $middleware->api(append: [AddServerTiming::class]);
        $middleware->alias([
            'order.open' => EnsureOrderIsOpen::class,
        ]);
        $middleware->redirectGuestsTo('/login');
    })
    ->withExceptions(function (Exceptions $exceptions): void {
        //
    })->create();
```

10.4.3 A gate for the price list

`can:manage-prices` asks a *gate* named `manage-prices`. Gates are defined once, usually in the `boot` method of `AppServiceProvider`, as a function that receives the signed-in user and returns true or false [3]. **Listing 10.9** defines ours: only the owner may manage prices.

Listing 10.9 `app/Providers/AppServiceProvider.php`: one gate, `manage-prices`, for the owner only.

```php
<?php

namespace App\Providers;

use App\Models\User;
use Illuminate\Support\Facades\Gate;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        //
    }

    public function boot(): void
    {
        // Only the owner changes the price list.
        Gate::define('manage-prices', fn (User $user) => $user->isOwner());
    }
}
```

The same gate is used in three places, and that is its strength. In the routes, `->middleware('can:manage-prices')` answers 403 to a cashier who types the URL of the price list by hand. In Blade, `@can('manage-prices') ... @endcan` hides the **Prices** link from cashiers, and the chapter's `layouts/app.blade.php` does exactly that, along with a **Sign out** button showing the user's name inside an `@auth` block. And in any PHP code, `Gate::allows('manage-prices')` or `$user->can('manage-prices')` asks the same question. Hiding the link is only courtesy; the route middleware is the lock. A rule that applies to one kind of model, such as "only the owner may delete an order that has not started", fits a *policy* class better: a class with one method per action, which Laravel finds by the model's name [3]. Exercise 10.4 asks where each kind belongs.

Table 10.1 collects the answers that the staff pages and the API of this chapter can give when something is refused, and which part of the application gives each.

Table 10.1 The refusals of this chapter's application: status code, meaning and the part that answers.

Status	Meaning	Who answers	Example
302 to `/login`	a guest asked for a staff page	`auth` middleware (browser)	opening `/counter` signed out
401 Unauthorized	no valid session or token	`auth` / `auth:sanctum` (JSON)	API call without a token
403 Forbidden	signed in, but not allowed	`can:` middleware, a gate or a form request	a cashier opening `/prices`
409 Conflict	the order is closed	`order.open` (Chapter 9)	paying a departed order by API
419 Page Expired	the form's CSRF token is missing or old	request-forgery protection	a form left open overnight
422 Unprocessable	the input breaks a rule	validation, form requests	paying more than the balance
429 Too Many Requests	too many attempts	`throttle`	the sixth sign-in in a minute

10.5 An API for Programs

10.5.1 Installing Sanctum

A new Laravel 13 project has no API routes file. One command adds everything an API needs: `php artisan install:api` installs Laravel Sanctum with Composer, publishes its migration for the `personal_access_tokens` table, creates `routes/api.php` and registers it [4]. Our `User` model already uses the `HasApiTokens` trait that the command asks for, and our `bootstrap/app.php` already lists the `api` routes file. Every route in that file gets the prefix `/api` and the `api` middleware group, which has no sessions and no cookies: API requests are stateless and must prove themselves on every call.

10.5.2 Issuing and revoking tokens

Listing 10.10 is the whole token counter: one method to issue a token and one to revoke the token that the current request used.

Listing 10.10 `app/Http/Controllers/Api/TokenController.php`: a password in, a token out; and a way to give it back.

```php
<?php

namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Validation\ValidationException;

class TokenController extends Controller
{
    // POST /api/tokens: e-mail + password in, a token out (shown once).
    public function store(Request $request)
    {
        $data = $request->validate([
            'email' => ['required', 'email'],
            'password' => ['required', 'string'],
            'device' => ['required', 'string', 'max:60'],
        ]);

        $user = User::where('email', $data['email'])->first();
        if (! $user || ! Hash::check($data['password'], $user->password)) {
            throw ValidationException::withMessages([
                'email' => 'These credentials do not match our records.',
            ]);
        }

        return response()->json([
            'token' => $user->createToken($data['device'])->plainTextToken,
            'user' => ['name' => $user->name, 'role' => $user->role->value],
        ], 201);
    }

    // DELETE /api/tokens/current: sign this device out.
    public function destroy(Request $request)
    {
        $request->user()->currentAccessToken()->delete();

        return response()->noContent();
    }
}
```

The password check is the same as in the sign-in form, but done explicitly with `Hash::check`, because no session is involved. `createToken($data['device'])` makes a new random token, stores its hash with the device name the app sent, such as “driver-phone-rina”, and returns an object whose `plainTextToken` is the only copy of the token in plain text that will ever exist. The API sends it back once, with status 201 Created; after that, not even the server can show it again, because it only has the hash. `destroy` deletes the token that the current request was made with: `currentAccessToken()` is the token Sanctum matched. The app calls it when the driver signs out; the owner can remove a lost phone’s token from the database, or delete all of a user’s tokens with `$user->tokens()->delete()`.

10.5.3 The API routes

Listing 10.11 lists the API. The token route is public, with the same throttle as the sign-in form; everything else needs `auth:sanctum`, and the two routes that change an order are guarded by Chapter 9’s `order.open` exactly as the web routes are.

Listing 10.11 `routes/api.php`: tokens, customers, orders and payments, behind `auth:sanctum`.

```php
<?php

use App\Http\Controllers\Api\CustomerController;
use App\Http\Controllers\Api\OrderController;
use App\Http\Controllers\Api\PaymentController;
use App\Http\Controllers\Api\TokenController;
use Illuminate\Support\Facades\Route;

// Every URL here starts with /api (Laravel adds the prefix).
Route::post('tokens', [TokenController::class, 'store'])
    ->middleware('throttle:5,1');

Route::middleware('auth:sanctum')->group(function () {
    Route::delete('tokens/current', [TokenController::class, 'destroy']);

    Route::get('customers', [CustomerController::class, 'index']);
    Route::post('customers', [CustomerController::class, 'store']);

    Route::get('orders', [OrderController::class, 'index']);
    Route::post('orders', [OrderController::class, 'store']);
    Route::get('orders/{order}', [OrderController::class, 'show']);

    Route::middleware('order.open')->group(function () {
        Route::patch('orders/{order}/status',
            [OrderController::class, 'updateStatus']);
        Route::post('orders/{order}/payments',
            [PaymentController::class, 'store']);
    });
});
```

Because API requests send `Accept: application/json`, every refusal in **Table 10.1** arrives as JSON: `{"message": "Unauthenticated."}` with 401, `{"message": "..."}` with 409 from our middleware, and for validation errors the 422 body with a `message` and an `errors` object that lists the messages per field. A phone app can therefore show the exact message under the exact field, just as the Blade forms do.

10.5.4 One way to create an order

The web counter and the API both create orders. If each controller had its own copy of "find the service, compute the total, save as queued", the two copies would drift apart the first time one of them is changed. **Listing 10.12** moves that work into an *action* class, a plain class with one public method, which both controllers receive from Laravel's service container.

Listing 10.12 `app/Actions/CreateOrder.php`: the one place where checked input becomes a priced order.

```php
<?php

namespace App\Actions;

use App\Domain\PriceCalculator;
use App\Enums\OrderStatus;
use App\Models\Order;
use App\Models\Service;

// One place that turns checked input into a priced, saved order.
// The web page and the API both call it, so they can never disagree.
class CreateOrder
{
    public function handle(array $data): Order
    {
        $service = Service::findOrFail($data['service_id']);
        $pickup = (bool) ($data['pickup'] ?? false);

        return Order::create([
            ...$data,
            'pickup' => $pickup,
            'total' => PriceCalculator::total($service->per_kg,
                $service->price, (string) $data['quantity'], $pickup),
            'status' => OrderStatus::Queued,
            'received_at' => now(),
        ]);
    }
}
```

The web `OrderController` of this chapter now calls `$createOrder->handle($request->validated())` in its `store` method instead of pricing the order itself; the rest of the controller is unchanged from Chapter 9. **Listing 10.13** is the API controller for orders.

Listing 10.13 `app/Http/Controllers/Api/OrderController.php`: list, create, show and move orders, answering with resources.

```php
<?php

namespace App\Http\Controllers\Api;

use App\Actions\CreateOrder;
use App\Enums\OrderStatus;
use App\Http\Controllers\Controller;
use App\Http\Requests\StoreOrderRequest;
use App\Http\Requests\UpdateOrderStatusRequest;
use App\Http\Resources\OrderResource;
use App\Models\Order;
use Illuminate\Http\Request;
use Illuminate\Validation\Rule;

class OrderController extends Controller
{
    // GET /api/orders?status=finished&page=2
    public function index(Request $request)
    {
        $request->validate([
            'status' => ['nullable', Rule::enum(OrderStatus::class)],
        ]);

        $orders = Order::with(['customer', 'service'])
            ->withSum('payments', 'amount')
            ->when($request->query('status'),
                fn ($q, $status) => $q->where('status', $status))
            ->orderByDesc('received_at')->orderByDesc('id')
            ->paginate(20)
            ->withQueryString();

        return OrderResource::collection($orders);
    }

    public function store(StoreOrderRequest $request, CreateOrder $createOrder)
    {
        $order = $createOrder->handle($request->validated());

        return (new OrderResource($order->load(['customer', 'service'])))
            ->response()->setStatusCode(201);
    }
```

Listing 10.13 (Continued) *app/Http/Controllers/Api/OrderController.php: list, create, show and move orders, answering with resources.*

```
    public function show(Order $order)
    {
        return new OrderResource($order->load(['customer', 'service',
            'payments' => fn ($q) => $q->orderBy('paid_at')]));
    }

    // PATCH /api/orders/5/status   {"status": "finished"}
    public function updateStatus(UpdateOrderStatusRequest $request,
        Order $order)
    {
        $next = OrderStatus::from($request->validated('status'));
        $order->status = $next;
        if ($next === OrderStatus::Finished) {
            $order->finished_at = now();
        } else {
            $order->departed_at = now();
        }
        $order->save();

        return new OrderResource($order->load(['customer', 'service']));
    }
}
```

`index` validates its one query parameter, then builds the same kind of query as the web list of Chapter 8: customers and services eager-loaded, the sum of payments added by `withSum`, an optional status filter, twenty a page. `->withQueryString()` keeps `?status=finished` in the links to the next pages. `OrderResource::collection($orders)` turns the page into JSON with three parts: `data`, the orders; `links`, the URLs of the first, previous, next and last pages; and `meta`, with numbers such as `current_page`, `per_page` and `total` [5]. `store` reuses Chapter 9's `StoreOrderRequest`, so the web and the API accept exactly the same input, and answers 201 Created. `show` loads the payments too, in time order.

`updateStatus` is the API's way of moving an order along. A browser presses a button that says "the next step, whatever it is"; a program should say what it believes the next step is, `{"status": "finished"}`, so that two drivers pressing at the same moment cannot move an order two steps. **Listing 10.14** checks that request with Chapter 8's `OrderWorkflow`.

Listing 10.14 `app/Http/Requests/UpdateOrderStatusRequest.php`: the wanted status must be the one the workflow allows next.

```php
<?php

namespace App\Http\Requests;

use App\Domain\OrderWorkflow;
use App\Enums\OrderStatus;
use DomainException;
use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rule;
use Illuminate\Validation\Validator;

class UpdateOrderStatusRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return ['status' => ['required', Rule::enum(OrderStatus::class)]];
    }

    // The workflow of Chapter 8 decides which step is allowed.
    public function after(): array
    {
        return [
            function (Validator $validator) {
                if ($validator->errors()->isNotEmpty()) {
                    return;
                }
                $order = $this->route('order')->load('payments');
                try {
                    $next = OrderWorkflow::advance($order->status,
                        $order->balance());
                } catch (DomainException $e) {
                    $validator->errors()->add('status', $e->getMessage());
                    return;
                }
                if ($this->input('status') !== $next->value) {
                    $validator->errors()->add('status', "An order that is "
                        . "{$order->status->value} can only become "
                        . "{$next->value}.");
                }
            },
        ];
    }
}
```

Listing 10.14 (Continued) *app/Http/Requests/UpdateOrderStatusRequest.php: the wanted status must be the one the workflow allows next.*

The rule `Rule::enum(OrderStatus::class)` rejects words that are not statuses at all. The `after` hook then asks the workflow for the next status, given the order's balance. If the workflow refuses, because the laundry would leave unpaid, its own message becomes the error: "Collect Rp 34.000 before handing over the laundry." If the workflow allows a step but not the one requested, the error says which step is allowed. A departed order never reaches this request, because `order.open` stops it first with 409.

The payment API (**Listing 10.15**) reuses Chapter 9's `StorePaymentRequest`, with its check against the balance, and answers with the whole order, payments included, so the app can show the new balance at once.

Listing 10.15 `app/Http/Controllers/Api/PaymentController.php`: record a payment, answer with the updated order.

```php
<?php

namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use App\Http\Requests\StorePaymentRequest;
use App\Http\Resources\OrderResource;
use App\Models\Order;

class PaymentController extends Controller
{
    // POST /api/orders/5/payments   {"amount": 34000, "method": "qris"}
    public function store(StorePaymentRequest $request, Order $order)
    {
        $order->payments()->create([...$request->validated(),
            'paid_at' => now()]);

        return (new OrderResource($order->load(['customer', 'service',
            'payments' => fn ($q) => $q->orderBy('paid_at')])))
            ->response()->setStatusCode(201);
    }
}
```

Drivers also meet new customers at their doors. **Listing 10.16** checks a new customer: the phone is reduced to its digits first, so “0812-3456-7890” and “0812 3456 7890” are the same number, then it must look like an Indonesian mobile number and must not belong to another customer yet. The API’s `CustomerController`, in the chapter’s folder, has an `index` that looks customers up by phone and a `store` that uses this request and answers 201.

Listing 10.16 `app/Http/Requests/StoreCustomerRequest.php`: a new customer, with the phone reduced to digits and checked for duplicates.

```php
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class StoreCustomerRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;
    }

    // "0812-3456-7890" and "0812 3456 7890" are the same number.
    protected function prepareForValidation(): void
    {
        $this->merge([
            'phone' => preg_replace('/[^0-9]/', '', (string) $this->phone),
        ]);
    }

    public function rules(): array
    {
        return [
            'name' => ['required', 'string', 'min:2', 'max:80'],
            'phone' => ['required', 'regex:/^08[0-9]{8,11}$/',
                'unique:customers,phone'],
            'address' => ['nullable', 'string', 'max:200'],
        ];
    }

    public function messages(): array
    {
        return ['phone.regex' => 'Write the phone as 08..., 10 to 13 digits.'];
    }
}
```

10.6 API Resources

Figure 10.2 shows what `OrderResource` does with an order: some fields pass through unchanged, some are converted, some are computed, relations become nested objects when they were loaded, and the rest never leaves the server.

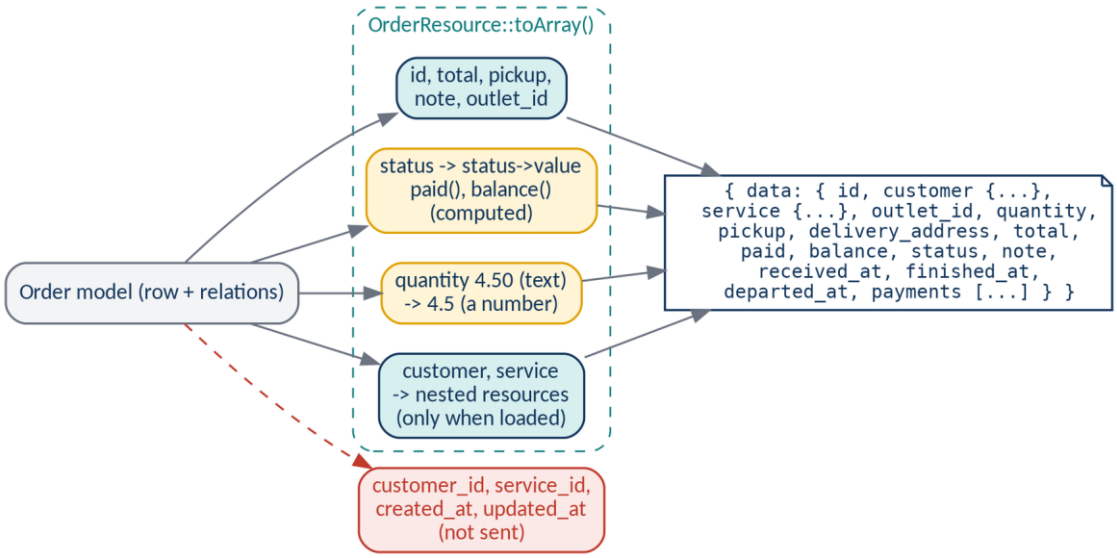


Figure 10.2 What `OrderResource` does with an order: fields passed through, converted or computed, relations nested when loaded, and internal columns left out.

Listing 10.17 is the class, made with `php artisan make:resource OrderResource`.

Listing 10.17 `app/Http/Resources/OrderResource.php`: the JSON shape of an order.

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class OrderResource extends JsonResource
{
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'customer' => new CustomerResource($this->whenLoaded('customer')),
            'service' => new ServiceResource($this->whenLoaded('service')),
            'outlet_id' => $this->outlet_id,
            'quantity' => (float) $this->quantity,
            'pickup' => $this->pickup,
            'delivery_address' => $this->delivery_address,
            'total' => $this->total,
            'paid' => $this->paid(),
            'balance' => $this->balance(),
            'status' => $this->status->value,
            'note' => $this->note,
            'received_at' => $this->received_at,
            'finished_at' => $this->finished_at,
            'departed_at' => $this->departed_at,
            'payments' => PaymentResource::collection(
                $this->whenLoaded('payments')),
        ];
    }
}
```

Inside `toArray`, `$this` stands for the order, so `$this->total` reads the order's total [5]. `'quantity' => (float) $this->quantity` turns the text `"4.50"` that the `decimal:2` cast produces into the number `4.5`, which is what a program expects of a quantity. `paid()` and `balance()` come from the model; this chapter's `Order::paid()` uses the `payments_sum_amount` that `withSum` adds when the query provided it, and the loaded payments otherwise, so a list of

twenty orders costs one extra query, not twenty. `$this->status->value` sends the word `"finished"`, not the enum object. `whenLoaded('customer')` includes the customer only if the controller loaded it; otherwise the key is left out entirely, and no hidden query runs. `created_at`, `updated_at` and the raw foreign keys are simply not listed, so they are not sent.

The three small resources are just as plain. `CustomerResource` sends `id`, `name`, `phone` and `address`. `ServiceResource` (**Listing 10.18**) sends the code, the name and unit that the `Service` model of Chapter 7 derives from the description, and the price; the description itself stays inside.

Listing 10.18 `app/Http/Resources/ServiceResource.php`: a line of the price list, as a phone needs it.

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class ServiceResource extends JsonResource
{
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,            // worked out from description
            'unit' => $this->unit,            // kg, item or pair
            'price' => $this->price,
        ];
    }
}
```

A single resource is wrapped in a `data` key: the phone receives `{"data": {"id": 12, "customer": {...}, ...}}`. The wrapper leaves room for other top-level keys, such as the `links` and `meta` of a page, and it is the same for single orders and lists, which makes the app's code simpler. The times, `received_at` and the others, are sent in this chapter as Laravel serializes them by default, which Chapter 12 examines closely when the same API is built a second time in Spring Boot.

10.7 Testing

10.7.1 A unit test for the price rule

Listing 10.19 checks `PriceCalculator` alone. It extends PHPUnit's own `TestCase`, not Laravel's, because it needs nothing from the framework: no database, no configuration, no HTTP.

Listing 10.19 `tests/Unit/PriceCalculatorTest.php`: six price cases, one test method.

```php
<?php

namespace Tests\Unit;

use App\Domain\PriceCalculator;
use PHPUnit\Framework\Attributes\DataProvider;
use PHPUnit\Framework\TestCase;

// No framework, no database: the rule of Chapter 3, checked on its own.
class PriceCalculatorTest extends TestCase
{
    public static function cases(): array
    {
        return [
            'per kg' => [true, 8000, '4.5', false, 36000],
            'minimum 3 kg' => [true, 8000, '2.5', false, 24000],
            'minimum 3 kg, pickup' => [true, 8000, '2.5', true, 34000],
            'items counted whole' => [false, 35000, '1.5', false, 70000],
            'free pickup from 100k' => [false, 65000, '2', true, 130000],
            'two decimals' => [true, 15000, '3.35', false, 50250],
        ];
    }

    #[DataProvider('cases')]
    public function test_total(bool $perKg, int $price, string $quantity,
        bool $pickup, int $expected): void
    {
        $this->assertSame($expected,
            PriceCalculator::total($perKg, $price, $quantity, $pickup));
    }
}
```

The `#[DataProvider('cases')]` attribute runs `test_total` once for every row of `cases()`, and a failure names the row, such as “minimum 3 kg, pickup”, so you know at once which rule broke. The six rows cover the rules of Chapter 3: the price per kilogram, the three-kilogram minimum, the pickup fee, whole items, free pickup from Rp 100.000, and a weight with two decimals. The expected totals are not guesses: they were computed with Chapter 8’s `PriceCalculator` in plain PHP, and they agree with the ASP.NET Core `PriceCalculator` of Chapter 5, as Chapter 8’s check showed.

10.7.2 Feature tests for signing in

Listing 10.20 checks the sign-in, the permissions and the tokens through pretend HTTP requests.

Listing 10.20 `tests/Feature/AuthTest.php`: guests, sign-in, wrong passwords, the owner’s price list and API tokens.

```php
<?php

namespace Tests\Feature;

use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class AuthTest extends TestCase
{
    use RefreshDatabase;

    public function test_guests_are_sent_to_the_sign_in_page(): void
    {
        $this->get('/counter')->assertRedirect('/login');
    }

    public function test_the_public_pages_stay_public(): void
    {
        $this->get('/outlets')->assertOk();
    }

    public function test_staff_sign_in_and_reach_the_counter(): void
    {
        $user = User::factory()->create();

        $this->post('/login', ['email' => $user->email,
            'password' => 'password'])->assertRedirect('/counter');

        $this->assertAuthenticatedAs($user);
    }

    public function test_a_wrong_password_is_refused(): void
    {
        $user = User::factory()->create();

        $this->from('/login')->post('/login', ['email' => $user->email,
            'password' => 'wrong'])
            ->assertRedirect('/login')->assertSessionHasErrors('email');
        $this->assertGuest();
    }
```

Listing 10.20 (Continued) *tests/Feature/AuthTest.php: guests, sign-in, wrong passwords, the owner's price list and API tokens.*

```
    public function test_only_the_owner_sees_the_price_list(): void
    {
        $this->actingAs(User::factory()->create())
            ->get('/prices')->assertForbidden();
        $this->actingAs(User::factory()->owner()->create())
            ->get('/prices')->assertOk();
    }

    public function test_a_token_opens_the_api_and_can_be_revoked(): void
    {
        $user = User::factory()->create();

        $token = $this->postJson('/api/tokens', ['email' => $user->email,
            'password' => 'password', 'device' => 'counter-tablet'])
            ->assertCreated()->json('token');

        $this->withToken($token)->getJson('/api/orders')->assertOk();
        $this->withToken($token)->deleteJson('/api/tokens/current')
            ->assertNoContent();
        $this->assertDatabaseCount('personal_access_tokens', 0);
    }

    public function test_the_api_refuses_requests_without_a_token(): void
    {
        $this->getJson('/api/orders')->assertUnauthorized();
    }
}
```

Each method is one short story with an expected ending. `$this->get('/counter')` sends a GET request through the whole application without a network; `assertRedirect('/login')` checks the status and the `Location` header. `actingAs($user)` signs a user in for the next request without going through the form, which keeps the tests about permissions short. `from('/login')` sets the page the request pretends to come from, so that a failed validation redirects back there. `postJson` and `getJson` send JSON and ask for JSON, as a phone would; `withToken($token)` adds the `Authorization: Bearer` header. The last lines of `test_a_token_opens_the_api_and_can_be_revoked` check the database directly: after revoking, no token is left.

`RefreshDatabase` gives every test an empty database with all migrations applied [9]. A new Laravel project's `phpunit.xml` runs the tests on an in-memory SQLite database, so the tests never touch the database you use for trying the application, and they are fast.

10.7.3 Feature tests for the API

Listing 10.21 checks the API as the drivers' app will use it.

Listing 10.21 `tests/Feature/OrderApiTest.php`: prices, validation, the workflow, closed orders, payments and filters, through the API.

```php
<?php

namespace Tests\Feature;

use App\Models\Customer;
use App\Models\Order;
use App\Models\User;
use Database\Seeders\PriceListSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Laravel\Sanctum\Sanctum;
use Tests\TestCase;

class OrderApiTest extends TestCase
{
    use RefreshDatabase;

    protected function setUp(): void
    {
        parent::setUp();
        $this->seed(PriceListSeeder::class);        // services and outlets
        Sanctum::actingAs(User::factory()->create());
    }

    private function newOrder(): array
    {
        return $this->postJson('/api/orders', [
            'customer_id' => Customer::factory()->create()->id,
            'outlet_id' => 'O01', 'service_id' => 'S0001',
            'quantity' => '2,5', 'pickup' => true,
            'delivery_address' => 'Jl. Kenanga 5, Surabaya',
        ])->assertCreated()->json('data');
    }

    public function test_an_order_is_priced_by_the_rule(): void
    {
        $order = $this->newOrder();

        $this->assertSame(34000, $order['total']);   // 3 kg x 8,000 + 10,000
        $this->assertSame(34000, $order['balance']);
        $this->assertSame('queued', $order['status']);
    }
```

Listing 10.21 (Continued) *tests/Feature/OrderApiTest.php: prices, validation, the workflow, closed orders, payments and filters, through the API.*

```
    public function test_bad_input_gets_422_with_every_error(): void
    {
        $this->postJson('/api/orders', ['quantity' => 0, 'pickup' => true])
            ->assertUnprocessable()
            ->assertJsonValidationErrors(['customer_id', 'outlet_id',
                'service_id', 'quantity', 'delivery_address']);
    }

    public function test_the_steps_follow_the_workflow(): void
    {
        $url = '/api/orders/' . $this->newOrder()['id'];

        $this->patchJson("$url/status", ['status' => 'departed'])
            ->assertUnprocessable();                    // skips a step
        $this->patchJson("$url/status", ['status' => 'finished'])
            ->assertOk()->assertJsonPath('data.status', 'finished');
        $this->patchJson("$url/status", ['status' => 'departed'])
            ->assertUnprocessable()                     // not paid yet
            ->assertJsonValidationErrors('status');

        $this->postJson("$url/payments", ['amount' => 34000,
            'method' => 'qris'])->assertCreated()
            ->assertJsonPath('data.balance', 0);
        $this->patchJson("$url/status", ['status' => 'departed'])
            ->assertOk();
    }

    public function test_a_departed_order_is_closed(): void
    {
        $order = Order::factory()->departed()->create();

        $this->postJson("/api/orders/{$order->id}/payments",
            ['amount' => 1000, 'method' => 'cash'])
            ->assertStatus(409);
    }

    public function test_a_payment_cannot_exceed_the_balance(): void
    {
        $url = '/api/orders/' . $this->newOrder()['id'];

        $this->postJson("$url/payments", ['amount' => 40000,
            'method' => 'cash'])
            ->assertUnprocessable()->assertJsonValidationErrors('amount');
    }
```

Listing 10.21 (Continued) *tests/Feature/OrderApiTest.php: prices, validation, the workflow, closed orders, payments and filters, through the API.*

```
    public function test_the_list_can_be_filtered_by_status(): void
    {
        Order::factory()->count(3)->create();
        Order::factory()->finished()->count(2)->create();

        $this->getJson('/api/orders?status=finished')
            ->assertOk()
            ->assertJsonCount(2, 'data')
            ->assertJsonPath('meta.total', 2);
    }
}
```

`setUp` runs before every test: it seeds the real price list and outlets, which the orders need, and `Sanctum::actingAs` signs in a cashier for the API without creating a real token [4]. The helper

`newOrder` creates Budi's pickup order of Section 9.8 through the API, with the Indonesian comma in "2,5", and returns the `data` part of the answer. The tests then read like the shop's rules. An order of 2.5 kg with pickup costs Rp 34.000. A request with nothing right in it returns 422 with an error for every field that needs one. An order can go from queued to finished, cannot skip to departed, cannot depart unpaid, and departs after payment. A departed order refuses payments with 409. A payment above the balance is refused. And the status filter counts correctly, which `meta.total` confirms.

10.7.4 Running the tests

```
php artisan test
```

runs every test in `tests/Unit` and `tests/Feature` and prints a line per test and a summary. `php artisan test --filter=OrderApiTest` runs one class, which is handy while you work on it. When a test fails, the report names the test method, the line of the assertion that failed, and what was expected against what arrived, for example an expected status of 422 against a received 201. Read that line first: it tells you whether the application did something new (the rule was broken) or whether the test described the rule wrongly. For a failed JSON assertion, `->dump()` placed just before the assertion prints the whole response body, so you can see what the application actually sent. And when you fix a bug, run the whole suite, not only the test you were looking at: the point of automated tests is that they also watch the rules you are not thinking about at the moment. A good habit, which Dimas now follows, is to run `php artisan test` before every change is copied to the shop's computer, and to copy nothing while a single test is red. Laravel projects may also use Pest, a testing tool with a shorter syntax that runs on top of PHPUnit and runs PHPUnit test classes such as ours unchanged [12]. The chapter's `README.md` lists the number of tests in each file. These tests could not be run where this book was written, because Laravel could not be installed there; they are written to the Laravel 13 testing documentation, and the expected price totals were computed and checked in plain PHP.

10.8 Running It

With the Chapter 9 project in place, run `php artisan install:api` once, copy the chapter's files over the project, run `php artisan migrate:fresh --seed` and start the server on port 4100, as the `README.md` describes. Then try the application as each of the people in the story.

1. Open http://localhost:4100/counter. You land on the sign-in page. Sign in as `cashier@laundry.test` with the password `password`: you reach the counter, and the menu has no **Prices** link. Type `/prices` in the address bar: the answer is 403.
2. Sign out, sign in as `owner@laundry.test`: the **Prices** link is there and works.
3. Type a wrong password six times within a minute: the sixth attempt is refused with 429 before the password is even checked.
4. In a terminal, ask for a token as the driver would, with the first `curl` command of the `README.md`. The answer is a JSON object with a `token` and the user's name and role. Use the token with the second command to list the orders, and add `?status=finished` to see only the finished ones.
5. Try the same list without the `Authorization` header: the answer is 401 with `{"message":"Unauthenticated."}`.

On Windows, PowerShell's `curl` is a different program, so type `curl.exe` as the `README.md` says; on macOS and Linux `curl` is already the right one. VS Code's REST Client extension is a friendly alternative that keeps the requests in a file.

10.9 Making It Better

Token abilities. A token can carry *abilities*, a list of words chosen when it is created: `createToken('driver-phone', ['orders:read', 'payments:write'])`. Routes can then demand them with Sanctum's `abilities` and `ability` middleware, and code can ask `$request->user()->tokenCan('payments:write')` [4]. A driver's phone that only needs to read orders and record payments should not carry a token that could also create customers.

Tokens that expire. By default Sanctum tokens do not expire. The `expiration` setting in `config/sanctum.php` gives every token a lifetime in minutes, and `php artisan sanctum:prune-expired` removes the dead ones [4]. A lifetime of a few weeks, with a fresh sign-in after that, is a sensible default for the drivers' phones.

Rate limits for the API. The API as a whole deserves a limit too. `throttle:api` uses a named limiter that can allow, for example, 60 requests a minute per user, and it protects the counter computer from a buggy app that asks the same question in a loop.

Policies as the rules grow. One gate is enough for one rule. When the rules become "a cashier may edit a customer she registered today, the owner any customer", write an `OrderPolicy` or `CustomerPolicy` with a method per action; `can:update,customer` in a route then passes the bound model to the policy [3].

JSON:API. Laravel 13 adds resources that follow the JSON:API specification, with standard names for relationships, included records and sparse fieldsets [1]. For a public API used by many different clients, a published standard saves writing and explaining one's own conventions; for the shop's own phone app, the plain resources of this chapter are simpler.

10.10 Using It for Real

- **One account per person, never shared.** A shared "kasir" account makes every log line useless: the question "who changed this price?" has no answer. Create an account for each cashier and driver, and delete it the day they leave.
- **Passwords that only their owners know.** The demo accounts' password "password" must never exist on the shop's computer. Let each person choose a long password, and never ask for it by WhatsApp. Laravel stores only bcrypt hashes, but a weak password is guessed, not cracked [11].
- **HTTPS everywhere outside your own laptop.** Session cookies and bearer tokens are as good as passwords while they travel. On a real server, serve the application only over HTTPS and set `APP_DEBUG=false`, so that error pages do not show code, settings or data to strangers.
- **A token per device, revoked when the device is gone.** When a phone is lost or a driver leaves, delete the tokens at once.
- **Personal data is protected by law.** Customers' names, phone numbers and addresses are personal data under UU PDP 27/2022. Sending only the needed fields in each resource, keeping accounts personal and revoking access promptly are part of treating that data with care.

- **Tests for every rule that once cost money.** The wrong price of Section 10.1 would have been caught by the first test in **Listing 10.20** that forbids the price list to a cashier. Every time a mistake reaches the shop, write the test that would have caught it before you fix it.

10.11 Common Mistakes, and How to Avoid Them

- **Checking the role in the view only.** Hiding the **Prices** link does not protect `/prices`; the route middleware, the gate or the policy must refuse the request.
- **Telling a stranger which part was wrong.** "No user with this e-mail" or "Wrong password" helps someone guess accounts. Give one message for both.
- **Forgetting `session()->regenerate()` after sign-in.** It leaves the door open to session fixation. Always regenerate after a successful sign-in, and invalidate on sign-out.
- **Sign-out by GET.** A link or an image can then sign people out from any page. Use a POST form with `@csrf`.
- **Returning models instead of resources from an API.** The JSON then follows the database and leaks columns. Return resources, and list every field on purpose.
- **Using `whenLoaded` without loading.** A resource that uses `whenLoaded('customer')` sends no customer if the controller forgot `with('customer')`; one that reads `$this->customer->name` instead runs a query per order. Load what the resource needs in the controller.
- **Storing or logging the plain token.** `plainTextToken` must be shown once to its device and never written to a log, a database column or an error report.
- **Tests that depend on each other or on your data.** A test that assumes "there are 3 orders" breaks when the demo seeder changes. Use `RefreshDatabase` and create what each test needs inside the test.
- **Testing only the happy path.** Most of the tests in this chapter check refusals: 401, 403, 409, 422. Those are the rules that protect the shop, and they are the ones that break silently.

10.12 Chapter Summary

- Authentication answers "who are you?" and authorization "may you do this?"; failures are 401 and 403 respectively, and a browser guest is redirected to the sign-in page.
- Users have a role, owner or cashier, stored as a backed enum; passwords are stored only as hashes, thanks to the `hashed` cast.
- A hand-written `LoginController` signs staff in with `Auth::attempt`, regenerates the session, and signs them out by invalidating it. Routes are grouped into public, guest-only, `auth`, and owner-only with `can:manage-prices`, a gate defined in `AppServiceProvider`. `throttle` limits sign-in attempts.
- Laravel Sanctum issues API tokens that are stored as hashes, sent as bearer tokens, and revoked per device. The API routes live in `routes/api.php` behind `auth:sanctum` and reuse the form requests and the `order.open` middleware of Chapter 9.
- An action class, `CreateOrder`, lets the web counter and the API create orders in one way only. `UpdateOrderStatusRequest` lets programs move an order to a named status, checked by Chapter 8's workflow.
- API resources decide the JSON: which fields, under which names, converted or computed, with relations nested only when loaded, and lists paginated with `links` and `meta`.

- Unit tests check plain classes such as `PriceCalculator`; feature tests send requests through the whole application with `RefreshDatabase`, `actingAs` and `Sanctum::actingAs`, and check status codes, JSON paths and the database. `php artisan test` runs them all in seconds.

10.13 Exercises

1. For each request, name the status code of the answer and the part of the application that gives it: (a) a guest opens `/counter` in a browser; (b) a program calls `GET /api/orders` without a token; (c) a cashier opens `/prices`; (d) the sixth wrong sign-in within one minute; (e) a signed-in cashier submits the payment form of an order that another cashier has just marked as picked up.
2. Write a feature test class that proves that a cashier can neither open `/prices/create` nor send the form to `POST /prices`, that the owner can open the form, and that a guest is sent to sign in first.
3. Change the token issue so that the owner's tokens may do everything while a cashier's tokens carry only the abilities `orders:write` and `payments:write`, and protect the customer routes of the API with a new ability `customers:write`. Which lines change, and what does a cashier's phone receive when it tries to create a customer?
4. A new rule: "a cashier may correct the note of an order only on the day it was received; the owner always". Should it be a gate, a policy or a form request? Sketch where the code goes and how a route uses it.
5. Explain why `TokenController::store` can show the token only once, what exactly the `personal_access_tokens` table stores, and what the owner should do when a driver's phone is stolen.
6. Call `GET /api/orders/{id}` for Sari's order and look at `received_at`. Laravel sends it as it serializes dates by default. What do the letter `Z` and the digits after the seconds mean, and why does the hour differ from the time on the counter's clock in Surabaya?

10.14 Solutions to the Exercises

1. (a) 302 redirect to `/login`, from the `auth` middleware, with the location set by `redirectGuestsTo`. (b) 401 with `{"message":"Unauthenticated."}`, from `auth:sanctum`; there is no redirect because the request asks for JSON. (c) 403, from the `can:manage-prices` middleware, which asks the gate. (d) 429, from `throttle:5,1` on the POST route; the controller does not even run, so the password is not checked. (e) A redirect back to the order page with the red note "Order #… has left the shop and is closed.", from Chapter 9's `order.open` middleware, which answers people with a note instead of 409; the payment is not recorded.

2. **Listing 10.22** is one solution. It seeds the price list because the price form lists existing services, and it checks the database as well as the status: a 403 that still saved the new service would be a bug. The data sent to `POST /prices` is valid by Chapter 8's rules, so a failure could only come from the permission, not from validation.

Listing 10.22 `tests/Feature/PriceListAccessTest.php`: the price list is the owner's, on every route.

```php
<?php

namespace Tests\Feature;

use App\Models\User;
use Database\Seeders\PriceListSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

// Exercise 10.2: the price list belongs to the owner, on every route.
class PriceListAccessTest extends TestCase
{
    use RefreshDatabase;

    protected function setUp(): void
    {
        parent::setUp();
        $this->seed(PriceListSeeder::class);
    }

    public function test_a_cashier_cannot_open_or_send_the_price_form(): void
    {
        $cashier = User::factory()->create();

        $this->actingAs($cashier)->get('/prices/create')->assertForbidden();
        $this->actingAs($cashier)->post('/prices', [
            'description' => 'Carpet (per kg)', 'price' => 20000,
        ])->assertForbidden();
        $this->assertDatabaseMissing('services',
            ['description' => 'Carpet (per kg)']);
    }

    public function test_the_owner_can_open_the_price_form(): void
    {
        $this->actingAs(User::factory()->owner()->create())
            ->get('/prices/create')->assertOk();
    }

    public function test_a_guest_is_sent_to_sign_in_first(): void
    {
        $this->get('/prices/create')->assertRedirect('/login');
    }
}
```

3. In `TokenController::store`, choose the abilities from the role: `$abilities = $user->isOwner() ? ['*'] : ['orders:write', 'payments:write'];` and create the token with `$user->createToken($data['device'], $abilities)`. In `bootstrap/app.php`, register Sanctum's two middleware under their usual names by adding `'abilities' => \Laravel\Sanctum\Http\Middleware\CheckAbilities::class` and `'ability' => \Laravel\Sanctum\Http\Middleware\CheckForAnyAbility::class` to the `alias` array [4]. In `routes/api.php`, add `->middleware('abilities:customers:write')` to the `POST customers` route. A cashier's phone that tries to create a customer then receives 403, because its token lacks the ability, while the owner's token, with `*`, has every ability. Tokens issued before the change keep the abilities they were created with, so ask the drivers to sign in again.

4. The rule concerns one kind of model, an order, and depends on the user and on that order, so it belongs in a policy: an `OrderPolicy` with a method `updateNote(User $user, Order $order): bool` that returns `$user->isOwner() || $order->received_at->isToday()`. Laravel finds `App\Policies\OrderPolicy` for `App\Models\Order` by its name. The route that saves the note uses `->middleware('can:updateNote,order')`, and the note form's request may also call `$this->user()->can('updateNote', $this->route('order'))` in `authorize()`. A gate would work but

would collect unrelated rules in one provider; a form request alone would protect only the routes that use that request.

5. `createToken` generates a random string, stores only its SHA-256 hash in `personal_access_tokens`, together with the user's id, the device name, the abilities, and the times of creation and last use, and returns the plain string once. Because only the hash is kept, the server cannot show the token again, and a copy of the database does not give anyone working tokens [4]. When a phone is stolen, the owner deletes that device's row (for example with `$user->tokens()->where('name', 'driver-phone-rina')->delete()` in `php artisan tinker`), which locks that phone out at once, and asks the driver to change her password if it might have been saved on the phone.

6. Laravel serializes dates in ISO 8601 format in the UTC time zone, with microseconds: for example `2026-09-25T01:15:00.000000Z`. The `Z` ("Zulu") means UTC, and `.000000` are the microseconds. Surabaya is on Western Indonesian Time, seven hours ahead of UTC, so an order received at 08:15 on the counter's clock is sent as 01:15 UTC. The value is correct, but a program has to convert it, and another framework may choose a different format for the same moment. Chapter 12 turns exactly this into a rule of the API contract and changes `OrderResource` accordingly.

References

[1] Laravel, "Release Notes," *Laravel 13.x Documentation*, Mar. 17, 2026. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/releases

[2] Laravel, "Authentication," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/authentication

[3] Laravel, "Authorization," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/authorization

[4] Laravel, "Laravel Sanctum," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/sanctum

[5] Laravel, "Eloquent: API Resources," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/eloquent-resources

[6] Laravel, "HTTP Tests," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/http-tests

[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] M. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage," RFC 6750, IETF, Oct. 2012. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6750

[9] Laravel, "Database Testing," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/database-testing

[10] Laravel, "Hashing," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/hashing

[11] OWASP Foundation, "Authentication Cheat Sheet," *OWASP Cheat Sheet Series*. Accessed: Sep. 25, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

[12] Pest, "Pest: The elegant PHP testing framework," pestphp.com. Accessed: Sep. 25, 2026. [Online]. Available: https://pestphp.com/

Chapter 11

Spring Boot I: Initializr, REST Controllers and Spring Data JPA

Profitina Laundry now runs on two frameworks: the ASP.NET Core counter of Chapters 3 to 6 and the Laravel counter and API of Chapters 7 to 10. This chapter builds the shop a third time, in Java with Spring Boot, the framework that a great deal of the world's business software, from banks to logistics companies, is written in. The goal is not a third copy for its own sake. Seeing the same customers, services, orders and payments in a third framework separates what is essential about a web application from what is only one framework's habit. By the end of the chapter you will be able to create a Spring Boot 4.1 project with Spring Initializr and explain every part of it; describe what Spring's container, dependency injection and Boot's auto-configuration do for you; map the four tables of the shop to JPA entities, including a text primary key, a table whose name is an SQL keyword and enums stored as lowercase words; get queries from Spring Data JPA repositories without writing their implementation; build REST controllers that validate JSON input and answer JSON in the same field names as the Laravel API; and check the shop's rules in plain Java, with output identical to Chapter 8's PHP check.

11.1 Raka's Question

Raka is Dimas's nephew, the one who is writing the drivers' phone app. He studies informatics, and in his course the language of the back end is Java. When he read the Laravel API of Chapter 10 to write his app against it, he asked a question that every developer asks sooner or later: "Would this look the same in Spring Boot?" Bu Wati has her own version of the question. An accounting firm that keeps the books of several laundries in Surabaya has offered to connect its system to hers, and its system is written in Java.

So this chapter answers Raka's question by building the Profitina Laundry API in Spring Boot, with the same four tables, the same price list read from the same `Service.csv`, the same rules (the three-kilogram minimum, whole items, the pickup fee and free pickup from Rp 100.000, and laundry that leaves only when paid) and the same JSON field names. Chapter 12 then compares the two APIs field by field, and Chapters 13 and 14 grow the Spring version into a full application with pages, security and tests.

Spring Boot 4 is the current generation. Version 4.0 was released in November 2025 [1], version 4.1 in June 2026, and the version used in this book, 4.1.1, in August 2026 [2]. It runs on Java 17 or newer and is tested up to Java 26; this book uses Java 25, the current long-term-support release of the JDK [3], [4]. On the web side it embeds Tomcat 11, and under its data access sits Hibernate ORM 7, the most widely used implementation of the Jakarta Persistence standard [3], [5].

11.2 The Idea in Plain Words

11.2.1 A container that builds your objects

Chapter 2 described dependency injection: instead of a class building the objects it needs with `new`, it declares them as constructor parameters, and something else builds them and hands them in. In ASP.NET Core that something is the service container of `Program.cs`; in Laravel it is the service container that builds controllers and form requests. In Spring it is the *application context*, the oldest and most famous of them all [6].

When a Spring Boot application starts, the context scans the application's packages for classes marked with annotations such as `@RestController`, `@Component` or `@Service`, and for interfaces that extend Spring Data's `JpaRepository`. Each of them becomes a *bean*: one object, built once, that the context manages. When it builds `OrderController`, it sees that the constructor needs an `OrderRepository`, a `CustomerRepository` and a `ServiceRepository`, builds or finds those beans, and passes them in. Your code never calls `new OrderController(...)`; it only declares what it needs.

11.2.2 Boot: sensible defaults, switched on by what you depend on

Plain Spring can do almost anything, but it used to require a great deal of configuration to do it. *Spring Boot* adds three things that remove most of that work. *Starters* are dependencies that bring a coherent set of libraries in one line: `spring-boot-starter-webmvc` brings Spring MVC, Jackson for JSON and an embedded Tomcat; `spring-boot-starter-data-jpa` brings Hibernate and Spring Data JPA. *Auto-configuration* looks at what is on the class path and configures it with sensible defaults: if Hibernate and an H2 database driver are present and no database is configured, Boot creates an in-memory H2 database and connects Hibernate to it. And an *embedded server* means that the application is an ordinary Java program with a `main` method, started with `mvn spring-boot:run` or `java -jar`, with no separate server to install.

Spring Boot 4 split its auto-configuration into smaller modules, one per technology, so that an application loads only what it uses; the web starter is now named `spring-boot-starter-webmvc`, and every main starter has a matching test starter [7]. The projects in this book use the new names.

11.2.3 Entities, repositories and the SQL you do not write

Jakarta Persistence (JPA) is the Java standard for mapping classes to tables [8]. An *entity* is a class marked `@Entity`, with one field per column and annotations for the details that cannot be guessed: the table's name, the primary key, the length of a text column, a relation to another entity. Hibernate reads those annotations and writes the SQL: `INSERT` when you save a new object, `UPDATE` when you change one inside a transaction, `SELECT` with the right joins when you ask for one.

Spring Data JPA goes one step further. You declare an interface such as `OrderRepository extends JpaRepository<Order, Long>`, and Spring creates the implementation when the application starts: `findById`, `findAll`, `save`, `count`, `deleteById` and a dozen more come for free. Better still, a method whose name follows a pattern, such as `findByPhone(String phone)` or `findByStatusOrderByReceivedAtDescIdDesc(...)`, is turned into a query from its name [9]. It is the same idea as Eloquent's `Customer::where('phone', $phone)->first()`, written as a method signature instead of a chain of calls.

11.2.4 Controllers that speak JSON

A class marked `@RestController` answers HTTP requests with data instead of pages. Each method is mapped to a URL and a method by annotations such as `@GetMapping("/{id}")`; its parameters are filled from the URL (`@PathVariable`), the query string (`@RequestParam`) and the JSON body (`@RequestBody`); and whatever it returns is turned into JSON by Jackson [10]. `@Valid` in front of a body parameter runs Jakarta Validation on it before the method is called, using annotations such as `@NotBlank` and `@DecimalMin` on the fields of the request class [11]. **Figure 11.1** follows one request, the creation of an order, through these parts.

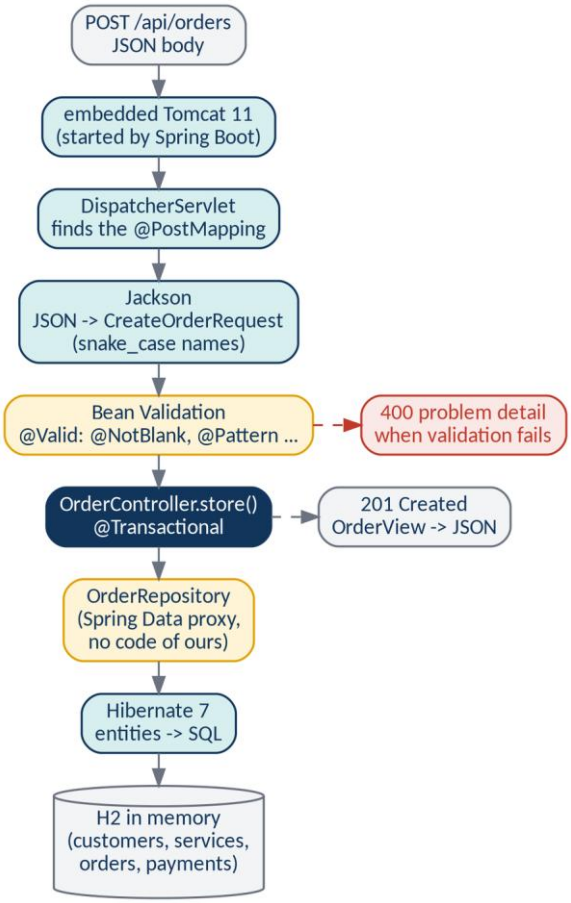


Figure 11.1 One `POST /api/orders` through a Spring Boot application: embedded Tomcat, the DispatcherServlet, Jackson, Bean Validation, the controller, a Spring Data repository, Hibernate and the database.

Table 11.1 sets the parts of Spring Boot next to the Laravel parts of Chapters 7 to 10 that play the same role. Most of the chapter is learning to recognize an old friend in new clothes.

Table 11.1 The same jobs in Laravel and in Spring Boot.

Job	Laravel (Chapters 7 to 10)	Spring Boot (this chapter)
start a project	`composer create-project laravel/laravel`	Spring Initializr, https://start.spring.io
dependencies	`composer.json`	`pom.xml` (Maven)
settings	`.env`, `config/*.php`	`application.properties`
the model	Eloquent model	JPA `@Entity`
queries	`Order::where(...)->get()`	Spring Data repository methods
the schema	migrations	Hibernate `ddl-auto` here; Flyway or Liquibase later
routes	`routes/api.php`	`@GetMapping`, `@PostMapping` on methods
input rules	form requests	records with Jakarta Validation annotations
JSON shape	API resources	response records (Jackson)
seed data	seeders	a `CommandLineRunner` bean
run it	`php artisan serve`	`mvn spring-boot:run`

11.3 Creating the Project

11.3.1 Spring Initializr

Every Spring Boot project starts the same way, at Spring Initializr [12]. The web page https://start.spring.io, the "Spring Initializr" command in VS Code's Spring Boot extension pack and the same function in other editors all ask the same questions: Maven or Gradle (choose Maven), Java, the Spring Boot version (4.1.x), the group `com.profitina`, the artifact `laundry`, Java 25, and the dependencies. Choose four: **Spring Web**, **Spring Data JPA**, **Validation** and **H2 Database**. Initializr answers with a zip file that contains a complete, runnable project; the chapter's `spring` folder is that project with the Profitina Laundry code added. Chapter 1 installed the JDK 25 and Maven that it needs, and the chapter's `README.md` repeats the commands for Windows, macOS and Linux.

11.3.2 The build file

Listing 11.1 is the project's `pom.xml`, the file Maven reads to know what the project is and what it needs.

Listing 11.1 `pom.xml`: Spring Boot 4.1.1 as the parent, Java 25, and five dependencies.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>4.1.1</version>
    <relativePath/>
  </parent>
  <groupId>com.profitina</groupId>
  <artifactId>laundry</artifactId>
  <version>11.0.0</version>
  <properties>
    <java.version>25</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webmvc</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
```

Listing 11.1 (Continued) *pom.xml: Spring Boot 4.1.1 as the parent, Java 25, and five dependencies.*

```
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

The `<parent>` is the most important part. By inheriting from `spring-boot-starter-parent` version 4.1.1, the project inherits a list of tested versions for hundreds of libraries, which is why none of the dependencies below it has a `<version>`: Spring Boot has already chosen versions of Hibernate, Jackson, Tomcat and the H2 driver that work together. `<java.version>25</java.version>` makes Maven compile for Java 25. The H2 driver has `<scope>runtime</scope>` because our code never refers to it directly, and the test starter has `<scope>test</scope>` because it is needed only by the tests. The `spring-boot-maven-plugin`

adds the `spring-boot:run` goal and packs the application into a single runnable jar with `mvn package`.

11.3.3 The layout and the entry point

The project follows Maven's standard layout. Java code lives under `src/main/java/com/profitina/laundry`, in three packages: `domain` for the entities and the rules, `repository` for the Spring Data interfaces, and `web` for the controllers and the records they read and write. Settings and data files live in `src/main/resources`, and tests in `src/test/java`. **Listing 11.2** is the class that starts everything.

Listing 11.2 `LaundryApplication.java`: the whole application starts here.

```
package com.profitina.laundry;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class LaundryApplication {

    public static void main(String[] args) {
        SpringApplication.run(LaundryApplication.class, args);
    }
}
```

`@SpringBootApplication` switches on three things at once: this class is a source of configuration, auto-configuration is enabled, and component scanning starts in this class's package, `com.profitina.laundry`, and covers every package below it. That is why the three packages must be *below* the application class; a controller placed in `com.profitina.other` would never be found. `SpringApplication.run` creates the application context, builds every bean, starts the embedded Tomcat on port 8080 and waits for requests.

11.3.4 Settings

Listing 11.3 holds the settings that differ from Boot's defaults.

Listing 11.3 `application.properties`: an in-memory database, snake_case JSON and problem details for errors.

```
spring.application.name=profitina-laundry
server.port=8080

# An in-memory H2 database: empty at every start, filled by DemoData.
spring.datasource.url=jdbc:h2:mem:laundry
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.open-in-view=false

# JSON names in snake_case (customer_id), the same as the Laravel API.
spring.jackson.property-naming-strategy=SNAKE_CASE
# Errors as RFC 9457 problem details.
spring.mvc.problemdetails.enabled=true
```

`jdbc:h2:mem:laundry` is an H2 database [13] that lives in memory and disappears when the application stops. That is ideal for a first contact: every start is a clean shop, filled by the start-up code of Section 11.7. `ddl-auto=create-drop` lets Hibernate create the tables from the entities at start-up and drop them at the end; that is fine for an in-memory database and wrong for a real one, where the tables must be managed by migrations, as Chapter 9 showed. `open-in-view=false` switches off a Spring convenience that keeps the database session open while the JSON is written;

with it off, every controller method must load what it needs inside its transaction, which makes the queries visible instead of hidden.

The last two settings concern the outside world. `property-naming-strategy=SNAKE_CASE` makes Jackson write and read JSON field names in snake_case: the Java field `deliveryAddress` becomes `delivery_address`, exactly as in the Laravel API, so the same phone app can talk to both [14]. `problemdetails.enabled=true` makes Spring answer errors in the standard *problem details* format of RFC 9457, a JSON object with `type`, `title`, `status` and `detail` [15].

11.4 The Domain: Entities

11.4.1 The four tables

Figure 11.2 shows the four tables and their Java classes. They are the tables of Chapter 5's `LaundryDbContext` and of Chapter 7's migrations, column for column; the only table left out is `outlets`, which the Spring version does not need yet.

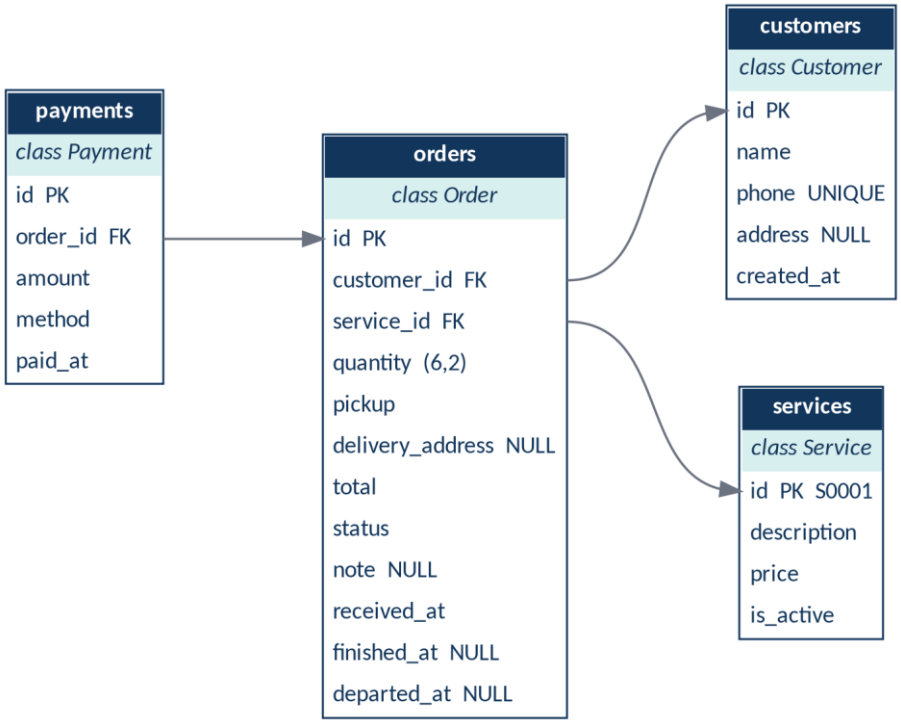


Figure 11.2 The four tables of Profitina Laundry and the entity classes that map them; arrows go from a foreign key to the primary key it refers to.

11.4.2 Customer

Listing 11.4 is the simplest entity.

Listing 11.4 `Customer.java`: a customer, with a unique phone number.

```
package com.profitina.laundry.domain;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import java.time.LocalDateTime;

@Entity
@Table(name = "customers")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 80)
    private String name;

    @Column(nullable = false, length = 20, unique = true)
    private String phone;

    @Column(length = 200)
    private String address;

    @Column(nullable = false)
    private LocalDateTime createdAt;

    protected Customer() {                // for JPA only
    }

    public Customer(String name, String phone, String address,
                    LocalDateTime createdAt) {
        this.name = name;
        this.phone = phone;
        this.address = address;
        this.createdAt = createdAt;
    }
```

Listing 11.4 (Continued) *Customer.java: a customer, with a unique phone number.*

```
    public Long getId() { return id; }
    public String getName() { return name; }
    public String getPhone() { return phone; }
    public String getAddress() { return address; }
    public LocalDateTime getCreatedAt() { return createdAt; }
}
```

`@Entity` marks the class for Hibernate and `@Table(name = "customers")` names its table; without it, the table would be called `customer`. `@Id` with `@GeneratedValue(strategy = GenerationType.IDENTITY)` makes `id` the primary key, numbered by the database itself, like Laravel's `$table->id()`. `@Column` gives the details that the migrations of Chapter 7 gave: the name is required and at most 80 characters, the phone at most 20 and unique, the address at most 200 and optional. Spring Boot's naming strategy turns the field `createdAt` into the column `created_at` automatically, so the column names match Laravel's without further annotations.

JPA requires a constructor without arguments, so that Hibernate can create an empty object and fill it from a row. Making it `protected` lets Hibernate use it while stopping our own code from creating a customer without a name. The public constructor takes everything a new customer

needs. There are getters but no setters: a customer is changed by methods that say what they do, which Chapter 13 adds, not by assigning fields one by one.

11.4.3 Service, and a name that is already taken

Listing 11.5 maps the price list. Its primary key is not a number but the code of Chapter 3, such as `S0001`, so there is no `@GeneratedValue`: the code is ours, and we set it.

Listing 11.5 `Service.java`: a line of the price list, keyed by its own code, with the name and unit worked out from the description.

```
package com.profitina.laundry.domain;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

/** A line of the price list, e.g. S0001 "Wash, dry & fold (per kg)". */
@Entity
@Table(name = "services")
public class Service {

    @Id
    @Column(length = 5)
    private String id;                      // S0001: our code, not counted

    @Column(nullable = false, length = 60)
    private String description;

    @Column(nullable = false)
    private long price;                     // rupiah per kg, item or pair

    @Column(name = "is_active", nullable = false)
    private boolean active = true;

    protected Service() {
    }

    public Service(String id, String description, long price) {
        this.id = id;
        this.description = description;
        this.price = price;
    }

    /** "Wash, dry & fold (per kg)" -> "Wash, dry & fold". */
    public String name() {
        int cut = description.lastIndexOf(" (");
        return cut < 0 ? description : description.substring(0, cut);
    }

    public boolean perKg() {
        return description.endsWith("(per kg)");
    }
```

Listing 11.5 (Continued) *Service.java: a line of the price list, keyed by its own code, with the name and unit worked out from the description.*

```
    public String unit() {
        return perKg() ? "kg"
                : description.endsWith("(per pair)") ? "pair" : "item";
    }

    public String getId() { return id; }
    public String getDescription() { return description; }
    public long getPrice() { return price; }
    public boolean isActive() { return active; }
}
```

The field `active` is stored in the column `is_active`, the name Chapter 7's migration uses, thanks to `@Column(name = "is_active")`. The methods `name()`, `perKg()` and `unit()` do exactly what the `Name`, `PerKg` and `Unit` properties of Chapter 5's C# `Service` and the accessors of Chapter 7's Eloquent `Service` do: "Wash, dry & fold (per kg)" has the name "Wash, dry & fold" and the unit "kg"; "Shoes deep clean (per pair)" has the unit "pair"; everything else is counted in items. The rule lives in one place per framework and is the same in all three.

The class is called `Service`, like the model in the other two frameworks, and that creates one small problem in Spring: Spring also has an annotation named `@Service`, for the classes of the service layer that Chapter 13 introduces. Java cannot import two things with the same short name into one file, so a file that uses both must write one of them in full, `@org.springframework.stereotype.Service`. Chapter 13 does exactly that. Keeping the name `Service` for the price list is worth that small cost, because the same word then means the same thing in all three versions of the shop.

11.4.4 Order and Payment

Listing 11.6 is the heart of the domain. Two details stand out before any code: the table is named `orders`, because `ORDER` is a reserved word of SQL (`ORDER BY`) and a table named `order` would need quoting in every query; and the class is created with a constructor that prices the order, so that no order can exist without its total.

Listing 11.6 `Order.java`: an order, priced when it is created, moved along by the shop's workflow.

```java
package com.profitina.laundry.domain;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import org.hibernate.annotations.BatchSize;

@Entity
@Table(name = "orders")                    // ORDER is a word of SQL
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "customer_id")
    private Customer customer;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "service_id")
    private Service service;

    @Column(nullable = false, precision = 6, scale = 2)
    private BigDecimal quantity;

    @Column(nullable = false)
    private boolean pickup;

    @Column(length = 200)
    private String deliveryAddress;
```

Listing 11.6 (Continued) *Order.java: an order, priced when it is created, moved along by the shop's workflow.*

```
    @Column(nullable = false)
    private long total;                     // rupiah, fixed at creation

    @Column(nullable = false, length = 12)
    private OrderStatus status = OrderStatus.QUEUED;

    @Column(length = 200)
    private String note;

    @Column(nullable = false)
    private LocalDateTime receivedAt;

    private LocalDateTime finishedAt;

    private LocalDateTime departedAt;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL,
               orphanRemoval = true)
    @BatchSize(size = 50)                   // payments of 50 orders at once
    private List<Payment> payments = new ArrayList<>();

    protected Order() {
    }

    /** A new order, priced by the shop's rule the moment it is made. */
    public Order(Customer customer, Service service, BigDecimal quantity,
                 boolean pickup, String deliveryAddress, String note,
                 LocalDateTime receivedAt) {
        this.customer = customer;
        this.service = service;
        this.quantity = quantity.setScale(2, RoundingMode.UNNECESSARY);
        this.pickup = pickup;
        this.deliveryAddress = deliveryAddress;
        this.note = note;
        this.receivedAt = receivedAt;
        this.total = LaundryRules.total(service.perKg(), service.getPrice(),
                quantity, pickup);
    }
```

Listing 11.6 (Continued) *Order.java: an order, priced when it is created, moved along by the shop's workflow.*

```
    /** One step along the workflow, stamped with the time it happened. */
    public void advance(LocalDateTime when) {
        status = LaundryRules.advance(status, balance());
        if (status == OrderStatus.FINISHED) {
            finishedAt = when;
        } else {
            departedAt = when;
        }
    }

    public long paid() {
        return payments.stream().mapToLong(Payment::getAmount).sum();
    }

    public long balance() {
        return total - paid();
    }

    public Long getId() { return id; }
    public Customer getCustomer() { return customer; }
    public Service getService() { return service; }
    public BigDecimal getQuantity() { return quantity; }
    public boolean isPickup() { return pickup; }
    public String getDeliveryAddress() { return deliveryAddress; }
    public long getTotal() { return total; }
    public OrderStatus getStatus() { return status; }
    public String getNote() { return note; }
    public LocalDateTime getReceivedAt() { return receivedAt; }
    public LocalDateTime getFinishedAt() { return finishedAt; }
    public LocalDateTime getDepartedAt() { return departedAt; }
    public List<Payment> getPayments() { return payments; }
}
```

`@ManyToOne(fetch = FetchType.LAZY, optional = false)` says that many orders belong to one customer, that every order must have one, and that the customer should not be loaded until someone asks for it. `@JoinColumn(name = "customer_id")` names the foreign-key column. The default for `@ManyToOne` is *eager* loading, which fetches the customer with every order whether it is needed or not; lazy is the better default, and Section 11.5 shows how to load the relations when they are needed. `@OneToMany(mappedBy = "order", ...)` is the other side of `Payment`'s relation: `mappedBy` says that the foreign key lives in the `payments` table, in the field `order` of `Payment`. `cascade = CascadeType.ALL` saves a new payment added to the list when the order is saved, and `orphanRemoval = true` deletes a payment removed from the list. `@BatchSize(size = 50)`, from Hibernate, makes the payments of up to fifty orders load in a single query when a list of orders needs them, instead of one query per order.

The quantity is a `BigDecimal` with precision 6 and scale 2, the same `decimal(6,2)` as Chapter 7's migration. `setScale(2, RoundingMode.UNNECESSARY)` stores "3" as 3.00 and throws an exception if a quantity with more than two decimals ever slipped past validation, instead of silently rounding it. Money is a `long` of whole rupiah; there is no floating-point number anywhere near a price.

The constructor asks `LaundryRules` for the total, and `advance` asks it for the next status, which it then stamps with the time: `finishedAt` for finished, `departedAt` for departed. `paid()` and `balance()` compute what Chapter 5's `Paid` and `Balance` properties computed. **Listing 11.7** is the payment, the last of the four entities.

Listing 11.7 `Payment.java`: money received for an order; an order may be paid in parts.

```
package com.profitina.laundry.domain;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import java.time.LocalDateTime;

/** Money received for an order; an order may be paid in parts. */
@Entity
@Table(name = "payments")
public class Payment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "order_id")
    private Order order;

    @Column(nullable = false)
    private long amount;

    @Column(nullable = false, length = 10)
    private PaymentMethod method;

    @Column(nullable = false)
    private LocalDateTime paidAt;

    protected Payment() {
    }

    public Payment(Order order, long amount, PaymentMethod method,
                   LocalDateTime paidAt) {
        this.order = order;
        this.amount = amount;
        this.method = method;
        this.paidAt = paidAt;
    }
```

Listing 11.7 (Continued) *Payment.java: money received for an order; an order may be paid in parts.*

```
    public Long getId() { return id; }
    public Order getOrder() { return order; }
    public long getAmount() { return amount; }
    public PaymentMethod getMethod() { return method; }
    public LocalDateTime getPaidAt() { return paidAt; }
}
```

11.4.5 Statuses stored as the other frameworks store them

Java enums are named in capitals by convention, `QUEUED`, while the Laravel database of Chapter 7 stores the words `queued`, `finished` and `departed`. JPA's usual choice, `@Enumerated(EnumType.STRING)`, would store `QUEUED`, and the two applications could not share a database or even agree in their JSON. **Listing 11.8** gives each enum constant its word, and **Listing 11.9** tells JPA to store that word.

Listing 11.8 `OrderStatus.java`: three statuses, each with the word that the database and the JSON use.

```
package com.profitina.laundry.domain;

/** The life of an order; value() is the word stored and sent as JSON. */
public enum OrderStatus {
    QUEUED("queued"),
    FINISHED("finished"),
    DEPARTED("departed");

    private final String value;

    OrderStatus(String value) {
        this.value = value;
    }

    public String value() {
        return value;
    }

    /** The status that follows this one, or null when the order has left. */
    public OrderStatus next() {
        return switch (this) {
            case QUEUED -> FINISHED;
            case FINISHED -> DEPARTED;
            case DEPARTED -> null;
        };
    }

    /** "finished" -> FINISHED; an unknown word gives an exception. */
    public static OrderStatus fromValue(String value) {
        for (OrderStatus s : values()) {
            if (s.value.equals(value)) {
                return s;
            }
        }
        throw new IllegalArgumentException("Unknown status: " + value);
    }
}
```

Listing 11.9 `OrderStatusConverter.java`: stores `queued`, not `QUEUED`, in every status column.

```
package com.profitina.laundry.domain;

import jakarta.persistence.AttributeConverter;
import jakarta.persistence.Converter;

/** Stores "queued", not "QUEUED": the same text as the Laravel app. */
@Converter(autoApply = true)
public class OrderStatusConverter
        implements AttributeConverter<OrderStatus, String> {

    @Override
    public String convertToDatabaseColumn(OrderStatus status) {
        return status == null ? null : status.value();
    }

    @Override
    public OrderStatus convertToEntityAttribute(String value) {
        return value == null ? null : OrderStatus.fromValue(value);
    }
}
```

An `AttributeConverter` has two methods, one for each direction, and `@Converter(autoApply = true)` applies it to every field of type `OrderStatus` in every entity, so `Order` needs no annotation

on its `status` field [8]. `PaymentMethod` and its converter, in the chapter's folder, do the same for `cash`, `qris` and `transfer`. `next()` is the Java twin of the PHP enum's `next()` in Chapter 7: queued to finished, finished to departed, and `null` after that.

11.4.6 The rules, in plain Java

Listing 11.10 is the Spring version of Chapter 8's `PriceCalculator` and `OrderWorkflow`, in one class that knows nothing about Spring, JPA or HTTP.

Listing 11.10 `LaundryRules.java`: the price rule and the workflow, with the same whole-number arithmetic as the PHP version.

```
package com.profitina.laundry.domain;

import java.math.BigDecimal;
import java.util.Locale;

/**
 * The shop's rules as plain Java: no Spring, no database. The same
 * arithmetic as PriceCalculator and OrderWorkflow in the Laravel app.
 */
public final class LaundryRules {
    public static final long MINIMUM_KG = 3;
    public static final long PICKUP_FEE = 10_000;
    public static final long FREE_PICKUP_FROM = 100_000;

    private LaundryRules() {
    }

    /** 4.5 -> 450: the quantity in hundredths, exact. */
    public static long hundredths(BigDecimal quantity) {
        return quantity.movePointRight(2).longValueExact();
    }

    /** At least 3 kg for per-kg services; whole items otherwise. */
    public static long total(boolean perKg, long price, BigDecimal quantity,
                             boolean pickup) {
        long h = hundredths(quantity);
        long billed = perKg ? Math.max(h, MINIMUM_KG * 100)
                            : (h + 99) / 100 * 100;
        long subtotal = (price * billed + 50) / 100;       // half up
        long fee = pickup && subtotal < FREE_PICKUP_FROM ? PICKUP_FEE : 0;
        return subtotal + fee;
    }

    /** The next status, or an exception that says why there is none. */
    public static OrderStatus advance(OrderStatus status, long balance) {
        OrderStatus next = status.next();
        if (next == null) {
            throw new IllegalStateException(
                    "This order has already left the shop.");
        }
        if (next == OrderStatus.DEPARTED && balance > 0) {
            throw new IllegalStateException("Collect " + rupiah(balance)
                    + " before handing over the laundry.");
        }
        return next;
    }
```

Listing 11.10 (Continued) *LaundryRules.java: the price rule and the workflow, with the same whole-number arithmetic as the PHP version.*

```
    /** 36000 -> "Rp 36.000", as Money::rp does in PHP. */
    public static String rupiah(long amount) {
        return "Rp " + String.format(Locale.ROOT, "%,d", amount)
                .replace(',', '.');
    }
}
```

The arithmetic is Chapter 8's, step for step. The quantity becomes hundredths: 4.5 becomes 450, exactly, because `movePointRight(2)` on a `BigDecimal` moves the decimal point without any rounding, and `longValueExact` would throw an exception rather than drop a digit. A per-kilogram service bills at least 300 hundredths, three kilograms; other services round up to whole units with `(h + 99) / 100 * 100`. The subtotal is price times billed hundredths divided by 100, rounded half up by adding 50 before the integer division. Pickup costs Rp 10.000 below a subtotal of Rp 100.000 and nothing from there on. `advance` refuses to move a departed order and refuses to let laundry leave with money owing, with the same messages as the PHP `OrderWorkflow`, and `rupiah` writes "Rp 36.000" as `Money::rp` does.

Because the class needs nothing but the JDK, it can be compiled and run without Maven, Spring or a network. The chapter's `check/RulesCheck.java` runs the nine price cases and four workflow steps of Chapter 8's `tools/core_check.php`:

```
javac -d out -sourcepath src/main/java check/RulesCheck.java
java -cp out RulesCheck
```

```
Prices (Problem 8.1)
  Wash, dry & fold              4.50 kg            36000
  Wash, dry & fold              2.00 kg            24000
  Wash, dry & fold              4.02 kg            32160
  Ironing only                  3.35 kg            20100
  Curtain                       4.40 kg   pickup   76000
  Bed cover, any size           2.00 item pickup   80000
  Dry cleaning - suit 2 pieces  2.00 item pickup  130000
  Dry cleaning - shirt/blouse   5.00 item pickup  100000
  Shoes deep clean              1.50 pair          90000
Workflow (Problem 8.2)
  Queued   owing  25000 -> Finished
  Finished owing  25000 -> Collect Rp 25.000 before handing over the laundry.
  Finished owing      0 -> Departed
  Departed owing      0 -> This order has already left the shop.
```

This output was produced by the program, and compared byte for byte with the output of `php tools/core_check.php` from Chapter 8: the two are identical. Three frameworks, two languages and one set of prices. Whatever else differs between the versions of the shop, a customer pays the same amount in all of them.

11.5 Repositories

Listing 11.11 is a whole repository: an interface with one extra method, and nothing else to write.

Listing 11.11 `CustomerRepository.java`: every standard operation for customers, plus two methods derived from their names.

```
package com.profitina.laundry.repository;

import com.profitina.laundry.domain.Customer;
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;

public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Optional<Customer> findByPhone(String phone);

    boolean existsByPhone(String phone);
}
```

`JpaRepository<Customer, Long>` says that the repository manages `Customer` entities whose id is a `Long`. Spring Data reads the name `findByPhone`: "find" means a query, "By" starts the condition, "Phone" is the property `phone`. The generated query is `select c from Customer c where c.phone = ?1`, and the `Optional` return type means "one customer or none". `existsByPhone` asks the same question with a `boolean` answer, which is cheaper when the customer itself is not needed [9]. `ServiceRepository` extends `JpaRepository<Service, String>`, because the service's id is its code, and adds `findByActiveTrueOrderByIdAsc`, the active price list in code order. `PaymentRepository` needs no extra methods yet.

Listing 11.12 is the repository that the order endpoints use.

Listing 11.12 `OrderRepository.java`: pages of orders, filtered by status, and one order with everything it needs.

```
package com.profitina.laundry.repository;

import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import java.util.Optional;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.EntityGraph;
import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderRepository extends JpaRepository<Order, Long> {

    @EntityGraph(attributePaths = {"customer", "service"})
    Page<Order> findAllByOrderByReceivedAtDescIdDesc(Pageable pageable);

    @EntityGraph(attributePaths = {"customer", "service"})
    Page<Order> findByStatusOrderByReceivedAtDescIdDesc(OrderStatus status,
                                                        Pageable pageable);

    @EntityGraph(attributePaths = {"customer", "service", "payments"})
    Optional<Order> findWithDetailsById(Long id);
}
```

The long names are long because they say everything. `findByStatusOrderByReceivedAtDescIdDesc(OrderStatus status, Pageable pageable)` means: orders with this status, newest first, and orders received in the same second by id, one page at a time. The `Pageable` parameter carries the page number and size, and the `Page` result carries the orders of that page together with the total count, from which the number of pages follows.

`@EntityGraph(attributePaths = {"customer", "service"})` is the Spring Data way of saying what Eloquent says with `Order::with(['customer', 'service'])`: load these lazy relations together with the orders, in the same query. Without it, writing the JSON for a page of twenty

orders would load each order's customer and service with a separate query: the "N+1 queries" problem that Chapter 8 met in Laravel. `findWithDetailsById` also loads the payments, for the page of a single order.

11.6 REST Controllers

11.6.1 Records for what goes out

An entity is not a good answer for an API, for the same reasons as a Laravel model in Chapter 10: it follows the database, it would expose everything, and serializing its lazy relations outside a transaction fails. The Spring version therefore answers with *records*, Java's compact immutable data classes, which play the role of Laravel's API resources. **Listing 11.13** is the order.

Listing 11.13 `OrderView.java`: the JSON of one order, built from the entity.

```
package com.profitina.laundry.web;

import com.profitina.laundry.domain.Order;
import java.math.BigDecimal;
import java.time.LocalDateTime;

/** The JSON of one order: the same field names as Laravel's resource. */
public record OrderView(
        Long id,
        CustomerView customer,
        ServiceView service,
        BigDecimal quantity,
        boolean pickup,
        String deliveryAddress,
        long total,
        long paid,
        long balance,
        String status,
        String note,
        LocalDateTime receivedAt,
        LocalDateTime finishedAt,
        LocalDateTime departedAt) {

    public static OrderView of(Order o) {
        return new OrderView(o.getId(), CustomerView.of(o.getCustomer()),
                ServiceView.of(o.getService()), o.getQuantity(),
                o.isPickup(), o.getDeliveryAddress(), o.getTotal(), o.paid(),
                o.balance(), o.getStatus().value(), o.getNote(),
                o.getReceivedAt(), o.getFinishedAt(), o.getDepartedAt());
    }
}
```

Each component of the record becomes a JSON field, and the snake_case naming strategy of **Listing 11.3** turns `deliveryAddress` into `delivery_address` and `receivedAt` into `received_at`: the same names as the Laravel resource. The static method `of` builds the record from an `Order`, reading the customer and the service, which must therefore be loaded; the repositories' entity graphs make sure they are. `CustomerView` and `ServiceView`, in the chapter's folder, are two more records of the same kind, and `ServiceView` uses the `name()` and `unit()` of the entity. A page of orders is wrapped in `PageView` (**Listing 11.14**), which gives the page the same `data` and `meta` parts as Laravel's paginated resources, with pages counted from 1.

Listing 11.14 `PageView.java`: a page as `data` and `meta`, counting pages from 1 as Laravel does.

```
package com.profitina.laundry.web;

import java.util.List;
import java.util.function.Function;
import org.springframework.data.domain.Page;

/** {data: [...], meta: {...}}, with pages counted from 1 as in Laravel. */
public record PageView<T>(List<T> data, Meta meta) {

    public record Meta(int currentPage, int perPage, long total,
                       int lastPage) {
    }

    public static <E, T> PageView<T> of(Page<E> page, Function<E, T> map) {
        return new PageView<>(page.getContent().stream().map(map).toList(),
                new Meta(page.getNumber() + 1, page.getSize(),
                        page.getTotalElements(),
                        Math.max(1, page.getTotalPages())));
    }
}
```

Spring Data counts pages from 0, and a `Page` also has a `getTotalPages()` of 0 when there are no results; `PageView.of` converts both to what a person and the Laravel API expect.

11.6.2 Records for what comes in

Listing 11.15 is the input of a new order, with its rules.

Listing 11.15 `CreateOrderRequest.java`: the fields of a new order and their rules, including one rule about two fields.

```
package com.profitina.laundry.web;

import jakarta.validation.constraints.AssertTrue;
import jakarta.validation.constraints.DecimalMax;
import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Digits;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;
import java.math.BigDecimal;

public record CreateOrderRequest(
        @NotNull Long customerId,
        @NotBlank String serviceId,
        @NotNull @DecimalMin("0.1") @DecimalMax("100")
        @Digits(integer = 3, fraction = 2) BigDecimal quantity,
        boolean pickup,
        @Size(max = 200) String deliveryAddress,
        @Size(max = 200) String note) {

    /** A rule about two fields at once: pickup needs an address. */
    @AssertTrue(message = "Where should the laundry be delivered?")
    public boolean isAddressGivenForPickup() {
        return !pickup || (deliveryAddress != null
                && !deliveryAddress.isBlank());
    }
}
```

Jackson fills the record from the JSON body, `customer_id` into `customerId` and so on, and `@Valid` in the controller then checks the annotations [11]. `@NotNull` and `@NotBlank` make fields required; `@DecimalMin("0.1")`, `@DecimalMax("100")` and `@Digits(integer = 3, fraction = 2)` are the Java spelling of Laravel's `min:0.1`, `max:100` and `decimal:0,2`. The rule "pickup needs an address"

involves two fields, so it cannot be an annotation on one of them. A method marked `@AssertTrue` is checked like a field that must be true, and it can look at the whole record: Jakarta Validation treats `isAddressGivenForPickup()` as a property named `addressGivenForPickup`. `CreateCustomerRequest` in the chapter's folder checks a new customer, and its *compact constructor*, the block without parameters that runs when the record is built, reduces the phone to digits before any rule sees it, as Chapter 10's `prepareForValidation` did.

11.6.3 The controllers

Listing 11.16 is the smallest controller, the price list.

Listing 11.16 `ServiceController.java`: the active price list, and one service by its code.

```
package com.profitina.laundry.web;

import com.profitina.laundry.repository.ServiceRepository;
import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.server.ResponseStatusException;

@RestController
@RequestMapping("/api/services")
public class ServiceController {

    private final ServiceRepository services;

    public ServiceController(ServiceRepository services) {
        this.services = services;          // handed in by Spring
    }

    @GetMapping
    public List<ServiceView> index() {
        return services.findByActiveTrueOrderByIdAsc().stream()
                .map(ServiceView::of).toList();
    }

    @GetMapping("/{id}")
    public ServiceView show(@PathVariable String id) {
        return services.findById(id).map(ServiceView::of)
                .orElseThrow(() -> new ResponseStatusException(
                        HttpStatus.NOT_FOUND, "No service " + id + "."));
    }
}
```

`@RequestMapping("/api/services")` sets the common start of the URLs; `@GetMapping` with no argument answers `GET /api/services`, and `@GetMapping("/{id}")` answers `GET /api/services/S0001`, with `@PathVariable String id` receiving "S0001". The repository arrives through the constructor: Spring sees the single constructor and injects the bean, with no annotation needed. When a service does not exist, `ResponseStatusException` with `HttpStatus.NOT_FOUND` makes Spring answer 404, in the problem-details format. `CustomerController`, in the chapter's folder, lists customers, looks one up with `?phone=`, and creates one with `POST /api/customers`, answering 409 Conflict if the phone is already registered.

Listing 11.17 is the controller for orders.

Listing 11.17 `OrderController.java`: list, show, create and move orders, each inside a transaction.

```
package com.profitina.laundry.web;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.domain.Service;
import com.profitina.laundry.repository.CustomerRepository;
import com.profitina.laundry.repository.OrderRepository;
import com.profitina.laundry.repository.ServiceRepository;
import jakarta.validation.Valid;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.http.HttpStatus;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.server.ResponseStatusException;

@RestController
@RequestMapping("/api/orders")
public class OrderController {

    private final OrderRepository orders;
    private final CustomerRepository customers;
    private final ServiceRepository services;

    public OrderController(OrderRepository orders,
                           CustomerRepository customers,
                           ServiceRepository services) {
        this.orders = orders;
        this.customers = customers;
        this.services = services;
    }
```

Listing 11.17 (Continued) *OrderController.java: list, show, create and move orders, each inside a transaction.*

```
    /** GET /api/orders?status=finished&page=2 (pages start at 1). */
    @GetMapping
    @Transactional(readOnly = true)
    public PageView<OrderView> index(
            @RequestParam(required = false) String status,
            @RequestParam(defaultValue = "1") int page) {
        PageRequest pageable = PageRequest.of(Math.max(page, 1) - 1, 20);
        Page<Order> result = status == null
                ? orders.findAllByOrderByReceivedAtDescIdDesc(pageable)
                : orders.findByStatusOrderByReceivedAtDescIdDesc(
                        parseStatus(status), pageable);
        return PageView.of(result, OrderView::of);
    }

    @GetMapping("/{id}")
    @Transactional(readOnly = true)
    public OrderView show(@PathVariable Long id) {
        return OrderView.of(find(id));
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    @Transactional
    public OrderView store(@Valid @RequestBody CreateOrderRequest in) {
        Customer customer = customers.findById(in.customerId())
                .orElseThrow(() -> unprocessable("No customer "
                        + in.customerId() + "."));
        Service service = services.findById(in.serviceId())
                .filter(Service::isActive)
                .orElseThrow(() -> unprocessable("The service "
                        + in.serviceId() + " is not offered."));
        Order order = new Order(customer, service, in.quantity(),
                in.pickup(), in.deliveryAddress(), in.note(), now());
        return OrderView.of(orders.save(order));
    }
```

Listing 11.17 (Continued) *OrderController.java: list, show, create and move orders, each inside a transaction.*

```
    /** PATCH /api/orders/5/status {"status": "finished"}: one step on. */
    @PatchMapping("/{id}/status")
    @Transactional
    public OrderView updateStatus(@PathVariable Long id,
                                  @Valid @RequestBody StatusRequest in) {
        Order order = find(id);
        OrderStatus wanted = parseStatus(in.status());
        if (wanted != order.getStatus().next()) {
            throw new ResponseStatusException(HttpStatus.CONFLICT,
                    "An order that is " + order.getStatus().value()
                    + " cannot become " + wanted.value() + ".");
        }
        try {
            order.advance(now());
        } catch (IllegalStateException e) {
            throw new ResponseStatusException(HttpStatus.CONFLICT,
                    e.getMessage());
        }
        return OrderView.of(order);        // saved when the method commits
    }

    private Order find(Long id) {
        return orders.findWithDetailsById(id)
                .orElseThrow(() -> new ResponseStatusException(
                        HttpStatus.NOT_FOUND, "No order " + id + "."));
    }

    private static OrderStatus parseStatus(String value) {
        try {
            return OrderStatus.fromValue(value);
        } catch (IllegalArgumentException e) {
            throw unprocessable(e.getMessage());
        }
    }

    private static ResponseStatusException unprocessable(String reason) {
        return new ResponseStatusException(HttpStatus.UNPROCESSABLE_CONTENT,
                reason);
    }

    private static LocalDateTime now() {
        return LocalDateTime.now().truncatedTo(ChronoUnit.SECONDS);
    }
}
```

`index` receives two query parameters, both optional: `@RequestParam(required = false) String status` is `null` when the URL has no `?status=`, and `@RequestParam(defaultValue = "1") int page` is 1 when there is no `?page=`. It asks for the right page with `PageRequest.of(page - 1, 20)` and wraps the result with `PageView.of(result, OrderView::of)`: `OrderView::of` is a *method reference*, the function that turns each order into its view.

Every method carries `@Transactional`. A read-only transaction for the reads tells Hibernate that nothing will change, which lets it skip some work; a read-write transaction for the writes groups everything the method does into one unit that either happens completely or not at all. It also keeps the database session open while `OrderView.of` reads the lazy relations. `store` answers 201 Created because of `@ResponseStatus(HttpStatus.CREATED)`. It finds the customer and the active service, answering 422 if either is missing, then creates the order with the constructor that prices it, and saves it. `updateStatus` checks that the wanted status is the next one and lets `Order.advance` apply the workflow; it needs no `save` call, because Hibernate notices the change to an entity loaded inside the transaction and writes it when the transaction commits. In this

chapter a refused step is answered with 409 Conflict. Chapter 12 compares that choice with the Laravel API's, which answers 422, and settles the question.

Putting `@Transactional` on controller methods is acceptable for a first, small API. Chapter 13 moves the work into a *service layer*, classes marked `@org.springframework.stereotype.Service` that own the transactions and the use of the repositories, so that the controllers only translate between HTTP and the domain.

11.7 Loading the Price List

Laravel filled its database with seeders; Spring Boot applications usually do the same with a bean that runs once after start-up. Any bean that implements `CommandLineRunner` has its `run` method called when the application context is ready. **Listing 11.18** is ours.

Listing 11.18 `DemoData.java`: the real price list from `Service.csv`, and the three customers of Chapters 5 and 7 with their orders.

```
package com.profitina.laundry;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.Payment;
import com.profitina.laundry.domain.PaymentMethod;
import com.profitina.laundry.domain.Service;
import com.profitina.laundry.repository.CustomerRepository;
import com.profitina.laundry.repository.OrderRepository;
import com.profitina.laundry.repository.ServiceRepository;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigDecimal;
import java.nio.charset.StandardCharsets;
import java.time.LocalDate;
import java.time.LocalDateTime;
import org.springframework.boot.CommandLineRunner;
import org.springframework.core.io.ClassPathResource;
import org.springframework.stereotype.Component;

/** Spring's counterpart of Laravel's seeders: runs once at start-up. */
@Component
public class DemoData implements CommandLineRunner {

    private final ServiceRepository services;
    private final CustomerRepository customers;
    private final OrderRepository orders;

    public DemoData(ServiceRepository services, CustomerRepository customers,
                    OrderRepository orders) {
        this.services = services;
        this.customers = customers;
        this.orders = orders;
    }

    @Override
    public void run(String... args) throws IOException {
        if (services.count() > 0) {
            return;                          // already filled: do nothing
        }
        loadPriceList();
```

Listing 11.18 (Continued) *DemoData.java: the real price list from Service.csv, and the three customers of Chapters 5 and 7 with their orders.*

```
        LocalDateTime today = LocalDate.now().atStartOfDay();
        Customer sari = customers.save(new Customer("Sari Wulandari",
                "081200000101", "Jl. Kertajaya 12, Surabaya", today));
        Customer budi = customers.save(new Customer("Budi Santoso",
                "081200000102", null, today));
        customers.save(new Customer("Dewi Lestari", "081200000103",
                "Jl. Manyar 7, Surabaya", today));

        Order first = new Order(sari, service("S0001"), new BigDecimal("4.5"),
                false, null, null, today.minusDays(1).plusHours(8));
        first.getPayments().add(new Payment(first, 36_000, PaymentMethod.QRIS,
                today.minusDays(1).plusHours(8)));
        first.advance(today.plusHours(9));              // queued -> finished
        orders.save(first);

        Order second = new Order(budi, service("S0002"), new BigDecimal("3"),
                false, null, null, today.plusMinutes(450));
        second.getPayments().add(new Payment(second, 20_000,
                PaymentMethod.CASH, today.plusMinutes(450)));
        orders.save(second);

        orders.save(new Order(budi, service("S0008"), new BigDecimal("2"),
                true, "Jl. Mulyosari 30, Surabaya", null,
                today.plusHours(10)));
    }

    /** data/Service.csv: the same price list as Chapters 3, 5 and 7. */
    private void loadPriceList() throws IOException {
        var csv = new ClassPathResource("data/Service.csv");
        try (var in = new BufferedReader(new InputStreamReader(
                csv.getInputStream(), StandardCharsets.UTF_8))) {
            in.readLine();                  // the header line
            for (String line; (line = in.readLine()) != null; ) {
                int first = line.indexOf(',');
                int last = line.lastIndexOf(',');
                String description = line.substring(first + 1, last)
                        .replace("\"", "");
                long price = new BigDecimal(line.substring(last + 1))
                        .longValueExact();
                services.save(new Service(line.substring(0, first),
                        description, price));
            }
        }
    }
```

Listing 11.18 (Continued) *DemoData.java: the real price list from Service.csv, and the three customers of Chapters 5 and 7 with their orders.*

```
    private Service service(String id) {
        return services.findById(id).orElseThrow();
    }
}
```

`@Component` makes the class a bean, and its constructor receives three repositories. `run` first checks whether the price list is already there, so that the class stays harmless if the database is ever made permanent. `loadPriceList` reads `data/Service.csv` from the class path, the same file as in Chapters 3, 5 and 7, copied into `src/main/resources/data`. The CSV parsing is deliberately simple: the id is before the first comma, the price after the last, and the description, which may contain commas and therefore quotes, is everything between. Then Sari, Budi and Dewi are created with the orders they have in Chapter 7's seeder. Sari's order is priced by the constructor

at Rp 36.000, paid by QRIS and advanced to finished; Budi's express order of 3 kg has a first payment of Rp 20.000; and his two bed covers with pickup cost Rp 80.000.

11.8 Running It

With JDK 25 and Maven installed as in Chapter 1, open a terminal in the chapter's `spring` folder and run:

```
mvn spring-boot:run
```

The first run downloads Spring Boot and its libraries from Maven Central, which takes a few minutes; later runs start in seconds. When the log shows that Tomcat has started on port 8080, open http://localhost:8080/api/services in a browser: the twelve services of `Service.csv` appear as JSON, each with its `id`, `name`, `unit` and `price`. http://localhost:8080/api/orders shows the three demo orders as a page with `data` and `meta`. To create an order, use the `curl` command in the chapter's `README.md`; it registers Dewi's first order, 2.5 kg of "Wash, dry & fold" with pickup, and the answer carries a `total` of 34000, three kilograms at Rp 8.000 plus the pickup fee, exactly as the Laravel API computes it in Chapter 10. Send the same request with `"quantity": 0` and without the address: the answer is 400 with a problem-details body. Stop the application with Ctrl+C; because the database is in memory, the next start is a clean shop again.

In VS Code, with the Extension Pack for Java and the Spring Boot Extension Pack, open the `spring` folder, wait until the Java projects are loaded, and press the **Run** link above `main` in `LaundryApplication.java`; the Spring Boot Dashboard in the side bar starts and stops the application as well.

The environment in which this book was written could not download Maven packages, so the application itself was not started there. Everything in `domain/LaundryRules.java` and `check/RulesCheck.java` was compiled and run, with the output shown in Section 11.4.6; the other Java files were compiled against small stand-ins for the Spring, Jakarta and JUnit types they use, with no errors or warnings. The chapter's `README.md` says exactly what was run and how.

11.9 Testing the Rules

Listing 11.19 is the Spring project's first test. Like Chapter 10's `PriceCalculatorTest`, it checks the rules alone, without starting Spring.

Listing 11.19 `LaundryRulesTest.java`: the price rule and the workflow, tested without Spring.

```
package com.profitina.laundry.domain;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import java.math.BigDecimal;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

/** No Spring, no database: the rules alone, fast. */
class LaundryRulesTest {

    @ParameterizedTest
    @CsvSource({
        "true, 8000, 4.5, false, 36000",
        "true, 8000, 2.5, false, 24000",
        "true, 8000, 2.5, true, 34000",
        "false, 35000, 1.5, false, 70000",
        "false, 65000, 2, true, 130000",
        "true, 15000, 3.35, false, 50250",
    })
    void totalFollowsTheRule(boolean perKg, long price, String quantity,
                             boolean pickup, long expected) {
        assertEquals(expected, LaundryRules.total(perKg, price,
                new BigDecimal(quantity), pickup));
    }

    @Test
    void laundryLeavesOnlyWhenPaid() {
        assertEquals(OrderStatus.FINISHED,
                LaundryRules.advance(OrderStatus.QUEUED, 25_000));
        assertThrows(IllegalStateException.class,
                () -> LaundryRules.advance(OrderStatus.FINISHED, 25_000));
        assertEquals(OrderStatus.DEPARTED,
                LaundryRules.advance(OrderStatus.FINISHED, 0));
    }
}
```

`@ParameterizedTest` with `@CsvSource` runs `totalFollowsTheRule` once per line, converting each comma-separated value to the parameter's type; the six lines are the six cases of Chapter 10's PHP test, with the same expected totals. `assertThrows` checks that advancing a finished, unpaid order throws the exception that the controller turns into an error. `mvn test` runs the test with JUnit, which the test starter provides. Chapter 14 adds tests that start the whole application and send it HTTP requests, as Chapter 10 did for Laravel.

11.10 Making It Better

A database that stays. Replacing the H2 URL with a file-based or server database, SQLite as in Chapter 13, or MySQL or PostgreSQL in a shop with several computers, keeps the data between starts. At the same time `ddl-auto` should become `validate` or `none`, and the schema should be managed by migrations with Flyway or Liquibase, which Spring Boot 4 supports through their own starters [7].

Profiles. `application-dev.properties` and `application-prod.properties` hold settings that differ between a laptop and the shop, and `--spring.profiles.active=prod` chooses one; the demo data can be limited to the `dev` profile with `@Profile("dev")` on `DemoData`, the Spring version of Chapter 9's environment check.

Developer tools. The `spring-boot-devtools` dependency restarts the application automatically when a class changes, which makes the edit-and-try cycle almost as quick as PHP's.

Health and metrics. `spring-boot-starter-actuator` adds endpoints such as `/actuator/health` that a server or a monitoring service can ask whether the application is alive.

The same error shape as Laravel. Problem details are a good standard, but they are not the shape that the Laravel API sends. Chapter 12 writes one class that makes every error of the Spring API look like Laravel's, so that the phone app needs only one way of reading errors.

11.11 Using It for Real

- **Start from Initializr, not from a copy.** A copied project carries old versions and forgotten settings. Initializr always gives the current Boot version and a consistent set of libraries.
- **Keep the rules out of the framework.** `LaundryRules` compiles and runs with the JDK alone, and its output could be compared with the PHP version's. For any business whose rules must be the same in several systems, such as a cooperative's interest calculation or a clinic's queue numbers, a plain class plus a shared table of test cases is the cheapest guarantee.
- **Match the names at the edges.** A second system that talks to the first, like the accounting firm's, is easiest to connect when the JSON field names and the stored words are the same. The snake_case strategy and the enum converters cost a few lines here and save a translation layer later.
- **Treat `ddl-auto` as a prototype tool.** Let Hibernate create the tables while the model is still changing every hour; switch to migrations before the first real order is stored.
- **Personal data is personal data in every language.** The Spring API holds the same customers' names, phone numbers and addresses as the Laravel one, and the obligations of UU PDP 27/2022 apply to it in the same way. Until Chapter 14 adds Spring Security, run this API only on your own computer.

11.12 Common Mistakes, and How to Avoid Them

- **Placing classes outside the application's package.** Component scanning starts at the package of `LaundryApplication`; a controller in a sibling package is never found, and its URLs answer 404.
- **Naming a table `order`.** `ORDER` is an SQL keyword. Name the table `orders` with `@Table`.
- **Forgetting the no-argument constructor.** Hibernate needs one to create entities from rows. Make it `protected`, so that your own code cannot use it by accident.
- **Eager relations everywhere.** `@ManyToOne` loads eagerly by default. Make relations lazy and load them with `@EntityGraph` where they are needed.
- **Serializing entities directly.** Jackson may follow lazy relations outside a transaction and fail, or follow `order.payments[0].order` in a circle. Answer with records.
- **Storing enums by position or by Java name.** `EnumType.ORDINAL` stores 0, 1, 2 and breaks when a constant is inserted; `EnumType.STRING` stores `QUEUED`, which other programs do not expect. Store an agreed word with a converter.
- **Money in `double`.** 0.1 + 0.2 is not 0.3 in binary floating point. Use `long` for whole rupiah and `BigDecimal` for quantities.
- **Pages from 0 in the URL.** Spring Data counts pages from 0, people and the Laravel API from 1. Convert once, in one place.

- **Forgetting `@Transactional` on a method that reads lazy relations.** With `open-in-view=false`, reading `order.getCustomer().getName()` outside a transaction throws a `LazyInitializationException`. Load what you need with an entity graph, inside a transaction.
- **Relying on `ddl-auto=update` for a real database.** It adds columns but never removes or renames them, and it cannot move data. Use migrations.

11.13 Chapter Summary

- Spring Boot 4.1 runs on Java 17 or newer (this book uses Java 25) and brings starters, auto-configuration and an embedded Tomcat 11; a project starts at Spring Initializr, and `pom.xml` inherits tested library versions from `spring-boot-starter-parent`.
- Spring's application context scans the packages below `@SpringBootApplication`, creates beans and injects them through constructors.
- JPA entities map the four tables of the shop: generated and text primary keys, `@ManyToOne` and `@OneToMany` relations, lazy loading, the table `orders`, and converters that store the same lowercase words as Laravel.
- `LaundryRules` holds the price rule and the workflow in plain Java; its check prints exactly what Chapter 8's PHP check prints.
- Spring Data JPA repositories are interfaces; their queries come from `JpaRepository` and from method names, and `@EntityGraph` loads relations with the main query.
- `@RestController` classes map URLs to methods, read path variables, query parameters and JSON bodies, validate records with Jakarta Validation and answer records as snake_case JSON with the same field names as the Laravel API.
- A `CommandLineRunner` bean loads the price list from `Service.csv` and the demo customers at start-up, as Laravel's seeders do.

11.14 Exercises

1. Write the Spring Data method names for these queries, and say what each returns: (a) the customer with a given phone; (b) all orders of one customer, newest first; (c) the number of finished orders; (d) whether any order uses a given service; (e) the active services whose description contains a given word, ignoring case.
2. Add `GET /api/customers/{id}/orders`, which returns a customer's orders, newest first, as a list of `OrderView`, and 404 if the customer does not exist. Write the controller and the repository method it needs.
3. `OrderRepository.findAllByOrderByReceivedAtDescIdDesc` carries `@EntityGraph(attributePaths = {"customer", "service"})`. How many SQL queries does `GET /api/orders` send for a page of twenty orders with and without the entity graph, counting the payments that `OrderView.of` needs through `paid()`? What does `@BatchSize(size = 50)` on `payments` change?
4. Work out by hand, with `LaundryRules.total`, the totals of: 1.2 kg of "Ironing only" (Rp 6.000 per kg) without pickup; 3 shirts of "Dry cleaning - shirt/blouse" (Rp 20.000 per item) with pickup; 7.25 kg of "Curtain" (Rp 15.000 per kg) with pickup. Show the hundredths, the billed quantity, the subtotal and the fee.
5. Explain what happens, step by step, when the JSON `{"customer_id": 1, "service_id": "S0001", "quantity": 2.555}` is sent to `POST /api/orders`: which part reads it, which check fails, and what the answer is.

6. For each of these Laravel pieces, name the Spring Boot piece of this chapter that does the same job, and one difference: a migration, a seeder, a form request, an API resource, `Order::with('customer')`.

11.15 Solutions to the Exercises

1. (a) `Optional<Customer> findByPhone(String phone)`, one customer or none. (b) `List<Order> findByCustomerIdOrderByReceivedAtDesc(Long customerId)`; `CustomerId` walks from `Order.customer` to its `id`, and the list may be empty. (c) `long countByStatus(OrderStatus status)`, a number. (d) `boolean existsByServiceId(String serviceId)`. (e) `List<Service> findByActiveTrueAndDescriptionContainingIgnoreCase(String word)`; `Containing` adds the `%` wildcards on both sides and `IgnoreCase` compares in one case [9].

2. Add to `OrderRepository` the method of 1(b) with an entity graph, so that the views can read the customer and the service: `@EntityGraph(attributePaths = {"customer", "service"}) List<Order> findByCustomerIdOrderByReceivedAtDesc(Long customerId);`. **Listing 11.20** is the controller. It checks first that the customer exists, because an empty list would otherwise hide a mistyped id, and it reads inside a read-only transaction because `OrderView.of` reads the payments through `paid()`.

Listing 11.20 `CustomerOrdersController.java`: a customer's orders, newest first, or 404.

```
package com.profitina.laundry.web;

import com.profitina.laundry.repository.CustomerRepository;
import com.profitina.laundry.repository.OrderRepository;
import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.server.ResponseStatusException;

/** Exercise 11.2: GET /api/customers/{id}/orders, newest first. */
@RestController
public class CustomerOrdersController {

    private final CustomerRepository customers;
    private final OrderRepository orders;

    public CustomerOrdersController(CustomerRepository customers,
                                    OrderRepository orders) {
        this.customers = customers;
        this.orders = orders;
    }

    @GetMapping("/api/customers/{id}/orders")
    @Transactional(readOnly = true)
    public List<OrderView> index(@PathVariable Long id) {
        if (!customers.existsById(id)) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND,
                    "No customer " + id + ".");
        }
        return orders.findByCustomerIdOrderByReceivedAtDesc(id).stream()
                .map(OrderView::of).toList();
    }
}
```

3. Without the entity graph: one query for the page of orders, one to count all orders for `meta.total`, then for each of the twenty orders one query for its customer, one for its service and one for its payments, 2 + 60 = 62 queries (fewer if several orders share a customer or service already loaded in the same transaction). With the entity graph, customers and services arrive with the orders, leaving 2 + 20 = 22. `@BatchSize(size = 50)` makes Hibernate load the payments of up to fifty orders in one query the first time any of them is needed, so the twenty payment queries become one: 3 queries in all.

4. Ironing: 1.2 kg is 120 hundredths, below the minimum, so 300 are billed; 6000 × 300 = 1 800 000, plus 50, divided by 100 gives 18 000; no pickup; total Rp 18.000. Shirts: 3 items are 300 hundredths, already whole; 20000 × 300/100 = 60 000; pickup below Rp 100.000 adds 10 000; total Rp 70.000. Curtain: 7.25 kg is 725 hundredths, above the minimum; 15000 × 725 = 10 875 000, plus 50 and divided by 100 gives 108 750 (the remainder of 50 is dropped by the integer division); the subtotal is at least Rp 100.000, so pickup is free; total Rp 108.750.

5. Tomcat receives the request and hands it to the DispatcherServlet, which finds `OrderController.store` by the URL and the method. Jackson reads the body into `CreateOrderRequest`: `customer_id` becomes `customerId`, `quantity` becomes the `BigDecimal` 2.555, `pickup` is missing and becomes `false`. `@Valid` then checks the annotations: `@Digits(integer = 3, fraction = 2)` fails, because 2.555 has three decimals; the other rules pass, and `isAddressGivenForPickup()` is true because there is no pickup. Spring throws a `MethodArgumentNotValidException` before the method body runs, and answers 400 Bad Request with a problem-details body. No order is created. (Chapter 12 changes the answer to 422 with a Laravel-style `errors` object.)

6. A migration: Hibernate's `ddl-auto` in this chapter; unlike a migration it keeps no history and cannot move data, so real projects use Flyway or Liquibase. A seeder: the `DemoData CommandLineRunner`; it runs at every start, not on demand, so it checks first whether the data is there. A form request: a record with Jakarta Validation annotations and `@Valid`; checks that need the database are written in the controller (Chapter 12 collects them in one place). An API resource: a response record such as `OrderView`; the field names come from the naming strategy instead of being written as array keys. `Order::with('customer')`: `@EntityGraph(attributePaths = {"customer"})` on a repository method; it is fixed on the method instead of chosen at each call.

References

[1] Spring, "Spring Boot 4.0.0 available now," spring.io blog, Nov. 20, 2025. Accessed: Sep. 25, 2026. [Online]. Available: https://spring.io/blog/2025/11/20/spring-boot-4-0-0-available-now/

[2] Spring, "Spring Boot 4.1.1 available now," spring.io blog, Aug. 20, 2026. Accessed: Sep. 25, 2026. [Online]. Available: https://spring.io/blog/2026/08/20/spring-boot-4-1-1-available-now/

[3] Spring, "System Requirements," *Spring Boot 4.1.1 Reference Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-boot/system-requirements.html

[4] OpenJDK, "JDK 25," openjdk.org. Accessed: Sep. 25, 2026. [Online]. Available: https://openjdk.org/projects/jdk/25/

[5] Hibernate, "Hibernate ORM documentation." Accessed: Sep. 25, 2026. [Online]. Available: https://hibernate.org/orm/documentation/

[6] Spring, "The IoC Container," *Spring Framework Reference Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans.html

[7] Spring, “Modularizing Spring Boot,” spring.io blog, Oct. 28, 2025. Accessed: Sep. 25, 2026. [Online]. Available: https://spring.io/blog/2025/10/28/modularizing-spring-boot/

[8] Eclipse Foundation, “Jakarta Persistence 3.2,” *Jakarta EE Specifications*. Accessed: Sep. 25, 2026. [Online]. Available: https://jakarta.ee/specifications/persistence/3.2/

[9] Spring, “JPA Query Methods,” *Spring Data JPA Reference Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

[10] Spring, “Annotated Controllers,” *Spring Framework Reference Documentation: Web on Servlet Stack*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller.html

[11] Eclipse Foundation, “Jakarta Validation 3.1,” *Jakarta EE Specifications*. Accessed: Sep. 25, 2026. [Online]. Available: https://jakarta.ee/specifications/bean-validation/3.1/

[12] Spring, “Spring Initializr.” Accessed: Sep. 25, 2026. [Online]. Available: https://start.spring.io

[13] H2 Database Engine, “H2 Database Engine.” Accessed: Sep. 25, 2026. [Online]. Available: https://www.h2database.com/

[14] Spring, “JSON,” *Spring Boot Reference Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-boot/reference/features/json.html

[15] M. Nottingham, E. Wilde, and S. Dalal, “Problem Details for HTTP APIs,” RFC 9457, IETF, Jul. 2023. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9457

Chapter 12

Evaluation and Discussion III: One REST API in Laravel and Spring Boot

Chapters 9 to 11 built two APIs for the same laundry. The Laravel API of Chapter 10 has sign-in tokens, form requests, resources and tests; the Spring Boot API of Chapter 11 has entities, repositories, records and the same rules in plain Java. Raka's phone app should be able to talk to either of them without knowing which one answers. This chapter checks whether that is true, finds the places where it is not, and fixes them, one problem at a time, in both frameworks. By the end of it you will be able to write an API contract precise enough to be checked by a program; turn every difference between two implementations into a decision recorded in that contract; make Laravel and Spring Boot agree on the envelope of an answer, the format of dates, the shape of errors, the status codes of refusals, payments and a daily report; and check any server against the contract with a small PHP program whose run you can reproduce on your own computer.

12.1 Where We Left Off

Chapter 9 gave the Laravel application migrations that change tables holding data, factories and seeders that fill a laptop with a believable month, the `EnsureOrderIsOpen` middleware that closes departed orders, and form requests such as `StoreOrderRequest` and `StorePaymentRequest`. Chapter 10 added staff sign-in, a gate for the price list, an API behind Sanctum tokens, resources that shape its JSON, and unit and feature tests. Chapter 11 built the same shop in Spring Boot: entities for customers, services, orders and payments; Spring Data repositories; REST controllers that read and write records; and `LaundryRules`, whose check prints exactly what Chapter 8's PHP check prints.

The rules are the same in both, down to the rupiah. The APIs are not. Chapter 10's resources wrap every answer in `data`, while Chapter 11's controllers return bare objects and lists. Chapter 10 sends times the way Laravel serializes them by default, in UTC with microseconds, while Chapter 11 sends Java's local date-time. Laravel answers a broken rule with 422 and a list of errors per field; Spring answers with 400 and a problem-details object, or with 409 for a refused status change. Spring has no payment endpoint and neither has a daily report for programs. None of these is a bug in one framework. Each is a decision that nobody has made yet, and a phone app cannot work with two answers to the same question.

12.2 How to Approach an API Problem

An API problem is different from an algorithm problem: the hard part is rarely the code, and almost always the agreement. Every problem of this chapter is worked in the same seven steps.

1. **State the rule of the contract** in one or two sentences, with the exact URL, fields and status codes.
2. **Work an example by hand**: one request and the exact answer the contract requires, byte for byte where it matters.

3. **Compare the two implementations** against the example, and write down every difference.
4. **Decide** what the contract says where they differ, and why. Prefer the answer that is easier for the client, that follows an HTTP standard, and that the other framework can produce without fighting itself.
5. **Change each implementation** as little as possible to keep the decision.
6. **Check** with a test in each framework and with the contract checker of Section 12.11.
7. **Look at the tempting wrong solution**, the change that seems to fix the problem but breaks something else, and see why it fails.

Figure 12.1 shows the arrangement that these steps build. The contract sits in the middle; the Laravel application, the Spring Boot application and a small reference server written in plain PHP each implement it; and one checker program verifies all three the same way.

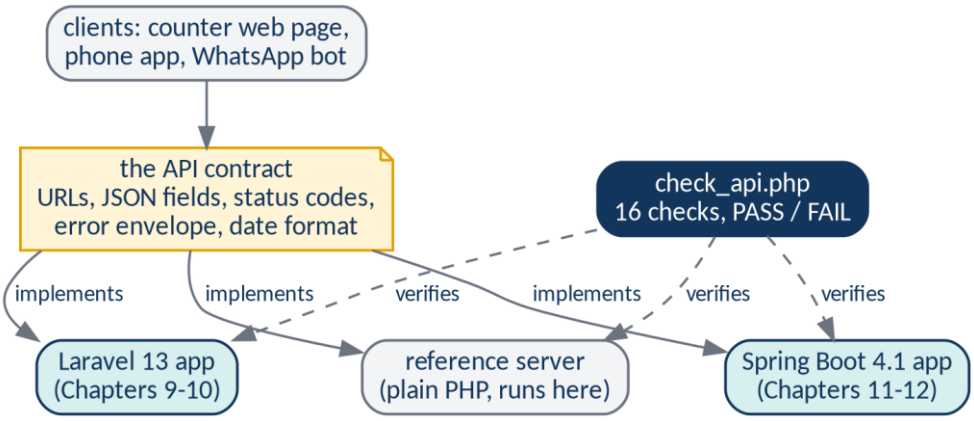


Figure 12.1 One contract, three implementations and one checker: the clients depend only on the contract, and `check_api.php` verifies every server against it.

The reference server deserves a word. The environment in which this book was written could not install Laravel or Spring Boot, as each chapter's `README.md` explains, but it could run PHP. So the chapter's `contract/reference/server.php` implements the whole contract in about 240 lines of plain PHP with SQLite, using the very rule files of Chapter 8, and the checker's output in Section 12.11 is the real output of a run against it. On your own computer, the same checker runs against the Laravel and Spring applications.

12.3 The Contract

Table 12.1 lists the endpoints of the contract. Every URL starts with `/api`; every request and answer body is JSON; every answer that carries one resource or one list wraps it in a top-level `data` key.

Table 12.1 The endpoints of the Profitina Laundry API contract; a question mark marks an optional field.

Method and URL	Body	Success
`GET /api/services`		200, `data`: active services, by code
`POST /api/customers`	`name`, `phone`, `address`?	201, `data`: the customer
`GET /api/orders?status=&page=`		200, `data`: orders, `meta`: page numbers
`POST /api/orders`	`customer_id`, `service_id`, `quantity`, `pickup`, `delivery_address`?, `note`?	201, `data`: the order
`GET /api/orders/{id}`		200, `data`: the order with `payments`
`PATCH /api/orders/{id}/status`	`status`	200, `data`: the order
`POST /api/orders/{id}/payments`	`amount`, `method`	201, `data`: the order with `payments`
`GET /api/reports/daily?date=YYYY-MM-DD`		200, `data`: the day's figures

The rules for the content of the answers are as short as possible, so that they can be kept.

- **Names.** Field names are snake_case (`delivery_address`). Status and method values are lowercase words (`queued`, `finished`, `departed`; `cash`, `qris`, `transfer`).
- **Numbers.** Money is a whole number of rupiah. Quantities are numbers, not text, with at most two decimals.
- **Times.** Every date-time is local shop time, written `YYYY-MM-DDTHH:MM:SS`, such as `2026-09-25T08:15:00`, and a time that has not happened yet is `null`, never missing.
- **An order** carries at least `id`, `customer` (`id`, `name`, `phone`), `service` (`id`, `name`, `unit`, `price`), `quantity`, `pickup`, `delivery_address`, `total`, `paid`, `balance`, `status`, `note`, `received_at`, `finished_at` and `departed_at`, where the balance is defined by Equation (12.1). A client must ignore fields it does not know, so an implementation may send more, as Laravel's `outlet_id`.

$$balance = total - \sum_{p \in P} amount_p \tag{12.1}$$

Here P is the set of payments recorded for the order and $amount_p$ is the amount of payment p. Because no payment may exceed what is still owed (Problem 12.6), the balance always lies between 0 and the total.

Refusals follow **Table 12.2**. The body of every refusal has a `message`; the body of a 422 also has `errors`, an object whose keys are the field names of the request and whose values are lists of messages. This is the shape that Laravel's validation already produces, and the reason for choosing it is simple: a client can show each message under the field it belongs to.

Table 12.2 The refusals of the contract, and the body each one carries.

Status	When	Body
401	no valid token (Laravel; Spring from Chapter 14)	`message`
404	the order, service or customer does not exist	`message`
409	the order has departed and is closed	`message`
422	the input breaks a rule, including a refused status step	`message`, `errors`

12.4 Problem 12.1: The Price List for Programs

12.4.1 The problem

`GET /api/services` answers 200 with `{"data": [...]}`: every active service, ordered by code, each with `id`, `name`, `unit` and `price`.

12.4.2 Understanding it by hand

The first line of `Service.csv` is `S0001,"Wash, dry & fold (per kg)",8000.00`. The contract's first element of `data` is therefore `{"id": "S0001", "name": "Wash, dry & fold", "unit": "kg", "price": 8000}`: the name without the part in brackets, the unit taken from it, the price as a whole number.

12.4.3 Comparing the two

The Laravel API of Chapter 10 has no services endpoint at all; its `ServiceResource` exists, but only inside orders. The Spring API of Chapter 11 has one, but it answers with a bare list `[...]` instead of `{"data": [...]}`. The decision: the endpoint exists in both, and wraps the list in `data`, as every other answer of the contract does.

12.4.4 The solution in both frameworks

In Laravel, **Listing 12.1** is a two-method controller that reuses the resource of Chapter 10; `ServiceResource::collection` wraps the list in `data` by itself.

Listing 12.1 `Api/ServiceController.php` (Laravel): the active price list as a resource collection.

```php
<?php

namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use App\Http\Resources\ServiceResource;
use App\Models\Service;

class ServiceController extends Controller
{
    // GET /api/services: the active price list, S0001 first.
    public function index()
    {
        return ServiceResource::collection(
            Service::where('is_active', true)->orderBy('id')->get());
    }

    public function show(Service $service)
    {
        return new ServiceResource($service);
    }
}
```

Listing 12.2 is the API routes file of this chapter: Chapter 10's routes plus the services and the daily report of Problem 12.7.

Listing 12.2 `routes/api.php` (Laravel): the routes of Chapter 10, the price list and the daily report.

```php
<?php

use App\Http\Controllers\Api\CustomerController;
use App\Http\Controllers\Api\OrderController;
use App\Http\Controllers\Api\PaymentController;
use App\Http\Controllers\Api\ReportController;
use App\Http\Controllers\Api\ServiceController;
use App\Http\Controllers\Api\TokenController;
use Illuminate\Support\Facades\Route;

// Every URL here starts with /api (Laravel adds the prefix).
Route::post('tokens', [TokenController::class, 'store'])
    ->middleware('throttle:5,1');

Route::middleware('auth:sanctum')->group(function () {
    Route::delete('tokens/current', [TokenController::class, 'destroy']);

    Route::get('services', [ServiceController::class, 'index']);
    Route::get('services/{service}', [ServiceController::class, 'show']);

    Route::get('customers', [CustomerController::class, 'index']);
    Route::post('customers', [CustomerController::class, 'store']);

    Route::get('orders', [OrderController::class, 'index']);
    Route::post('orders', [OrderController::class, 'store']);
    Route::get('orders/{order}', [OrderController::class, 'show']);

    Route::middleware('order.open')->group(function () {
        Route::patch('orders/{order}/status',
            [OrderController::class, 'updateStatus']);
        Route::post('orders/{order}/payments',
            [PaymentController::class, 'store']);
    });

    Route::get('reports/daily', [ReportController::class, 'daily']);
});
```

In Spring there is no automatic wrapper, so **Listing 12.3** writes one: a generic record with a single component, which Jackson writes as `{"data": ...}`.

Listing 12.3 `DataView.java` (Spring): the contract's envelope in one line.

```
package com.profitina.laundry.web;

/** The contract wraps every answer in {"data": ...}, as Laravel does. */
public record DataView<T>(T data) {
}
```

Listing 12.4 uses it; the only change from Chapter 11 is the return type and the `new DataView<>(...)` around the result.

Listing 12.4 `ServiceController.java` (Spring): the same list, now in the envelope.

```
package com.profitina.laundry.web;

import com.profitina.laundry.repository.ServiceRepository;
import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.server.ResponseStatusException;

@RestController
@RequestMapping("/api/services")
public class ServiceController {

    private final ServiceRepository services;

    public ServiceController(ServiceRepository services) {
        this.services = services;
    }

    @GetMapping
    public DataView<List<ServiceView>> index() {
        return new DataView<>(services.findByActiveTrueOrderByIdAsc()
                .stream().map(ServiceView::of).toList());
    }

    @GetMapping("/{id}")
    public DataView<ServiceView> show(@PathVariable String id) {
        return new DataView<>(services.findById(id).map(ServiceView::of)
                .orElseThrow(() -> new ResponseStatusException(
                        HttpStatus.NOT_FOUND, "No service " + id + ".")));
    }
}
```

`CustomerController` and `OrderController` of the Spring application wrap their answers in `DataView` in the same way; `PageView` needs no wrapper, because it already has `data` and `meta`.

12.4.5 A tempting wrong solution

Returning the models themselves, `Service::all()` in Laravel or `services.findAll()` in Spring, gives an answer in a few characters. It also sends `description` and `is_active` (and in Laravel `created_at` and `updated_at`), omits `name` and `unit`, which are computed, and ties every phone app to the columns of today's database. The day the owner splits the description into two columns, every client breaks. An answer is a promise; it should be written on purpose, in a resource or a record.

12.5 Problem 12.2: One Way to Write a Time

12.5.1 The problem

Every date-time in an answer is local shop time, written `YYYY-MM-DDTHH:MM:SS`, and a time that has not happened is `null`.

12.5.2 Understanding it by hand

Sari's order was received on September 25, 2026 at 08:15 by the clock on the counter in Surabaya. The contract's `received_at` is `2026-09-25T08:15:00`. Western Indonesian Time is seven hours ahead of UTC, so the same moment in UTC is 01:15.

12.5.3 Comparing the two

Chapter 10's `OrderResource` passes the Carbon object to the JSON encoder, which writes Laravel's default: ISO 8601 in UTC with microseconds, `2026-09-25T01:15:00.000000Z` (Exercise 10.6). Chapter 11's `OrderView` passes a `LocalDateTime` to Jackson, which writes the local time without a zone; for a time with zero seconds and no fraction, Java's own `toString()` would even write `2026-09-25T08:15`, without the seconds. Both are valid ISO 8601, and they describe the same moment, but a client comparing strings, or a person reading them, sees three different values.

The decision is local time without a zone, with seconds, and nothing else. The shop's day, its reports and its receipts all live in Surabaya time; a client in the shop never needs to convert. A system that spans time zones would choose UTC with an explicit offset instead; what matters is that the contract says which one.

12.5.4 The solution in both frameworks

Laravel's resource formats each time itself. **Listing 12.5** defines the format once, as a constant, and uses the null-safe operator `?->` so that a missing time becomes `null`. The application's time zone must be `Asia/Jakarta` in `config/app.php`, as every chapter's `README.md` asks, so that the Carbon objects are in shop time before they are formatted.

Listing 12.5 `OrderResource.php` (Laravel): every time in the contract's format.

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class OrderResource extends JsonResource
{
    // The contract's one date-time format: 2026-09-25T08:15:00, local time.
    public const TIME = 'Y-m-d\TH:i:s';

    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'customer' => new CustomerResource($this->whenLoaded('customer')),
            'service' => new ServiceResource($this->whenLoaded('service')),
            'outlet_id' => $this->outlet_id,
            'quantity' => (float) $this->quantity,
            'pickup' => $this->pickup,
            'delivery_address' => $this->delivery_address,
            'total' => $this->total,
            'paid' => $this->paid(),
            'balance' => $this->balance(),
            'status' => $this->status->value,
            'note' => $this->note,
            'received_at' => $this->received_at->format(self::TIME),
            'finished_at' => $this->finished_at?->format(self::TIME),
            'departed_at' => $this->departed_at?->format(self::TIME),
            'payments' => PaymentResource::collection(
                $this->whenLoaded('payments')),
        ];
    }
}
```

`PaymentResource` uses the same constant for `paid_at`. In Spring, the format is fixed per field with Jackson's `@JsonFormat`, as **Listing 12.6** shows; the same record also gains the payment list that Problem 12.6 needs.

Listing 12.6 `OrderView.java` (Spring): times with a fixed pattern, and the payments only when asked for.

```
package com.profitina.laundry.web;

import com.fasterxml.jackson.annotation.JsonFormat;
import com.fasterxml.jackson.annotation.JsonInclude;
import com.profitina.laundry.domain.Order;
import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.List;

/** The JSON of one order, exactly as the contract of Chapter 12 says. */
public record OrderView(
        Long id,
        CustomerView customer,
        ServiceView service,
        BigDecimal quantity,
        boolean pickup,
        String deliveryAddress,
        long total,
        long paid,
        long balance,
        String status,
        String note,
        @JsonFormat(pattern = TIME) LocalDateTime receivedAt,
        @JsonFormat(pattern = TIME) LocalDateTime finishedAt,
        @JsonFormat(pattern = TIME) LocalDateTime departedAt,
        @JsonInclude(JsonInclude.Include.NON_NULL)
        List<PaymentView> payments) {

    /** The contract's one date-time format: 2026-09-25T08:15:00. */
    static final String TIME = "yyyy-MM-dd'T'HH:mm:ss";

    /** Without the payment list, like Laravel when it is not loaded. */
    public static OrderView of(Order o) {
        return build(o, null);
    }

    public static OrderView withPayments(Order o) {
        return build(o, o.getPayments().stream().map(PaymentView::of)
                .toList());
    }
```

Listing 12.6 (Continued) *OrderView.java (Spring): times with a fixed pattern, and the payments only when asked for.*

```
    private static OrderView build(Order o, List<PaymentView> payments) {
        return new OrderView(o.getId(), CustomerView.of(o.getCustomer()),
                ServiceView.of(o.getService()), o.getQuantity(),
                o.isPickup(), o.getDeliveryAddress(), o.getTotal(), o.paid(),
                o.balance(), o.getStatus().value(), o.getNote(),
                o.getReceivedAt(), o.getFinishedAt(), o.getDepartedAt(),
                payments);
    }
}
```

`'T'` in the pattern is quoted because it is a literal letter, not a code for a date field. `TIME` is a constant of the record so that `PaymentView` can use the same pattern. Because the controllers truncate every time to whole seconds before it is stored, no fraction of a second can appear.

12.5.5 A tempting wrong solution

Changing Laravel's `config/app.php` time zone to UTC "to match Spring" seems to align the two, and makes everything worse: the report of Problem 12.7 would then cut the day at 07:00 Surabaya time, and every order received between midnight and seven in the morning would be

counted on the wrong day. Time zones are chosen for people, and the format is chosen for programs; do not change the first to fix the second.

12.6 Problem 12.3: One Shape for Every Error

12.6.1 The problem

A request that breaks a rule is answered with 422 and `{"message": "...", "errors": {"field": ["..."]}}`, where the field names are those of the request, in snake_case. Any other refusal has at least a `message`.

12.6.2 Understanding it by hand

`POST /api/customers` with `{"name": "S", "phone": "12345"}` must be answered with 422 and an `errors` object with the keys `name` (too short) and `phone` (not a mobile number). The order of the keys does not matter; their names do.

12.6.3 Comparing the two

Laravel already answers this way: it is the framework's own format for validation errors. Chapter 11's Spring application, with problem details switched on, answers 400 with a problem-details body such as `{"type": "about:blank", "title": "Bad Request", "status": 400, "detail": "Invalid request content."}`. The information about which field failed is not in the body at all, and if it were, the field would be called by its Java name, `serviceId` rather than `service_id`. Its `ResponseStatusException`s are also answered as problem details, with the reason in `detail` rather than `message`.

The decision follows Laravel. Its shape carries the information a client needs, and it is also what most PHP and JavaScript clients already expect. The HTTP standard defines 422 Unprocessable Content for exactly this case, a request that is well formed but whose content cannot be processed [1]; 400 remains for a body that is not valid JSON at all.

12.6.4 The solution in Spring

Two classes do the work. **Listing 12.7** is an exception that carries a map of errors, for the rules that annotations cannot express, such as "this phone is already registered".

Listing 12.7 `ApiValidationException.java` (Spring): a broken rule, with the messages per field.

```
package com.profitina.laundry.web;

import java.util.List;
import java.util.Map;

/** Input that breaks a rule: becomes 422 with {"errors": {...}}. */
public class ApiValidationException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    private final transient Map<String, List<String>> errors;

    public ApiValidationException(Map<String, List<String>> errors) {
        super("The given data was invalid.");
        this.errors = errors;
    }

    public ApiValidationException(String field, String message) {
        this(Map.of(field, List.of(message)));
    }

    public Map<String, List<String>> getErrors() {
        return errors;
    }
}
```

Listing 12.8 turns every error into the contract's shape. A class marked `@RestControllerAdvice` applies its `@ExceptionHandler` methods to exceptions thrown by any controller, before Spring's default handling [2].

Listing 12.8 `ApiExceptionHandler.java` (Spring): validation errors, broken rules, status exceptions and unreadable bodies, each in the contract's shape.

```java
package com.profitina.laundry.web;

import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.server.ResponseStatusException;

/** Every error in the contract's shape: {"message": ..., "errors": ...}. */
@RestControllerAdvice
public class ApiExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, Object>> invalid(
            MethodArgumentNotValidException e) {
        Map<String, List<String>> errors = new LinkedHashMap<>();
        for (FieldError f : e.getBindingResult().getFieldErrors()) {
            errors.computeIfAbsent(snake(f.getField()),
                    k -> new ArrayList<>()).add(f.getDefaultMessage());
        }
        return unprocessable(errors);
    }

    @ExceptionHandler(ApiValidationException.class)
    public ResponseEntity<Map<String, Object>> rule(
            ApiValidationException e) {
        return unprocessable(e.getErrors());
    }

    @ExceptionHandler(ResponseStatusException.class)
    public ResponseEntity<Map<String, Object>> status(
            ResponseStatusException e) {
        String reason = e.getReason() == null ? "" : e.getReason();
        return ResponseEntity.status(e.getStatusCode())
                .body(Map.of("message", reason));
    }
```

Listing 12.8 (Continued) *ApiExceptionHandler.java (Spring): validation errors, broken rules, status exceptions and unreadable bodies, each in the contract's shape.*

```
    @ExceptionHandler(HttpMessageNotReadableException.class)
    public ResponseEntity<Map<String, Object>> unreadable(
            HttpMessageNotReadableException e) {
        return ResponseEntity.badRequest().body(Map.of("message",
                "The request body is not valid JSON for this endpoint."));
    }

    private static ResponseEntity<Map<String, Object>> unprocessable(
            Map<String, List<String>> errors) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("message", "The given data was invalid.");
        body.put("errors", errors);
        return ResponseEntity.status(HttpStatus.UNPROCESSABLE_CONTENT)
                .body(body);
    }

    /** customerId -> customer_id: the name the client sent. */
    static String snake(String name) {
        return name.replaceAll("([a-z0-9])([A-Z])", "$1_$2").toLowerCase();
    }
}
```

`invalid` handles the exception that `@Valid` throws. It walks the field errors, converts each field name to snake_case with the regular expression in `snake`, which puts an underscore between a lowercase letter or digit and the capital that follows it, and collects the messages per field in a `LinkedHashMap`, which keeps them in the order they were found. `rule` handles our own exception, `status` gives every `ResponseStatusException` a body with its reason as `message`, and `unreadable` answers a body that is not JSON with 400. `unprocessable` builds the 422 body once for both validation paths. Because this advice handles the exceptions itself, the problem-details setting of Chapter 11 no longer shapes these answers; it stays on for the errors this class does not handle.

The customer controller now answers a duplicate phone with `new ApiValidationException("phone", "The phone has already been taken.")`, the same message as Laravel's `unique` rule, instead of Chapter 11's 409.

12.6.5 A tempting wrong solution

Configuring Jackson to write Java names in snake_case (as Chapter 11 does) and hoping that the error field names follow is the tempting shortcut. They do not: the names in a `FieldError` are Java property paths, produced by the validator, not by Jackson, and they stay `serviceId`. The conversion has to happen where the error body is built.

12.7 Problem 12.4: Every Error at Once

12.7.1 The problem

`POST /api/orders` with several broken rules answers 422 with all of them in `errors` at once, including the rule that involves two fields: when `pickup` is true, `delivery_address` is required.

12.7.2 Understanding it by hand

`{"quantity": 0, "pickup": true}` breaks four rules: no customer, no service, a quantity below 0.1, and pickup without an address. The contract requires `errors` to contain at least `customer_id`,

`service_id`, `quantity` and `delivery_address`. (Laravel also names `outlet_id`, its extra required field; a client ignores keys it does not know.)

12.7.3 Comparing the two

Laravel's `StoreOrderRequest` checks all its rules together, `required_if_accepted:pickup` among them, and reports every failure: it already keeps the contract. Chapter 11's Spring version fails in two ways. Its two-field rule is an `@AssertTrue` method, `isAddressGivenForPickup()`, and the validator reports it under the property name `addressGivenForPickup`, which becomes `address_given_for_pickup`, a key no client knows. And its checks that need the database, "the customer exists" and "the service is offered", run in the controller only after `@Valid` has passed, so a request with both kinds of mistakes gets its errors in two rounds.

12.7.4 The solution in Spring

Listing 12.9 removes the `@AssertTrue` method; the record keeps only the rules that concern one field.

Listing 12.9 `CreateOrderRequest.java` (Spring): one-field rules only.

```
package com.profitina.laundry.web;

import jakarta.validation.constraints.DecimalMax;
import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Digits;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;
import java.math.BigDecimal;

/** The pickup-needs-an-address rule now lives in OrderController. */
public record CreateOrderRequest(
        @NotNull Long customerId,
        @NotBlank String serviceId,
        @NotNull @DecimalMin("0.1") @DecimalMax("100")
        @Digits(integer = 3, fraction = 2) BigDecimal quantity,
        boolean pickup,
        @Size(max = 200) String deliveryAddress,
        @Size(max = 200) String note) {
}
```

The controller's `store` method, the first part of **Listing 12.10**, no longer puts `@Valid` on its parameter. It asks the Jakarta `Validator`, injected through the constructor, to validate the record, turns each violation's property path into a snake_case key, and then adds its own checks to the same map: the customer, the service and the delivery address. Only if the map is still empty does it create the order; otherwise it throws one `ApiValidationException` with every error in it.

Listing 12.10 `OrderController.java` (Spring): every error at once, the workflow as 422, closed orders as 409, and payments.

```java
package com.profitina.laundry.web;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.domain.LaundryRules;
import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.domain.PaymentMethod;
import com.profitina.laundry.domain.Service;
import com.profitina.laundry.repository.CustomerRepository;
import com.profitina.laundry.repository.OrderRepository;
import com.profitina.laundry.repository.PaymentRepository;
import com.profitina.laundry.repository.ServiceRepository;
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Valid;
import jakarta.validation.Validator;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.http.HttpStatus;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.server.ResponseStatusException;

@RestController
@RequestMapping("/api/orders")
public class OrderController {

    private final OrderRepository orders;
    private final CustomerRepository customers;
    private final ServiceRepository services;
    private final PaymentRepository payments;
    private final Validator validator;
```

Listing 12.10 (Continued) *OrderController.java (Spring): every error at once, the workflow as 422, closed orders as 409, and payments.*

```
    public OrderController(OrderRepository orders,
                           CustomerRepository customers,
                           ServiceRepository services,
                           PaymentRepository payments, Validator validator) {
        this.orders = orders;
        this.customers = customers;
        this.services = services;
        this.payments = payments;
        this.validator = validator;
    }

    @GetMapping
    @Transactional(readOnly = true)
    public PageView<OrderView> index(
            @RequestParam(required = false) String status,
            @RequestParam(defaultValue = "1") int page) {
        PageRequest pageable = PageRequest.of(Math.max(page, 1) - 1, 20);
        Page<Order> result = status == null
                ? orders.findAllByOrderByReceivedAtDescIdDesc(pageable)
                : orders.findByStatusOrderByReceivedAtDescIdDesc(
                        parseStatus(status), pageable);
        return PageView.of(result, OrderView::of);
    }

    @GetMapping("/{id}")
    @Transactional(readOnly = true)
    public DataView<OrderView> show(@PathVariable Long id) {
        return new DataView<>(OrderView.withPayments(find(id)));
    }

    /** Every broken rule is reported at once, as Laravel does. */
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    @Transactional
    public DataView<OrderView> store(@RequestBody CreateOrderRequest in) {
        Map<String, List<String>> errors = new LinkedHashMap<>();
        for (ConstraintViolation<CreateOrderRequest> v
                : validator.validate(in)) {
            add(errors, ApiExceptionHandler.snake(
                    v.getPropertyPath().toString()), v.getMessage());
        }
        Customer customer = in.customerId() == null ? null
                : customers.findById(in.customerId()).orElse(null);
        if (in.customerId() != null && customer == null) {
            add(errors, "customer_id", "The selected customer is invalid.");
        }
```

Listing 12.10 (Continued) *OrderController.java (Spring): every error at once, the workflow as 422, closed orders as 409, and payments.*

```
        Service service = in.serviceId() == null ? null
                : services.findById(in.serviceId())
                        .filter(Service::isActive).orElse(null);
        if (in.serviceId() != null && service == null) {
            add(errors, "service_id", "The selected service is invalid.");
        }
        if (in.pickup() && (in.deliveryAddress() == null
                || in.deliveryAddress().isBlank())) {
            add(errors, "delivery_address",
                    "Where should the laundry be delivered?");
        }
        if (!errors.isEmpty()) {
            throw new ApiValidationException(errors);
        }
        Order order = new Order(customer, service, in.quantity(),
                in.pickup(), in.deliveryAddress(), in.note(), now());
        return new DataView<>(OrderView.of(orders.save(order)));
    }

    @PatchMapping("/{id}/status")
    @Transactional
    public DataView<OrderView> updateStatus(@PathVariable Long id,
            @Valid @RequestBody StatusRequest in) {
        Order order = find(id);
        ensureOpen(order);
        OrderStatus wanted = parseStatus(in.status());
        OrderStatus next;
        try {
            next = LaundryRules.advance(order.getStatus(), order.balance());
        } catch (IllegalStateException e) {
            throw new ApiValidationException("status", e.getMessage());
        }
        if (wanted != next) {
            throw new ApiValidationException("status", "An order that is "
                    + order.getStatus().value() + " can only become "
                    + next.value() + ".");
        }
        order.advance(now());
        return new DataView<>(OrderView.of(order));
    }
```

Listing 12.10 (Continued) *OrderController.java (Spring): every error at once, the workflow as 422, closed orders as 409, and payments.*

```
    @PostMapping("/{id}/payments")
    @ResponseStatus(HttpStatus.CREATED)
    @Transactional
    public DataView<OrderView> pay(@PathVariable Long id,
            @Valid @RequestBody CreatePaymentRequest in) {
        Order order = find(id);
        ensureOpen(order);
        PaymentMethod method;
        try {
            method = PaymentMethod.fromValue(in.method());
        } catch (IllegalArgumentException e) {
            throw new ApiValidationException("method",
                    "The selected method is invalid.");
        }
        if (in.amount() > order.balance()) {
            throw new ApiValidationException("amount", "Only "
                    + LaundryRules.rupiah(order.balance())
                    + " is still owed.");
        }
        payments.save(order.addPayment(in.amount(), method, now()));
        return new DataView<>(OrderView.withPayments(order));
    }

    private Order find(Long id) {
        return orders.findWithDetailsById(id)
                .orElseThrow(() -> new ResponseStatusException(
                        HttpStatus.NOT_FOUND, "No order " + id + "."));
    }

    /** The job of Laravel's EnsureOrderIsOpen middleware. */
    private static void ensureOpen(Order order) {
        if (order.getStatus() == OrderStatus.DEPARTED) {
            throw new ResponseStatusException(HttpStatus.CONFLICT, "Order #"
                    + order.getId() + " has left the shop and is closed.");
        }
    }

    private static OrderStatus parseStatus(String value) {
        try {
            return OrderStatus.fromValue(value);
        } catch (IllegalArgumentException e) {
            throw new ApiValidationException("status",
                    "The selected status is invalid.");
        }
    }
```

Listing 12.10 (Continued) *OrderController.java (Spring): every error at once, the workflow as 422, closed orders as 409, and payments.*

```
    private static void add(Map<String, List<String>> errors, String field,
                            String message) {
        errors.computeIfAbsent(field, k -> new ArrayList<>()).add(message);
    }

    private static LocalDateTime now() {
        return LocalDateTime.now().truncatedTo(ChronoUnit.SECONDS);
    }
}
```

The same approach is how a Laravel form request works inside: rules first, then the `after` hooks, then one list of errors. Writing it by hand in Spring takes about fifteen lines, and it keeps the knowledge in one method.

12.7.5 A tempting wrong solution

Keeping `@Valid` and renaming the `@AssertTrue` method to `isDeliveryAddress()` so that its property becomes `deliveryAddress` looks clever. It collides with the real `deliveryAddress` component, confuses anyone reading the record, and still leaves the database checks in a second round. A rule about two fields and a rule that needs the database both belong in code that can see everything, and report into the same list.

12.8 Problem 12.5: Moving an Order, and Closed Orders

12.8.1 The problem

`PATCH /api/orders/{id}/status` with `{"status": "..."}` moves an order one step along the workflow of **Figure 12.2**. A step that the workflow does not allow, a skipped step or a departure with money owing, is answered with 422 and `errors.status`. Any change to a departed order, a status change or a payment, is answered with 409 and a `message`.

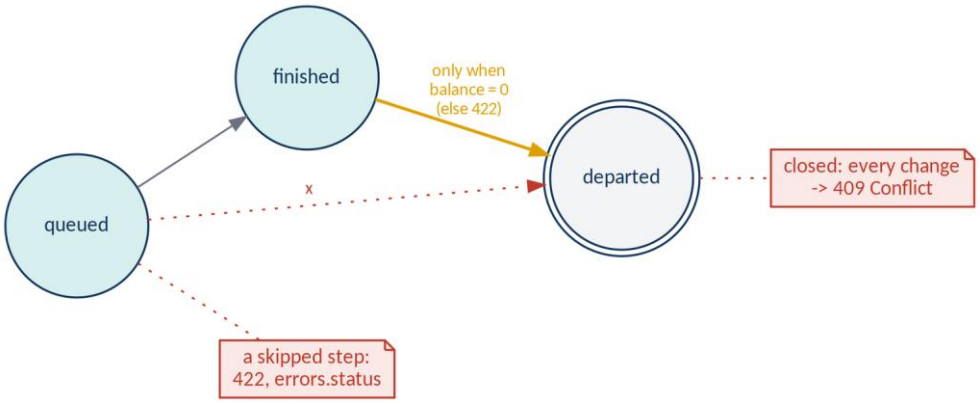


Figure 12.2 The order's workflow as the contract states it: one step at a time, departure only when the balance is 0, and a departed order closed to every change.

12.8.2 Understanding it by hand

A new order of Rp 34.000 is queued. `departed` skips `finished`: 422. `finished`: 200, and `finished_at` gets the current time. `departed` with Rp 34.000 owing: 422, with the workflow's message "Collect Rp 34.000 before handing over the laundry." After a payment of Rp 34.000, `departed`: 200. Then a payment of Rp 1.000: 409, because the order is closed.

12.8.3 Comparing the two

Laravel's API keeps this already: `UpdateOrderStatusRequest` turns the workflow's refusals into 422 errors on `status`, and Chapter 9's `order.open` middleware answers 409 for departed orders. Chapter 11's Spring API answers every refused step with 409, including the skipped step and the unpaid departure.

The two status codes mean different things, and the contract should keep them apart. 422 says "your request was understood but breaks a rule about its content": the client can fix the request, for example by collecting the payment first. 409 says "the resource is in a state that conflicts with any such request": nothing the client sends will change a departed order [1]. A phone app can then show 422 errors next to the button, and 409 as a notice that the order is closed and the screen should be refreshed.

12.8.4 The solution in Spring

In **Listing 12.10**, `updateStatus` first calls `ensureOpen`, the Spring counterpart of Chapter 9's middleware, which throws a `ResponseStatusException` with 409 for a departed order, with the same message as Laravel's. It then asks `LaundryRules.advance` for the next status, turning the rule's refusal into an `ApiValidationException` on `status`, and compares the next status with the one requested, again answering 422 if they differ. Only then does it call `order.advance`. Laravel needs no change.

12.8.5 A tempting wrong solution

Letting the client send only "advance", without naming the status, as the web counter's button does, avoids the comparison altogether. It also lets two drivers who press "picked up" at the same moment move a finished order to departed and then fail the second request with a confusing 409, or, worse, with a web form's "advance", move a queued order two steps. Naming the wanted status makes a request safe to repeat: the second identical request is refused instead of doing something new.

12.9 Problem 12.6: Payments

12.9.1 The problem

`POST /api/orders/{id}/payments` with `{"amount": ..., "method": "..."}` records a payment of at least Rp 1.000 by cash, QRIS or transfer, never more than the balance of Equation (12.1), and answers 201 with the whole order, its `payments` list included.

12.9.2 Understanding it by hand

The order of Problem 12.5 owes Rp 34.000. A payment of Rp 40.000 is refused with 422 and `errors.amount` ("Only Rp 34.000 is still owed."). A payment of Rp 34.000 by QRIS is recorded, and the answer shows `"balance": 0` and a `payments` list whose first element has `"method": "qris"` and an `id`.

12.9.3 Comparing the two

Laravel has the endpoint since Chapter 10, with Chapter 9's `StorePaymentRequest`. Spring has none. The decision is simply the Laravel behavior; the work is in Spring.

12.9.4 The solution in Spring

Listing 12.11 is the request record. The amount is a `Long`, not a `long`, so that a missing amount is `null` and fails `@NotNull`, instead of silently becoming 0.

Listing 12.11 `CreatePaymentRequest.java` (Spring): amount and method.

```
package com.profitina.laundry.web;

import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;

public record CreatePaymentRequest(
        @NotNull @Min(1000) Long amount,
        @NotBlank String method) {
}
```

The `pay` method of **Listing 12.10** checks, in order: the order is open (409), the method is one of the three words (422 on `method`), and the amount is not more than the balance (422 on `amount`). It then calls `order.addPayment`, a new method of the `Order` entity that creates the `Payment`, adds it to the order's list and returns it, and saves that payment at once with `payments.save(...)`, through the `PaymentRepository` that the controller now also receives. Because the payment's id is generated by the database, saving it sends the insert immediately, so the new payment has its `id` when `OrderView.withPayments` writes the answer; later, when the transaction commits, the cascade of Chapter 11 finds the payment already saved and does nothing more. **Listing 12.12** is the payment in the answer.

Listing 12.12 `PaymentView.java` (Spring): one payment, with its time in the contract's format.

```
package com.profitina.laundry.web;

import com.fasterxml.jackson.annotation.JsonFormat;
import com.profitina.laundry.domain.Payment;
import java.time.LocalDateTime;

public record PaymentView(
        Long id,
        long amount,
        String method,
        @JsonFormat(pattern = OrderView.TIME) LocalDateTime paidAt) {

    public static PaymentView of(Payment p) {
        return new PaymentView(p.getId(), p.getAmount(),
                p.getMethod().value(), p.getPaidAt());
    }
}
```

12.9.5 A tempting wrong solution

Leaving out `payments.save` and relying on the cascade at the end of the method works for the database: when the transaction commits, the payment in the order's list is saved. But the answer is written *before* the commit, while the new payment still has `id` null, and the client receives `"id": null`. It is a small bug that a test catches at once and a person clicking through the app never sees, which is exactly why the tests of Section 12.11 look at the answer, not only at the status code.

12.10 Problem 12.7: The Daily Report

12.10.1 The problem

`GET /api/reports/daily?date=YYYY-MM-DD` answers with the day's figures: `orders_received`, the number of orders received that day; `billed`, the sum of their totals; `collected`, the sum of the payments made that day, whichever day their orders came in; and `by_status`, the number of the day's orders in each status, with every status present, even at 0. A missing or badly written date is a 422 on `date`. The counts must satisfy Equation (12.2).

$$\sum_{s \in S} n_s = N \tag{12.2}$$

Here S is the set of statuses {queued, finished, departed}, n_s is the number of the day's orders now in status s, and N is `orders_received`. The equation says that every order received that day is counted in exactly one status.

12.10.2 Understanding it by hand

Suppose that on September 25 two orders came in, one of Rp 34.000 that is now departed and one of Rp 36.000 still queued, and that the only payments that day were Rp 34.000 for the first order and Rp 20.000 for an order received the day before. The report is `orders_received` 2, `billed` 70000, `collected` 54000, and `by_status` `{"queued": 1, "finished": 0, "departed": 1}`; $1 + 0 + 1 = 2$, as Equation (12.2) requires. Note that `collected` is not a part of `billed`: money collected today may belong to yesterday's orders.

12.10.3 Comparing the two

Neither API has this endpoint. Chapter 8 built a web report of payments per day and method in Laravel; its query is the starting point. The "one day" is the half-open interval from midnight to the next midnight, $[d, d + 1)$, so that an order received at exactly 00:00 belongs to one day only.

12.10.4 The solution in both frameworks

Listing 12.13 is the Laravel controller.

Listing 12.13 `Api/ReportController.php` (Laravel): counts per status, the day's totals and payments, every status present.

```php
<?php

namespace App\Http\Controllers\Api;

use App\Enums\OrderStatus;
use App\Http\Controllers\Controller;
use App\Models\Order;
use App\Models\Payment;
use Illuminate\Http\Request;
use Illuminate\Support\Carbon;
use Illuminate\Support\Facades\DB;

class ReportController extends Controller
{
    // GET /api/reports/daily?date=2026-09-25
    public function daily(Request $request)
    {
        $request->validate(['date' => ['required', 'date_format:Y-m-d']]);

        $from = Carbon::createFromFormat('Y-m-d', $request->query('date'))
            ->startOfDay();
        $to = $from->copy()->addDay();              // [from, to): one day

        $orders = Order::query()
            ->where('received_at', '>=', $from)
            ->where('received_at', '<', $to);

        $counts = (clone $orders)->toBase()
            ->select('status', DB::raw('count(*) as n'))
            ->groupBy('status')
            ->pluck('n', 'status');

        $byStatus = [];
        foreach (OrderStatus::cases() as $status) {   // every status, even 0
            $byStatus[$status->value] = (int) ($counts[$status->value] ?? 0);
        }
```

Listing 12.13 (Continued) *Api/ReportController.php (Laravel): counts per status, the day's totals and payments, every status present.*

```
        return response()->json(['data' => [
            'date' => $from->format('Y-m-d'),
            'orders_received' => array_sum($byStatus),
            'billed' => (int) (clone $orders)->sum('total'),
            'collected' => (int) Payment::query()
                ->where('paid_at', '>=', $from)
                ->where('paid_at', '<', $to)
                ->sum('amount'),
            'by_status' => $byStatus,
        ]]);
    }
}
```

The validation rule `date_format:Y-m-d` refuses anything but a date written as the contract says. `$orders` is a query builder for the day's orders, and `clone` makes a copy for each use, because a builder is changed by every `where` added to it. The count per status is done by the database with `GROUP BY`; `toBase()` asks for plain rows instead of `Order` models, so the enum cast does not interfere with `pluck('n', 'status')`, which returns the counts keyed by status word. The loop over `OrderStatus::cases()` adds the statuses that had no orders, with 0, so that a client never has to guess whether a missing key means zero.

The Spring version needs two new repository methods. **Listing 12.14** adds them to `OrderRepository` as JPQL queries with `@Query`. `countByStatusBetween` returns a list of `StatusCount`, a small *projection* interface whose getters Spring Data fills from the query's aliases `status` and `n` [3]. `billedBetween` sums the totals; `coalesce(..., 0)` turns the `null` that `SUM` returns for no rows into 0.

Listing 12.14 `OrderRepository.java` (Spring): the queries of Chapter 11 plus the two the report needs.

```
package com.profitina.laundry.repository;

import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.EntityGraph;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface OrderRepository extends JpaRepository<Order, Long> {

    @EntityGraph(attributePaths = {"customer", "service"})
    Page<Order> findAllByOrderByReceivedAtDescIdDesc(Pageable pageable);

    @EntityGraph(attributePaths = {"customer", "service"})
    Page<Order> findByStatusOrderByReceivedAtDescIdDesc(OrderStatus status,
                                                        Pageable pageable);

    @EntityGraph(attributePaths = {"customer", "service", "payments"})
    Optional<Order> findWithDetailsById(Long id);

    /** One row per status: the shape the daily report needs. */
    interface StatusCount {
        OrderStatus getStatus();

        long getN();
    }

    @Query("""
            select o.status as status, count(o) as n
            from Order o
            where o.receivedAt >= :from and o.receivedAt < :to
            group by o.status""")
    List<StatusCount> countByStatusBetween(@Param("from") LocalDateTime from,
                                           @Param("to") LocalDateTime to);
```

Listing 12.14 (Continued) *OrderRepository.java (Spring): the queries of Chapter 11 plus the two the report needs.*

```
    @Query("""
            select coalesce(sum(o.total), 0)
            from Order o
            where o.receivedAt >= :from and o.receivedAt < :to""")
    long billedBetween(@Param("from") LocalDateTime from,
                       @Param("to") LocalDateTime to);
}
```

`PaymentRepository` gains `collectedBetween`, the same kind of sum over `paidAt`. **Listing 12.15** is the controller, and the record `DailyReport` in the chapter's folder holds the five figures, which the snake_case strategy writes as `orders_received` and `by_status`.

Listing 12.15 `ReportController.java` (Spring): the same report, with a `LinkedHashMap` that lists every status in order.

```java
package com.profitina.laundry.web;

import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.repository.OrderRepository;
import com.profitina.laundry.repository.PaymentRepository;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeParseException;
import java.util.LinkedHashMap;
import java.util.Map;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ReportController {

    private final OrderRepository orders;
    private final PaymentRepository payments;

    public ReportController(OrderRepository orders,
                            PaymentRepository payments) {
        this.orders = orders;
        this.payments = payments;
    }

    /** GET /api/reports/daily?date=2026-09-25 */
    @GetMapping("/api/reports/daily")
    @Transactional(readOnly = true)
    public DataView<DailyReport> daily(
            @RequestParam(required = false) String date) {
        LocalDate day = parseDate(date);
        LocalDateTime from = day.atStartOfDay();
        LocalDateTime to = from.plusDays(1);   // [from, to): one whole day
```

Listing 12.15 (Continued) *ReportController.java (Spring): the same report, with a LinkedHashMap that lists every status in order.*

```
        Map<String, Long> byStatus = new LinkedHashMap<>();
        for (OrderStatus s : OrderStatus.values()) {
            byStatus.put(s.value(), 0L);       // every status, even if 0
        }
        for (OrderRepository.StatusCount c
                : orders.countByStatusBetween(from, to)) {
            byStatus.put(c.getStatus().value(), c.getN());
        }
        long received = byStatus.values().stream()
                .mapToLong(Long::longValue).sum();

        return new DataView<>(new DailyReport(day.toString(), received,
                orders.billedBetween(from, to),
                payments.collectedBetween(from, to), byStatus));
    }

    private static LocalDate parseDate(String date) {
        if (date == null || date.isBlank()) {
            throw new ApiValidationException("date",
                    "The date field is required.");
        }
        try {
            return LocalDate.parse(date);
        } catch (DateTimeParseException e) {
            throw new ApiValidationException("date",
                    "The date field must match the format Y-m-d.");
        }
    }
}
```

12.10.5 A tempting wrong solution

Selecting the day with `WHERE date(received_at) = ?` reads naturally and is correct on a small table. It is also slow on a large one, because applying a function to the column prevents the database from using an index on `received_at`, and on some databases `date()` interprets the stored time in the server's zone rather than the shop's. The interval form `received_at >= :from AND received_at < :to` compares the column directly, can use an index, and states exactly which moments belong to the day. (The reference server uses a text comparison of the first ten characters, which is correct for its stored format and fine for its size, and a good example of a shortcut that should stay in a toy.)

12.11 Checking the Contract

12.11.1 The checker

Listing 12.16 is the contract as a program. It needs nothing but PHP: it sends each request with PHP's own HTTP stream functions, reads the status code from the first response header, decodes the JSON, and prints PASS or FAIL for each of sixteen checks.

Listing 12.16 `contract/check_api.php`: sixteen checks of the contract, against any server.

```php
<?php

// The Profitina Laundry API contract as a program. Plain PHP, no packages:
//   php check_api.php http://127.0.0.1:8000 [token]
// Point it at the Laravel app, the Spring Boot app or any other server.

$base = rtrim($argv[1] ?? 'http://127.0.0.1:8000', '/');
$token = $argv[2] ?? null;
$passed = $failed = 0;

// Sends one request; returns [status code, decoded JSON body].
function call(string $method, string $path, ?array $body = null): array
{
    global $base, $token;
    $headers = ['Accept: application/json',
                'Content-Type: application/json'];
    if ($token !== null) {
        $headers[] = "Authorization: Bearer $token";
    }
    $context = stream_context_create(['http' => [
        'method' => $method,
        'header' => implode("\r\n", $headers),
        'content' => $body === null ? '' : json_encode($body),
        'ignore_errors' => true,          // give us the 4xx bodies too
    ]]);
    $raw = @file_get_contents($base . $path, false, $context);
    if ($raw === false) {
        fwrite(STDERR, "Cannot reach $base$path\n");
        exit(2);
    }
    preg_match('/^HTTP\/\S+ (\d{3})/', $http_response_header[0], $m);
    return [(int) $m[1], json_decode($raw, true)];
}

function check(string $what, bool $ok): void
{
    global $passed, $failed;
    $ok ? $passed++ : $failed++;
    printf("%s  %s\n", $ok ? 'PASS' : 'FAIL', $what);
}

$time = '/^\d{4}-\d\d-\d\dT\d\d:\d\d:\d\d$/';
```

Listing 12.16 (Continued) *contract/check_api.php: sixteen checks of the contract, against any server.*

```
[$s, $j] = call('GET', '/api/services');
$first = $j['data'][0] ?? [];
check('GET /api/services -> 200, S0001 first, 8000 per kg',
    $s === 200 && ($first['id'] ?? '') === 'S0001'
    && ($first['unit'] ?? '') === 'kg' && ($first['price'] ?? 0) === 8000);

$phone = '0812' . random_int(10000000, 99999999);   // new on every run
[$s, $j] = call('POST', '/api/customers', ['name' => 'Sri Rahayu',
    'phone' => substr($phone, 0, 4) . '-' . substr($phone, 4)]);
$customer = $j['data'] ?? [];
check('POST /api/customers -> 201, phone kept as digits only',
    $s === 201 && ($customer['phone'] ?? '') === $phone);
[$s, $j] = call('POST', '/api/customers', ['name' => 'Sri Rahayu',
    'phone' => $phone]);
check('the same phone again -> 422, errors.phone',
    $s === 422 && isset($j['errors']['phone']));

[$s, $j] = call('POST', '/api/orders', [
    'customer_id' => $customer['id'] ?? 0, 'outlet_id' => 'O01',
    'service_id' => 'S0001', 'quantity' => 2.5, 'pickup' => true,
    'delivery_address' => 'Jl. Kenanga 5, Surabaya']);
$o = $j['data'] ?? [];
check('POST /api/orders -> 201, total 34000 (3 kg + pickup), queued',
    $s === 201 && ($o['total'] ?? 0) === 34000
    && ($o['balance'] ?? 0) === 34000 && ($o['status'] ?? '') === 'queued');
check('times are YYYY-MM-DDTHH:MM:SS, finished_at is null',
    preg_match($time, $o['received_at'] ?? '') === 1
    && array_key_exists('finished_at', $o) && $o['finished_at'] === null);
$url = '/api/orders/' . ($o['id'] ?? 0);

[$s, $j] = call('POST', '/api/orders', ['quantity' => 0, 'pickup' => true]);
$fields = array_keys($j['errors'] ?? []);
check('a bad order -> 422, all of its errors at once',
    $s === 422 && is_string($j['message'] ?? null)
    && ! array_diff(['customer_id', 'service_id', 'quantity',
        'delivery_address'], $fields));
```

Listing 12.16 (Continued) *contract/check_api.php: sixteen checks of the contract, against any server.*

```
$move = fn (string $to) => call('PATCH', "$url/status", ['status' => $to]);
[$s, $j] = $move('departed');
check('queued -> departed skips a step: 422, errors.status',
    $s === 422 && isset($j['errors']['status']));
[$s, $j] = $move('finished');
check('queued -> finished: 200, finished_at set',
    $s === 200 && preg_match($time, $j['data']['finished_at'] ?? '') === 1);
[$s, $j] = $move('departed');
check('finished -> departed before payment: 422, errors.status',
    $s === 422 && isset($j['errors']['status']));

[$s, $j] = call('POST', "$url/payments", ['amount' => 40000,
    'method' => 'cash']);
check('paying more than the balance -> 422, errors.amount',
    $s === 422 && isset($j['errors']['amount']));
[$s, $j] = call('POST', "$url/payments", ['amount' => 34000,
    'method' => 'qris']);
check('paying the balance -> 201, balance 0, payments[0].method qris',
    $s === 201 && ($j['data']['balance'] ?? -1) === 0
    && ($j['data']['payments'][0]['method'] ?? '') === 'qris');

[$s, $j] = $move('departed');
check('finished -> departed after payment: 200',
    $s === 200 && ($j['data']['status'] ?? '') === 'departed');
[$s, $j] = call('POST', "$url/payments", ['amount' => 1000,
    'method' => 'cash']);
check('a departed order is closed: 409 with a message',
    $s === 409 && is_string($j['message'] ?? null));

[$s, $j] = call('GET', '/api/orders?status=departed');
check('GET /api/orders?status=departed -> 200, data[] and meta',
    $s === 200 && array_is_list($j['data'] ?? null)
    && ($j['meta']['total'] ?? 0) >= 1
    && ($j['meta']['current_page'] ?? 0) === 1);
[$s, $j] = call('GET', '/api/orders/999999');
check('GET /api/orders/999999 -> 404 with a message',
    $s === 404 && is_string($j['message'] ?? null));

[$s, $j] = call('GET', '/api/reports/daily?date=' . date('Y-m-d'));
$r = $j['data'] ?? [];
check('GET /api/reports/daily -> 200, the counts add up',
    $s === 200 && ($r['orders_received'] ?? 0) >= 1
    && ($r['collected'] ?? 0) >= 34000
    && array_sum($r['by_status'] ?? []) === $r['orders_received']);
```

Listing 12.16 (Continued) *contract/check_api.php: sixteen checks of the contract, against any server.*

```
printf("\n%d passed, %d failed\n", $passed, $failed);
exit($failed === 0 ? 0 : 1);
```

The checker follows the life of one order, as a driver's app would: it reads the price list, registers a new customer with a phone number chosen at random, so that it can run again and again against the same database, tries the same phone a second time, creates an order, breaks every rule of a new order at once, walks the order through its statuses with a refused skip and a refused unpaid departure, pays too much and then the right amount, closes the order, tries to pay a closed order, lists the departed orders, asks for an order that does not exist, and finally reads today's report and checks Equation (12.2). `'ignore_errors' => true` is what lets it read the bodies of 4xx answers; without it PHP would treat them as failures and return nothing.

12.11.2 A real run

Started in one terminal with `php -S 127.0.0.1:8000 reference/server.php` and checked from another with `php check_api.php http://127.0.0.1:8000`, the reference server gives this output, which is the real output of the run made where this book was written, on an empty database and again on the same database:

```
PASS  GET /api/services -> 200, S0001 first, 8000 per kg
PASS  POST /api/customers -> 201, phone kept as digits only
PASS  the same phone again -> 422, errors.phone
PASS  POST /api/orders -> 201, total 34000 (3 kg + pickup), queued
PASS  times are YYYY-MM-DDTHH:MM:SS, finished_at is null
PASS  a bad order -> 422, all of its errors at once
PASS  queued -> departed skips a step: 422, errors.status
PASS  queued -> finished: 200, finished_at set
PASS  finished -> departed before payment: 422, errors.status
PASS  paying more than the balance -> 422, errors.amount
PASS  paying the balance -> 201, balance 0, payments[0].method qris
PASS  finished -> departed after payment: 200
PASS  a departed order is closed: 409 with a message
PASS  GET /api/orders?status=departed -> 200, data[] and meta
PASS  GET /api/orders/999999 -> 404 with a message
PASS  GET /api/reports/daily -> 200, the counts add up

16 passed, 0 failed
```

The reference server is not a third application for the shop. It is the contract made executable in the simplest possible way, and its value is that it proves the checker: when the checker reports a failure against Laravel or Spring, the fault is in that application, not in the checker. The chapter's `README.md` shows how to point the checker at the two real applications; the Laravel API needs the token of Chapter 10 as the second argument, while the Spring API has no sign-in until Chapter 14.

Before the changes of this chapter, the checker would have stopped the Chapter 11 Spring application at its first check, because `data` was missing from the price list, and the Chapter 10 Laravel application at its first check as well, because there was no price-list endpoint; further down, the time format, the error shape and the missing payment endpoint would have failed in turn. That is how a contract check is used in practice: run it, fix the first failure, run it again.

12.11.3 Tests inside each framework

The checker tests from outside. Each framework also gets tests of its own, which run without a server and without a network, in the style of Chapter 10. **Listing 12.17** adds the points that Chapter 10's tests did not cover: the price list, the time format and the report. `travelTo` fixes the clock at a known moment, so the expected `received_at` can be written exactly [4].

Listing 12.17 `tests/Feature/ContractTest.php` (Laravel): the contract's new points, with a fixed clock.

```php
<?php

namespace Tests\Feature;

use App\Models\Customer;
use App\Models\User;
use Database\Seeders\PriceListSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Laravel\Sanctum\Sanctum;
use Tests\TestCase;

// The points of the contract that Chapter 10's API did not yet keep.
class ContractTest extends TestCase
{
    use RefreshDatabase;

    protected function setUp(): void
    {
        parent::setUp();
        $this->seed(PriceListSeeder::class);
        Sanctum::actingAs(User::factory()->create());
    }

    public function test_the_price_list_is_an_endpoint(): void
    {
        $this->getJson('/api/services')->assertOk()
            ->assertJsonPath('data.0.id', 'S0001')
            ->assertJsonPath('data.0.unit', 'kg')
            ->assertJsonPath('data.0.price', 8000);
    }

    public function test_times_use_the_contract_format(): void
    {
        $this->travelTo('2026-09-25 08:15:00');

        $this->postJson('/api/orders', [
            'customer_id' => Customer::factory()->create()->id,
            'outlet_id' => 'O01', 'service_id' => 'S0001', 'quantity' => 4.5,
        ])->assertCreated()
            ->assertJsonPath('data.received_at', '2026-09-25T08:15:00')
            ->assertJsonPath('data.finished_at', null);
    }
```

Listing 12.17 (Continued) *tests/Feature/ContractTest.php (Laravel): the contract's new points, with a fixed clock.*

```php
    public function test_the_daily_report(): void
    {
        $this->getJson('/api/reports/daily')
            ->assertStatus(422)->assertJsonValidationErrors('date');

        $this->getJson('/api/reports/daily?date=2026-09-25')
            ->assertOk()
            ->assertJsonPath('data.orders_received', 0)
            ->assertJsonPath('data.by_status.queued', 0);
    }
}
```

Listing 12.18 is the Spring application's first test that starts the whole application. `@SpringBootTest` starts it with its real configuration and its in-memory database, filled by `DemoData`; `@AutoConfigureMockMvc` provides `MockMvc`, which sends requests to the controllers without a network, as Laravel's `postJson` does [5]. In Spring Boot 4 this annotation comes from the `spring-boot-starter-webmvc-test` starter, which the chapter's `pom.xml` adds [6].

Listing 12.18 `OrderApiTest.java` (Spring): the price rule, every error at once, the whole life of an order and the report, through MockMvc.

```
package com.profitina.laundry.web;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

import com.jayway.jsonpath.JsonPath;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.webmvc.test.autoconfigure.AutoConfigureMockMvc;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

/** The whole application, started once, driven through HTTP calls. */
@SpringBootTest
@AutoConfigureMockMvc
class OrderApiTest {

    @Autowired
    private MockMvc mvc;

    private String send(String method, String url, String json, int expect)
            throws Exception {
        var request = switch (method) {
            case "POST" -> post(url);
            case "PATCH" -> patch(url);
            default -> get(url);
        };
        if (json != null) {
            request.contentType(MediaType.APPLICATION_JSON).content(json);
        }
        return mvc.perform(request).andExpect(status().is(expect))
                .andReturn().getResponse().getContentAsString();
    }

    @Test
    void anOrderIsPricedByTheRule() throws Exception {
        mvc.perform(post("/api/orders")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("""
                            {"customer_id": 1, "service_id": "S0001",
                             "quantity": 2.5, "pickup": true,
                             "delivery_address": "Jl. Kenanga 5"}"""))
                .andExpect(status().isCreated())
                .andExpect(jsonPath("$.data.total").value(34000))
                .andExpect(jsonPath("$.data.status").value("queued"));
```

Listing 12.18 (Continued) *OrderApiTest.java (Spring): the price rule, every error at once, the whole life of an order and the report, through MockMvc.*

```
    }

    @Test
    void badInputGets422WithEveryError() throws Exception {
        mvc.perform(post("/api/orders")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("{\"quantity\": 0, \"pickup\": true}"))
                .andExpect(status().is(422))
                .andExpect(jsonPath("$.errors.customer_id").exists())
                .andExpect(jsonPath("$.errors.service_id").exists())
                .andExpect(jsonPath("$.errors.quantity").exists())
                .andExpect(jsonPath("$.errors.delivery_address").exists());
    }

    @Test
    void theWholeLifeOfAnOrder() throws Exception {
        String body = send("POST", "/api/orders", """
                {"customer_id": 2, "service_id": "S0001", "quantity": 2.5,
                 "pickup": true, "delivery_address": "Jl. Kenanga 5"}""",
                201);
        String url = "/api/orders/" + JsonPath.read(body, "$.data.id");
        String to = "{\"status\": \"%s\"}";

        send("PATCH", url + "/status", to.formatted("departed"), 422);
        send("PATCH", url + "/status", to.formatted("finished"), 200);
        send("PATCH", url + "/status", to.formatted("departed"), 422);
        send("POST", url + "/payments",
                "{\"amount\": 40000, \"method\": \"cash\"}", 422);
        send("POST", url + "/payments",
                "{\"amount\": 34000, \"method\": \"qris\"}", 201);
        send("PATCH", url + "/status", to.formatted("departed"), 200);
        send("POST", url + "/payments",
                "{\"amount\": 1000, \"method\": \"cash\"}", 409);
    }

    @Test
    void theDailyReportNeedsADate() throws Exception {
        send("GET", "/api/reports/daily", null, 422);
        mvc.perform(get("/api/reports/daily?date=2026-09-25"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.data.by_status.queued").exists());
    }
}
```

`jsonPath("$.data.total").value(34000)` checks one value in the answer with a JSONPath expression; `JsonPath.read` takes the new order's id out of the body so that the following requests can use it. The helper `send` keeps the life of an order to one line per step, with the expected status at the end, so the test reads like the hand-worked example of Problem 12.5.

12.12 Using It for Real

- **Write the contract before the second client.** One client can live with whatever the server sends. The day a second client or a second server appears, write down the rules, as **Table 12.1** and **Table 12.2** do, and make them executable.
- **Publish it in a standard format.** The OpenAPI Specification describes URLs, fields, types and status codes in a file that tools can read, turn into documentation and use to generate client code [7]. The tables of Section 12.3 are a readable summary of what such a file would say.

- **Add, never change.** A new field, a new endpoint or a new optional parameter does not break clients that ignore what they do not know. Renaming or removing a field does; do it in a new version of the API, such as `/api/v2`, while the old one keeps working until every client has moved.
- **Keep one list of refusals.** Status codes chosen case by case drift apart. A short table of what 404, 409 and 422 mean in this API, applied everywhere, is worth more than any single clever choice.
- **Check every server the same way.** A checker such as `check_api.php` runs in seconds before every release, on a laptop or on a build server, and it checks what the client sees, which the framework's own tests do not.

12.13 Common Mistakes, and How to Avoid Them

- **Letting frameworks choose the contract.** Default date formats, default error bodies and default envelopes differ; choose each one on purpose and write it down.
- **Comparing times as text across zones.** `2026-09-25T01:15:00Z` and `2026-09-25T08:15:00` are the same moment and different strings. Fix one format and one zone in the contract.
- **Mixing up 400, 409 and 422.** 400 is for a body that cannot be read, 422 for content that breaks a rule, 409 for a resource whose state forbids the request.
- **Error keys in the server's own names.** `serviceId` in a Java error, or `service` where the request said `service_id`, cannot be matched to a field by the client. Use the request's field names.
- **Errors in two rounds.** A client that fixes the three errors it was shown and then receives a fourth feels cheated. Collect every error before answering.
- **Missing keys for zero.** A report that leaves out statuses with no orders forces every client to guess; send every key.
- **Checking only status codes in tests.** An answer with the right status and `"id": null` still breaks the client. Check the fields the client uses.

12.14 Chapter Summary

- A contract fixes the endpoints (**Table 12.1**), the envelope `data`, snake_case names, lowercase status words, whole rupiah, one date-time format in shop time, and the refusals of **Table 12.2**.
- Problem 12.1 added the price list to Laravel and the `data` envelope to Spring (`DataView`).
- Problem 12.2 fixed every time to `YYYY-MM-DDTHH:MM:SS` in shop time: an explicit format in Laravel's resources and `@JsonFormat` in Spring's records.
- Problem 12.3 gave Spring Laravel's 422 error shape with an exception handler that converts field names to snake_case.
- Problem 12.4 made Spring report every error of a new order at once by validating by hand and adding the database checks to the same map.
- Problem 12.5 separated 422 for a refused step from 409 for a closed order, following the meaning of the two HTTP codes.
- Problem 12.6 added payments to Spring, saving each payment at once so that the answer carries its id; the balance follows Equation (12.1).
- Problem 12.7 added a daily report to both, over the half-open day $[d, d + 1)$, with every status present so that Equation (12.2) holds.

- `check_api.php` checks any server against the contract; its run against the reference server passed all sixteen checks, and each framework has its own tests for the same points.

12.15 Exercises

1. Add to the contract an optional filter `GET /api/orders?customer_id=5`. Write the contract's sentence for it, and the change in each framework's `index` method.
2. Without running anything, go through `check_api.php` against the Chapter 11 Spring application (before this chapter's changes). Which check fails first, and why? Name three later checks that would also fail, and the problem of this chapter that fixes each.
3. Laravel accepts `"quantity": "2,5"` because `StoreOrderRequest` turns the comma into a dot; Spring's Jackson refuses the string. Should the contract allow a quantity written as text with a comma? Argue for one answer, and say what each framework must change to follow it.
4. Write two more checks for `check_api.php`: that `GET /api/customers?phone=` with the new customer's phone finds exactly that customer, and that `GET /api/orders/{id}` of the new order carries its payments. Which endpoint needs to change in which framework for the first check to pass?
5. The accounting firm wants the report as numbers per payment method as well. Design the extra field of the answer so that old clients keep working, and write the Laravel query that fills it.
6. A second shop in Bali, one hour ahead of Surabaya, will use the same API. What happens to the contract's time rule, and what would you change: the format, the zone, or both?

12.16 Solutions to the Exercises

1. Contract: "`GET /api/orders` accepts an optional `customer_id`; when it is given, only that customer's orders are listed; an id that is not a number is a 422 on `customer_id`." Laravel: add `'customer_id' => ['nullable', 'integer']` to the rules in `index`, and `->when($request->integer('customer_id'), fn ($q, $id) => $q->where('customer_id', $id))` to the query. Spring: add `@RequestParam(required = false) Long customerId` to `index`, and a repository method `findByCustomerIdOrderByReceivedAtDescIdDesc(Long customerId, Pageable pageable)` with the same entity graph, chosen when `customerId` is not null; Spring answers a non-numeric value with a 400 conversion error, so to keep the contract, take the parameter as a `String` and throw `ApiValidationException("customer_id", ...)` when it does not parse.

2. The first check fails: `GET /api/services` answers a bare list, so `$j['data'][0]` is missing and the check "S0001 first" reports FAIL (Problem 12.1). Later: "a bad order → 422" fails, because Spring answers 400 with problem details and no `errors` (Problems 12.3 and 12.4); "times are YYYY-MM-DDTHH:MM:SS" may fail, because Chapter 11 has no fixed format (Problem 12.2); "queued → departed skips a step: 422" fails with 409 (Problem 12.5); every payment check fails with 404 or 405, because there is no payment endpoint (Problem 12.6); and the report check fails with 404 (Problem 12.7). Note that the checker does not stop at the first failure; it runs every check, and some checks that depend on an earlier one, such as the order checks after a failed customer creation, fail as a consequence.

3. The contract should say that a quantity is a JSON number, and nothing else. Commas are a matter of how people write numbers on a form, and forms belong to the client: the phone app turns what the driver typed into a number before it sends it, exactly as the Laravel web form's `prepareForValidation` does for the browser. Accepting text would force every server to parse locale-dependent strings. Spring already follows this rule. Laravel may keep its tolerance, because

it is invisible to clients that send numbers, but the contract does not promise it; a stricter Laravel API would add the rule `'quantity' => [... 'numeric' ...]` only for JSON requests, which is not worth the complexity.

4. First check: `[$s, $j] = call('GET', '/api/customers?phone=' . $phone); check('GET /api/customers?phone= finds the new customer', $s === 200 && count($j['data'] ?? []) === 1 && ($j['data'][0]['id'] ?? 0) === ($customer['id'] ?? -1));`. Second: `[$s, $j] = call('GET', $url); check('GET /api/orders/{id} carries its payments', $s === 200 && count($j['data']['payments'] ?? []) === 1);`, placed after the payment. Laravel's customer `index` answers with a paginated collection whose `data` is a list, which passes. Spring's Chapter 12 `CustomerController` already wraps its list in `DataView`, which passes too; the reference server has no customer list and needs a `GET /api/customers` branch.

5. Add a new field, `collected_by_method`, an object with the keys `cash`, `qris` and `transfer`, each present even at 0; old clients ignore it, and `collected` stays as it is, equal to the sum of the three. Laravel: `$byMethod = Payment::query()->where('paid_at', '>=', $from)->where('paid_at', '<', $to)->toBase()->select('method', DB::raw('sum(amount) as s'))->groupBy('method')->pluck('s', 'method');` followed by a loop over `PaymentMethod::cases()` that writes `(int) ($byMethod[$m->value] ?? 0)` for each method, exactly as the report does for statuses.

6. With a second zone, "local shop time" is ambiguous: 08:15 in Bali is 07:15 in Surabaya. Keep the format and change the zone rule: either send every time with its offset, `2026-09-25T08:15:00+08:00`, so each shop's times stay readable and every moment is unambiguous, or send UTC with `Z` and let clients show local time. The daily report then needs a zone too: "the day" of the Bali shop runs from its own midnight, so the report takes the outlet (or its zone) as a parameter. Because changing the format of an existing field breaks old clients, introduce the new form in `/api/v2`, or add new fields such as `received_at_utc`, while the old ones keep their meaning for the Surabaya shops.

References

[1] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[2] Spring, "Exceptions," *Spring Framework Reference Documentation: Web on Servlet Stack*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-exceptionhandler.html

[3] Spring, "Projections," *Spring Data JPA Reference Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-data/jpa/reference/repositories/projections.html

[4] Laravel, "Mocking: Interacting With Time," *Laravel 13.x Documentation*. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/mocking#interacting-with-time

[5] Spring, "MockMvc," *Spring Framework Reference Documentation: Testing*. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-framework/reference/testing/mockmvc.html

[6] Spring, "Modularizing Spring Boot," spring.io blog, Oct. 28, 2025. Accessed: Sep. 25, 2026. [Online]. Available: https://spring.io/blog/2025/10/28/modularizing-spring-boot/

[7] OpenAPI Initiative, "OpenAPI Specification v3.2.0," Sep. 2025. Accessed: Sep. 25, 2026. [Online]. Available: https://spec.openapis.org/oas/v3.2.0.html

Chapter 13

Spring Boot II: Layered Architecture, Validation and Full CRUD with Thymeleaf

Chapters 11 and 12 gave Profitina Laundry a Spring Boot API that speaks the same JSON as the Laravel one: the price list, customers, orders that move from queued to finished to departed, payments and a daily report. It works, but it has two weaknesses that every first API has. All the work sits in the controllers, which read the database, apply the shop's rules and decide the HTTP answer in the same methods; and there is nothing for the people at the counter, who need pages, not JSON. This chapter fixes both. By the end of it you will be able to split a Spring Boot application into controller, service and repository layers and say which code belongs where; move business rules and transactions out of controllers into services that both an API and web pages can call; validate form input with Jakarta Bean Validation and show every message next to the right field; build complete create, read, update and delete pages with Thymeleaf; handle "not found" in one place; keep the data in SQLite between starts; and test the service layer without starting Spring.

13.1 Where We Left Off

At the end of Chapter 12 the Spring Boot project had four entities (`Customer`, `Service`, `Order`, `Payment`), the rules of the shop in the plain Java class `LaundryRules`, Spring Data repositories, and five REST controllers. The contract of Chapter 12 checked it, field by field, against the Laravel API: `POST /api/orders` answers 201 with `{"data": {...}}`, a bad order answers 422 with every error at once, a departed order is closed with 409, and so on.

Look again at Chapter 12's `OrderController.store`, which takes a new order. In one method it validates the request, looks up the customer and the service in their repositories, checks that a pickup has a delivery address, collects the errors, builds the order, saves it, and wraps the result in JSON. `updateStatus` reads an order, checks that it is still open, asks `LaundryRules` for the next step, compares it with the status the client wanted, and stamps the time. Each method carries `@Transactional`. Chapter 11 warned that this is acceptable for a first, small API and promised that Chapter 13 would move the work into a *service layer*. That promise is due, because Bu Wati now wants something the API alone cannot give.

The Wiyung outlet, opened in January 2026, still takes orders on paper. Its counter has a small computer with a browser, and Bu Wati wants the same counter screens there as in the other outlets: a list of customers to search, a form to add a customer or correct a phone number, a form to take in a bag of laundry, the queue of orders, and an order page that shows the total, the payments and the balance, with a button that moves the order along and refuses to hand over laundry that is not paid. The accounting firm that connected to the API in Chapter 11 must keep working exactly as before.

If the pages were written the way Chapter 12 wrote the API, every rule would exist twice: once in the JSON controller and once in the page controller. The first time someone changed the pickup rule in one place and not the other, the Wiyung counter and the phone app would disagree about the price of the same bag. This chapter therefore does the work in the right order: first the layers (Sections 13.2 to 13.5), then the pages on top of them (Sections 13.6 to 13.8), then SQLite, running and testing (Sections 13.9 and 13.10). The code is in `Chapter13/spring`; it starts as a copy of Chapter 12's project.

13.2 The Idea in Plain Words

13.2.1 One class that does everything

A controller method that reads the database, applies rules, starts a transaction and chooses an HTTP status is quick to write and easy to follow while there is only one of it. The trouble starts with the second caller. A page that takes orders needs every check that `store` makes, but it must answer with a form and red messages instead of JSON and status 422. A nightly job that closes forgotten orders needs the workflow rule but has no HTTP request at all. Each new caller copies the rules, and copies drift apart. Testing is hard too: to check "laundry leaves only when paid", a test must build an HTTP request, start the web layer and a database, and read the answer out of JSON or HTML. Slow tests are tests that people stop running.

13.2.2 Three layers, one direction

The standard answer, used by all three frameworks in this book, is to split the application into *layers*, each with one kind of responsibility, and to let each layer call only the layer below it. **Figure 13.1** shows the layers of this chapter's application.

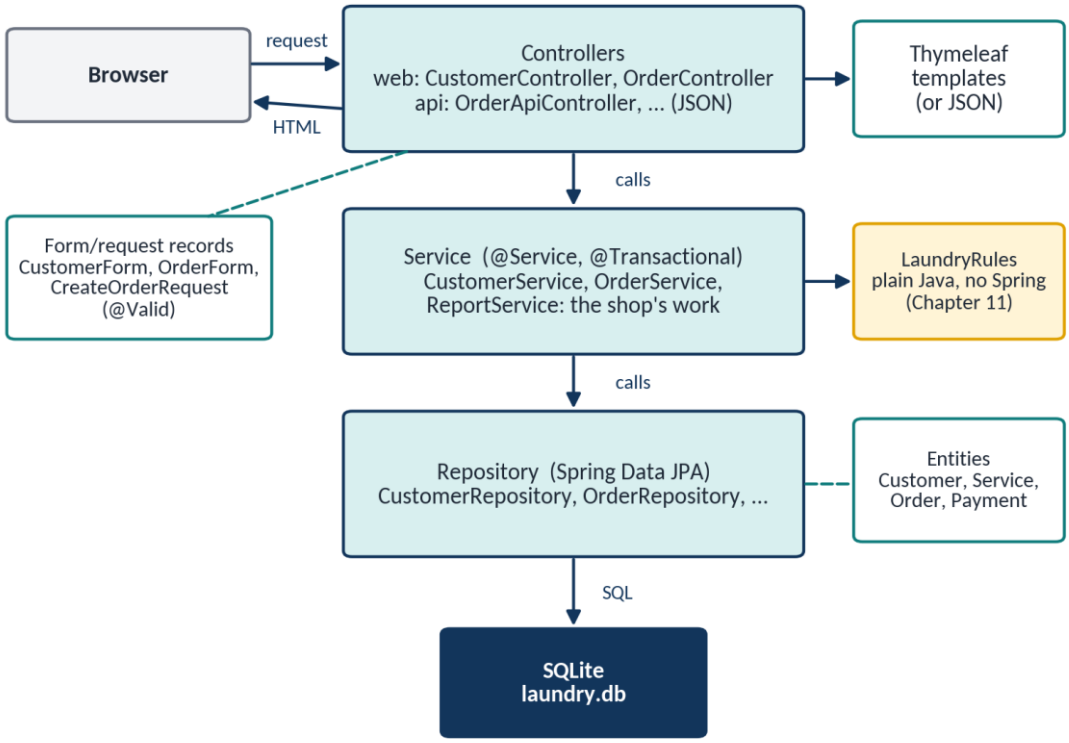


Figure 13.1 The layers of the Spring Boot laundry application: page and API controllers speak HTTP, services do the shop's work inside transactions, repositories speak SQL; the rules themselves are plain Java from Chapter 11.

> **DEFINITION.** In a *layered architecture*, the **controller** (presentation) layer turns HTTP requests into method calls and results into HTML or JSON; the **service** layer carries out one business operation per method, applies the business rules and decides where a database transaction starts and ends; the **repository** (persistence) layer loads and stores entities. Calls go downwards only: a controller calls services, a service calls repositories, and never the other way round.

The idea is older than any of the three frameworks. Fowler calls it the *Service Layer* pattern: a layer of services that sets the boundary of the application, offers the operations it supports, and coordinates the application's response in each operation [1]. Spring's vocabulary follows it: a class annotated `@Controller` or `@RestController` is found by the web layer, a class annotated `@Service` is an ordinary bean that other beans may call, and an interface that extends `JpaRepository` is implemented for you by Spring Data [2], [3].

The interesting consequence is at the top of **Figure 13.1**. Once the work lives in services, there can be two kinds of controller side by side: the JSON controllers of Chapter 12, now thin, and new page controllers that render Thymeleaf templates. Both call the same `OrderService.create`, so both apply the same rules by construction. On the right sits `LaundryRules`, unchanged since Chapter 11: plain Java that the services call, and that can be checked with nothing but the JDK.

13.2.3 What each layer may know

A good way to decide where a line of code belongs is to ask what it needs to *know*.

Table 13.1 What each layer may know, and what it must not know.

Layer	Knows about	Must not know about	Example in this chapter
Controller	URLs, request parameters, JSON and forms, redirects, which template to show	SQL, transactions, prices	`OrderController.advance`
Service	the shop's operations and rules, transactions	HTML, URLs, HTTP status codes	`OrderService.advance`
Repository	tables, queries	rules, HTTP	`OrderRepository.billedBetween`
Rules	numbers, statuses	everything else	`LaundryRules.advance`

Keep **Table 13.1** in mind for the rest of the chapter: follow a click on "Hand over to customer" from the page controller to the service to the repository, and at each step the code does only what its row allows.

13.2.4 Transactions: all or nothing

Taking a payment involves two steps: find out how much is still owed, then store the payment. If two cashiers take a payment for the same order at the same moment, both could read "Rp 34.000 owed" and both could store Rp 34.000, so the customer pays twice. A *transaction* groups database operations so that they succeed or fail together, and the database isolates it from other transactions running at the same time. In Spring you mark a method, or a whole class, with `@Transactional`; Spring starts a transaction before the method runs, commits it when the method returns normally, and rolls it back when the method throws an unchecked exception [2]. A transaction should cover one business operation, and a business operation is exactly what a service method is. That is why this chapter removes `@Transactional` from every controller and puts it on the services.

> **KEY INSIGHT:** The service layer is where three things meet: one business operation, one set of rules and one transaction. When you are unsure where code belongs, ask “is this part of what the shop *does*?” If yes, it goes in a service.

13.3 Where the Code Goes

Chapter 12’s project kept its JSON controllers in a package called `web`. With pages arriving, `web` becomes the home of the page controllers, and the JSON controllers move to a new package `api`. **Table 13.2** is the whole map.

Table 13.2 The packages of the Chapter 13 project.

Package	Contents	New in this chapter?
`domain`	`Customer`, `Service`, `Order`, `Payment`, statuses, `LaundryRules`	only `Customer.change`
`repository`	Spring Data interfaces	a few query methods
`service`	`CustomerService`, `OrderService`, `ReportService`, three exceptions	all new
`api`	`OrderApiController` and the other JSON controllers, request and view records, `ApiExceptionHandler`	moved from `web`, now thin
`web`	page controllers, form records, `Forms`, `WebErrorAdvice`	all new

The JSON controllers get the suffix `Api`: `OrderApiController`, `CustomerApiController`, `ServiceApiController`, `ReportApiController`. This is not only tidiness. Spring names a bean after its class, `orderController` for `OrderController`, and two classes with the same simple name in different packages would both claim that name, so the application would refuse to start. The URLs, the JSON and the contract of Chapter 12 do not change at all.

13.4 The Service Layer

13.4.1 Three exceptions

A service must be able to say “no” without knowing whether its caller will turn the answer into JSON or into a red message on a page. It does so with three exceptions (**Listing 13.1**, **Listing 13.2** and **Listing 13.3**).

Listing 13.1 `RuleException.java`: input that breaks a rule, with its messages per field.

```
package com.profitina.laundry.service;

import java.util.List;
import java.util.Map;

/**
 * Input that breaks one of the shop's rules, field by field. The API turns
 * it into 422 {"errors": {...}}; the pages show each message by its field.
 */
public class RuleException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    private final transient Map<String, List<String>> errors;

    public RuleException(Map<String, List<String>> errors) {
        super("The given data was invalid.");
        this.errors = errors;
    }

    public RuleException(String field, String message) {
        this(Map.of(field, List.of(message)));
    }

    public Map<String, List<String>> getErrors() {
        return errors;
    }
}
```

Listing 13.2 `NotFoundException.java`: a record that does not exist.

```
package com.profitina.laundry.service;

/** Asked for a record that does not exist: 404 in the API and the pages. */
public class NotFoundException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public NotFoundException(String what, Object id) {
        super("No " + what + " " + id + ".");
    }
}
```

Listing 13.3 `ConflictException.java`: a request that the record's present state does not allow.

```
package com.profitina.laundry.service;

/** Well formed, but not possible in the record's present state: 409. */
public class ConflictException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public ConflictException(String message) {
        super(message);
    }
}
```

`RuleException` is Chapter 12's `ApiValidationException` under a name that no longer mentions the API: it carries a map from field name to messages, exactly the `errors` object of the contract. The field names are the API's snake_case names (`customer_id`), because the rules were written for the API first; Section 13.6 shows how the pages translate them. `NotFoundException` builds the same sentence as Chapter 12 ("No order 7."), and `ConflictException` covers the one 409 of the contract, the closed order, plus a new one for customers in Section 13.4.2. All three extend `RuntimeException`. That matters twice: unchecked exceptions need no `throws` clause on the way

up, and Spring rolls a transaction back only for unchecked exceptions by default [2], so a rule that fails halfway through a service method undoes whatever the method had already written.

13.4.2 Customers

The pages must edit and delete customers, which the API never needed. The entity gains one method, `change` (**Listing 13.4**); a customer's creation time still has no setter at all, because it never changes.

Listing 13.4 `Customer.java`: Chapter 11's customer with one new method, `change`.

```
package com.profitina.laundry.domain;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import java.time.LocalDateTime;

@Entity
@Table(name = "customers")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 80)
    private String name;

    @Column(nullable = false, length = 20, unique = true)
    private String phone;

    @Column(length = 200)
    private String address;

    @Column(nullable = false)
    private LocalDateTime createdAt;

    protected Customer() {                // for JPA only
    }

    public Customer(String name, String phone, String address,
                    LocalDateTime createdAt) {
        this.name = name;
        this.phone = phone;
        this.address = address;
        this.createdAt = createdAt;
    }
```

Listing 13.4 (Continued) *Customer.java: Chapter 11's customer with one new method, change.*

```
    /** The only fields a form may change; created_at never changes. */
    public void change(String name, String phone, String address) {
        this.name = name;
        this.phone = phone;
        this.address = address;
    }

    public Long getId() { return id; }
    public String getName() { return name; }
    public String getPhone() { return phone; }
    public String getAddress() { return address; }
    public LocalDateTime getCreatedAt() { return createdAt; }
}
```

The repository gains three derived queries (**Listing 13.5**).

Listing 13.5 `CustomerRepository.java`: queries derived from method names.

```
package com.profitina.laundry.repository;

import com.profitina.laundry.domain.Customer;
import java.util.List;
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;

public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Optional<Customer> findByPhone(String phone);

    boolean existsByPhone(String phone);

    boolean existsByPhoneAndIdNot(String phone, Long id);

    List<Customer> findByPhoneContainingOrderByName(String digits);

    List<Customer> findByNameContainingIgnoreCaseOrderByName(String name);
}
```

`existsByPhoneAndIdNot` answers the uniqueness question for an edit: when customer 5 is saved with a phone number, no *other* customer may have it. `findByNameContainingIgnoreCaseOrderByName` becomes `where lower(name) like lower(?)` with the argument wrapped in `%` signs, bound as a parameter, so a customer named `' or 1=1 --` is merely an odd name [3]. **Listing 13.6** is the service.

Listing 13.6 `CustomerService.java`: search, create, update and delete customers, inside transactions.

```java
package com.profitina.laundry.service;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.repository.CustomerRepository;
import com.profitina.laundry.repository.OrderRepository;
import java.time.Clock;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.List;
import java.util.Optional;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/** Everything the shop does with customers, in one place. */
@Service
@Transactional
public class CustomerService {

    private final CustomerRepository customers;
    private final OrderRepository orders;
    private final Clock clock;

    public CustomerService(CustomerRepository customers,
                           OrderRepository orders, Clock clock) {
        this.customers = customers;
        this.orders = orders;
        this.clock = clock;
    }

    /** By name, or by phone when the text is digits. */
    @Transactional(readOnly = true)
    public List<Customer> search(String text) {
        String q = text == null ? "" : text.trim();
        if (q.isEmpty()) {
            return customers.findAll(Sort.by("name"));
        }
        String digits = q.replaceAll("[^0-9]", "");
        return q.matches("[0-9 +-]+")
                ? customers.findByPhoneContainingOrderByName(digits)
                : customers.findByNameContainingIgnoreCaseOrderByName(q);
    }
```

Listing 13.6 (Continued) *CustomerService.java: search, create, update and delete customers, inside transactions.*

```
    @Transactional(readOnly = true)
    public Optional<Customer> findByPhone(String phone) {
        return customers.findByPhone(phone.replaceAll("[^0-9]", ""));
    }

    @Transactional(readOnly = true)
    public Customer get(Long id) {
        return customers.findById(id)
                .orElseThrow(() -> new NotFoundException("customer", id));
    }

    public Customer create(String name, String phone, String address) {
        if (customers.existsByPhone(phone)) {     // Laravel: unique rule
            throw new RuleException("phone",
                    "The phone has already been taken.");
        }
        return customers.save(new Customer(name.trim(), phone,
                blankToNull(address), now()));
    }

    public Customer update(Long id, String name, String phone,
                           String address) {
        Customer customer = get(id);
        if (customers.existsByPhoneAndIdNot(phone, id)) {
            throw new RuleException("phone",
                    "The phone has already been taken.");
        }
        customer.change(name.trim(), phone, blankToNull(address));
        return customer;            // saved at commit (dirty checking)
    }

    public void delete(Long id) {
        Customer customer = get(id);
        if (orders.existsByCustomerId(id)) {
            throw new ConflictException(customer.getName()
                    + " has orders and cannot be deleted.");
        }
        customers.delete(customer);
    }

    private LocalDateTime now() {
        return LocalDateTime.now(clock).truncatedTo(ChronoUnit.SECONDS);
    }
```

Listing 13.6 (Continued) *CustomerService.java: search, create, update and delete customers, inside transactions.*

```
    private static String blankToNull(String s) {
        return s == null || s.isBlank() ? null : s.trim();
    }
}
```

The constructor receives two repositories and a `Clock`. It does not create them: Spring sees that `CustomerService` is a `@Service` with one constructor, finds beans of the three types and passes them in. This is *constructor injection*, the form of dependency injection that Chapter 2 recommended and that makes the class easy to test with stand-ins. The `Clock` is a new bean (**Listing 13.7**); services ask it for the time instead of calling `LocalDateTime.now()`, so that a test can fix the time to “25 September 2026, 09:00 in Surabaya”.

Listing 13.7 `LaundryApplication.java`: the entry point, and the shop's clock in Western Indonesian Time.

```
package com.profitina.laundry;

import java.time.Clock;
import java.time.ZoneId;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class LaundryApplication {

    public static void main(String[] args) {
        SpringApplication.run(LaundryApplication.class, args);
    }

    /** The shop's clock, Surabaya time; a test can pass a fixed one. */
    @Bean
    Clock clock() {
        return Clock.system(ZoneId.of("Asia/Jakarta"));
    }
}
```

The class-level `@Transactional` makes each public method one transaction; `search`, `findByPhone` and `get` narrow it to `readOnly = true`, a hint that lets Hibernate skip looking for changes. The methods take plain values, not form objects: a service must not know whether it was called from a page or from JSON, and that keeps the dependency pointing downwards, as in **Figure 13.1**.

`update` looks as if it forgot to save. It loads the customer, calls `change`, and returns. This is JPA's *dirty checking*: an entity loaded inside a transaction is *managed*, and when the transaction commits, Hibernate compares it with the state it loaded and writes an `UPDATE` for whatever changed. `delete` protects the shop's history: a customer who has orders cannot be removed, because the orders would lose their owner. The database would refuse too, through the foreign key, but with a message no cashier can read; the service refuses first, with a sentence, as a `ConflictException`.

13.4.3 Orders

Listing 13.8 adds `existsByCustomerId` to Chapter 12's order repository; the rest is unchanged, including the entity graphs that load an order's customer and service in the same query.

Listing 13.8 `OrderRepository.java`: pages, one order with its details, and the report queries.

```
package com.profitina.laundry.repository;

import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.EntityGraph;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface OrderRepository extends JpaRepository<Order, Long> {

    @EntityGraph(attributePaths = {"customer", "service"})
    Page<Order> findAllByOrderByReceivedAtDescIdDesc(Pageable pageable);

    @EntityGraph(attributePaths = {"customer", "service"})
    Page<Order> findByStatusOrderByReceivedAtDescIdDesc(OrderStatus status,
                                                        Pageable pageable);

    @EntityGraph(attributePaths = {"customer", "service", "payments"})
    Optional<Order> findWithDetailsById(Long id);

    boolean existsByCustomerId(Long customerId);

    /** One row per status: the shape the daily report needs. */
    interface StatusCount {
        OrderStatus getStatus();

        long getN();
    }

    @Query("""
            select o.status as status, count(o) as n
            from Order o
            where o.receivedAt >= :from and o.receivedAt < :to
            group by o.status""")
    List<StatusCount> countByStatusBetween(@Param("from") LocalDateTime from,
                                           @Param("to") LocalDateTime to);
```

Listing 13.8 (Continued) *OrderRepository.java: pages, one order with its details, and the report queries.*

```
    @Query("""
            select coalesce(sum(o.total), 0)
            from Order o
            where o.receivedAt >= :from and o.receivedAt < :to""")
    long billedBetween(@Param("from") LocalDateTime from,
                       @Param("to") LocalDateTime to);
}
```

Listing 13.9 is where the shop's work now happens.

Listing 13.9 `OrderService.java`: taking orders, moving them along and taking money, with every rule in one place.

```
package com.profitina.laundry.service;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.domain.LaundryRules;
import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.domain.PaymentMethod;
import com.profitina.laundry.domain.Service;
import com.profitina.laundry.repository.CustomerRepository;
import com.profitina.laundry.repository.OrderRepository;
import com.profitina.laundry.repository.PaymentRepository;
import com.profitina.laundry.repository.ServiceRepository;
import java.math.BigDecimal;
import java.time.Clock;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.transaction.annotation.Transactional;

/** Taking orders, moving them along, taking money: the shop's work. */
// Written in full: the entity "Service" already owns the short name.
@org.springframework.stereotype.Service
@Transactional
public class OrderService {

    /** What a new order needs, from a form or from JSON alike. */
    public record NewOrder(Long customerId, String serviceId,
                           BigDecimal quantity, boolean pickup,
                           String deliveryAddress, String note) {
    }

    private final OrderRepository orders;
    private final CustomerRepository customers;
    private final ServiceRepository services;
    private final PaymentRepository payments;
    private final Clock clock;
```

Listing 13.9 (Continued) *OrderService.java: taking orders, moving them along and taking money, with every rule in one place.*

```
    public OrderService(OrderRepository orders, CustomerRepository customers,
                        ServiceRepository services,
                        PaymentRepository payments, Clock clock) {
        this.orders = orders;
        this.customers = customers;
        this.services = services;
        this.payments = payments;
        this.clock = clock;
    }

    /** Page numbers start at 1 for people; Spring Data counts from 0. */
    @Transactional(readOnly = true)
    public Page<Order> page(OrderStatus status, int page, int size) {
        PageRequest request = PageRequest.of(Math.max(page, 1) - 1, size);
        return status == null
                ? orders.findAllByOrderByReceivedAtDescIdDesc(request)
                : orders.findByStatusOrderByReceivedAtDescIdDesc(status,
                        request);
    }

    @Transactional(readOnly = true)
    public Order get(Long id) {
        return orders.findWithDetailsById(id)
                .orElseThrow(() -> new NotFoundException("order", id));
    }

    @Transactional(readOnly = true)
    public List<Service> activeServices() {
        return services.findByActiveTrueOrderByIdAsc();
    }

    @Transactional(readOnly = true)
    public Service service(String id) {
        return services.findById(id)
                .orElseThrow(() -> new NotFoundException("service", id));
    }
```

Listing 13.9 (Continued) *OrderService.java: taking orders, moving them along and taking money, with every rule in one place.*

```
    /** The rules that need the database; an empty map means all is well. */
    @Transactional(readOnly = true)
    public Map<String, List<String>> check(NewOrder in) {
        Map<String, List<String>> errors = new LinkedHashMap<>();
        if (in.customerId() != null
                && !customers.existsById(in.customerId())) {
            add(errors, "customer_id", "The selected customer is invalid.");
        }
        if (in.serviceId() != null && !in.serviceId().isBlank()
                && services.findById(in.serviceId())
                        .filter(Service::isActive).isEmpty()) {
            add(errors, "service_id", "The selected service is invalid.");
        }
        if (in.pickup() && (in.deliveryAddress() == null
                || in.deliveryAddress().isBlank())) {
            add(errors, "delivery_address",
                    "Where should the laundry be delivered?");
        }
        return errors;
    }

    public Order create(NewOrder in) {
        Map<String, List<String>> errors = check(in);
        if (!errors.isEmpty()) {
            throw new RuleException(errors);
        }
        Customer customer = customers.findById(in.customerId()).orElseThrow();
        Service service = services.findById(in.serviceId()).orElseThrow();
        return orders.save(new Order(customer, service, in.quantity(),
                in.pickup(), in.pickup() ? in.deliveryAddress() : null,
                in.note(), now()));
    }

    /**
     * One step along queued -> finished -> departed. With a wanted status
     * (the API sends one) it must be exactly the next step.
     */
    public Order advance(Long id, OrderStatus wanted) {
        Order order = get(id);
        ensureOpen(order);
        OrderStatus next;
        try {
            next = LaundryRules.advance(order.getStatus(), order.balance());
        } catch (IllegalStateException e) {
            throw new RuleException("status", e.getMessage());
        }
```

Listing 13.9 (Continued) *OrderService.java: taking orders, moving them along and taking money, with every rule in one place.*

```
        if (wanted != null && wanted != next) {
            throw new RuleException("status", "An order that is "
                    + order.getStatus().value() + " can only become "
                    + next.value() + ".");
        }
        order.advance(now());
        return order;
    }

    public Order pay(Long id, long amount, PaymentMethod method) {
        Order order = get(id);
        ensureOpen(order);
        if (amount > order.balance()) {
            throw new RuleException("amount", "Only "
                    + LaundryRules.rupiah(order.balance())
                    + " is still owed.");
        }
        payments.save(order.addPayment(amount, method, now()));
        return order;
    }

    /** Laundry that has left the shop takes no more steps or money. */
    private static void ensureOpen(Order order) {
        if (order.getStatus() == OrderStatus.DEPARTED) {
            throw new ConflictException("Order #" + order.getId()
                    + " has left the shop and is closed.");
        }
    }

    private static void add(Map<String, List<String>> errors, String field,
                            String message) {
        errors.computeIfAbsent(field, k -> new ArrayList<>()).add(message);
    }

    private LocalDateTime now() {
        return LocalDateTime.now(clock).truncatedTo(ChronoUnit.SECONDS);
    }
}
```

The first thing to notice is the annotation written in full, `@org.springframework.stereotype.Service`. Chapter 11 met the same clash: the entity `Service` owns the short name in this file, and Java cannot import two different classes called `Service` into one file. The comment in the listing says why, so the next reader does not "fix" it.

`NewOrder` is a small record that describes a new order independently of where it came from; the JSON controller builds it from `CreateOrderRequest`, the page controller from a form. The rules that need the database live in `check`, which returns a map of errors instead of throwing. That design keeps a promise of the contract: a bad order is reported with *all* its errors at once, the ones Bean Validation finds (a missing customer, a quantity of 0) together with the ones only the database can find (a customer that does not exist, a withdrawn service) and the pickup rule. The controller merges both lists and answers once. `create` calls `check` again and throws if anything is wrong, so a caller that forgets to check cannot slip a bad order past the service.

`advance` is the old `updateStatus` without the HTTP. It loads the order, refuses a closed order with `ConflictException`, asks `LaundryRules.advance` for the next step (which also refuses to let unpaid laundry depart), and, when the caller named the status it wanted, insists that it is exactly the next one. The API always names it (`{"status": "finished"}`); the page's button does not, and passes `null`, meaning "whatever comes next". `pay` refuses an amount above the balance and

records the payment. Because each method runs in one transaction, reading the balance and inserting the payment happen together.

Money is still handled by the rules of Chapter 11, and it is worth restating them, because every page in this chapter shows their results. If q is the quantity and $h = 100q$ the quantity in hundredths (4.5 kg is 450), the billed hundredths b are $\max(h, 300)$ for a per-kilogram service and h rounded up to whole items otherwise. With a price p per unit, the subtotal and the total are

$$S = \left\lfloor \frac{p \cdot b + 50}{100} \right\rfloor, \qquad T = S + \begin{cases} 10000 & \text{pickup and } S < 100000 \\ 0 & \text{otherwise} \end{cases} \tag{13.1}$$

so 2.5 kg of wash-and-fold at Rp 8.000 with pickup costs $8000 \times 300/100 = 24000$ plus the Rp 10.000 fee, Rp 34.000. If the payments add up to P, the balance shown on the order page is

$$B = T - P \tag{13.2}$$

and the service never lets P grow past T, so B never becomes negative.

13.4.4 The daily report

Listing 13.10 moves the arithmetic of Chapter 12's report into a service.

Listing 13.10 `ReportService.java`: the figures of one day.

```
package com.profitina.laundry.service;

import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.repository.OrderRepository;
import com.profitina.laundry.repository.PaymentRepository;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.EnumMap;
import java.util.Map;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/** The figures of one day, for the owner's report. */
@Service
@Transactional(readOnly = true)
public class ReportService {

    /** Orders per status, money billed and money collected on one day. */
    public record DayFigures(LocalDate day, Map<OrderStatus, Long> byStatus,
                             long billed, long collected) {

        public long received() {
            return byStatus.values().stream().mapToLong(Long::longValue).sum();
        }
    }

    private final OrderRepository orders;
    private final PaymentRepository payments;

    public ReportService(OrderRepository orders, PaymentRepository payments) {
        this.orders = orders;
        this.payments = payments;
    }

    public DayFigures daily(LocalDate day) {
        LocalDateTime from = day.atStartOfDay();
        LocalDateTime to = from.plusDays(1);   // [from, to): one whole day
        Map<OrderStatus, Long> byStatus = new EnumMap<>(OrderStatus.class);
        for (OrderStatus s : OrderStatus.values()) {
            byStatus.put(s, 0L);               // every status, even if 0
        }
        for (OrderRepository.StatusCount c
                : orders.countByStatusBetween(from, to)) {
            byStatus.put(c.getStatus(), c.getN());
        }
        return new DayFigures(day, byStatus, orders.billedBetween(from, to),
```

Listing 13.10 (Continued) *ReportService.java: the figures of one day.*

```
                payments.collectedBetween(from, to));
    }
}
```

It returns `DayFigures`, a record that uses the `OrderStatus` enum as map keys; turning the enum into the words `queued`, `finished` and `departed` of the JSON is the controller's job. An `EnumMap` keeps the statuses in their natural order, so the report always lists them the same way.

13.5 Thin API Controllers

With the work in the services, the JSON controllers become short. **Listing 13.11** is the order controller; compare it with Chapter 12's.

Listing 13.11 `OrderApiController.java`: read JSON, call `OrderService`, write JSON.

```java
package com.profitina.laundry.api;

import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.domain.PaymentMethod;
import com.profitina.laundry.service.OrderService;
import com.profitina.laundry.service.OrderService.NewOrder;
import com.profitina.laundry.service.RuleException;
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Valid;
import jakarta.validation.Validator;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

/** Thin: read JSON, call OrderService, write JSON. No rules, no SQL. */
@RestController
@RequestMapping("/api/orders")
public class OrderApiController {

    private final OrderService orders;
    private final Validator validator;

    public OrderApiController(OrderService orders, Validator validator) {
        this.orders = orders;
        this.validator = validator;
    }

    @GetMapping
    public PageView<OrderView> index(
            @RequestParam(required = false) String status,
            @RequestParam(defaultValue = "1") int page) {
        OrderStatus wanted = status == null ? null : parseStatus(status);
        return PageView.of(orders.page(wanted, page, 20), OrderView::of);
    }
```

Listing 13.11 (Continued) *OrderApiController.java: read JSON, call OrderService, write JSON.*

```
    @GetMapping("/{id}")
    public DataView<OrderView> show(@PathVariable Long id) {
        return new DataView<>(OrderView.withPayments(orders.get(id)));
    }

    /** Every broken rule is reported at once, as Laravel does. */
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public DataView<OrderView> store(@RequestBody CreateOrderRequest in) {
        NewOrder order = new NewOrder(in.customerId(), in.serviceId(),
                in.quantity(), in.pickup(), in.deliveryAddress(), in.note());
        Map<String, List<String>> errors = new LinkedHashMap<>();
        for (ConstraintViolation<CreateOrderRequest> v
                : validator.validate(in)) {
            errors.computeIfAbsent(ApiExceptionHandler.snake(
                    v.getPropertyPath().toString()), k -> new ArrayList<>())
                    .add(v.getMessage());
        }
        orders.check(order).forEach((field, messages) -> errors
                .computeIfAbsent(field, k -> new ArrayList<>())
                .addAll(messages));
        if (!errors.isEmpty()) {
            throw new RuleException(errors);
        }
        return new DataView<>(OrderView.of(orders.create(order)));
    }

    @PatchMapping("/{id}/status")
    public DataView<OrderView> updateStatus(@PathVariable Long id,
            @Valid @RequestBody StatusRequest in) {
        return new DataView<>(OrderView.of(
                orders.advance(id, parseStatus(in.status()))));
    }

    @PostMapping("/{id}/payments")
    @ResponseStatus(HttpStatus.CREATED)
    public DataView<OrderView> pay(@PathVariable Long id,
            @Valid @RequestBody CreatePaymentRequest in) {
        PaymentMethod method;
        try {
            method = PaymentMethod.fromValue(in.method());
        } catch (IllegalArgumentException e) {
            throw new RuleException("method",
                    "The selected method is invalid.");
        }
        return new DataView<>(OrderView.withPayments(
```

Listing 13.11 (Continued) *OrderApiController.java: read JSON, call OrderService, write JSON.*

```
                orders.pay(id, in.amount(), method)));
    }

    private static OrderStatus parseStatus(String value) {
        try {
            return OrderStatus.fromValue(value);
        } catch (IllegalArgumentException e) {
            throw new RuleException("status",
                    "The selected status is invalid.");
        }
    }
}
```

`store` shows the merged validation. Spring's `Validator` checks the Bean Validation constraints of `CreateOrderRequest` (the record is unchanged from Chapter 12), `orders.check` adds the

database rules, and only if both are empty does `create` run. `ApiExceptionHandler.snake` turns `customerId`, the Java name of a record component, into `customer_id`, the name the client sent. Nothing in the class starts a transaction, touches a repository, or knows the pickup fee. The other three JSON controllers are equally thin (**Listing 13.12**, **Listing 13.13**, **Listing 13.14**).

Listing 13.12 `CustomerApiController.java`: customers in JSON.

```
package com.profitina.laundry.api;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.service.CustomerService;
import jakarta.validation.Valid;
import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/customers")
public class CustomerApiController {

    private final CustomerService customers;

    public CustomerApiController(CustomerService customers) {
        this.customers = customers;
    }

    @GetMapping
    public DataView<List<CustomerView>> index(
            @RequestParam(required = false) String phone) {
        List<Customer> found = phone == null ? customers.search("")
                : customers.findByPhone(phone).stream().toList();
        return new DataView<>(found.stream().map(CustomerView::of).toList());
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public DataView<CustomerView> store(
            @Valid @RequestBody CreateCustomerRequest in) {
        return new DataView<>(CustomerView.of(
                customers.create(in.name(), in.phone(), in.address())));
    }
}
```

Listing 13.13 `ServiceApiController.java`: the price list in JSON.

```
package com.profitina.laundry.api;

import com.profitina.laundry.service.OrderService;
import java.util.List;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/services")
public class ServiceApiController {

    private final OrderService orders;

    public ServiceApiController(OrderService orders) {
        this.orders = orders;
    }

    @GetMapping
    public DataView<List<ServiceView>> index() {
        return new DataView<>(orders.activeServices().stream()
                .map(ServiceView::of).toList());
    }

    @GetMapping("/{id}")
    public DataView<ServiceView> show(@PathVariable String id) {
        return new DataView<>(ServiceView.of(orders.service(id)));
    }
}
```

Listing 13.14 `ReportApiController.java`: the daily report in JSON.

```
package com.profitina.laundry.api;

import com.profitina.laundry.service.ReportService;
import com.profitina.laundry.service.ReportService.DayFigures;
import com.profitina.laundry.service.RuleException;
import java.time.LocalDate;
import java.time.format.DateTimeParseException;
import java.util.LinkedHashMap;
import java.util.Map;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ReportApiController {

    private final ReportService reports;

    public ReportApiController(ReportService reports) {
        this.reports = reports;
    }

    /** GET /api/reports/daily?date=2026-09-25 */
    @GetMapping("/api/reports/daily")
    public DataView<DailyReport> daily(
            @RequestParam(required = false) String date) {
        DayFigures f = reports.daily(parseDate(date));
        Map<String, Long> byStatus = new LinkedHashMap<>();
        f.byStatus().forEach((status, n) -> byStatus.put(status.value(), n));
        return new DataView<>(new DailyReport(f.day().toString(),
                f.received(), f.billed(), f.collected(), byStatus));
    }

    private static LocalDate parseDate(String date) {
        if (date == null || date.isBlank()) {
            throw new RuleException("date", "The date field is required.");
        }
        try {
            return LocalDate.parse(date);
        } catch (DateTimeParseException e) {
            throw new RuleException("date",
                    "The date field must match the format Y-m-d.");
        }
    }
}
```

Parsing the date stays in the report controller, because “the text in the query string is not a date” is a question about HTTP input, not about the shop. Finally, **Listing 13.15** is Chapter 12’s error handler, now listening for the service’s exceptions.

Listing 13.15 `ApiExceptionHandler.java`: every API error in the contract’s shape.

```java
package com.profitina.laundry.api;

import com.profitina.laundry.service.ConflictException;
import com.profitina.laundry.service.NotFoundException;
import com.profitina.laundry.service.RuleException;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.server.ResponseStatusException;

/** Every error in the contract's shape: {"message": ..., "errors": ...}. */
@RestControllerAdvice(basePackages = "com.profitina.laundry.api")
public class ApiExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, Object>> invalid(
            MethodArgumentNotValidException e) {
        Map<String, List<String>> errors = new LinkedHashMap<>();
        for (FieldError f : e.getBindingResult().getFieldErrors()) {
            errors.computeIfAbsent(snake(f.getField()),
                    k -> new ArrayList<>()).add(f.getDefaultMessage());
        }
        return unprocessable(errors);
    }

    @ExceptionHandler(RuleException.class)
    public ResponseEntity<Map<String, Object>> rule(
            RuleException e) {
        return unprocessable(e.getErrors());
    }
```

Listing 13.15 (Continued) *ApiExceptionHandler.java: every API error in the contract's shape.*

```
    @ExceptionHandler(ResponseStatusException.class)
    public ResponseEntity<Map<String, Object>> status(
            ResponseStatusException e) {
        String reason = e.getReason() == null ? "" : e.getReason();
        return ResponseEntity.status(e.getStatusCode())
                .body(Map.of("message", reason));
    }

    @ExceptionHandler(NotFoundException.class)
    public ResponseEntity<Map<String, Object>> notFound(NotFoundException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(Map.of("message", e.getMessage()));
    }

    @ExceptionHandler(ConflictException.class)
    public ResponseEntity<Map<String, Object>> conflict(ConflictException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT)
                .body(Map.of("message", e.getMessage()));
    }

    @ExceptionHandler(HttpMessageNotReadableException.class)
    public ResponseEntity<Map<String, Object>> unreadable(
            HttpMessageNotReadableException e) {
        return ResponseEntity.badRequest().body(Map.of("message",
                "The request body is not valid JSON for this endpoint."));
    }

    private static ResponseEntity<Map<String, Object>> unprocessable(
            Map<String, List<String>> errors) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("message", "The given data was invalid.");
        body.put("errors", errors);
        return ResponseEntity.status(HttpStatus.UNPROCESSABLE_CONTENT)
                .body(body);
    }

    /** customerId -> customer_id: the name the client sent. */
    static String snake(String name) {
        return name.replaceAll("([a-z0-9])([A-Z])", "$1_$2").toLowerCase();
    }
}
```

`basePackages = "com.profitina.laundry.api"` limits the handler to the JSON controllers. Without it, a `RuleException` thrown while a *page* is being served would also be answered with JSON and status 422, which is exactly what a browser must not receive. The mapping is the contract's: `RuleException` becomes 422 with `errors`, `NotFoundException` 404 and `ConflictException` 409, each with a `message`.

Because the URLs, the request records, the view records and the error shapes are those of Chapter 12, the contract program of Chapter 12 applies unchanged; running it against this chapter's application is the quickest way to prove that the refactoring changed nothing a client can see.

13.6 Validating Form Input

13.6.1 Jakarta Bean Validation

Spring Boot validates with *Jakarta Bean Validation* 3.1, implemented by Hibernate Validator and brought in by `spring-boot-starter-validation` [4]. You put constraint annotations on fields; a

validator checks an object against them and returns a list of violations, each with a field name and a message. **Table 13.3** lists the constraints the forms of this chapter use.

Table 13.3 The Bean Validation constraints used by the forms.

Annotation	Checks that the value…	Used on
`@NotBlank`	is not null and contains a non-space character	name, phone, service, payment method
`@NotNull`	is present	customer, quantity, amount
`@Size(min, max)`	has between `min` and `max` characters	name, address, note
`@Pattern(regexp = r)`	matches the regular expression *r*	phone
`@DecimalMin`, `@DecimalMax`	lies between two decimal bounds	quantity
`@Digits(integer, fraction)`	has at most so many digits before and after the point	quantity
`@Min`	is at least a whole number	payment amount

Two facts save a lot of confusion. Most constraints treat `null` as *valid*, so an optional address with `@Size(max = 200)` may be left empty; a required field says so with `@NotNull` or `@NotBlank`. And constraints check *form*, not *meaning*: whether a phone number is already taken, or whether an order still owes money, can only be answered with the database, which is why those checks stay in the services.

13.6.2 Form records

Listing 13.16 is what the customer form sends.

Listing 13.16 `CustomerForm.java`: the customer form, with the same limits as the API's request.

```
package com.profitina.laundry.web;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;

/** What the customer form sends; never the entity itself. */
public record CustomerForm(
        @NotBlank(message = "Please enter the customer's name.")
        @Size(min = 2, max = 80) String name,
        @NotBlank(message = "Please enter a phone number.")
        @Pattern(regexp = "^08[0-9]{8,11}$",
                 message = "Write the phone as 08..., 10 to 13 digits.")
        String phone,
        @Size(max = 200) String address) {

    /** Keep only digits, as the API does: "0812-3456-7890" -> digits. */
    public CustomerForm {
        phone = phone == null ? null : phone.replaceAll("[^0-9]", "");
    }

    public static CustomerForm empty() {
        return new CustomerForm("", "", "");
    }
}
```

A Java *record* is a compact class whose components are fixed at construction; Spring MVC creates it from the submitted fields by calling its constructor, and Thymeleaf reads its accessors to fill the inputs again. The *compact constructor* runs before the components are stored and keeps only the

digits of the phone, exactly as Chapter 11's `CreateCustomerRequest` does, so "0812-3456-7890" is checked and saved as `081234567890`.

Why not bind the form directly to the `Customer` entity? Because the entity has fields no form may set, such as `id` and `createdAt`. If the entity were the form object, anyone could add `createdAt=2001-01-01` to the request and Spring would bind it. This attack is called *mass assignment*, and a separate record closes it by construction: only its components can be bound. **Listing 13.17** and **Listing 13.18** describe the other two forms.

Listing 13.17 `OrderForm.java`: the new-order form, which knows how to become a `NewOrder`.

```
package com.profitina.laundry.web;

import com.profitina.laundry.service.OrderService.NewOrder;
import jakarta.validation.constraints.DecimalMax;
import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Digits;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;
import java.math.BigDecimal;

/** The new-order form: the same limits as CreateOrderRequest. */
public record OrderForm(
        @NotNull(message = "Choose a customer.") Long customerId,
        @NotBlank(message = "Choose a service.") String serviceId,
        @NotNull(message = "Enter the weight or the number of items.")
        @DecimalMin("0.1") @DecimalMax("100")
        @Digits(integer = 3, fraction = 2) BigDecimal quantity,
        boolean pickup,
        @Size(max = 200) String deliveryAddress,
        @Size(max = 200) String note) {

    public NewOrder toNewOrder() {
        return new NewOrder(customerId, serviceId, quantity, pickup,
                deliveryAddress, note);
    }
}
```

Listing 13.18 `PaymentForm.java`: the payment form.

```
package com.profitina.laundry.web;

import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;

public record PaymentForm(
        @NotNull(message = "Enter the amount.")
        @Min(value = 1000, message = "At least Rp 1.000.") Long amount,
        @NotBlank String method) {
}
```

The fields are wrapper types (`Long`, `BigDecimal`), not primitives, so that an empty field arrives as `null` and `@NotNull` reports it in plain words; a primitive `long` would turn an empty field into a conversion error with a technical message.

13.6.3 Rule messages on the right field

The services speak the API's field names, `customer_id` and `delivery_address`; the forms have `customerId` and `deliveryAddress`. **Listing 13.19** translates one into the other and attaches each message to the form field it belongs to.

Listing 13.19 `Forms.java`: the service's rule messages become errors on form fields.

```
package com.profitina.laundry.web;

import java.util.List;
import java.util.Map;
import org.springframework.validation.BindingResult;

/** Puts the service's rule messages next to the right form fields. */
final class Forms {

    private Forms() {
    }

    /** {"customer_id": [...]} -> an error on the form field customerId. */
    static void reject(BindingResult result, Map<String, List<String>> errors) {
        errors.forEach((field, messages) -> messages.forEach(
                message -> result.rejectValue(camel(field), "rule", message)));
    }

    /** customer_id -> customerId: the rules speak the API's names. */
    static String camel(String snake) {
        StringBuilder out = new StringBuilder();
        boolean up = false;
        for (char c : snake.toCharArray()) {
            if (c == '_') {
                up = true;
            } else {
                out.append(up ? Character.toUpperCase(c) : c);
                up = false;
            }
        }
        return out.toString();
    }
}
```

`BindingResult` is Spring's container for everything that went wrong while binding and validating one form object; `rejectValue` adds an error to a field, and the template shows it next to that field exactly like a Bean Validation message.

13.7 Pages with Thymeleaf

13.7.1 The customer pages

Listing 13.20 is the complete customer controller: list and search, new, create, edit, update and delete.

Listing 13.20 `CustomerController.java`: the full CRUD for customers as pages.

```java
package com.profitina.laundry.web;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.service.ConflictException;
import com.profitina.laundry.service.CustomerService;
import com.profitina.laundry.service.RuleException;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

/** The customer pages: list and search, add, edit, delete. */
@Controller
@RequestMapping("/customers")
public class CustomerController {

    private final CustomerService customers;

    public CustomerController(CustomerService customers) {
        this.customers = customers;
    }

    @GetMapping
    public String list(@RequestParam(required = false) String q, Model model) {
        model.addAttribute("customers", customers.search(q));
        model.addAttribute("q", q);
        return "customers/list";
    }

    @GetMapping("/new")
    public String newForm(Model model) {
        model.addAttribute("form", CustomerForm.empty());
        return "customers/form";
    }
```

Listing 13.20 (Continued) *CustomerController.java: the full CRUD for customers as pages.*

```
    @PostMapping
    public String create(@Validated @ModelAttribute("form") CustomerForm form,
                         BindingResult errors, RedirectAttributes flash) {
        if (errors.hasErrors()) {
            return "customers/form";
        }
        try {
            Customer c = customers.create(form.name(), form.phone(),
                    form.address());
            flash.addFlashAttribute("message", c.getName() + " was added.");
            return "redirect:/customers";
        } catch (RuleException e) {
            Forms.reject(errors, e.getErrors());
            return "customers/form";
        }
    }

    @GetMapping("/{id}/edit")
    public String editForm(@PathVariable Long id, Model model) {
        Customer c = customers.get(id);
        model.addAttribute("form",
                new CustomerForm(c.getName(), c.getPhone(), c.getAddress()));
        model.addAttribute("customerId", id);
        return "customers/form";
    }

    @PostMapping("/{id}")
    public String update(@PathVariable Long id,
                         @Validated @ModelAttribute("form") CustomerForm form,
                         BindingResult errors, Model model,
                         RedirectAttributes flash) {
        model.addAttribute("customerId", id);
        if (errors.hasErrors()) {
            return "customers/form";
        }
        try {
            customers.update(id, form.name(), form.phone(), form.address());
            flash.addFlashAttribute("message", "Changes saved.");
            return "redirect:/customers";
        } catch (RuleException e) {
            Forms.reject(errors, e.getErrors());
            return "customers/form";
        }
    }
```

Listing 13.20 (Continued) *CustomerController.java: the full CRUD for customers as pages.*

```
    @PostMapping("/{id}/delete")
    public String delete(@PathVariable Long id, RedirectAttributes flash) {
        try {
            customers.delete(id);
            flash.addFlashAttribute("message", "Customer deleted.");
        } catch (ConflictException e) {
            flash.addFlashAttribute("error", e.getMessage());
        }
        return "redirect:/customers";
    }
}
```

The mapping is the conventional one for pages rendered on the server (**Table 13.4**).

Table 13.4 The URLs of the customer pages.

Method and URL	Action	Answer
`GET /customers?q=...`	list and search	the list page
`GET /customers/new`	show an empty form	the form page
`POST /customers`	create	redirect to the list, or the form with errors
`GET /customers/{id}/edit`	show a filled form	the form page
`POST /customers/{id}`	update	redirect to the list, or the form with errors
`POST /customers/{id}/delete`	delete	redirect to the list with a message

HTML forms can only send `GET` and `POST`, so updating and deleting use `POST`. Deleting is never a `GET`: search engines and browsers that prefetch follow links, and a link that deletes would delete.

Look at `create`. The parameter `@Validated @ModelAttribute("form") CustomerForm form` asks Spring MVC to build the record from the request, validate it, and put it in the model under the name `form`. The parameter right after it, `BindingResult errors`, receives the validation errors. It *must* come immediately after the validated parameter; if it is missing, Spring throws an exception instead of letting the method handle the errors. When there are errors the method returns the form template again, with the user's typing and the messages still in the model. When the service refuses ("The phone has already been taken."), `Forms.reject` adds the refusal to the same `BindingResult`, and the page shows it under the phone box.

13.7.2 Post/Redirect/Get

When a customer is created successfully, the method does not render the list. It returns `"redirect:/customers"`, which makes the browser load the list with a new `GET`. **Figure 13.2** shows why, with the order form as the example.

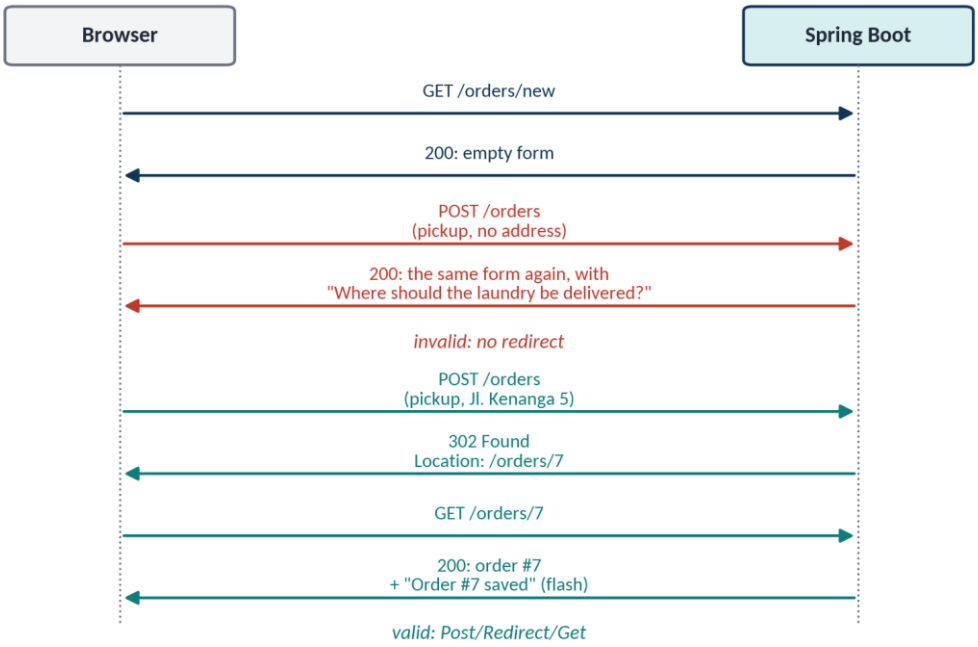


Figure 13.2 Post/Redirect/Get: an invalid form is answered with the same page and its messages; a valid one with a redirect, so that refreshing the page never takes the same order twice.

If the page were rendered as the answer to the `POST`, pressing F5 (or going back and forward) would make the browser send the `POST` again, and the shop would have two identical orders. After a redirect, the page on screen is the answer to a harmless `GET` that may be repeated any number of times. The pattern is called *Post/Redirect/Get* (PRG) and relies on the redirect status codes of HTTP [5]. A message such as “Order #7 saved: Rp 34.000.” cannot travel in the model across a redirect, because the model lives for one request only; `RedirectAttributes.addFlashAttribute` keeps it in the session for exactly one following request, which is why it is called a *flash* attribute. An *invalid* form, on the other hand, is answered at once with the form: nothing was saved, so there is nothing to repeat, and the cashier’s typing is not lost.

13.7.3 Templates

Thymeleaf templates are HTML files with extra attributes in the `th:` namespace, kept in `src/main/resources/templates` [6]. They open in a browser as ordinary HTML with their example text, so a designer can work on them without running the application. **Listing 13.21** holds the parts every page shares.

Listing 13.21 `fragments.html`: the page head, the navigation and the flash messages.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:fragment="head(title)">
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title th:text="${title} + ' | Profitina Laundry'">Profitina Laundry</title>
  <link rel="stylesheet" th:href="@{/css/app.css}">
</head>
<body>
<header th:fragment="nav">
  <strong>Profitina Laundry</strong>
  <nav>
    <a th:href="@{/orders(status='queued')}">Queue</a>
    <a th:href="@{/orders}">All orders</a>
    <a th:href="@{/orders/new}">New order</a>
    <a th:href="@{/customers}">Customers</a>
  </nav>
</header>
<div th:fragment="flash">
  <p class="ok" th:if="${message}" th:text="${message}">Saved.</p>
  <p class="bad" th:if="${error}" th:text="${error}">Not allowed.</p>
</div>
</body>
</html>
```

`th:fragment="head(title)"` declares a reusable piece with a parameter; a page includes it with `th:replace="~{fragments :: head('Customers')}"`. `@{/orders}` builds a URL relative to the application’s context path, so the pages keep working if the application is deployed under `/laundry/`. `${message}` reads the flash attribute, and `th:if` removes the element when there is none. **Listing 13.22** is the customer list.

Listing 13.22 `customers/list.html`: the searchable customer list.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="~{fragments :: head('Customers')}"></head>
<body>
<header th:replace="~{fragments :: nav}"></header>
<main>
  <h1>Customers</h1>
  <div th:replace="~{fragments :: flash}"></div>
  <form method="get" th:action="@{/customers}" class="inline">
    <input name="q" th:value="${q}" placeholder="Name or phone">
    <button>Search</button>
    <a class="button" th:href="@{/customers/new}">New customer</a>
  </form>
  <table>
    <thead><tr><th>Name</th><th>Phone</th><th>Address</th><th></th></tr></thead>
    <tbody>
    <tr th:each="c : ${customers}">
      <td th:text="${c.name}">Sari Wulandari</td>
      <td th:text="${c.phone}">081200000101</td>
      <td th:text="${c.address} ?: '-'">Jl. Kertajaya 12</td>
      <td class="actions">
        <a th:href="@{/orders/new(customerId=${c.id})}">New order</a>
        <a th:href="@{/customers/{id}/edit(id=${c.id})}">Edit</a>
        <form method="post" th:action="@{/customers/{id}/delete(id=${c.id})}"
              onsubmit="return confirm('Delete this customer?')">
          <button class="link">Delete</button>
        </form>
      </td>
    </tr>
    <tr th:if="${customers.isEmpty()}"><td colspan="4">No customers found.</td></tr>
    </tbody>
  </table>
</main>
</body>
</html>
```

`th:each` repeats the row once per customer. `th:text` puts text into an element and escapes it: a customer named `<script>alert(1)</script>` is shown as those characters, never executed. This protection against cross-site scripting (XSS) is Thymeleaf's default; `th:utext`, which does not escape, is only for HTML you wrote yourself. The elvis operator `${c.address} ?: '-'` prints a dash for a customer without an address. **Listing 13.23** serves both "new" and "edit".

Listing 13.23 `customers/form.html`: one form for adding and editing, with messages next to each field.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="~{fragments :: head('Customer')}"></head>
<body>
<header th:replace="~{fragments :: nav}"></header>
<main>
  <h1 th:text="${customerId} ? 'Edit customer' : 'New customer'">Customer</h1>
  <form method="post" th:object="${form}"
        th:action="${customerId} ? @{/customers/{id}(id=${customerId})}
                                 : @{/customers}">
    <p class="bad" th:each="err : ${#fields.globalErrors()}" th:text="${err}"></p>
    <label>Name
      <input th:field="*{name}" th:errorclass="invalid" autofocus>
    </label>
    <p class="bad" th:errors="*{name}"></p>
    <label>Phone
      <input th:field="*{phone}" th:errorclass="invalid" placeholder="08...">
    </label>
    <p class="bad" th:errors="*{phone}"></p>
    <label>Address (for pickup and delivery)
      <input th:field="*{address}" th:errorclass="invalid">
    </label>
    <p class="bad" th:errors="*{address}"></p>
    <button>Save</button>
    <a th:href="@{/customers}">Cancel</a>
  </form>
</main>
</body>
</html>
```

`th:object="${form}"` makes the record the object of the form, and `th:field="*{name}"` does three things at once: it sets the input's `name` and `id` to `name`, fills its `value` from the record, and, with `th:errorclass="invalid"`, adds the CSS class `invalid` when the field has an error. `th:errors="*{phone}"` prints the messages for the phone field, whether they came from `@Pattern` or from the service. The form's target depends on whether `customerId` is in the model: an edit posts to `/customers/5`, a new customer to `/customers`.

13.7.4 Orders and payments

Listing 13.24 drives the counter.

Listing 13.24 `OrderController.java`: the queue, a new order, one order, status steps and payments.

```java
package com.profitina.laundry.web;

import com.profitina.laundry.domain.LaundryRules;
import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.domain.PaymentMethod;
import com.profitina.laundry.service.ConflictException;
import com.profitina.laundry.service.CustomerService;
import com.profitina.laundry.service.OrderService;
import com.profitina.laundry.service.RuleException;
import java.util.List;
import java.util.Map;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

/** The counter pages: the queue, a new order, one order and its money. */
@Controller
@RequestMapping("/orders")
public class OrderController {

    private final OrderService orders;
    private final CustomerService customers;

    public OrderController(OrderService orders, CustomerService customers) {
        this.orders = orders;
        this.customers = customers;
    }
```

Listing 13.24 (Continued) *OrderController.java: the queue, a new order, one order, status steps and payments.*

```
    @GetMapping
    public String list(@RequestParam(required = false) String status,
                       @RequestParam(defaultValue = "1") int page,
                       Model model) {
        OrderStatus wanted = parse(status);
        model.addAttribute("orders", orders.page(wanted, page, 30));
        model.addAttribute("status", wanted);
        model.addAttribute("statuses", OrderStatus.values());
        return "orders/list";
    }

    @GetMapping("/new")
    public String newForm(@RequestParam(required = false) Long customerId,
                          Model model) {
        model.addAttribute("form", new OrderForm(customerId, "", null,
                false, "", ""));
        return fillChoices(model);
    }

    @PostMapping
    public String create(@Validated @ModelAttribute("form") OrderForm form,
                         BindingResult errors, Model model,
                         RedirectAttributes flash) {
        Map<String, List<String>> rules = orders.check(form.toNewOrder());
        Forms.reject(errors, rules);            // all problems at once
        if (errors.hasErrors()) {
            return fillChoices(model);
        }
        Order order = orders.create(form.toNewOrder());
        flash.addFlashAttribute("message", "Order #" + order.getId()
                + " saved: " + LaundryRules.rupiah(order.getTotal()) + ".");
        return "redirect:/orders/" + order.getId();
    }

    @GetMapping("/{id}")
    public String detail(@PathVariable Long id, Model model) {
        if (!model.containsAttribute("payment")) {
            model.addAttribute("payment", new PaymentForm(null, "cash"));
        }
        model.addAttribute("order", orders.get(id));
        model.addAttribute("methods", PaymentMethod.values());
        return "orders/detail";
    }
```

Listing 13.24 (Continued) *OrderController.java: the queue, a new order, one order, status steps and payments.*

```
    /** One step along the workflow; the service decides which step. */
    @PostMapping("/{id}/advance")
    public String advance(@PathVariable Long id, RedirectAttributes flash) {
        try {
            Order order = orders.advance(id, null);
            flash.addFlashAttribute("message", "Order #" + id + " is now "
                    + order.getStatus().value() + ".");
        } catch (RuleException e) {
            flash.addFlashAttribute("error",
                    e.getErrors().values().iterator().next().get(0));
        } catch (ConflictException e) {
            flash.addFlashAttribute("error", e.getMessage());
        }
        return "redirect:/orders/" + id;
    }

    @PostMapping("/{id}/payments")
    public String pay(@PathVariable Long id,
                      @Validated @ModelAttribute("payment") PaymentForm payment,
                      BindingResult errors, Model model,
                      RedirectAttributes flash) {
        if (!errors.hasErrors()) {
            try {
                orders.pay(id, payment.amount(),
                        PaymentMethod.fromValue(payment.method()));
                flash.addFlashAttribute("message", "Payment recorded.");
                return "redirect:/orders/" + id;
            } catch (RuleException e) {
                Forms.reject(errors, e.getErrors());
            } catch (ConflictException e) {
                flash.addFlashAttribute("error", e.getMessage());
                return "redirect:/orders/" + id;
            }
        }
        return detail(id, model);
    }

    /** "queued" -> QUEUED; empty or unknown text means "all orders". */
    private static OrderStatus parse(String status) {
        try {
            return status == null ? null : OrderStatus.fromValue(status);
        } catch (IllegalArgumentException e) {
            return null;
        }
    }
```

Listing 13.24 (Continued) *OrderController.java: the queue, a new order, one order, status steps and payments.*

```
    private String fillChoices(Model model) {
        model.addAttribute("customers", customers.search(""));
        model.addAttribute("services", orders.activeServices());
        return "orders/form";
    }
}
```

Notice how little each method does. `create` asks the service to `check` the new order and adds any problems to the form, so the cashier sees every message at once, as the API client does; only a clean form is created. `advance` calls the service with `null` (“the next step”), puts the outcome in a flash message and redirects; it does not know that payments exist, the service does. When the service says no (“Collect Rp 34.000 before handing over the laundry.”), the refusal becomes a red

line on the order page. `list` reads the status filter from the query string and quietly shows all orders for an unknown word, which is friendlier for a page than an error. `HomeController` (**Listing 13.25**) opens the shop on the queue.

Listing 13.25 `HomeController.java`: the front page is the queue.

```
package com.profitina.laundry.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {

    /** The shop opens on the queue. */
    @GetMapping("/")
    public String home() {
        return "redirect:/orders?status=queued";
    }
}
```

The three order templates follow the patterns of the customer templates: **Listing 13.26** is the queue, **Listing 13.27** the new-order form, **Listing 13.28** one order with its payments.

Listing 13.26 `orders/list.html`: one page of orders, filtered by status.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="~{fragments :: head('Orders')}"></head>
<body>
<header th:replace="~{fragments :: nav}"></header>
<main th:with="R=${T(com.profitina.laundry.domain.LaundryRules)}">
  <h1 th:text="${status} ? 'Orders: ' + ${status.value()} : 'All orders'">
    Orders</h1>
  <div th:replace="~{fragments :: flash}"></div>
  <p>
    <a th:each="s : ${statuses}" th:href="@{/orders(status=${s.value()})}"
       th:text="${s.value()}" th:classappend="${s == status} ? 'current'">
      queued</a>
    <a th:href="@{/orders}">all</a>
  </p>
  <table>
    <thead><tr><th>#</th><th>Received</th><th>Customer</th><th>Service</th>
      <th class="num">Qty</th><th class="num">Total</th><th>Status</th></tr>
    </thead>
    <tbody>
    <tr th:each="o : ${orders.content}">
      <td><a th:href="@{/orders/{id}(id=${o.id})}" th:text="${o.id}">1</a></td>
      <td th:text="${#temporals.format(o.receivedAt, 'dd MMM HH:mm')}">01 Sep</td>
      <td th:text="${o.customer.name}">Sari</td>
      <td th:text="${o.service.name()}">Wash, dry &amp;amp; fold</td>
      <td class="num"
          th:text="${o.quantity} + ' ' + ${o.service.unit()}">4.50 kg</td>
      <td class="num" th:text="${R.rupiah(o.total)}">Rp 36.000</td>
      <td th:text="${o.status.value()}">queued</td>
    </tr>
    <tr th:if="${orders.isEmpty()}"><td colspan="7">No orders here.</td></tr>
    </tbody>
  </table>
  <p th:if="${orders.totalPages > 1}"
     th:text="'Page ' + ${orders.number + 1} + ' of ' + ${orders.totalPages}">
    Page 1 of 2</p>
</main>
</body>
</html>
```

Listing 13.27 `orders/form.html`: taking in a bag of laundry.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="~{fragments :: head('New order')}"></head>
<body>
<header th:replace="~{fragments :: nav}"></header>
<main th:with="R=${T(com.profitina.laundry.domain.LaundryRules)}">
  <h1>New order</h1>
  <form method="post" th:action="@{/orders}" th:object="${form}">
    <label>Customer
      <select th:field="*{customerId}" th:errorclass="invalid">
        <option value="">-- choose --</option>
        <option th:each="c : ${customers}" th:value="${c.id}"
                th:text="${c.name} + ' (' + ${c.phone} + ')'">Sari</option>
      </select>
    </label>
    <p class="bad" th:errors="*{customerId}"></p>
    <label>Service
      <select th:field="*{serviceId}" th:errorclass="invalid">
        <option value="">-- choose --</option>
        <option th:each="s : ${services}" th:value="${s.id}"
                th:text="${s.description} + ', ' + ${R.rupiah(s.price)}">
          Wash, dry &amp;amp; fold (per kg), Rp 8.000</option>
      </select>
    </label>
    <p class="bad" th:errors="*{serviceId}"></p>
    <label>Quantity (kg, items or pairs)
      <input th:field="*{quantity}" th:errorclass="invalid" inputmode="decimal">
    </label>
    <p class="bad" th:errors="*{quantity}"></p>
    <label class="check"><input type="checkbox" th:field="*{pickup}">
      Pickup and delivery (Rp 10.000 below Rp 100.000)</label>
    <label>Delivery address
      <input th:field="*{deliveryAddress}" th:errorclass="invalid">
    </label>
    <p class="bad" th:errors="*{deliveryAddress}"></p>
    <label>Note
      <input th:field="*{note}" placeholder="e.g. separate the white shirts">
    </label>
    <p class="bad" th:errors="*{note}"></p>
    <button>Save order</button>
  </form>
</main>
</body>
</html>
```

Listing 13.28 `orders/detail.html`: an order with its total, payments and balance, and the button for the next step.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="~{fragments :: head('Order')}"></head>
<body>
<header th:replace="~{fragments :: nav}"></header>
<main th:with="R=${T(com.profitina.laundry.domain.LaundryRules)}">
  <h1 th:text="'Order #' + ${order.id}">Order #1</h1>
  <div th:replace="~{fragments :: flash}"></div>
  <dl>
    <dt>Customer</dt><dd th:text="${order.customer.name}">Sari</dd>
    <dt>Service</dt>
    <dd th:text="${order.service.name()} + ', ' + ${order.quantity} + ' '
                + ${order.service.unit()}">Wash, dry &amp;amp; fold, 4.50 kg</dd>
    <dt>Pickup</dt>
    <dd th:text="${order.pickup} ? ${order.deliveryAddress} : 'no'">no</dd>
    <dt>Status</dt><dd th:text="${order.status.value()}">queued</dd>
    <dt>Total</dt><dd th:text="${R.rupiah(order.total)}">Rp 36.000</dd>
    <dt>Paid</dt><dd th:text="${R.rupiah(order.paid())}">Rp 0</dd>
    <dt>Balance</dt><dd th:text="${R.rupiah(order.balance())}">Rp 36.000</dd>
  </dl>
  <form method="post" th:action="@{/orders/{id}/advance(id=${order.id})}"
        th:unless="${order.status.value() == 'departed'}">
    <button th:text="${order.status.value() == 'queued'}
                     ? 'Mark as finished' : 'Hand over to customer'">Next</button>
  </form>
  <h2>Payments</h2>
  <table>
    <tr th:each="p : ${order.payments}">
      <td th:text="${#temporals.format(p.paidAt, 'dd MMM HH:mm')}">01 Sep</td>
      <td th:text="${p.method.value()}">cash</td>
      <td class="num" th:text="${R.rupiah(p.amount)}">Rp 10.000</td>
    </tr>
    <tr th:if="${order.payments.isEmpty()}"><td>No payments yet.</td></tr>
  </table>
  <form method="post" th:if="${order.balance() > 0}" th:object="${payment}"
        th:action="@{/orders/{id}/payments(id=${order.id})}" class="inline">
    <input th:field="*{amount}" th:errorclass="invalid" placeholder="Amount">
    <select th:field="*{method}">
      <option th:each="m : ${methods}" th:value="${m.value()}"
              th:text="${m.value()}">cash</option>
    </select>
    <button>Record payment</button>
    <p class="bad" th:errors="*{amount}"></p>
  </form>
</main>
</body>
```

Listing 13.28 (Continued) *orders/detail.html: an order with its total, payments and balance, and the button for the next step.*

```
</html>
```

A few Thymeleaf features appear here for the first time. `orders` in the list is a Spring Data `Page`, so the template reads `orders.content` for the rows and `orders.totalPages` for the pager line. `#temporals.format(o.receivedAt, 'dd MMM HH:mm')` formats a `LocalDateTime`. `${o.service.name()}` calls a method: `Service` has `name()` and `unit()` rather than getters, because Chapter 11 derived them from the description. `T(com.profitina.laundry.domain.LaundryRules)` refers to a Java class so that the template can call the static `rupiah`; the pages store it once in a local variable `R` with `th:with`. Calling a small, pure formatting function from a template is fine; calling a service or a repository from a template is not, because it would move work out of the layers where it belongs.

The payment form appears only while the balance is positive, and the next-step button disappears once the order has departed. Hiding a button is a courtesy, not a protection: a request can still be sent by hand, which is why the service checks the same conditions again.

13.8 One Place for Errors

What happens when someone opens `/orders/999` and there is no order 999? `OrderService.get` throws `NotFoundException`. Without further instruction Spring would answer 500, “Internal Server Error”, which is wrong (nothing broke; the request asked for something that does not exist) and unfriendly. **Listing 13.29** fixes this for every page at once.

Listing 13.29 `WebErrorAdvice.java`: every `NotFoundException` on a page becomes a 404 page.

```
package com.profitina.laundry.web;

import com.profitina.laundry.service.NotFoundException;
import org.springframework.http.HttpStatus;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;

/** One place that turns "not found" into a friendly 404 page. */
@ControllerAdvice(basePackages = "com.profitina.laundry.web")
public class WebErrorAdvice {

    @ExceptionHandler(NotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String notFound(NotFoundException e, Model model) {
        model.addAttribute("message", e.getMessage());
        return "error/404";
    }
}
```

A class annotated `@ControllerAdvice` contributes to many controllers at once, here those of the `web` package; its `@ExceptionHandler` methods run when a controller method throws the given exception [7]. `@ResponseStatus(HttpStatus.NOT_FOUND)` sets the status, and the method names the template **Listing 13.30**.

Listing 13.30 `error/404.html`: the page for a record that does not exist.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="~{fragments :: head('Not found')}"></head>
<body>
<header th:replace="~{fragments :: nav}"></header>
<main>
  <h1>Not found</h1>
  <p th:text="${message}">That record does not exist.</p>
  <p><a th:href="@{/orders}">Back to the orders</a></p>
</main>
</body>
</html>
```

Pages and API now each have one error advice, limited to their own package: the same `NotFoundException` becomes JSON with status 404 for the phone app and a friendly page with status 404 for the cashier.

13.9 SQLite, Configuration and Running It

Chapter 11 kept the data in H2, a database in memory that forgets everything when the application stops, and promised SQLite for this chapter. **Listing 13.31** is the build file.

Listing 13.31 `pom.xml`: Chapter 12's build plus Thymeleaf and SQLite; H2 stays for the tests.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>4.1.1</version>
    <relativePath/>
  </parent>
  <groupId>com.profitina</groupId>
  <artifactId>laundry</artifactId>
  <version>13.0.0</version>
  <properties>
    <java.version>25</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webmvc</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <!-- SQLite for the shop, as promised in Chapter 11 -->
    <dependency>
      <groupId>org.xerial</groupId>
      <artifactId>sqlite-jdbc</artifactId>
      <version>3.53.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-community-dialects</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
```

Listing 13.31 (Continued) *pom.xml: Chapter 12's build plus Thymeleaf and SQLite; H2 stays for the tests.*

```
    <!-- H2 in memory for the tests only -->
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webmvc-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

The parent `spring-boot-starter-parent` 4.1.1 fixes the versions of the Spring libraries and of Hibernate, Jackson and Thymeleaf, so most dependencies need no version. Spring Boot 4.1 was released on June 30, 2026, and its open-source support runs until July 31, 2027, according to the calendar collected by endoflife.date [8]; Section 15.7 compares that calendar with those of .NET and Laravel. The SQLite driver 3.53.2.0 comes from the xerial project [9], and `SQLiteDialect` from Hibernate's community dialects teaches Hibernate SQLite's flavor of SQL [10]; neither is in Spring Boot's managed list, so both give a version. H2 moves to the `test` scope. **Listing 13.32** is the configuration of the application, and **Listing 13.33** the configuration that the tests use instead.

Listing 13.32 `application.properties`: SQLite in a file, the schema kept up to date, snake_case JSON.

```
spring.application.name=profitina-laundry
server.port=8080

# SQLite: a file next to the project, kept between starts. DemoData fills
# it on the first start only.
spring.datasource.url=jdbc:sqlite:laundry.db?foreign_keys=true
spring.datasource.driver-class-name=org.sqlite.JDBC
spring.datasource.hikari.maximum-pool-size=1
spring.jpa.database-platform=org.hibernate.community.dialect.SQLiteDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.open-in-view=false

# JSON names in snake_case (customer_id), the same as the Laravel API.
spring.jackson.property-naming-strategy=SNAKE_CASE
```

Listing 13.33 `src/test/resources/application.properties`: a fresh H2 database for every test run.

```
# The tests use a fresh H2 database in memory, as in Chapter 12.
spring.datasource.url=jdbc:h2:mem:laundry-test
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.open-in-view=false
spring.jackson.property-naming-strategy=SNAKE_CASE
```

Each setting deserves a line. `foreign_keys=true` switches on SQLite's foreign-key checks, which are off by default. `maximum-pool-size=1` gives the application a single connection: SQLite allows one writer at a time, and one connection avoids "database is locked" errors in a small shop. `ddl-auto=update` lets Hibernate create the tables on the first start and add missing columns later; it never deletes or renames anything, so real projects move to migrations (Flyway or Liquibase in Spring, EF Core migrations in Chapter 5, Laravel migrations in Chapter 9). `open-in-view=false`, kept from Chapter 11, closes the database session when the service call ends, so every page's data must be loaded inside a service, which is exactly where the entity graphs of the repositories load it. Chapter 11's `DemoData` still fills the price list and three customers, but only when the table is empty, so restarting the application keeps the shop's data.

To run the application on Windows, macOS or Linux, install JDK 25 and Maven as described in Chapter 1, open `Chapter13/spring` in a terminal (or in VS Code with the Java and Spring Boot extension packs), and type

```
mvn spring-boot:run
```

After Maven has downloaded the libraries (only the first time) and compiled the project, Spring Boot starts Tomcat on port 8080. Open `http://localhost:8080`; you land on the queue, which already holds the demo orders of Chapter 11. A good first walk through the shop:

1. Open **Customers**, search for "sari", then add a new customer with the phone "0813-1111-2222". The list shows the phone as `081311112222`.
2. Try adding a second customer with "0813 1111 2222". The form comes back with "The phone has already been taken." under the phone box.
3. Open **New order**, choose the new customer and "Wash, dry & fold", type `2.5`, tick pickup, leave the address empty and save. The form comes back with "Where should the laundry be delivered?". Type "Jl. Kenanga 5" and save: the order page says Rp 34.000, the minimum and the pickup fee at work (Equation (13.1)).
4. Press **Mark as finished**, then **Hand over to customer**. The page says "Collect Rp 34.000 before handing over the laundry."
5. Record Rp 40.000 in cash: "Only Rp 34.000 is still owed." Record Rp 20.000 in cash and Rp 14.000 by QRIS, then hand the order over. The balance is Rp 0 (Equation (13.2)) and the status `departed`.

Chapter 11's rule check still runs with the JDK alone, from the `spring` folder:

```
javac -d out -sourcepath src/main/java check/RulesCheck.java
java -cp out RulesCheck
```

It printed this on the machine where the book was written:

```
Prices (Problem 8.1)
  Wash, dry & fold             4.50 kg            36000
  Wash, dry & fold             2.00 kg            24000
  Wash, dry & fold             4.02 kg            32160
  Ironing only                 3.35 kg            20100
  Curtain                      4.40 kg   pickup   76000
  Bed cover, any size          2.00 item pickup   80000
  Dry cleaning - suit 2 pieces 2.00 item pickup  130000
  Dry cleaning - shirt/blouse  5.00 item pickup  100000
  Shoes deep clean             1.50 pair          90000
Workflow (Problem 8.2)
  Queued   owing  25000 -> Finished
  Finished owing  25000 -> Collect Rp 25.000 before handing over the laundry.
  Finished owing      0 -> Departed
  Departed owing      0 -> This order has already left the shop.
```

That machine could not download Spring's libraries, so the application itself was not started there. Every Java file of the project was compiled against small stand-ins for the Spring, Jakarta and JUnit types it uses, with no errors, and the rules above were compiled and run for real. The chapter's README says exactly this and asks you to run `mvn test` first, as the real check.

13.10 Testing the Service Layer

The rules have been tested since Chapter 11 (`LaundryRulesTest`), and the API since Chapter 12 (`OrderApiTest`, now in the `api` package and unchanged otherwise). The service layer is new, and it can be tested without Spring, without a web server and without a database. **Listing 13.34** does it with *mocks*: stand-ins for the repositories that answer what the test tells them to.

Listing 13.34 `OrderServiceTest.java`: the service layer with mocked repositories and a fixed clock.

```
package com.profitina.laundry.service;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.domain.PaymentMethod;
import com.profitina.laundry.domain.Service;
import com.profitina.laundry.repository.CustomerRepository;
import com.profitina.laundry.repository.OrderRepository;
import com.profitina.laundry.repository.PaymentRepository;
import com.profitina.laundry.repository.ServiceRepository;
import java.math.BigDecimal;
import java.time.Clock;
import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.Optional;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

/** The service layer on its own: no Spring, no database, a fixed clock. */
class OrderServiceTest {

    private final OrderRepository orders = mock(OrderRepository.class);
    private final CustomerRepository customers =
            mock(CustomerRepository.class);
    private final ServiceRepository services = mock(ServiceRepository.class);
    private final Clock clock = Clock.fixed(
            Instant.parse("2026-09-25T02:00:00Z"), ZoneId.of("Asia/Jakarta"));
    private OrderService service;
    private Order order;
```

Listing 13.34 (Continued) *OrderServiceTest.java: the service layer with mocked repositories and a fixed clock.*

```
    @BeforeEach
    void setUp() {
        service = new OrderService(orders, customers, services,
                mock(PaymentRepository.class), clock);
        Customer sari = new Customer("Sari Wulandari", "081200000101", null,
                LocalDateTime.now(clock));
        Service wash = new Service("S0001", "Wash, dry & fold (per kg)",
                8000);
        order = new Order(sari, wash, new BigDecimal("4.5"), false, null,
                null, LocalDateTime.now(clock));
        when(orders.findWithDetailsById(7L)).thenReturn(Optional.of(order));
    }

    @Test
    void finishingStampsTheShopTime() {
        service.advance(7L, null);
        assertEquals(OrderStatus.FINISHED, order.getStatus());
        assertEquals(LocalDateTime.of(2026, 9, 25, 9, 0),
                order.getFinishedAt());
    }

    @Test
    void unpaidLaundryCannotLeave() {
        service.advance(7L, null);                      // queued -> finished
        RuleException e = assertThrows(RuleException.class,
                () -> service.advance(7L, OrderStatus.DEPARTED));
        assertEquals("Collect Rp 36.000 before handing over the laundry.",
                e.getErrors().get("status").get(0));
        assertEquals(OrderStatus.FINISHED, order.getStatus());
    }

    @Test
    void aStepCannotBeSkipped() {
        RuleException e = assertThrows(RuleException.class,
                () -> service.advance(7L, OrderStatus.DEPARTED));
        assertEquals("An order that is queued can only become finished.",
                e.getErrors().get("status").get(0));
    }
```

Listing 13.34 (Continued) *OrderServiceTest.java: the service layer with mocked repositories and a fixed clock.*

```
    @Test
    void overpaymentIsRefused() {
        RuleException e = assertThrows(RuleException.class,
                () -> service.pay(7L, 40_000, PaymentMethod.CASH));
        assertEquals("Only Rp 36.000 is still owed.",
                e.getErrors().get("amount").get(0));
    }

    @Test
    void paidLaundryLeavesAndIsThenClosed() {
        service.advance(7L, null);
        service.pay(7L, 36_000, PaymentMethod.QRIS);
        service.advance(7L, null);
        assertEquals(OrderStatus.DEPARTED, order.getStatus());
        assertThrows(ConflictException.class,
                () -> service.pay(7L, 1_000, PaymentMethod.CASH));
    }

    @Test
    void pickupNeedsAnAddress() {
        when(customers.existsById(1L)).thenReturn(true);
        var errors = service.check(new OrderService.NewOrder(1L, null,
                BigDecimal.ONE, true, " ", null));
        assertTrue(errors.containsKey("delivery_address"));
        assertEquals(1, errors.size());
    }
}
```

`mock(OrderRepository.class)` creates an object that implements the interface and returns empty answers; `when(orders.findWithDetailsById(7L)).thenReturn(Optional.of(order))` teaches it one answer. The clock is fixed at 02:00 UTC, which is 09:00 in Surabaya, so `finishingStampsTheShopTime` can check the exact time. `unpaidLaundryCannotLeave` is the heart of the chapter as a test: with nothing paid on a Rp 36.000 order, the step to `departed` must be refused with the exact message and the status must stay `finished`. Because the service receives its dependencies through its constructor, the test simply calls `new OrderService(...)`: no Spring, no annotations, and the six tests run in a fraction of a second.

Run all the tests with

```
mvn test
```

Maven prints how many tests ran, failed and had errors, and for each failure the test's name with the expected and actual values. Chapter 14 adds tests of the web layer on its own, of the queries on their own, and of security.

13.11 Making It Better

Edit the price list from a page. Chapter 10 gave Laravel's owner a price-list page protected by a gate; the Spring pages still read the list that `DemoData` loaded. A `PriceController` with a form record is a CRUD exactly like the customers, and Chapter 14 adds the login that restricts it to the owner.

Page through the customers. The customer list shows everyone. With a few hundred regulars that is fine; with thousands it is not. The repository can return a `Page` as the orders do.

Lock the order while paying. One transaction per payment prevents most double payments. Two cashiers pressing "Record payment" in the same instant could still both read the old balance

before either writes. SQLite lets one writer at a time and our pool has one connection, so this application cannot hit the problem; on MySQL or PostgreSQL the service would lock the order row (`@Lock(LockModeType.PESSIMISTIC_WRITE)` on the repository method) or use a version column for optimistic locking.

One template layout. The pages repeat their `<head>` and navigation through fragments. The Thymeleaf Layout Dialect, or Thymeleaf's own fragment expressions with a layout template, would give one layout in which each page fills a slot, like Blade's `@extends` or Razor's `_Layout.cshtml`.

13.12 Using It for Real

Nothing in this chapter is specific to laundry except the four entities and `LaundryRules`. To adapt the application to another small business, work from the rules outward. A tailor has customers, a price list of services (take in a waist, shorten trousers, sew a kebaya), orders with a promised date, and payments, usually a deposit on the day and the rest on collection: add a due date to the order, drop the kilogram rule, and write "a deposit of at least 30% before work starts" as a rule with its own test. A motorbike workshop has the same shape with a plate number next to the customer and spare parts as a second kind of line. A small clinic's appointment book is also a queue with states (booked, arrived, seen), but there the rules concern people's health, and the application must only record and remind, never decide: whether someone should see a doctor today is for the doctor or the midwife to say, not the program.

The order of work is always the same: rules in one plain class, tested; then entities; then repositories; then one service method for each thing the business does; then the controllers and pages, which should end up the thinnest part of the code.

13.13 Profitina in Practice

Profitina, the AI-visibility platform whose name this book's running example borrows, is written in Python with FastAPI rather than in Spring Boot, and it keeps the same separation. Request handlers turn HTTP into calls; a layer of services carries out operations such as scanning a website, computing a readiness score or renewing a subscription; the database is reached through one layer only. The rules that matter most to its users, such as how a subscription renews from a balance and which daily caps apply when earning PROFIT coins, live in plain functions with their own tests, and the full automated suite, well over a thousand tests, must pass before any release. The lesson carries across languages: when "laundry leaves only when paid" or "a subscription renews only with enough balance" lives in one place and is tested there, every page and every API that uses it is right at once.

13.14 Common Mistakes, and How to Avoid Them

- **Rules in controllers.** Symptom: the same `if` appears in a page controller and a JSON controller. Move it into a service method and call it from both.
- **Transactions on controllers.** `@Transactional` belongs on the service methods that do the work; a controller that needs a transaction is doing a service's job.
- **Two beans with the same name.** `web.OrderController` and `api.OrderController` both become the bean `orderController`, and the application does not start. Give them different class names.

- **An error advice for everything.** A `@RestControllerAdvice` without `basePackages` also catches exceptions from pages and answers them with JSON. Limit each advice to its package.
- **Binding forms to entities.** A form can then set fields it should not, such as `id` or `createdAt`. Use a form record with only the fields the form may change.
- **`BindingResult` in the wrong place.** It must follow the `@Validated` parameter immediately; otherwise the user sees an error page instead of the form with messages.
- **Showing a page as the answer to a successful `POST`.** Refreshing then submits again. Redirect after success, and use flash attributes for the message.
- **`LazyInitializationException` in a template.** With `open-in-view=false`, an association not loaded in the service cannot be loaded in the template. Load it in the repository with an entity graph.
- **`@Transactional` on a private method, or on a method called from inside the same class.** Spring applies transactions through a proxy around the bean, so only public methods called from another bean are wrapped.
- **Trusting hidden buttons.** A button hidden by the template can still be "pressed" with a hand-made request. The service must check every rule itself.

13.15 Chapter Summary

- A layered architecture separates controllers (HTTP), services (business operations, rules and transactions) and repositories (persistence), with calls going downwards only (**Figure 13.1**, **Table 13.1**).
- Chapter 12's JSON controllers moved to the package `api` and became thin; new page controllers live in `web`; both call the same services (**Table 13.2**).
- The services throw three exceptions: `RuleException` (422 or messages on form fields), `NotFoundException` (404) and `ConflictException` (409).
- `OrderService.check` returns the database rules as a map, so that both the API and the pages can report every problem at once; `create` checks again.
- The money rules of Chapter 11 are unchanged: the total by Equation (13.1), the balance by Equation (13.2).
- Form records with Jakarta Bean Validation constraints protect against bad input and mass assignment; `BindingResult` lets a controller show the form again with the constraint messages and the service's messages (**Table 13.3**).
- Thymeleaf templates reuse fragments, escape text by default and bind forms with `th:object` and `th:field`; a successful post is followed by a redirect (Post/Redirect/Get, **Figure 13.2**, **Table 13.4**).
- Each layer has one error advice: JSON for `api`, a 404 page for `web`.
- The data now live in SQLite; the tests use a fresh H2 database; the service layer is tested with Mockito and a fixed clock.

13.16 Exercises

1. By hand, using Equation (13.1), compute the totals of: (a) 1.2 kg of ironing only at Rp 6.000 per kg, no pickup; (b) 7.5 kg of express wash and fold at Rp 12.000, with pickup; (c) 2.5 pairs of shoes at Rp 45.000 per pair; (d) 4.02 kg of wash, dry and fold at Rp 8.000 with pickup.

2. For each piece of code, name the layer it belongs in and say why: (a) “if the customer has an address, suggest it as the delivery address”; (b) “show at most 30 orders per page”; (c) “the sum of all QRIS payments today”; (d) “turn the word `finished` in the URL into `OrderStatus.FINISHED`”.
3. Add a rule: express services (the description begins with “Express”) are accepted for at most 10 kg per order. Put the check where it belongs and write a test for it in `OrderServiceTest`.
4. A cashier searches for “0812 3456” and finds nobody, although customer `081234560001` exists. Explain why `CustomerService.search` does find her, and what would happen with the search text “08123 x”.
5. Explain what would go wrong if `OrderService.pay` were *not* transactional and two cashiers recorded a payment for the same order at the same time, and why this application is safe anyway.
6. Write the form record and the list of files for a price-list page (`/prices`) on which the owner changes a service’s price and switches it off.
7. Write a test in `OrderServiceTest` that checks that a new order for a withdrawn service is refused with the message “The selected service is invalid.”.

13.17 Solutions to the Exercises

1. (a) $h = 120$, per kg, so $b = \max(120,300) = 300$; $S = \lfloor(6000 \cdot 300 + 50)/100\rfloor = 18000$; no pickup, $T = 18000$. (b) $h = 750$, $b = 750$, $S = \lfloor(12000 \cdot 750 + 50)/100\rfloor = 90000$; with pickup and $S < 100000$, $T = 100000$. (c) Pairs are whole units: $h = 250$ is rounded up to $b = 300$, $S = 45000 \cdot 3 = 135000$, $T = 135000$. (d) $h = 402$, $b = 402$, $S = \lfloor(8000 \cdot 402 + 50)/100\rfloor = \lfloor 3216050/100\rfloor = 32160$; with pickup, $T = 42160$.

2. (a) Service (or the page controller, as a convenience for the form): suggesting an address is presentation, but if it became a rule (“deliver to the customer’s address unless another is given”) it would belong in `OrderService.create`. (b) Controller: page size is a choice of the screen; the service takes the page and the size as parameters. (c) Repository: a query with `sum` and a `where` on method and date, called by a service method that builds the report. (d) Controller: it is about the text of a URL.

3. The rule needs the service’s description, so it belongs in `OrderService.check`, after the service lookup:

```
        services.findById(in.serviceId() == null ? "" : in.serviceId())
            .filter(s -> s.getDescription().startsWith("Express"))
            .filter(s -> in.quantity() != null
                    && in.quantity().compareTo(new BigDecimal("10")) > 0)
            .ifPresent(s -> add(errors, "quantity",
                    "Express service takes at most 10 kg per order."));
```

A test, with the mock taught to return an express service:

```
    @Test
    void expressIsLimitedToTenKilos() {
        when(customers.existsById(1L)).thenReturn(true);
        when(services.findById("S0002")).thenReturn(Optional.of(new Service(
                "S0002", "Express wash & fold, 1 day (per kg)", 12000)));
        var errors = service.check(new OrderService.NewOrder(1L, "S0002",
                new BigDecimal("12"), false, null, null));
        assertEquals("Express service takes at most 10 kg per order.",
                errors.get("quantity").get(0));
    }
```

4. "0812 3456" matches `[0-9 +-]+`, so `search` strips it to the digits "08123456" and calls `findByPhoneContainingOrderByName`, which finds `081234560001`. "08123 x" contains a letter, so it is treated as a name and searched with `findByNameContainingIgnoreCaseOrderByName`, which finds nobody unless a name contains "08123 x".

5. Without a transaction, cashier A reads the balance (Rp 34.000), cashier B reads it a moment later (still Rp 34.000), both see that their Rp 34.000 is not more than the balance, and both insert a payment: the order is paid twice. With `@Transactional` on a database that isolates transactions and a lock on the order row (Section 13.11), B waits for A and then sees a balance of 0. Here, SQLite allows one writer at a time and the pool has one connection, so the two `pay` calls run one after the other anyway.

6. The record:

```
public record PriceForm(
        @NotBlank @Size(max = 60) String description,
        @NotNull @Min(value = 500, message = "At least Rp 500.") Long price,
        boolean active) {
}
```

Files: `Service` gains a method `change(String description, long price, boolean active)`; a `PriceService` (list all, get, update, with the fully written `@org.springframework.stereotype.Service`); a `PriceController` for `GET /prices`, `GET /prices/{id}/edit` and `POST /prices/{id}`; the templates `prices/list.html` and `prices/form.html`; a link in `fragments.html`. There is no delete: a service that was ever ordered must stay, so the form offers "active" instead. Because each order stored its own total, changing a price never changes an old receipt.

7. In `OrderServiceTest`:

```
    @Test
    void aWithdrawnServiceIsRefused() {
        Service old = new Service("S0009", "Blanket (per item)", 25000);
        old.withdraw();                // an assumed method: active = false
        when(customers.existsById(1L)).thenReturn(true);
        when(services.findById("S0009")).thenReturn(Optional.of(old));
        var errors = service.check(new OrderService.NewOrder(1L, "S0009",
                BigDecimal.ONE, false, null, null));
        assertEquals("The selected service is invalid.",
                errors.get("service_id").get(0));
    }
```

Chapter 11's `Service` has no method that switches a service off, so the test assumes the `withdraw` method that the price page of Exercise 6 needs anyway (it sets `active` to `false`).

References

[1] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002, ch. 9, "Service Layer."

[2] Broadcom, "Spring Boot Reference Documentation," version 4.1. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-boot/

[3] Broadcom, "Spring Data JPA Reference Documentation: Query Methods and Entity Graphs." Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-data/jpa/reference/

[4] Eclipse Foundation, *Jakarta Validation 3.1 Specification*, 2024. Accessed: Sep. 25, 2026. [Online]. Available: https://jakarta.ee/specifications/bean-validation/3.1/

[5] R. Fielding, M. Nottingham, and J. Reschke, “HTTP Semantics,” RFC 9110, IETF, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[6] The Thymeleaf Team, “Tutorial: Thymeleaf + Spring,” Thymeleaf 3.1. Accessed: Sep. 25, 2026. [Online]. Available: https://www.thymeleaf.org/documentation.html

[7] Broadcom, “Spring Framework Reference: Web on Servlet Stack (Spring MVC), Validation and Exceptions,” version 7.0. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc.html

[8] endoflife.date, “Spring Boot,” release and support dates. Accessed: Sep. 25, 2026. [Online]. Available: https://endoflife.date/spring-boot

[9] Xerial, “SQLite JDBC Driver,” release 3.53.2.0. Accessed: Sep. 25, 2026. [Online]. Available: https://github.com/xerial/sqlite-jdbc

[10] Hibernate, “Hibernate ORM Community Dialects: SQLiteDialect.” Accessed: Sep. 25, 2026. [Online]. Available: https://github.com/hibernate/hibernate-orm/tree/main/hibernate-community-dialects

Chapter 14

Spring Boot III: Spring Security, REST API Design and Automated Testing

Chapter 13 gave the Spring Boot laundry application pages for the counter and a service layer beneath both the pages and the API. Everything in it is still open to anyone who can reach the server: a passer-by on the shop's Wi-Fi can open the queue, and any program can take orders through the API. This chapter closes the doors and then checks, automatically, that they stay closed. By the end of it you will be able to store passwords safely and explain why a hash is not enough without a salt and a cost; protect the JSON API with bearer tokens in the style of Laravel Sanctum and the pages with a login form and a session; configure two Spring Security filter chains and follow a request through them; restrict an operation to the owner both at the URL and in the service; defend the decisions behind the REST API of Chapter 12 (URLs, methods, status codes, envelopes) and name what it still lacks; and write tests at four levels, from plain unit tests to tests of the whole application, knowing what each level can and cannot find.

14.1 Where We Left Off

Chapter 12 made the Spring Boot API speak the same JSON as the Laravel API of Chapters 9 and 10, and Chapter 13 put a service layer beneath it and pages on top. One part of the Laravel API was deliberately left out: its doors. In Chapter 10, Laravel's API answers every request without a valid token with `401 {"message": "Unauthenticated."}`, the phone app of the drivers signs in with `POST /api/tokens` and receives a token, and only the owner may do certain things. Raka, Dimas's nephew who writes the drivers' phone app, noticed the difference at once: against the Spring server, his app worked without signing in at all. That is convenient for a demonstration and unacceptable for a shop that keeps its customers' phone numbers and addresses.

Bu Wati has two wishes. The cashiers should log in at the counter with their own accounts, so that the shop can later see who took which order; and deleting a customer, a rare operation that removes a person's data, should be for her alone. Everything else a cashier does today, a cashier may keep doing.

The code of this chapter is in `Chapter14/spring`. It starts as a copy of Chapter 13 and adds a `security` package, two entities (`Staff` and `ApiToken`), a token controller, a login page, and tests on four levels.

14.2 The Idea in Plain Words

Security in a web application answers two questions for every request, in this order: *who is asking?* and *may they do this?* The first is **authentication**: the request proves an identity, with a password, a cookie that the server issued after a password, or a token. The second is **access control**: given the identity, the application decides whether the operation is allowed, here by the staff member's *role*. The two answers produce the two classic refusals of HTTP: 401 when the first question has no answer ("I do not know who you are"), 403 when the second answer is no ("I know who you are, and you may not") [1].

Two kinds of client reach the laundry, and they prove their identity differently.

A **browser** at the counter gets a login form. After a correct e-mail and password, the server creates a *session*, keeps the fact "this session belongs to Kasir 1" in its memory, and gives the browser a cookie with the session's random identifier. The browser sends the cookie with every following request automatically. Pages are convenient this way, and the convenience brings a danger called cross-site request forgery, discussed in Section 14.5.4.

A **program**, such as Raka's phone app or the accounting firm's integration, has no use for forms and cookies. It sends its e-mail and password once to `POST /api/tokens` and receives a long random *token*, which it then puts in a header of every request: `Authorization: Bearer 12|5Rk...`. The server keeps no session: each request carries its own proof. The app stores the token instead of the password, and signing out one phone deletes one token without affecting any other device.

Spring Security implements both with a *filter chain*: a list of filters that every request passes through before it reaches a controller [2]. One filter reads the session cookie, another reads the login form, our own filter reads the bearer token; a later filter compares the identity they found with the rules of the configuration and either lets the request through or answers 401 or 403. An application may have several chains, each for a part of its URLs, and this is what we need: one chain for `/api/**` with tokens and no sessions, one for everything else with a login form and sessions.

> **DEFINITION.** A **security filter chain** is an ordered list of servlet filters that Spring Security runs for every request matching the chain. The first chain whose *matcher* accepts the request handles it; the filters in that chain establish who the caller is and decide whether the request may reach the application.

14.3 Accounts and Passwords

14.3.1 Never store a password

The first rule of accounts is that the database must not contain the passwords. Databases leak: through backups copied to a laptop, a mistake in a query, or an attack. Many people reuse passwords, so a leaked list of passwords from a laundry opens their e-mail and their bank. What the application stores instead is a *password hash*: the output of a one-way function that is easy to compute from the password and practically impossible to reverse. At login the application hashes what was typed and compares the result with the stored hash.

A plain cryptographic hash such as SHA-256 is not enough, for two reasons. Two customers with the same password would have the same hash, and an attacker can prepare tables of the hashes of millions of common passwords in advance. The answer to both is a *salt*, a random value stored next to the hash and mixed into it, so that equal passwords give different hashes and no table can be prepared in advance. The second reason is speed: a modern graphics card computes billions of SHA-256 hashes per second, so an attacker who has the hash and the salt can still try every short password. Password hashing functions such as bcrypt, scrypt, Argon2 and PBKDF2 are therefore deliberately *slow*, with a configurable cost. OWASP's guidance lists them and recommends suitable work factors [3].

bcrypt's cost is a power of two. With cost c, one check performs about

$$N = 2^c \tag{14.1}$$

rounds of its expensive key setup, so each step of c doubles the work for the server and for any attacker alike. Spring's `BCryptPasswordEncoder` uses $c = 10$ by default [2]; Laravel's default is 12 [4]; ASP.NET Core Identity uses PBKDF2 with HMAC-SHA512 and 100,000 iterations [5]. A cost that makes one login take a few tens of milliseconds is invisible to a cashier and makes guessing expensive.

Spring Security wraps this in the `PasswordEncoder` interface. The configuration of Section 14.5.1 asks `PasswordEncoderFactories.createDelegatingPasswordEncoder()` for one. It produces stored values such as `{bcrypt}$2a$10$N9qo8uLOickgx2ZMRZoMye...`: the part in braces names the algorithm, `$2a$` is the bcrypt version, `10` the cost, then 22 characters of salt and 31 of hash. Because the algorithm is named in the stored value, a later version of the application can change the default to Argon2 and still check every old hash, upgrading each one the next time its owner logs in.

14.3.2 Staff and roles

Listing 14.1 and **Listing 14.2** are the accounts.

Listing 14.1 `Role.java`: two roles are all the shop needs.

```
package com.profitina.laundry.domain;

/** The owner runs the shop; a cashier takes orders and money. */
public enum Role {
    OWNER, CASHIER
}
```

Listing 14.2 `Staff.java`: a person who may log in, with a password hash and a role.

```
package com.profitina.laundry.domain;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

/** Someone who may log in. The password itself is never stored. */
@Entity
@Table(name = "staff")
public class Staff {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true, length = 120)
    private String email;

    @Column(nullable = false, length = 80)
    private String name;

    @Column(nullable = false, length = 100)
    private String passwordHash;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false, length = 10)
    private Role role;

    protected Staff() {
    }

    public Staff(String email, String name, String passwordHash, Role role) {
        this.email = email;
        this.name = name;
        this.passwordHash = passwordHash;
        this.role = role;
    }
```

Listing 14.2 (Continued) *Staff.java: a person who may log in, with a password hash and a role.*

```
    public Long getId() { return id; }
    public String getEmail() { return email; }
    public String getName() { return name; }
    public String getPasswordHash() { return passwordHash; }
    public Role getRole() { return role; }
}
```

`Staff` is a separate entity from `Customer`: customers never log in, and mixing the two would give every customer row a password column and a role that means nothing. The field is called `passwordHash`, never `password`, so that nobody is tempted to store the plain text in it. The role is stored as text (`@Enumerated(EnumType.STRING)`), so the column says `OWNER` rather than `0`, and adding a role later does not change the meaning of existing rows. **Listing 14.3** finds a staff member by e-mail, ignoring case, because people type `Owner@...` as often as `owner@...`.

Listing 14.3 `StaffRepository.java`: find a staff member by e-mail.

```
package com.profitina.laundry.repository;

import com.profitina.laundry.domain.Staff;
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;

public interface StaffRepository extends JpaRepository<Staff, Long> {

    Optional<Staff> findByEmailIgnoreCase(String email);
}
```

14.3.3 Telling Spring Security about our staff

The login form of the pages is handled entirely by Spring Security; it only needs to know how to find an account. That is the job of a `UserDetailsService` (**Listing 14.4**).

Listing 14.4 `StaffDetailsService.java`: from an e-mail to Spring Security's view of a user.

```
package com.profitina.laundry.security;

import com.profitina.laundry.domain.Staff;
import com.profitina.laundry.repository.StaffRepository;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Component;

/** Tells the login form how to find a staff member by e-mail. */
@Component
public class StaffDetailsService implements UserDetailsService {

    private final StaffRepository staff;

    public StaffDetailsService(StaffRepository staff) {
        this.staff = staff;
    }

    @Override
    public UserDetails loadUserByUsername(String email) {
        Staff s = staff.findByEmailIgnoreCase(email)
                .orElseThrow(() -> new UsernameNotFoundException(email));
        return User.withUsername(s.getEmail())
                .password(s.getPasswordHash())
                .roles(s.getRole().name())       // becomes ROLE_OWNER etc.
                .build();
    }
}
```

`User.withUsername(...).roles("OWNER")` creates a user who holds the permission `ROLE_OWNER`, in Spring's terms a `GrantedAuthority`. Spring Security's role checks such as `hasRole("OWNER")` add the prefix `ROLE_` themselves, so roles are written without it everywhere else. When the form is submitted, Spring Security calls `loadUserByUsername`, asks the password encoder whether the typed password matches the stored hash, and, if it does, starts a session. When no staff member has the e-mail, the method throws `UsernameNotFoundException`; Spring Security turns both "no such e-mail" and "wrong password" into the same message, so the form does not tell an attacker which e-mails exist.

Two demo accounts are created on the first start by **Listing 14.5**.

Listing 14.5 `DemoStaffSeeder.java`: the owner and one cashier, with hashed passwords.

```
package com.profitina.laundry.security;

import com.profitina.laundry.domain.Role;
import com.profitina.laundry.domain.Staff;
import com.profitina.laundry.repository.StaffRepository;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Component;

/** The two demo accounts, created once. Change them for real use. */
@Component
public class DemoStaffSeeder implements ApplicationRunner {

    private final StaffRepository staff;
    private final PasswordEncoder encoder;

    public DemoStaffSeeder(StaffRepository staff, PasswordEncoder encoder) {
        this.staff = staff;
        this.encoder = encoder;
    }

    @Override
    public void run(ApplicationArguments args) {
        if (staff.count() > 0) {
            return;
        }
        staff.save(new Staff("owner@profitina-laundry.example", "Owner",
                encoder.encode("Owner#2026"), Role.OWNER));
        staff.save(new Staff("kasir1@profitina-laundry.example", "Kasir 1",
                encoder.encode("Kasir#2026"), Role.CASHIER));
    }
}
```

The passwords are hashed on the way in by the same encoder that checks them. The addresses end in `.example`, a domain reserved for documentation, so no real mailbox is involved. The demo passwords are printed in this book and in the README: the first thing a real shop does is change them.

14.4 Tokens for the API

14.4.1 The idea of Sanctum's tokens

Chapter 10 used Laravel Sanctum's API tokens, and the Spring version copies their design so that Raka's app does not notice which server it talks to. A token has two parts separated by a bar, `12|5RkX...`: the number of a row in the table `api_tokens`, and a random secret of 40 characters. The table stores the row's owner, the name of the device, the time it was created and last used, and the SHA-256 hash of the secret, never the secret itself [4]. When a request arrives with a token, the server finds the row by its number, hashes the secret it was given and compares the two hashes.

Why is a fast hash acceptable here when Section 14.3.1 insisted on a slow one for passwords? Because the secret is not chosen by a person. Thirty random bytes carry

$$H = 30 \times 8 = 240 \text{ bits} \tag{14.2}$$

of randomness. Guessing it means trying on the order of 2^{239} values, which no amount of speed makes feasible, so there is nothing for a slow hash to protect against. The hash only ensures that a copy of the database does not contain working tokens.

14.4.2 The token table and the token service

Listing 14.6 is one row of the table, and **Listing 14.7** its repository.

Listing 14.6 `ApiToken.java`: one device's key to the API.

```
package com.profitina.laundry.domain;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import java.time.LocalDateTime;

/** One device's key to the API, as Laravel Sanctum keeps them. */
@Entity
@Table(name = "api_tokens")
public class ApiToken {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.EAGER, optional = false)
    @JoinColumn(name = "staff_id")
    private Staff staff;

    @Column(nullable = false, length = 60)
    private String device;

    @Column(nullable = false, length = 64)
    private String tokenHash;              // SHA-256 of the secret part

    @Column(nullable = false)
    private LocalDateTime createdAt;

    private LocalDateTime lastUsedAt;

    protected ApiToken() {
    }
```

Listing 14.6 (Continued) *ApiToken.java: one device's key to the API.*

```
    public ApiToken(Staff staff, String device, String tokenHash,
                    LocalDateTime createdAt) {
        this.staff = staff;
        this.device = device;
        this.tokenHash = tokenHash;
        this.createdAt = createdAt;
    }

    public void used(LocalDateTime when) {
        lastUsedAt = when;
    }

    public Long getId() { return id; }
    public Staff getStaff() { return staff; }
    public String getDevice() { return device; }
    public String getTokenHash() { return tokenHash; }
    public LocalDateTime getCreatedAt() { return createdAt; }
    public LocalDateTime getLastUsedAt() { return lastUsedAt; }
}
```

Listing 14.7 `ApiTokenRepository.java`: the standard methods are enough.

```
package com.profitina.laundry.repository;

import com.profitina.laundry.domain.ApiToken;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ApiTokenRepository extends JpaRepository<ApiToken, Long> {
}
```

The staff member is loaded eagerly (`FetchType.EAGER`) because every use of a token needs to know whose it is and which role they have. **Listing 14.8** does the work.

Listing 14.8 `TokenService.java`: issue, find and revoke tokens, keeping only hashes.

```java
package com.profitina.laundry.security;

import com.profitina.laundry.domain.ApiToken;
import com.profitina.laundry.domain.Staff;
import com.profitina.laundry.repository.ApiTokenRepository;
import com.profitina.laundry.repository.StaffRepository;
import com.profitina.laundry.service.RuleException;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.time.Clock;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.Base64;
import java.util.HexFormat;
import java.util.Optional;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * API tokens in the style of Laravel Sanctum: "id|secret" is shown once,
 * the table keeps only the SHA-256 of the secret.
 */
@Service
@Transactional
public class TokenService {

    /** A new token and the staff member it belongs to. */
    public record Issued(String token, Staff staff) {
    }

    private static final SecureRandom RANDOM = new SecureRandom();

    private final StaffRepository staff;
    private final ApiTokenRepository tokens;
    private final PasswordEncoder encoder;
    private final Clock clock;
```

Listing 14.8 (Continued) *TokenService.java: issue, find and revoke tokens, keeping only hashes.*

```
    public TokenService(StaffRepository staff, ApiTokenRepository tokens,
                        PasswordEncoder encoder, Clock clock) {
        this.staff = staff;
        this.tokens = tokens;
        this.encoder = encoder;
        this.clock = clock;
    }

    /** E-mail and password in, a token out; the same answer for any miss. */
    public Issued issue(String email, String password, String device) {
        Staff who = staff.findByEmailIgnoreCase(email)
                .filter(s -> encoder.matches(password, s.getPasswordHash()))
                .orElseThrow(() -> new RuleException("email",
                        "These credentials do not match our records."));
        byte[] raw = new byte[30];
        RANDOM.nextBytes(raw);
        String secret = Base64.getUrlEncoder().withoutPadding()
                .encodeToString(raw);                  // 40 characters
        ApiToken saved = tokens.save(new ApiToken(who, device,
                sha256(secret), now()));
        return new Issued(saved.getId() + "|" + secret, who);
    }

    /** The token behind "id|secret", if it exists and the secret fits. */
    public Optional<ApiToken> find(String plain) {
        int bar = plain.indexOf('|');
        if (bar < 1) {
            return Optional.empty();
        }
        long id;
        try {
            id = Long.parseLong(plain.substring(0, bar));
        } catch (NumberFormatException e) {
            return Optional.empty();
        }
        byte[] given = sha256(plain.substring(bar + 1))
                .getBytes(StandardCharsets.US_ASCII);
        Optional<ApiToken> token = tokens.findById(id).filter(t ->
                MessageDigest.isEqual(given,           // constant-time compare
                        t.getTokenHash().getBytes(StandardCharsets.US_ASCII)));
        token.ifPresent(t -> t.used(now()));
        return token;
    }
```

Listing 14.8 (Continued) *TokenService.java: issue, find and revoke tokens, keeping only hashes.*

```
    public void revoke(Long tokenId) {
        tokens.deleteById(tokenId);
    }

    static String sha256(String text) {
        try {
            return HexFormat.of().formatHex(MessageDigest.getInstance("SHA-256")
                    .digest(text.getBytes(StandardCharsets.UTF_8)));
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException(e);      // every JDK has it
        }
    }

    private LocalDateTime now() {
        return LocalDateTime.now(clock).truncatedTo(ChronoUnit.SECONDS);
    }
}
```

Three details in `TokenService` are security decisions, not style.

- `issue` gives the same answer, "These credentials do not match our records.", whether the e-mail is unknown or the password is wrong. This is the sentence Laravel uses, and it tells an attacker nothing about which e-mails belong to staff. The answer is a `RuleException` on the field `email`, so the API's error handler of Chapter 13 turns it into the usual 422 with `errors`.
- The random bytes come from `SecureRandom`, Java's cryptographically strong generator. `java.util.Random` produces numbers that can be predicted from a few earlier outputs and must never be used for secrets.
- `find` compares the hashes with `MessageDigest.isEqual`, which takes the same time whether the first character differs or the last. An ordinary `equals` stops at the first difference, and by measuring how long the server takes to refuse, an attacker could in principle discover a hash one character at a time. This is called a *timing attack*.

`find` also stamps `lastUsedAt`, so the owner can later see which phones still use the API and remove tokens that have been idle for months.

14.4.3 The filter

Listing 14.9 puts the token to work.

Listing 14.9 `BearerTokenFilter.java`: from a bearer token to an authenticated staff member.

```
package com.profitina.laundry.security;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.List;
import org.springframework.security.authentication
        .UsernamePasswordAuthenticationToken;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

/**
 * "Authorization: Bearer 12|secret" -> the staff member behind token 12.
 * Not a @Component: it must run inside the API chain only.
 */
public class BearerTokenFilter extends OncePerRequestFilter {

    private final TokenService tokens;

    public BearerTokenFilter(TokenService tokens) {
        this.tokens = tokens;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain)
            throws ServletException, IOException {
        String header = request.getHeader("Authorization");
        if (header != null && header.startsWith("Bearer ")) {
            tokens.find(header.substring(7).trim()).ifPresent(t -> {
                var role = new SimpleGrantedAuthority(
                        "ROLE_" + t.getStaff().getRole().name());
                SecurityContextHolder.getContext().setAuthentication(
                        UsernamePasswordAuthenticationToken.authenticated(
                                t.getStaff().getEmail(), t.getId(),
                                List.of(role)));
            });
        }
        chain.doFilter(request, response);    // no token: stay anonymous
    }
}
```

`OncePerRequestFilter` is Spring's base class for filters that must run exactly once per request. The filter reads the `Authorization` header; if it starts with `Bearer` and the token is valid, it places an authenticated identity in the `SecurityContextHolder`, the per-request storage from which the rest of Spring Security, and the controllers, learn who is asking. The identity's principal is the e-mail, its credentials the token's number (needed to sign this device out), and its only permission the role with its prefix. If there is no header, or the token is invalid, the filter does nothing and passes the request on: it does not decide what an anonymous request may do. That decision belongs to the chain's rules, which are in one place and easier to read than scattered `if` statements.

The class is not annotated `@Component`, and the comment says why. Spring Boot registers every filter bean it finds in the servlet container, where it would run for *every* request, including the pages. Created with `new` inside the API chain, it runs only there.

14.4.4 Signing in and signing out

Listing 14.10 is the API's front door.

Listing 14.10 `TokenApiController.java`: `POST /api/tokens` gives a token, `DELETE /api/tokens/current` takes it back.

```
package com.profitina.laundry.api;

import com.profitina.laundry.security.TokenService;
import jakarta.validation.Valid;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import org.springframework.http.HttpStatus;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

/** POST /api/tokens and DELETE /api/tokens/current, as in Chapter 10. */
@RestController
@RequestMapping("/api/tokens")
public class TokenApiController {

    public record TokenRequest(@NotBlank @Email String email,
                               @NotBlank String password,
                               @NotBlank @Size(max = 60) String device) {
    }

    public record UserView(String name, String role) {
    }

    public record TokenView(String token, UserView user) {
    }

    private final TokenService tokens;

    public TokenApiController(TokenService tokens) {
        this.tokens = tokens;
    }
```

Listing 14.10 (Continued) *TokenApiController.java: POST /api/tokens gives a token, DELETE /api/tokens/current takes it back.*

```
    /** The token is shown once; only its hash is kept. */
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public TokenView store(@Valid @RequestBody TokenRequest in) {
        TokenService.Issued t = tokens.issue(in.email(), in.password(),
                in.device());
        return new TokenView(t.token(), new UserView(t.staff().getName(),
                t.staff().getRole().name().toLowerCase()));
    }

    /** Signs this device out: its token stops working at once. */
    @DeleteMapping("/current")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void destroy(Authentication who) {
        tokens.revoke((Long) who.getCredentials());
    }
}
```

The answer to a successful sign-in has status 201, because a token, a new resource, was created:

```
{"token":"3|kq1V0cS2...","user":{"name":"Kasir 1","role":"cashier"}}
```

Unlike every other answer of the API, it is not wrapped in `{"data": ...}`; Chapter 10's Laravel controller returns the same bare object, and the contract follows Laravel. The token appears only in this answer. If the app loses it, it signs in again and receives a new one. `DELETE /api/tokens/current` reads the token's number from the credentials that the filter stored, deletes the row, and answers 204 with no body; from that moment the token opens nothing.

14.5 Two Filter Chains

14.5.1 The configuration

Listing 14.11 is the whole security configuration.

Listing 14.11 `SecurityConfig.java`: one chain for the API, one for the pages.

```
package com.profitina.laundry.security;

import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.method.configuration.*;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication
        .UsernamePasswordAuthenticationFilter;

/**
 * Two filter chains: the JSON API (bearer tokens, no session, no CSRF)
 * and the web pages (login form, session cookie, CSRF tokens in forms).
 */
@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Bean
    @Order(1)
    SecurityFilterChain api(HttpSecurity http, TokenService tokens)
            throws Exception {
        http.securityMatcher("/api/**")
            .authorizeHttpRequests(auth -> auth
                .requestMatchers(HttpMethod.POST, "/api/tokens").permitAll()
                .requestMatchers(HttpMethod.DELETE, "/api/customers/*")
                    .hasRole("OWNER")
                .anyRequest().authenticated())
            .addFilterBefore(new BearerTokenFilter(tokens),
                    UsernamePasswordAuthenticationFilter.class)
            .sessionManagement(s -> s
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(csrf -> csrf.disable())      // no cookies, so no CSRF risk
            .exceptionHandling(e -> e
```

Listing 14.11 (Continued) *SecurityConfig.java: one chain for the API, one for the pages.*

```
                .authenticationEntryPoint((request, response, ex) ->
                    json(response, 401, "Unauthenticated."))
                .accessDeniedHandler((request, response, ex) ->
                    json(response, 403, "This action is unauthorized.")));
        return http.build();
    }

    @Bean
    @Order(2)
    SecurityFilterChain web(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(auth -> auth
                .requestMatchers("/login", "/css/**", "/error").permitAll()
                .anyRequest().authenticated())
            .formLogin(form -> form
                .loginPage("/login")
                .usernameParameter("email")
                .defaultSuccessUrl("/", false))
            .logout(logout -> logout.logoutSuccessUrl("/login?logout"));
        return http.build();
    }

    /** The same {"message": ...} shape as every other API error. */
    private static void json(HttpServletResponse response, int status,
                             String message) throws IOException {
        response.setStatus(status);
        response.setContentType("application/json");
        response.getWriter().write("{\"message\":\"" + message + "\"}");
    }
}
```

There are two beans of type `SecurityFilterChain`, and **Figure 14.1** shows how a request finds its chain.

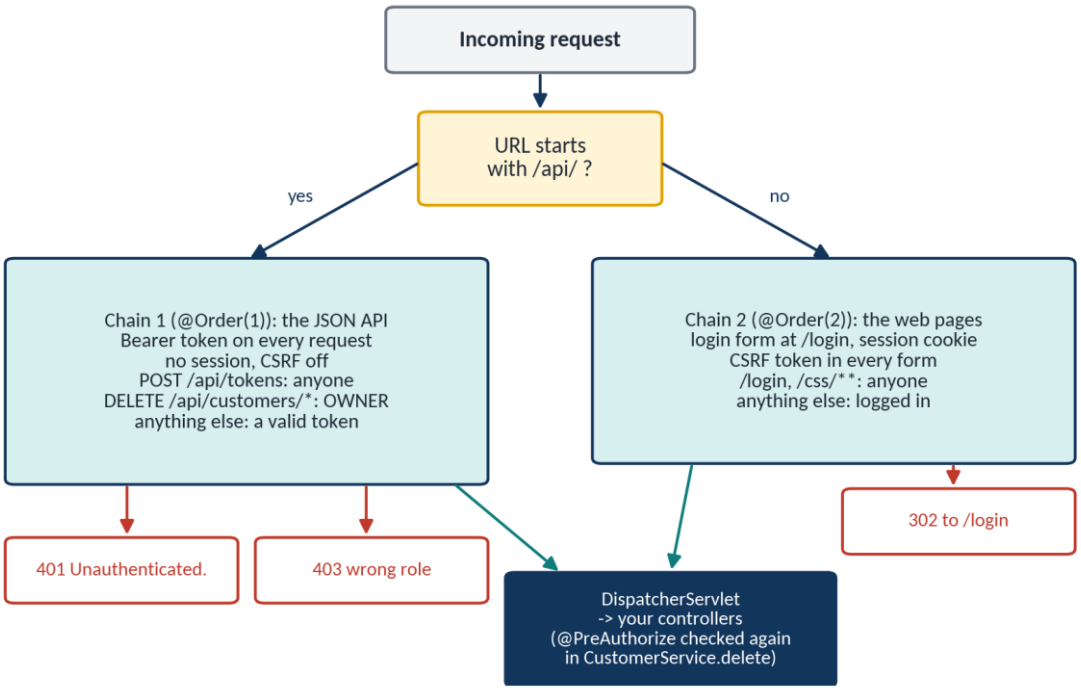


Figure 14.1 The two filter chains: requests under `/api/` go to the stateless chain with bearer tokens and JSON refusals; all others go to the chain with a login form, a session cookie and CSRF protection.

`@Order(1)` puts the API chain first. Its `securityMatcher("/api/**")` claims every URL that starts with `/api/`; any other URL falls through to the web chain, which has no matcher and so accepts

everything that is left. If the order were reversed, the web chain would claim the API's requests too and answer Raka's app with a redirect to a login page.

The API chain's rules are read from top to bottom, and the first rule that matches decides:

1. `POST /api/tokens` is open to everyone; one cannot require a token to obtain a token.
2. `DELETE /api/customers/*` requires the role `OWNER`.
3. Every other request needs an authenticated caller, of any role.

The chain adds `BearerTokenFilter` before Spring's own `UsernamePasswordAuthenticationFilter`, so the identity is known before any rule is checked. `SessionCreationPolicy.STATELESS` tells Spring Security never to create a session for the API, so every request must bring its token. CSRF protection is switched off for this chain only, for the reason given in Section 14.5.4. Finally, the two handlers write the contract's refusals: 401 with `{"message":"Unauthenticated."}` and 403 with `{"message":"This action is unauthorized."}`, the exact texts Laravel sends. Without them Spring Security would answer 401 with an empty body and 403 with its own error page.

The web chain is shorter. `/login`, the style sheet and the error page are open; every other page needs a logged-in staff member. `formLogin` switches on Spring Security's login processing: `POST /login` is handled by Spring Security, reading the fields `email` and `password`; success leads to the page the user wanted before logging in, or to `/`; failure to `/login?error`. `logout` handles `POST /logout` and returns to `/login?logout`. `@EnableMethodSecurity` on the class switches on the method checks of Section 14.5.5.

14.5.2 Following requests by hand

Before trusting a configuration, follow a few requests through it on paper. **Table 14.1** does it for six.

Table 14.1 Six requests followed through the two chains.

Request	Chain	Identity found	Rule that decides	Answer
`GET /api/orders`, no header	API	none	any request: authenticated	401 JSON
`GET /api/orders`, cashier's token	API	Kasir 1, `ROLE_CASHIER`	any request: authenticated	200
`DELETE /api/customers/2`, cashier's token	API	Kasir 1, `ROLE_CASHIER`	`DELETE /api/customers/*`: owner	403 JSON
`DELETE /api/customers/2`, owner's token	API	Owner, `ROLE_OWNER`	`DELETE /api/customers/*`: owner	204, or 409 if she has orders
`GET /orders`, no cookie	web	none	any request: authenticated	302 to `/login`
`POST /customers`, session, no CSRF field	web	Kasir 1	CSRF filter, before any rule	403

The fourth row shows a useful separation. Security decides whether the owner *may* delete; the service decides whether this customer *can* be deleted. A customer with orders is refused with 409 and "Sari Wulandari has orders and cannot be deleted." whoever asks.

14.5.3 The login page

Listing 14.12 and **Listing 14.13** draw the login form; Spring Security handles the rest.

Listing 14.12 `LoginController.java`: only the `GET`; Spring Security handles the `POST`.

```
package com.profitina.laundry.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class LoginController {

    /** Spring Security itself handles POST /login; we draw the form. */
    @GetMapping("/login")
    public String login() {
        return "login";
    }
}
```

Listing 14.13 `login.html`: e-mail and password, with messages for a failed login and a logout.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="~{fragments :: head('Log in')}"></head>
<body>
<main>
  <h1>Profitina Laundry</h1>
  <p class="bad" th:if="${param.error}">Wrong e-mail or password.</p>
  <p class="ok" th:if="${param.logout}">You have logged out.</p>
  <form method="post" th:action="@{/login}">
    <label>E-mail <input name="email" type="email" autofocus required></label>
    <label>Password <input name="password" type="password" required></label>
    <button>Log in</button>
  </form>
</main>
</body>
</html>
```

`${param.error}` reads the query string, so `/login?error` shows the message. The navigation fragment gains a logout button (**Listing 14.14**). Logging out is a `POST` in a form, not a link, for the same reason that deleting is never a `GET`.

Listing 14.14 `fragments.html`: the navigation now ends with a logout form.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:fragment="head(title)">
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title th:text="${title} + ' | Profitina Laundry'">Profitina Laundry</title>
  <link rel="stylesheet" th:href="@{/css/app.css}">
</head>
<body>
<header th:fragment="nav">
  <strong>Profitina Laundry</strong>
  <nav>
    <a th:href="@{/orders(status='queued')}">Queue</a>
    <a th:href="@{/orders}">All orders</a>
    <a th:href="@{/orders/new}">New order</a>
    <a th:href="@{/customers}">Customers</a>
  </nav>
  <form method="post" th:action="@{/logout}"><button>Log out</button></form>
</header>
<div th:fragment="flash">
  <p class="ok" th:if="${message}" th:text="${message}">Saved.</p>
  <p class="bad" th:if="${error}" th:text="${error}">Not allowed.</p>
</div>
</body>
</html>
```

14.5.4 CSRF: why the pages need it and the API does not

A browser attaches the shop's session cookie to *every* request for the shop's address, whichever page caused the request. Suppose Kasir 1 is logged in and, in another tab, opens a page on some other site that contains a hidden form posting to `http://laundry.local/customers/2/delete` and a line of script that submits it. The browser sends the cookie; the server sees a logged-in cashier's request. This is **cross-site request forgery** (CSRF).

The defense is a secret the other site cannot know. Spring Security's web chain gives each session a CSRF token and refuses every `POST`, `PUT`, `PATCH` or `DELETE` that does not carry it [2]. Thymeleaf inserts it automatically: a form written with `th:action` receives a hidden input `_csrf` with the token, which is why every form in Chapters 13 and 14 uses `th:action` instead of a plain `action` [6]. A form on another site cannot read the token, so its forged request fails with 403.

The API chain switches the protection off, and this is safe for a precise reason: the API does not use cookies. The token travels in a header that the program sets explicitly; a browser never adds it by itself, so a foreign page cannot make the browser send it. If the API ever accepted the session cookie, CSRF protection would have to come back.

14.5.5 A second line of defense in the service

The URL rule stops cashiers from deleting customers through the API. But the page controller of Chapter 13 also has a delete button, at `POST /customers/{id}/delete`, and the web chain has no rule about it. Adding a URL rule for every page that deletes something is easy to forget. The service, however, is the one place every delete passes through, so the check belongs there as well (**Listing 14.15**).

Listing 14.15 `CustomerService.java`: `delete` is for the owner, whoever calls it.

```java
package com.profitina.laundry.service;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.repository.CustomerRepository;
import com.profitina.laundry.repository.OrderRepository;
import java.time.Clock;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.List;
import java.util.Optional;
import org.springframework.data.domain.Sort;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/** Everything the shop does with customers, in one place. */
@Service
@Transactional
public class CustomerService {

    private final CustomerRepository customers;
    private final OrderRepository orders;
    private final Clock clock;

    public CustomerService(CustomerRepository customers,
                           OrderRepository orders, Clock clock) {
        this.customers = customers;
        this.orders = orders;
        this.clock = clock;
    }

    /** By name, or by phone when the text is digits. */
    @Transactional(readOnly = true)
    public List<Customer> search(String text) {
        String q = text == null ? "" : text.trim();
        if (q.isEmpty()) {
            return customers.findAll(Sort.by("name"));
        }
        String digits = q.replaceAll("[^0-9]", "");
        return q.matches("[0-9 +-]+")
                ? customers.findByPhoneContainingOrderByName(digits)
                : customers.findByNameContainingIgnoreCaseOrderByName(q);
    }
```

Listing 14.15 (Continued) *CustomerService.java: delete is for the owner, whoever calls it.*

```
    @Transactional(readOnly = true)
    public Optional<Customer> findByPhone(String phone) {
        return customers.findByPhone(phone.replaceAll("[^0-9]", ""));
    }

    @Transactional(readOnly = true)
    public Customer get(Long id) {
        return customers.findById(id)
                .orElseThrow(() -> new NotFoundException("customer", id));
    }

    public Customer create(String name, String phone, String address) {
        if (customers.existsByPhone(phone)) {     // Laravel: unique rule
            throw new RuleException("phone",
                    "The phone has already been taken.");
        }
        return customers.save(new Customer(name.trim(), phone,
                blankToNull(address), now()));
    }

    public Customer update(Long id, String name, String phone,
                           String address) {
        Customer customer = get(id);
        if (customers.existsByPhoneAndIdNot(phone, id)) {
            throw new RuleException("phone",
                    "The phone has already been taken.");
        }
        customer.change(name.trim(), phone, blankToNull(address));
        return customer;            // saved at commit (dirty checking)
    }

    @PreAuthorize("hasRole('OWNER')")
    public void delete(Long id) {
        Customer customer = get(id);
        if (orders.existsByCustomerId(id)) {
            throw new ConflictException(customer.getName()
                    + " has orders and cannot be deleted.");
        }
        customers.delete(customer);
    }

    private LocalDateTime now() {
        return LocalDateTime.now(clock).truncatedTo(ChronoUnit.SECONDS);
    }
```

Listing 14.15 (Continued) *CustomerService.java: delete is for the owner, whoever calls it.*

```
    private static String blankToNull(String s) {
        return s == null || s.isBlank() ? null : s.trim();
    }
}
```

`@PreAuthorize("hasRole('OWNER')")` makes Spring check the caller's role before the method runs and throw `AccessDeniedException` if it does not match; the API chain turns that into 403 JSON, the web chain into its 403 error page. Like `@Transactional`, it works through a proxy around the bean, so it applies to calls from other beans, which is exactly how controllers call services. **Listing 14.16** adds the API's `DELETE`.

Listing 14.16 `CustomerApiController.java`: the new `DELETE /api/customers/{id}`.

```
package com.profitina.laundry.api;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.service.CustomerService;
import jakarta.validation.Valid;
import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/customers")
public class CustomerApiController {

    private final CustomerService customers;

    public CustomerApiController(CustomerService customers) {
        this.customers = customers;
    }

    @GetMapping
    public DataView<List<CustomerView>> index(
            @RequestParam(required = false) String phone) {
        List<Customer> found = phone == null ? customers.search("")
                : customers.findByPhone(phone).stream().toList();
        return new DataView<>(found.stream().map(CustomerView::of).toList());
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public DataView<CustomerView> store(
            @Valid @RequestBody CreateCustomerRequest in) {
        return new DataView<>(CustomerView.of(
                customers.create(in.name(), in.phone(), in.address())));
    }
```

Listing 14.16 (Continued) *CustomerApiController.java: the new DELETE /api/customers/{id}.*

```
    /** Owner only (SecurityConfig and CustomerService both say so). */
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void destroy(@PathVariable Long id) {
        customers.delete(id);
    }
}
```

The URL rule and the method rule overlap on purpose. This is *defense in depth*: if a future change moves the URL or adds a new caller, one of the two checks still holds. OWASP's guidance on access control recommends exactly this: deny by default, and check on the server, at every entry point [7].

14.6 Designing the REST API

With the doors in place, it is time to step back and examine the API itself. Chapter 12 did not invent its shape: it copied the Laravel API so that both servers can be checked by one contract.

The decisions behind that shape are the ones every API designer faces, and knowing the reasons makes it possible to defend them, or to change them knowingly.

14.6.1 Resources, URLs and methods

REST, the style described by Fielding, organizes an API around *resources*, things with an address, which clients read and change with the uniform methods of HTTP [1]. The laundry's resources are the price list, customers, orders, the payments of an order and the daily report; tokens are resources too. **Table 14.2** is the complete API after this chapter.

Table 14.2 The Profitina Laundry API after Chapter 14.

Method and URL	Meaning	Success	Who
`POST /api/tokens`	sign in, create a token	201	anyone
`DELETE /api/tokens/current`	sign this device out	204	staff
`GET /api/services`	the active price list	200	staff
`GET /api/services/{id}`	one service	200	staff
`GET /api/customers?phone=`	find customers by phone	200	staff
`POST /api/customers`	a new customer	201	staff
`DELETE /api/customers/{id}`	remove a customer without orders	204	owner
`GET /api/orders?status=&page=`	one page of orders	200	staff
`POST /api/orders`	a new order	201	staff
`GET /api/orders/{id}`	one order with its payments	200	staff
`PATCH /api/orders/{id}/status`	the next step	200	staff
`POST /api/orders/{id}/payments`	a payment	201	staff
`GET /api/reports/daily?date=`	the figures of one day	200	staff

Three conventions run through the table. URLs name things with plural nouns (`/orders`, not `/getOrders`); the method says what happens to them. A resource that belongs to another is nested under it (`/orders/7/payments`), and nesting stops at one level. Filters and pages are query parameters (`?status=queued&page=2`), because they select from a resource rather than naming a new one.

The status change is the one place where the design had to choose. It could have been `PATCH /api/orders/7` with `{"status": "finished"}`, a partial update of the order. The contract uses a sub-resource, `PATCH /api/orders/7/status`, which makes it obvious in logs and in security rules that this request changes the status and nothing else. Some designers prefer an action URL such as `POST /api/orders/7/finish`; it is less RESTful and more explicit. Each choice is defensible; what matters is to choose once and apply it everywhere.

14.6.2 Safe and idempotent methods

HTTP gives methods two properties that clients and proxies rely on [1]. A **safe** method does not change anything: `GET` may be repeated, cached and prefetched freely. An **idempotent** method has the same effect whether it is sent once or several times: `PUT` and `DELETE` are idempotent, `POST` is not, and `PATCH` is idempotent only if the particular request is.

Look at the laundry's methods with these properties in mind. `DELETE /api/customers/5` sent twice deletes once and answers 404 the second time: the state of the server is the same. `PATCH /api/orders/7/status` with `{"status": "finished"}` is idempotent in its effect: the second time, the order is already finished, and the service refuses with 422 ("An order that is finished can only become departed.") without changing anything. This is exactly why the contract requires the client to name the wanted status instead of "move to the next step": a driver's phone that repeats a request after losing its signal does not skip a step.

`POST /api/orders/7/payments` is not idempotent. If the phone sends a Rp 20.000 payment, loses the connection before the answer arrives, and sends it again, the order is paid twice, unless the balance rule happens to refuse it. Section 14.8 describes the standard cure, an idempotency key.

14.6.3 Status codes and error bodies

Table 14.3 lists every status code the API uses.

Table 14.3 The status codes of the API and when each is used.

Code	Meaning in this API	Body
200 OK	a read or a status change succeeded	`{"data": ...}`
201 Created	a token, customer, order or payment was created	the new resource
204 No Content	a sign-out or deletion succeeded	none
401 Unauthorized	no valid token	`{"message": "Unauthenticated."}`
403 Forbidden	a valid token, but the role may not do this	`{"message": "This action is unauthorized."}`
404 Not Found	the record does not exist	`{"message": "No order 999."}`
409 Conflict	the record's state does not allow the request	`{"message": "Order #7 has left the shop and is closed."}`
422 Unprocessable Content	the input breaks a rule	`{"message": "The given data was invalid.", "errors": {...}}`

The difference between 422 and 409 is worth stating precisely, because the two are often confused. 422 means "what you sent is wrong": a quantity of 0, a pickup without an address, an overpayment; the client can fix the request and send it again. 409 means "what you sent may be fine, but the resource is in a state that does not allow it": a departed order cannot take a payment of any amount. RFC 9110 defines 409 as a conflict with the current state of the target resource and 422 as content that is well formed but cannot be processed [1].

The error body always has a `message` that a person can read, and for 422 an `errors` object that a program can use, keyed by the same field names the client sent. Raka's app puts each message under the input it belongs to, as Chapter 13's pages do. An alternative is the standard *problem details* format of RFC 9457, with `type`, `title`, `status` and `detail` [8]; Spring supports it out of the box. The laundry does not use it because the contract follows Laravel's shape, and one shape for all three servers is worth more to Raka than a standard shape on one of them.

14.6.4 Envelopes and pages

Every successful answer is wrapped: `{"data": {...}}` for one record, `{"data": [...], "meta": {"current_page": 1, "per_page": 15, "total": 42, "last_page": 3}}` for a page. The envelope costs a few bytes and gives room to grow: pagination facts, and perhaps later links or

warnings, can be added next to `data` without breaking clients that read only `data`. A bare JSON array, in contrast, can never receive a sibling field. The token answer of Section 14.4.4 is the one exception, inherited from Laravel.

Page numbers with a fixed page size (*offset pagination*) are simple and let a user jump to page 3. Their weakness shows on lists that change while being read: if a new order arrives while the phone is on page 1, page 2 starts one order later and one order appears twice. *Cursor pagination*, "the 15 orders after order 812", avoids that and suits endless scrolling; for a counter screen that lists at most a few dozen open orders, page numbers are fine.

14.6.5 What the API still lacks: versions

The URLs have no version number. As long as the phone app and the server are updated together, that is acceptable. The moment Raka publishes the app and some drivers do not update it, every change to the JSON must be *backward compatible*: fields may be added, never renamed or removed, and a new required field in a request breaks old apps. When a breaking change becomes unavoidable, the usual approach is to serve the old and the new shape side by side under `/api/v1/...` and `/api/v2/...`, or to select the version with a header, and to retire the old one after a announced date. Spring Framework 7 has built-in support for API versioning of this kind [9]. The laundry records it as a future improvement: adding `/v1` now to all three frameworks would change the contract without a user who needs it.

14.7 Automated Tests on Four Levels

14.7.1 The pyramid

A test proves one thing about the code, and a test that starts more of the application proves more, but runs slower and, when it fails, points less precisely at the cause. A healthy test suite therefore has many small tests at the bottom and a few large ones at the top, the *test pyramid* of **Figure 14.2**.

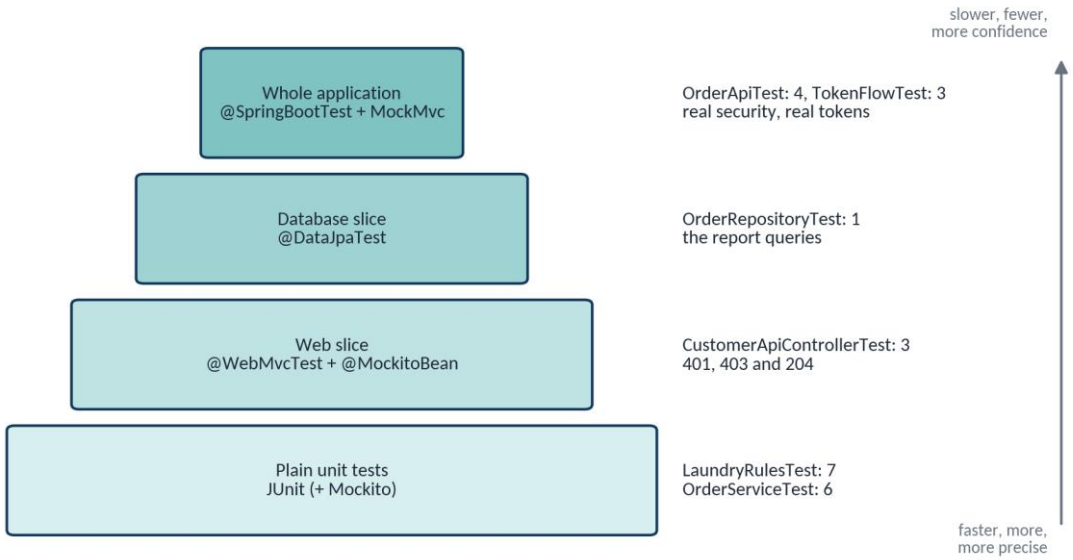


Figure 14.2 The four levels of tests in the Chapter 14 project, from plain unit tests at the bottom to tests of the whole application at the top.

Table 14.4 names the levels and what each can and cannot catch.

Table 14.4 The four levels of tests, with the 24 runs of this chapter.

Level	Starts	Test classes	Runs	Catches	Misses
Unit	nothing but the class	`LaundryRulesTest`, `OrderServiceTest`	13	wrong prices and rules	wiring, SQL, HTTP
Web slice	controllers, JSON, security	`CustomerApiControllerTest`	3	status codes, JSON, access rules	services, SQL
JPA slice	entities, repositories, H2	`OrderRepositoryTest`	1	wrong queries and mappings	HTTP, rules
Whole application	everything, with H2	`OrderApiTest`, `TokenFlowTest`	7	wrong wiring, end-to-end flows	little, but slowly

A *slice* test starts only the part of Spring Boot that one layer needs [10]. It is faster than starting everything and fails for fewer unrelated reasons. **Listing 14.17** adds the starters that provide the slices and the security test helpers; in Spring Boot 4 each technology brings its own test starter.

Listing 14.17 `pom.xml`: security, and one test starter per slice.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>4.1.1</version>
    <relativePath/>
  </parent>
  <groupId>com.profitina</groupId>
  <artifactId>laundry</artifactId>
  <version>14.0.0</version>
  <properties>
    <java.version>25</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webmvc</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-validation</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>
    <!-- SQLite for the shop, as promised in Chapter 11 -->
    <dependency>
      <groupId>org.xerial</groupId>
      <artifactId>sqlite-jdbc</artifactId>
      <version>3.53.2.0</version>
    </dependency>
    <dependency>
```

Listing 14.17 (Continued) *pom.xml: security, and one test starter per slice.*

```
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-community-dialects</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
    <!-- H2 in memory for the tests only -->
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webmvc-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

The unit level is already in place, written with JUnit 6 and Mockito [11]: the rules test of Chapter 11 with seven runs (six prices and one workflow), and the service test of Chapter 13 with six. The other levels follow.

14.7.2 The web layer alone

Listing 14.18 tests the customer API controller together with the real security configuration, and nothing else.

Listing 14.18 `CustomerApiControllerTest.java`: the web layer and the security rules, with the service mocked.

```
package com.profitina.laundry.api;

import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoInteractions;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

import com.profitina.laundry.security.SecurityConfig;
import com.profitina.laundry.security.TokenService;
import com.profitina.laundry.service.CustomerService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.webmvc.test.autoconfigure.WebMvcTest;
import org.springframework.context.annotation.Import;
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.test.web.servlet.MockMvc;

/** The web layer alone, with the real security rules. */
@WebMvcTest(CustomerApiController.class)
@Import(SecurityConfig.class)
class CustomerApiControllerTest {

    @Autowired
    MockMvc mvc;

    @MockitoBean
    CustomerService customers;

    @MockitoBean
    TokenService tokens;                    // SecurityConfig needs one

    @Test
    void withoutATokenTheApiSays401() throws Exception {
        mvc.perform(get("/api/customers"))
           .andExpect(status().isUnauthorized())
           .andExpect(jsonPath("$.message").value("Unauthenticated."));
    }
```

Listing 14.18 (Continued) *CustomerApiControllerTest.java: the web layer and the security rules, with the service mocked.*

```
    @Test
    @WithMockUser(roles = "CASHIER")
    void aCashierMayNotDelete() throws Exception {
        mvc.perform(delete("/api/customers/2"))
           .andExpect(status().isForbidden())
           .andExpect(jsonPath("$.message")
                   .value("This action is unauthorized."));
        verifyNoInteractions(customers);
    }

    @Test
    @WithMockUser(roles = "OWNER")
    void theOwnerDeletes() throws Exception {
        mvc.perform(delete("/api/customers/2"))
           .andExpect(status().isNoContent());
        verify(customers).delete(2L);
    }
}
```

`@WebMvcTest(CustomerApiController.class)` starts Spring MVC with this one controller, the JSON configuration and the security filter chains, but no services, repositories or database. `@Import(SecurityConfig.class)` makes sure our chains are used instead of Spring Boot's

default. `@MockitoBean` puts Mockito mocks into the application context in place of `CustomerService`, which the controller needs, and `TokenService`, which the API chain needs. `MockMvc` sends requests through the filters and the controller without a network.

`@WithMockUser(roles = "CASHIER")` runs one test as if a cashier were logged in, without tokens or passwords. The three tests follow the first three rows of **Table 14.1** and check the whole answer, status and JSON. `verifyNoInteractions(customers)` in the second test proves that the cashier's request was stopped *before* the controller: the service was never called. In this slice `CustomerService` is a mock, so its own `@PreAuthorize` does not run; the URL rule alone must refuse, and this test shows that it does.

14.7.3 The database queries

Listing 14.19 tests the repository's queries on a real, empty database.

Listing 14.19 `OrderRepositoryTest.java`: the report queries and `existsByCustomerId` against H2.

```
package com.profitina.laundry.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import com.profitina.laundry.domain.Customer;
import com.profitina.laundry.domain.Order;
import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.domain.Service;
import java.math.BigDecimal;
import java.time.LocalDateTime;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.data.jpa.test.autoconfigure.DataJpaTest;

/** The queries alone, on a fresh in-memory database per test. */
@DataJpaTest
class OrderRepositoryTest {

    @Autowired CustomerRepository customers;
    @Autowired ServiceRepository services;
    @Autowired OrderRepository orders;

    @Test
    void theDailyCountsAndTheCustomerCheck() {
        LocalDateTime nine = LocalDateTime.of(2026, 9, 25, 9, 0);
        Customer sari = customers.save(
                new Customer("Sari Wulandari", "081200000101", null, nine));
        Service wash = services.save(
                new Service("S0001", "Wash, dry & fold (per kg)", 8000));
        orders.save(new Order(sari, wash, new BigDecimal("4.5"), false, null,
                null, nine));
        Order b = new Order(sari, wash, new BigDecimal("3"), false, null,
                null, nine.plusHours(1));
        b.advance(nine.plusHours(5));                  // queued -> finished
        orders.save(b);
```

Listing 14.19 (Continued) *OrderRepositoryTest.java: the report queries and existsByCustomerId against H2.*

```
        var counts = orders.countByStatusBetween(nine.toLocalDate()
                .atStartOfDay(), nine.toLocalDate().plusDays(1).atStartOfDay());
        assertEquals(2, counts.size());
        assertEquals(1, counts.stream()
                .filter(c -> c.getStatus() == OrderStatus.FINISHED)
                .mapToLong(OrderRepository.StatusCount::getN).sum());
        assertEquals(60_000, orders.billedBetween(nine.minusHours(9),
                nine.plusHours(15)));
        assertTrue(orders.existsByCustomerId(sari.getId()));
    }
}
```

`@DataJpaTest` starts JPA, the repositories and an embedded database, runs each test in a transaction and rolls it back at the end, so tests cannot disturb each other. The test saves one customer, one service and two orders, finishes one, and checks the counts per status, the billed amount of the day ($36.000 + 24.000 = 60.000$ by the pricing rule of Section 13.4.3) and the query that protects customers with orders. A mistake in the JPQL of `countByStatusBetween`, or a mapping error in `Order`, fails here with a precise message, where the whole-application tests would only report a wrong number in a report.

14.7.4 The whole application

Listing 14.20 starts everything: the application context, the real security chains, the services, the repositories and a fresh H2 database with the demo accounts.

Listing 14.20 `TokenFlowTest.java`: sign in, use the token, sign out, and see the token refused.

```
package com.profitina.laundry.security;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

import com.jayway.jsonpath.JsonPath;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.webmvc.test.autoconfigure.AutoConfigureMockMvc;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

/** The whole application: real accounts, real hashes, real tokens. */
@SpringBootTest
@AutoConfigureMockMvc
class TokenFlowTest {

    @Autowired
    MockMvc mvc;

    private String login(String password, int expect) throws Exception {
        return mvc.perform(post("/api/tokens")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("""
                            {"email": "kasir1@profitina-laundry.example",
                             "password": "%s", "device": "test"}"""
                               .formatted(password)))
                .andExpect(status().is(expect))
                .andReturn().getResponse().getContentAsString();
    }

    @Test
    void aTokenOpensTheApiUntilItIsRevoked() throws Exception {
        String body = login("Kasir#2026", 201);
        String bearer = "Bearer " + JsonPath.read(body, "$.token");
        mvc.perform(get("/api/orders").header("Authorization", bearer))
           .andExpect(status().isOk());
        mvc.perform(delete("/api/tokens/current")
                        .header("Authorization", bearer))
           .andExpect(status().isNoContent());
        mvc.perform(get("/api/orders").header("Authorization", bearer))
           .andExpect(status().isUnauthorized());
    }
```

Listing 14.20 (Continued) *TokenFlowTest.java: sign in, use the token, sign out, and see the token refused.*

```
    @Test
    void aWrongPasswordGetsNoToken() throws Exception {
        String body = login("kasir#2026", 422);
        org.junit.jupiter.api.Assertions.assertEquals(
                "These credentials do not match our records.",
                JsonPath.read(body, "$.errors.email[0]"));
    }

    @Test
    void aMadeUpTokenIsRefused() throws Exception {
        mvc.perform(get("/api/orders")
                        .header("Authorization", "Bearer 1|not-the-secret"))
           .andExpect(status().isUnauthorized());
    }
}
```

`@SpringBootTest` with `@AutoConfigureMockMvc` starts the complete application and gives the test a `MockMvc` that goes through all its filters. The first test is the life of a token: a sign-in with the

cashier's real password, a successful request, a sign-out, and the same token refused. `JsonPath.read` pulls the token out of the answer. The second test checks that a wrong password (a lower-case `k`) receives 422 with Laravel's sentence; the third, that a token with a made-up secret opens nothing. These tests catch mistakes no smaller test can: a filter registered in the wrong chain, a password encoder that hashes differently when seeding and when checking, a token column too short for its hash.

Chapter 12's `OrderApiTest` also starts the whole application; after this chapter it would receive 401 for every request, since it sends no token. One annotation on the class, `@WithMockUser(roles = "CASHIER")`, makes it run as a cashier, and its four tests pass unchanged. It checks the order flow; `TokenFlowTest` checks the tokens; neither has to do the other's work.

Run everything with

```
mvn test
```

Surefire, Maven's test runner, reports 24 test runs in six classes, with a line per class and a summary; the reports for each class are written to `target/surefire-reports`. In a shop that makes changes regularly, this command runs before every release, and ideally on every push to the shared repository.

14.8 Making It Better

Idempotency keys. Let the phone app send a random `Idempotency-Key` header with every `POST`; the server stores the key with the answer and, when the same key arrives again, returns the stored answer instead of creating a second payment. Many payment APIs work this way.

Token expiry and a device list. Tokens now live until they are revoked. A page where the owner sees every token with its device and `lastUsedAt`, and a rule that removes tokens idle for 90 days, would close the gap left by a lost phone.

Limit login attempts. `POST /api/tokens` and the login form accept unlimited guesses. Counting failed attempts per e-mail and per address, and delaying or refusing after five in a minute, makes guessing impractical; Laravel's login throttling of Chapter 10 does this.

Record who did what. With accounts in place, `Order` and `Payment` can store the staff member who created them. The service reads the name from `SecurityContextHolder`, or better, receives it as a parameter from the controller, so that it stays testable.

HTTPS. Everything in this chapter assumes that passwords, cookies and tokens travel encrypted. Behind a reverse proxy such as nginx or Caddy with a certificate, set `server.forward-headers-strategy=native` so that Spring knows the original request was secure.

14.9 Using It for Real

Almost every small business application needs this chapter's pieces in the same arrangement: staff accounts with a slow password hash, a login form and session for people at a screen, tokens for programs, a short list of roles, and checks at the URL and in the service. A pharmacy counter would add a role for the pharmacist, who alone may hand over prescription medicines; a school canteen would give parents tokens for an app that shows their child's balance, each token limited to one child. When the number of roles grows beyond a handful, it is time to store *permissions* ("may delete customers", "may change prices") and give roles sets of permissions, instead of scattering role names through the code.

Do not build your own password storage or token scheme beyond what this chapter shows. Use the framework's password encoders, SecureRandom and constant-time comparisons, and follow OWASP's cheat sheets [7], [3]. Before a system with real customer data goes live, have someone who did not write it try to break it.

One more small habit makes a large difference: write a test for every refusal, not only for the paths that succeed. Tests that check that a cashier may *not* delete, that a revoked token no longer works, and that a wrong password does *not* produce a token are the tests that fail on the day someone accidentally loosens the security configuration. An open door produces no complaints from anyone; only a test notices it.

14.10 Profitina in Practice

Profitina uses the same two kinds of doors. Its web dashboard signs people in with a session, and its API issues tokens to programs: the MCP server through which AI assistants read a user's visibility reports, and scripts that customers write against the public API. Tokens are shown once and stored hashed, each can be revoked on its own, and access is checked twice, at the route and in the service that performs the operation. The platform's automated test suite, well over a thousand tests, follows the pyramid of this chapter: many fast tests of pure rules such as scoring and coin limits, fewer tests of single endpoints, and a smaller set of full flows such as "sign up, verify, run a scan, see the report", which must all pass before a release.

14.11 Common Mistakes, and How to Avoid Them

- **Storing passwords with SHA-256, or in plain text.** Use a `PasswordEncoder`; a fast hash of a human password can be reversed by guessing.
- **Registering a custom filter as a `@Component`.** It then runs for every request, outside its chain. Create it with `new` in the chain that needs it.
- **Chains in the wrong order.** A chain without a matcher placed first claims every request. Give the specific chain `@Order(1)` and a `securityMatcher`.
- **Disabling CSRF everywhere "because the API needs it".** Only a chain that uses no cookies may do without CSRF protection.
- **Plain `action=` in a Thymeleaf form.** The CSRF field is added only with `th:action`; the form then fails with 403.
- **Different answers for "no such e-mail" and "wrong password".** They tell an attacker which accounts exist.
- **Checking roles only in the page template.** Hiding a button is not access control. Check at the URL, in the service, or both.
- **`hasRole("ROLE_OWNER")`.** `hasRole` adds the prefix itself; write `hasRole("OWNER")`.
- **Tests that break after security is added.** A whole-application test without an identity now gets 401. Give it `@WithMockUser` or a real token, and add tests that check the refusals too.
- **Only whole-application tests.** They are slow and fail vaguely. Put most tests at the bottom of the pyramid.

14.12 Chapter Summary

- Authentication establishes who is asking; access control decides what they may do. Missing identity gives 401, a refusal 403.
- Passwords are stored as salted, deliberately slow hashes; bcrypt's work doubles with each step of its cost (Equation (14.1)). Spring's delegating encoder names the algorithm in the stored value.
- API tokens in the Sanctum style are `id|secret`; the table keeps only the SHA-256 of a 240-bit secret (Equation (14.2)), compared in constant time.
- Two filter chains (**Figure 14.1**): the API chain is stateless, reads bearer tokens and answers 401 and 403 in the contract's JSON; the web chain uses a login form, a session and CSRF tokens (**Table 14.1**).
- Deleting a customer is restricted to the owner at the URL and with `@PreAuthorize` in the service: defense in depth.
- The API's design follows REST: nouns in URLs, methods with defined safety and idempotence, a small set of status codes with 422 for bad input and 409 for a conflicting state, and envelopes for data and pages (**Table 14.2**, **Table 14.3**). Versioning is the main thing it still lacks.
- Tests come in four levels (**Figure 14.2**, **Table 14.4**): unit tests, a web slice with the real security rules, a JPA slice, and the whole application; 24 runs in all.

14.13 Exercises

1. Using Equation (14.1), how many times more work is a bcrypt check at cost 12 than at cost 10? If a check at cost 10 takes 60 ms on the shop's server, about how long does one at cost 12 take, and would a cashier notice?
2. For each request, name the chain, the rule that decides, and the status: (a) `POST /api/tokens` with a wrong password; (b) `GET /api/services` with the token `abc`; (c) `GET /customers` in a browser that logged out; (d) `POST /api/orders/7/payments` with a revoked token.
3. Explain why the API chain may disable CSRF protection but the web chain may not. What would have to change if Raka's app used the session cookie instead of a token?
4. A colleague proposes to store the plain token in `api_tokens` "so that the owner can see it again". Give two reasons against, and propose what the owner's page should show instead.
5. Is `DELETE /api/customers/5` idempotent in this API? Describe what the server answers the first and the second time, and argue whether the answers matter for idempotence.
6. Write a web slice test that checks that a cashier *may* create a customer through `POST /api/customers` and receives 201.
7. Add the rule "only the owner may read the daily report" in two ways: as a URL rule and with `@PreAuthorize`. Which tests from this chapter fail, and what must change in them?

14.14 Solutions to the Exercises

1. By Equation (14.1), $2^{12}/2^{10} = 4$ times more work, so about $4 \times 60 = 240$ ms. A cashier logs in once per shift, so a quarter of a second is not noticeable; for an attacker it divides the number of guesses per second by four.

2. (a) API chain; `POST /api/tokens` is open, so the request reaches the controller, and `TokenService.issue` throws `RuleException`: 422 with "These credentials do not match our records." (b) API chain; the filter finds no bar in `abc`, so the caller stays anonymous; "any request:

authenticated" decides: 401 JSON. (c) Web chain; no session; "any request: authenticated" decides: 302 redirect to `/login`. (d) API chain; the token's row was deleted, so `find` returns nothing and the caller is anonymous: 401 JSON.

3. CSRF works because browsers add cookies to requests automatically, whichever site caused them. The API chain does not read cookies at all; its proof is a header that only code holding the token can set, and a foreign page cannot make the browser add it. The web chain relies on a session cookie, so a foreign page could make the browser send a request with it; the CSRF token is what the foreign page cannot supply. If Raka's app used the session cookie, the API chain would have to enable CSRF protection and the app would have to fetch and send the token with every changing request, or the cookie would need the `SameSite=Strict` attribute and careful review.

4. A database that contains working tokens is as good as a list of passwords to anyone who obtains a backup; and the owner has no need to see a secret, only to know which devices have access. The page should list each token's number, device, owner, creation time and last use, with a button that revokes it. If a device loses its token, it signs in again and receives a new one.

5. Yes. The first request deletes the customer and answers 204; the second finds no customer and answers 404 ("No customer 5."). Idempotence is about the effect on the server's state, not about identical answers: after one request and after two, the customer is gone and nothing else changed. A client that repeats a `DELETE` after losing the answer should therefore treat 404 as "already done".

6. In `CustomerApiControllerTest`, with the imports for `post`, `MediaType` and `jsonPath`:

```
    @Test
    @WithMockUser(roles = "CASHIER")
    void aCashierMayAddACustomer() throws Exception {
        when(customers.create("Rina", "081300000001", null)).thenReturn(
                new Customer("Rina", "081300000001", null,
                        LocalDateTime.of(2026, 9, 25, 9, 0)));
        mvc.perform(post("/api/customers")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("""
                            {"name": "Rina", "phone": "0813-0000-0001"}"""))
           .andExpect(status().isCreated())
           .andExpect(jsonPath("$.data.phone").value("081300000001"));
    }
```

The phone is stripped to digits by the request record before the service is called, so the mock is taught the digits-only value. CSRF does not interfere, because the API chain disables it.

7. As a URL rule, add `.requestMatchers(HttpMethod.GET, "/api/reports/**").hasRole("OWNER")` before `anyRequest()` in the API chain. With `@PreAuthorize`, annotate `ReportService.daily` with `@PreAuthorize("hasRole('OWNER')")`. Either way, `OrderApiTest.theDailyReportNeedsADate` runs as a cashier and now receives 403 instead of 422 or 200. Give that one test its own `@WithMockUser(roles = "OWNER")`, and add a test that checks that a cashier receives 403 with "This action is unauthorized.". Doing both, as for deleting customers, protects the report also from a future page that shows it.

References

[1] R. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, IETF, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[2] Broadcom, “Spring Security Reference: Servlet Applications, Architecture, Password Storage and CSRF,” version 7.0. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-security/reference/

[3] OWASP Foundation, “Password Storage Cheat Sheet,” OWASP Cheat Sheet Series. Accessed: Sep. 25, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[4] Laravel, “Laravel Sanctum” and “Hashing,” Laravel 13.x documentation. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/sanctum

[5] Microsoft, “Introduction to Identity on ASP.NET Core” and “PasswordHasher,” ASP.NET Core 10 documentation. Accessed: Sep. 25, 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/security/authentication/identity

[6] The Thymeleaf Team, “Thymeleaf + Spring Security integration basics.” Accessed: Sep. 25, 2026. [Online]. Available: https://www.thymeleaf.org/doc/articles/springsecurity.html

[7] OWASP Foundation, “Authorization Cheat Sheet,” OWASP Cheat Sheet Series. Accessed: Sep. 25, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

[8] M. Nottingham, E. Wilde, and S. Dalal, “Problem Details for HTTP APIs,” RFC 9457, IETF, Jul. 2023. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9457

[9] Broadcom, “Spring Framework Reference: Web on Servlet Stack, API Versioning,” version 7.0. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc-versioning.html

[10] Broadcom, “Spring Boot Reference Documentation: Testing,” version 4.1. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-boot/reference/testing/

[11] JUnit Team, “JUnit User Guide,” JUnit 6. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.junit.org/

Chapter 15

Comparing the Three Frameworks: Performance, Productivity, Ecosystem, Choosing

Profitina Laundry now exists three times: in ASP.NET Core, in Laravel and in Spring Boot. Having built the same shop three times, you can compare the frameworks from experience rather than from rumor, and this chapter shows how to do it honestly. By the end of it you will be able to map every concept of one framework onto the other two; read a public benchmark critically and know what it does and does not say about your application; measure a framework yourself and check the numbers with a simple law of queues; compare productivity, ecosystem and security defaults with evidence; read the support calendars of .NET, Java, Spring Boot, PHP and Laravel; and choose a framework for a real project with reasons you can write down.

15.1 Where We Left Off

Bu Wati's nephew, Dimas, finished his diploma in informatics this year. He helped at the laundry during the holidays and saw all three versions of the counter program running on the same computer. Now a friend's family, who run a small catering business with twelve staff, has asked him to build their order system, and he has to pick one framework. His classmates each have an opinion. One says ASP.NET Core is the fastest, so it must be the best. Another says Laravel is the easiest to host, because every cheap hosting package has PHP. A third says Spring Boot is what the banks use, so it must be the most serious. All three statements contain some truth, and none of them is a reason.

This chapter gives Dimas, and you, a way to decide. It first lines the three frameworks up concept by concept (Section 15.3), because the biggest surprise after building the same application three times is how *similar* they are. Then it looks at the evidence people usually quote: performance benchmarks (Section 15.4), measures of productivity (Section 15.5), surveys of what developers use (Section 15.6), and the calendars that say how long each version receives fixes (Section 15.7). Security defaults get their own comparison (Section 15.8). Section 15.9 turns all of it into a decision procedure.

15.2 The Idea in Plain Words

Choosing a framework is choosing what a team will live with for years. The useful questions are therefore not "which is best?" but "which is best *for this project, this team and this place*?" **Table 15.1** lists the questions this chapter answers and the kind of evidence each needs.

Table 15.1 What to compare, and what kind of evidence each question needs.

Question	Kind of evidence	Where in this chapter
Will it be fast enough?	benchmarks, and your own measurement of your own workload	Section 15.4
How much work is a feature?	code size of the same feature, tools, learning curve	Section 15.5
Can we hire and get help?	developer surveys, package registries, local job and hosting market	Section 15.6
How long is a version supported?	official release and support calendars	Section 15.7
Is it safe by default?	defaults for passwords, XSS, CSRF, SQL	Section 15.8
Does it fit our team and our servers?	what the team knows, where the app must run	Section 15.9

Each kind of evidence has its own trap. A benchmark measures one workload on one machine; a survey measures who answered it; a calendar can change; your own measurement is only as good as its method. The rule that runs through the chapter is simple: *say where each number comes from, when it was measured, and what it measured.*

15.3 One Application, Three Vocabularies

15.3.1 The same concepts

Chapter 2 introduced the ideas that all modern web frameworks share: routing, controllers, models and an ORM, dependency injection, templates. After building the laundry three times, we can fill in the table with the real names (**Table 15.2**).

Table 15.2 The same concepts under three names.

Concept	ASP.NET Core 10	Laravel 13	Spring Boot 4.1
Language, runtime	C# 14, .NET 10	PHP 8.5	Java 25, JVM
New project	`dotnet new`	`laravel new`, `composer create-project`	Spring Initializr
Command-line helper	`dotnet` CLI	`php artisan`	`mvn` or `gradle`
Routing	attributes `[HttpGet("...")]` or `MapGet`	`routes/web.php`, `routes/api.php`	`@GetMapping` on methods
Controller	class deriving from `Controller`	class in `app/Http/Controllers`	`@Controller`, `@RestController`
Input validation	data annotations, `[ApiController]`	Form Request classes, `validate()`	Jakarta Bean Validation, `@Validated`
ORM style	EF Core, unit of work (`DbContext`)	Eloquent, active record	JPA/Hibernate, repositories
Schema changes	EF Core migrations	Laravel migrations	Flyway or Liquibase
Dependency injection	built in, constructor injection	service container, constructor injection	Spring container, constructor injection
Templates	Razor (`.cshtml`)	Blade (`.blade.php`)	Thymeleaf (`.html`)
JSON output	`System.Text.Json`, records	API Resources	Jackson 3, records
Errors in one JSON shape	`IExceptionHandler`, `ModelState`	the framework's own 422, 404 and 401 bodies	`@RestControllerAdvice`
Login and roles	Identity or an authentication handler, `[Authorize(Roles=...)]`	guards, Gates, `can` middleware	Spring Security filter chains, `@PreAuthorize`
Tests	xUnit, `WebApplicationFactory`	PHPUnit or Pest, `RefreshDatabase`	JUnit 6, `@WebMvcTest`, `@DataJpaTest`

Most rows differ only in spelling. Two rows show a real difference in design, and they are worth understanding.

Active record against unit of work. In Laravel's Eloquent an object knows how to save itself: `$order->update([...])` writes to the database at once. In EF Core and JPA an object is plain; a separate object (`DbContext`, the JPA `EntityManager`) tracks the objects loaded in a unit of work and writes all their changes when you call `SaveChanges` or when the transaction commits (the *dirty checking* of Section 13.5). Active record is quicker to start with and reads naturally in small applications; unit of work makes it easier to keep several changes together and to test rules without a database. Both reach the same result in our shop, but you must know which one you are using: in JPA, forgetting that an entity is managed means changes you did not intend may be saved; in Eloquent, forgetting that `update` saves immediately means half a change may be saved before an error.

How much is configured by convention. Laravel finds the table `orders` for the model `Order` and the columns by name, and routes live in one file you can read top to bottom. Spring Boot configures whatever it finds on the classpath (add the Security starter and every page needs a login) and finds controllers by scanning annotations. ASP.NET Core leaves more in `Program.cs`, which you read from top to bottom and which says in order what the application uses. None of these is better; each hides different things, and the first week with a new framework is mostly spent learning what it does without being asked.

15.3.2 The life of one request

The similarity goes deeper than names. Follow `POST /api/orders` from Chapter 16 through each framework, and the same stations appear in the same order.

1. **The web server receives the bytes.** ASP.NET Core has its own server, Kestrel, inside the application. Spring Boot embeds Tomcat. Laravel runs inside PHP, which is started by a web server (nginx with PHP-FPM, Apache, or `php artisan serve` while developing) for each request.
2. **A pipeline of middleware runs.** In ASP.NET Core it is the sequence of `app.Use...` calls in `Program.cs`: exception handling, authentication, access checks, then the endpoint. In Laravel it is the global middleware and the route's middleware (`auth:sanctum`, `can:`) declared in `bootstrap/app.php` and the route files. In Spring Boot it is the servlet filters, among them Spring Security's filter chain of Chapter 14. In all three, a middleware can answer on its own (a 401, a redirect to the login page) so that the request never reaches a controller.
3. **Routing picks one handler** by method and path: an attribute on a C# method, a line in `routes/api.php`, a `@PostMapping` on a Java method.
4. **Input is bound and validated.** The JSON body becomes an `OrderRequest` record in C# and Java or a validated array in a Laravel Form Request. Invalid input stops here with 422 and every error at once: Laravel's Form Requests do this by themselves, while ASP.NET Core and Spring Boot were taught to do the same in Chapters 12 and 16.
5. **Our code runs:** a thin controller method calls the service, the service applies the rules (`LaundryRules` in C# and Java, `PriceCalculator` and `OrderWorkflow` in PHP) inside a transaction, and the repository or model talks to the database through the ORM.
6. **The result is serialized** to JSON (`System.Text.Json`, an API Resource, Jackson) or rendered by a template, and exceptions on the way are turned into the contract's error bodies by one central handler.

Once you can name these six stations, a new framework is mostly a matter of finding out what each station is called there and where its configuration lives. It also tells you where to look when something goes wrong: a 401 comes from station 2, a 404 for a URL you are sure exists from station 3, a 422 from station 4, a 409 from station 5.

15.3.3 Testing compared

The test pyramid of Chapter 14 exists in all three frameworks, with different names for the same levels. Plain unit tests of the rules need no framework at all: JUnit in Java, xUnit or NUnit in C#, PHPUnit or Pest in PHP. Tests of the web layer send requests without opening a network port: `MockMvc` with `@WebMvcTest` in Spring Boot, `WebApplicationFactory` with an in-memory test server in ASP.NET Core, and in Laravel the `getJson` and `postJson` helpers of the test case, which call the application directly. Database tests use a throw-away database: `@DataJpaTest` rolls back after each test, Laravel's `RefreshDatabase` trait migrates a fresh database, and EF Core tests typically use SQLite in memory. Laravel's test helpers are the most compact (a whole API test fits in a few lines, as `CapstoneTest` in Chapter 16 shows); Spring Boot's slices are the most precise about what they start; ASP.NET Core's `WebApplicationFactory` starts the real application with replaced services, which sits between the two. None of the three makes testing hard, and in none of them is it optional.

15.3.4 The same endpoint three times

To compare code and speed fairly we need one small piece of work written the same way in all three. This chapter uses a price quote: `GET /api/quote?price=8000&unit=kg&quantity=2.5&pickup=true` returns the total of Equation (15.1), the rule of Chapters 8 and 11 that Section 13.4.3 restated, with b the billed hundredths (at least 300 for a kilogram service, whole units otherwise):

$$T = \left\lfloor \frac{p \cdot b + 50}{100} \right\rfloor + \begin{cases} 10000 & \text{pickup and subtotal} < 100000 \\ 0 & \text{otherwise} \end{cases} \tag{15.1}$$

Listing 15.1 is the ASP.NET Core version, a *minimal API*: no controller class, just a route and a function.

Listing 15.1 `PerfLab/Program.cs`: the quote endpoint as an ASP.NET Core minimal API.

```
// The same tiny endpoint in all three frameworks: price one laundry order.
// GET /api/quote?price=8000&unit=kg&quantity=2.5&pickup=true
//   -> {"price":8000,"unit":"kg","quantity":2.5,"pickup":true,"total":34000}
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Logging.ClearProviders();                // measure the work, not logs
var app = builder.Build();

app.MapGet("/api/quote",
    (long price, string unit, decimal quantity, bool pickup) =>
{
    // The rule of Chapters 8 and 11, in hundredths of a unit.
    long h = decimal.ToInt64(quantity * 100m);
    long billed = unit == "kg" ? Math.Max(h, 300) : (h + 99) / 100 * 100;
    long subtotal = (price * billed + 50) / 100;
    long total = subtotal + (pickup && subtotal < 100_000 ? 10_000 : 0);
    return Results.Json(new Quote(price, unit, quantity, pickup, total),
                        QuoteJson.Default.Quote);
});

app.Run();

public record Quote(long Price, string Unit, decimal Quantity, bool Pickup,
                    long Total);

// The JSON code for Quote is generated at compile time (no reflection).
[JsonSerializable(typeof(Quote))]
[JsonSourceGenerationOptions(
    PropertyNamingPolicy = JsonKnownNamingPolicy.CamelCase)]
public partial class QuoteJson : JsonSerializerContext
{
}
```

`CreateSlimBuilder` starts ASP.NET Core with fewer default features, and the `QuoteJson` class uses the source generator of `System.Text.Json`, which writes the JSON code at compile time instead of discovering the record's properties at run time. Both are optimizations that the .NET team recommends for small, fast services. **Listing 15.2** is the same endpoint in Laravel, reusing the `PriceCalculator` class of Chapter 8, and **Listing 15.3** in Spring Boot.

Listing 15.2 `routes/api.php`: the quote endpoint as a Laravel route.

```php
<?php

use App\Domain\PriceCalculator;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;

// The same tiny endpoint as PerfLab (ASP.NET Core) and QuoteController.
Route::get('/quote', function (Request $request) {
    $price = (int) $request->query('price');
    $unit = (string) $request->query('unit');
    $quantity = (string) $request->query('quantity');
    $pickup = $request->boolean('pickup');
    return response()->json([
        'price' => $price, 'unit' => $unit, 'quantity' => (float) $quantity,
        'pickup' => $pickup,
        'total' => PriceCalculator::total($unit === 'kg', $price, $quantity,
            $pickup),
    ]);
});
```

Listing 15.3 `QuoteController.java`: the quote endpoint as a Spring Boot controller.

```java
package com.profitina.perflab;

import java.math.BigDecimal;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

/** The same tiny endpoint as PerfLab (ASP.NET Core) and routes/api.php. */
@RestController
public class QuoteController {

    public record Quote(long price, String unit, BigDecimal quantity,
                        boolean pickup, long total) { }

    @GetMapping("/api/quote")
    public Quote quote(@RequestParam long price, @RequestParam String unit,
                       @RequestParam BigDecimal quantity,
                       @RequestParam(defaultValue = "false") boolean pickup) {
        // The rule of Chapters 8 and 11, in hundredths of a unit.
        long h = quantity.movePointRight(2).longValueExact();
        long billed = "kg".equals(unit) ? Math.max(h, 300)
                : (h + 99) / 100 * 100;
        long subtotal = (price * billed + 50) / 100;
        long total = subtotal + (pickup && subtotal < 100_000 ? 10_000 : 0);
        return new Quote(price, unit, quantity, pickup, total);
    }
}
```

Read the three side by side and the similarity is striking: a route, three parameters, one rule, one JSON object. The differences are in the language more than in the frameworks: the quantity arrives as C#'s `decimal`, Java's `BigDecimal` or a PHP string, and all three then compute in whole hundredths with integers, so no floating-point rounding can make one of them disagree by a rupiah.

15.4 Performance

15.4.1 What a public benchmark measures

The best-known public comparison of web frameworks was the TechEmpower Framework Benchmarks. For more than a decade it ran the same six small tests (plain JSON, one database query, many queries, updates, plain text, and "Fortunes") against hundreds of framework implementations on identical hardware and published the results as *rounds*. Round 23, published in February 2025, ran on new servers with 56-core Intel Xeon Gold 6330 processors and 40 Gbit/s networking [1]. It turned out to be the last round: on March 24, 2026 TechEmpower announced that it was sunsetting the project and archived its repository, keeping the past results as a snapshot [2]. Round 23 is therefore still "the latest" and will remain so, which is worth knowing when someone quotes a newer-sounding number.

The *Fortunes* test is the one closest to a real page: the server reads all rows of a small table through the framework's ORM, adds one row in memory, sorts them, and renders an HTML table through the framework's template engine with proper escaping [1]. **Figure 15.1** shows the Round 23 Fortunes throughput for the three frameworks, as compiled from the official results by Pham [3].

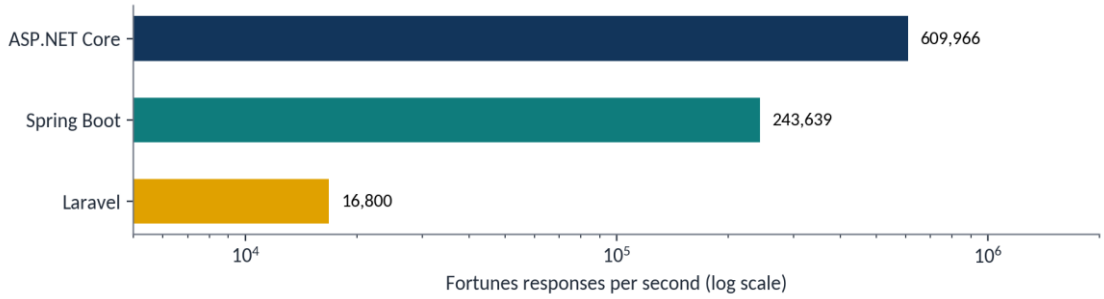


Figure 15.1 TechEmpower Round 23 (February 2025), Fortunes test: responses per second of the best ASP.NET Core, Spring and Laravel implementations, on a logarithmic scale.

The differences are large: ASP.NET Core answered about 2.5 times as many requests per second as Spring, and Spring about 14.5 times as many as Laravel. Before concluding anything, read what the numbers are.

- **They are the best entry of each family, tuned by experts.** The fastest ASP.NET Core entries use the leanest parts of the stack with raw database access; the Laravel entry runs on a production PHP set-up. Your application written the way this book writes it would be slower in all three.
- **The hardware is far larger than a laundry's server.** At 609,966 responses per second, ASP.NET Core could serve every order Profitina Laundry takes in a year in well under a second.
- **The logarithmic axis hides how much is enough.** Laravel's 16,800 responses per second is still about a million pages a minute. A shop, a school or a clinic sends perhaps a few requests per second at its busiest.

> **KEY INSIGHT:** A benchmark answers "how fast *can* this framework be?", not "how fast will *my* application be?" For most business applications, all three frameworks are far faster than they need to be, and the database, the network and the code you write decide the speed.

Performance still matters in two situations: when an application really serves heavy traffic (a national e-commerce site, a payment gateway), where a faster framework means fewer servers, and when the budget is one small server, where a lower memory footprint lets more fit on it.

15.4.2 Measuring it yourself

Instead of trusting anyone's numbers, measure your own workload on your own machine. **Listing 15.4** is a small load generator: a number of *workers* each send requests one after another for a fixed time, and the program reports requests per second and the latency that 50% and 99% of requests stayed under (the *p50* and *p99*).

Listing 15.4 `LoadGen/Program.cs`: a load generator reporting throughput and latency percentiles.

```
// A small load generator: N workers send GET requests for a fixed time and
// we report requests per second and latency percentiles.
// dotnet run -c Release -- <url> <seconds> <workers>
using System.Diagnostics;

string url = args.Length > 0 ? args[0]
    : "http://localhost:4150/api/quote?price=8000&unit=kg&quantity=2.5&pickup=true";
int seconds = args.Length > 1 ? int.Parse(args[1]) : 10;
int workers = args.Length > 2 ? int.Parse(args[2]) : 16;

using var http = new HttpClient(new SocketsHttpHandler
{
    MaxConnectionsPerServer = workers,
    PooledConnectionLifetime = TimeSpan.FromMinutes(5)
});

// Warm-up: let the JIT compile the hot paths on both sides first.
for (int i = 0; i < 2000; i++) (await http.GetAsync(url)).Dispose();

var latencies = new List<double>[workers];
long errors = 0;
var stop = DateTime.UtcNow.AddSeconds(seconds);
var clock = Stopwatch.StartNew();
await Task.WhenAll(Enumerable.Range(0, workers).Select(async w =>
{
    var mine = latencies[w] = new List<double>(200_000);
    while (DateTime.UtcNow < stop)
    {
        long start = Stopwatch.GetTimestamp();
        using var res = await http.GetAsync(url);
        await res.Content.ReadAsByteArrayAsync();
        if (!res.IsSuccessStatusCode) Interlocked.Increment(ref errors);
        mine.Add(Stopwatch.GetElapsedTime(start).TotalMilliseconds);
    }
}));
double elapsed = clock.Elapsed.TotalSeconds;

var all = latencies.SelectMany(l => l).Order().ToArray();
double P(double q) => all[(int)Math.Min(all.Length - 1, q * all.Length)];
Console.WriteLine($"workers {workers,3}: {all.Length / elapsed,9:F0} req/s   "
    + $"p50 {P(0.50):F2} ms   p99 {P(0.99):F2} ms   errors {errors}");
```

Two details make the measurement fair. The warm-up sends 2,000 requests before measuring, because .NET, the JVM and PHP's opcode cache all run code slower the first few times while they compile and cache it. And each worker records every latency, so the percentiles are computed from all requests, not averaged away.

The machine where this book was written could run only the ASP.NET Core version (the others need packages it could not download). PerfLab was built in Release mode and started; it answered its first request 446 ms after being launched, and the quote was right: `{"price":8000,"unit":"kg","quantity":2.5,"pickup":true,"total":34000}`, the 3 kg minimum plus the pickup fee. Then LoadGen ran for 10 seconds each with 1, 8 and 32 workers, on the same two-core virtual machine:

```
workers   1:      9408 req/s   p50 0.09 ms   p99 0.28 ms   errors 0
workers   8:     26238 req/s   p50 0.28 ms   p99 0.89 ms   errors 0
workers  32:     32566 req/s   p50 0.84 ms   p99 3.00 ms   errors 0
```

Figure 15.2 draws the same numbers.

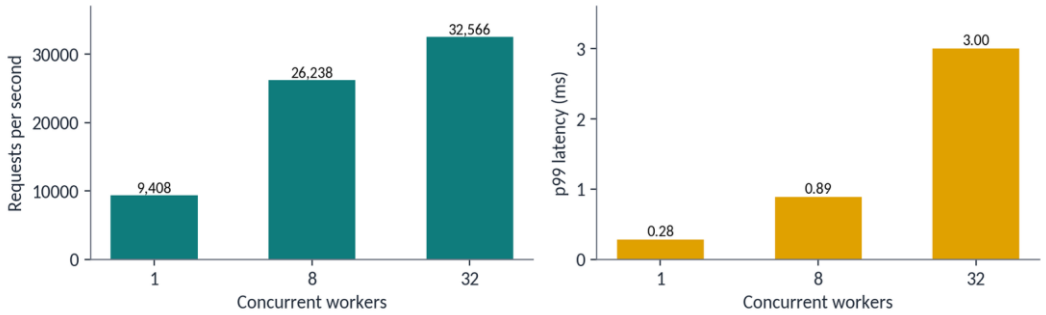


Figure 15.2 Our own measurement of the ASP.NET Core quote endpoint on a 2-vCPU virtual machine (September 25, 2026): throughput levels off after 8 workers while the p99 latency keeps growing.

The shape of the curve is the most useful lesson. From 1 to 8 workers, throughput almost triples (9,408 to 26,238 requests per second), because one worker leaves the processors idle while it waits for each answer. From 8 to 32 workers, four times as many, it grows by only a quarter, to 32,566: the two cores are nearly saturated, shared between the server and the load generator itself. Beyond that point, more workers mostly make each request wait longer in line: the p99 latency grows from 0.89 ms to 3.00 ms, more than three times.

15.4.3 Checking the numbers with Little's law

Queueing theory gives a check that anyone can do by hand. *Little's law* says that in a stable system the average number of requests in the system, L, equals the rate at which they arrive, λ, times the average time each spends in the system, W:

$$L = \lambda W \tag{15.2}$$

In a closed load test every worker always has exactly one request in flight, so L is the number of workers and λ is the measured throughput; Equation (15.2) then gives the average latency $W = L/\lambda$. For one worker, $W = 1/9408$ s ≈ 0.106 ms; for 8 workers, $W = 8/26238$ s ≈ 0.305 ms; for 32 workers, $W = 32/32566$ s ≈ 0.983 ms. Each average lies a little above the measured median (0.09, 0.28 and 0.84 ms) and well below the p99, which is what a latency distribution with a long tail looks like. Had Equation (15.2) given an average *below* the median, or far above the p99, the measurement would have been wrong somewhere, for example a worker that stopped early. A one-line calculation has confirmed the benchmark.

The same law answers a planning question. If the catering business expects at most 5 requests per second at noon and each takes about 20 ms including the database, then on average $L = 5 \times 0.02 = 0.1$ request is being served at any moment: one small server in any of the three frameworks is idle most of the time.

15.4.4 Start-up time and memory

Two other numbers matter for small deployments. *Start-up time* decides how quickly an application restarts after an update, or how quickly a new copy starts when traffic grows: PerfLab answered after about half a second; a Spring Boot application with JPA typically needs a few seconds on the JVM (much less when compiled ahead of time with GraalVM, which Spring Boot

supports); a PHP application has no start-up at all, because each request starts fresh on a server that is already running. *Memory* decides how many applications fit on one server: a PHP request uses a few megabytes and then releases them, while the .NET runtime and the JVM keep a warm process of tens to hundreds of megabytes. Measure both the same way as throughput, on your machine, with your application; the README of this chapter's code shows how.

15.5 Productivity

Productivity is harder to measure than speed, and claims about it deserve the same care. Three honest measures are available from this book.

Code size of the same feature. Counting whole applications says little, because each version of the capstone carries different extras (two data stores in ASP.NET Core, the Chapter 13 pages in Spring Boot). A fairer measure is the size of one *feature added to all three in the same way*. Chapter 16 adds two: deleting a customer, for the owner only and never when the customer has orders, and a public board that shows the queue by first name. **Table 15.3** counts the lines each needed, without blank lines and comments.

Table 15.3 Lines needed for the same two features in the Chapter 16 capstone, counted on September 25, 2026.

Feature	ASP.NET Core 10	Laravel 13	Spring Boot 4.1
Owner-only delete of a customer	about 20: service 10, controller 6, store query 4	about 12: controller 8, route 3, gate 1	about 17: service 9, controller 5, security rule 2, repository 1
Public board, controller and view	18 + 42	17 + 37, plus 1 route	21 + 35, plus 1 security rule

The board costs about the same everywhere, because most of it is HTML. The delete shows the frameworks' characters: Laravel's route-model binding, a one-line gate and `abort(409, ...)` make it the shortest; Spring Boot spends its lines on two independent checks (a URL rule and `@PreAuthorize`) and on a repository method that Eloquent's `$customer->orders()->exists()` does not need; ASP.NET Core writes the store's query by hand because its capstone supports two stores. Lines of code are a rough measure (a line of PHP and a line of Java do not carry the same meaning), but the result supports what building the shop three times suggests: for this kind of application no framework is dramatically shorter, and the differences come from design choices more than from the language.

Tools that write code for you. `php artisan make:model Order -mcr` creates a model, a migration and a resource controller in one command; `dotnet new` and `dotnet ef migrations add` and Spring Initializr do the same kind of work. All three have excellent support in VS Code, and Java and C# have the extra help of a compiler that checks types before the program runs, which catches many mistakes that PHP reports only when the line is executed (PHPStan and Larastan close much of that gap for Laravel).

The learning curve. Laravel is usually the quickest start for someone who knows PHP from a web programming course, because routing, models and templates are visible and few in number. ASP.NET Core and Spring Boot ask more of the beginner up front (a compiled language, dependency injection everywhere, more configuration to understand), and repay it on larger codebases with stronger types and tooling. This matches the experience of Chapters 3 to 14: the first Laravel chapter produced a working page soonest, and the Spring Boot chapters needed the most explanation of what happens without being asked.

The languages themselves. The three languages have grown towards each other in the last years, and the capstone uses the same modern features in all three. Records describe data in one line: `record OrderRequest(...)` in C#, `record CreateOrderRequest(...)` in Java; PHP has readonly classes and constructor promotion for the same purpose. Enumerations with values replace magic strings: `enum OrderStatus` in all three, and PHP's backed enums (`case Queued = 'queued'`) even convert to and from the stored text by themselves. Pattern matching replaces chains of `if`: the next status is a `switch` expression in C#'s `LaundryRules.Advance` and Java's `LaundryRules.advance`, and a `match` in the PHP enum's `next()` that `OrderWorkflow::advance` calls; C# warns and Java refuses to compile when a case is missing, and PHP throws `UnhandledMatchError` when the line runs. Nullability is explicit in C# (`string?`), declared with `?string` in PHP and handled with `Optional` and annotations in Java. The differences that remain are about *when* mistakes are found: C# and Java check types when the program is compiled, PHP when the line runs (unless a static analyzer such as PHPStan is used). For a team that likes the compiler to catch mistakes, that difference matters more than any benchmark.

15.6 Ecosystem and Community

15.6.1 What developers use

The largest public survey of developers is Stack Overflow's. The 2025 edition asked which web frameworks and technologies respondents had done extensive development work with in the past year; 23,678 respondents answered that question, 19,460 of them professional developers [4]. **Figure 15.3** shows the three frameworks of this book.

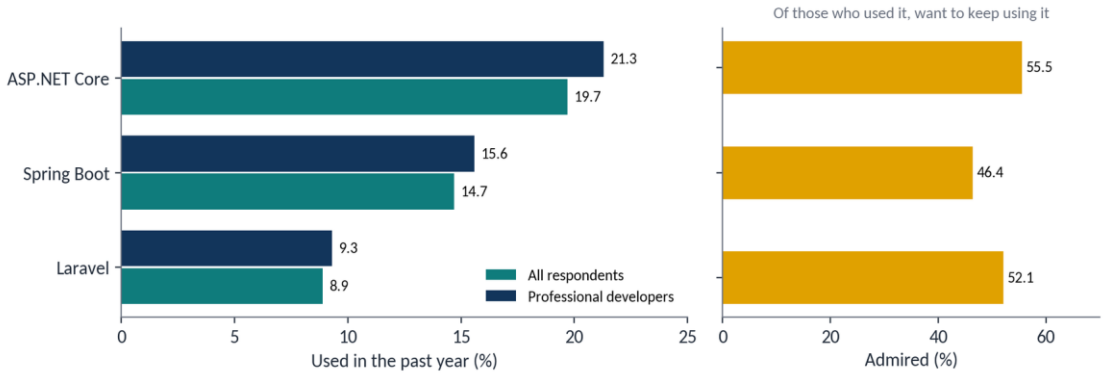


Figure 15.3 Stack Overflow Developer Survey 2025: share of respondents who used each framework in the past year, and share of its users who want to keep using it ("admired").

ASP.NET Core was used by 19.7% of all respondents and 21.3% of professionals, Spring Boot by 14.7% and 15.6%, Laravel by 8.9% and 9.3% [4]. For comparison, Node.js was used by 48.7% and Express by 19.9%, so all three of our frameworks are large, mainstream choices. The "admired" figure asks those who used a technology whether they want to continue: 55.5% for ASP.NET Core, 52.1% for Laravel and 46.4% for Spring Boot [4].

Read a survey the way you read a benchmark: who answered? Stack Overflow's respondents are self-selected visitors of an English-language site, and survey participation varies strongly by country and by kind of employer. The survey does not tell you what the job market in Surabaya or Makassar asks for, nor how many small Indonesian businesses run PHP on shared hosting. For those questions, look at local evidence: job advertisements in your city, the hosting packages local providers sell, and what the companies you want to work for actually use.

15.6.2 Packages, hosting and help

Each framework lives in a large package ecosystem: NuGet for .NET, Packagist (through Composer) for PHP, Maven Central for Java. For a shop application all three offer what is needed: database drivers, PDF receipts, QR codes, Excel export, payment gateway clients. Two practical differences remain. PHP runs on almost any hosting package, including the cheapest shared hosting, while ASP.NET Core and Spring Boot usually need a virtual server or a container platform, which costs a little more and needs a little more knowledge. And the size of the question-and-answer and tutorial literature is large for all three, but for Laravel much of it is written for beginners, for Spring Boot and ASP.NET Core much of it assumes larger systems.

15.7 Support Calendars

A framework version that no longer receives security fixes must not face the internet. Every team therefore needs to know, before it starts, how long the chosen versions will be supported and how often it will have to upgrade. **Table 15.4** collects the dates as of September 25, 2026, and **Figure 15.4** draws them.

Table 15.4 Release and support dates, collected from endoflife.date on September 25, 2026.

Version	Released	Full support until	Security fixes until
.NET 10 (LTS)	Nov 11, 2025	Nov 14, 2028	Nov 14, 2028
.NET 8 (LTS)	Nov 14, 2023	Nov 10, 2026	Nov 10, 2026
Java 25 (LTS, Oracle Premier)	Sep 16, 2025	Sep 30, 2030	Sep 30, 2030
Spring Boot 4.1 (open source)	Jun 30, 2026	Jul 31, 2027	Jul 31, 2027
Spring Boot 4.0 (open source)	Nov 30, 2025	Dec 31, 2026	Dec 31, 2026
PHP 8.5	Nov 20, 2025	Dec 31, 2027	Dec 31, 2029
Laravel 13	Mar 17, 2026	Sep 30, 2027	Mar 17, 2028
Laravel 12	Feb 24, 2025	Aug 13, 2026	Feb 24, 2027

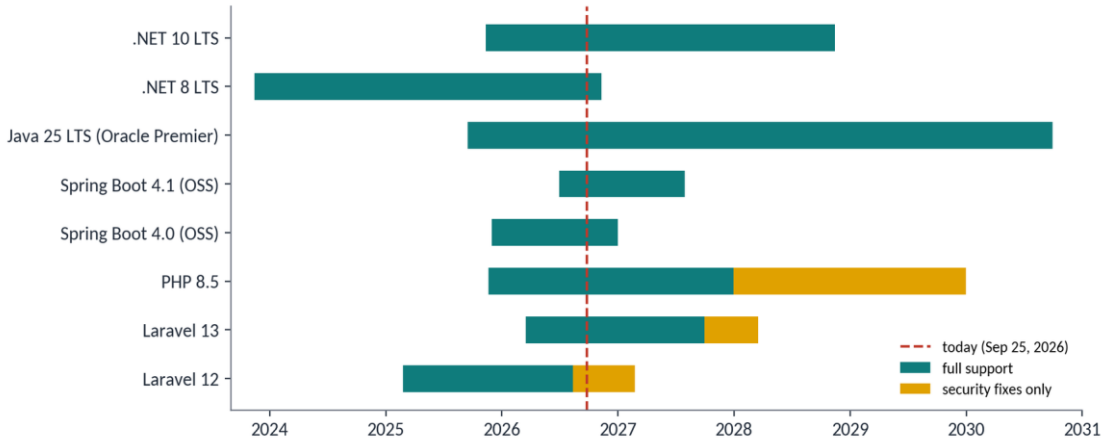


Figure 15.4 The support windows of the versions used in this book; the dashed line is September 25, 2026.

The patterns behind the dates are more useful than the dates themselves.

- **.NET** releases a new major version every November; even-numbered versions are *Long Term Support* (LTS) with three years of support [5]. .NET 10 is supported until November 2028; .NET 8, used by many existing projects, reaches its end on November 10, 2026, so projects on it must upgrade now.

- **Java** releases every six months and designates an LTS version every two years; Java 25 (September 2025) is the current LTS [6]. The JDK's support is long, but a Spring application also depends on Spring's own calendar.
- **Spring Boot** releases a minor version every six months (May or June and November), and each receives open-source fixes for about thirteen months [7], [8]. Commercial support extends this for paying customers. In practice a Spring Boot application should move to the next minor version at least once a year: 4.0 reaches the end of its open-source support on December 31, 2026, so our code uses 4.1.
- **PHP** gives each version two years of active support followed by two years of security fixes [9].
- **Laravel** releases one major version a year, around the first quarter, and provides bug fixes for 18 months and security fixes for two years [10]. Laravel 13 requires PHP 8.3 or newer and supports up to PHP 8.5.

A useful rule of thumb follows: a .NET team upgrades every two or three years, a Laravel team every year, a Spring Boot team every six to twelve months, but Spring Boot's minor upgrades are usually small. Plan the upgrade work into the project from the start, whichever you choose.

15.8 Security by Default

All three frameworks make the safe thing the easy thing, and they agree on most defaults (**Table 15.5**).

Table 15.5 Security defaults compared.

Concern	ASP.NET Core	Laravel	Spring Boot
SQL injection	EF Core parameters every value	Eloquent and the query builder bind every value	JPA and Spring Data bind every value
XSS in templates	Razor encodes `@value`	Blade `{{ }}` escapes	Thymeleaf `th:text` escapes
CSRF on forms	antiforgery tokens, added by tag helpers	`@csrf` token, checked by middleware	token added by Thymeleaf, checked by the security filter
Password hashing (default)	Identity: PBKDF2 with HMAC-SHA512, 100,000 iterations	bcrypt, cost 12	delegating encoder: bcrypt, cost 10
Validation errors	automatic 400 problem document with `[ApiController]`; the capstone switches it off and answers 422	422 JSON with every error, by default	the application's own handler; 422 since Chapter 12

The password defaults come from the frameworks' own code and documentation: ASP.NET Core Identity's `PasswordHasher` (version 3 format) uses PBKDF2 with HMAC-SHA512, a 128-bit salt and 100,000 iterations [11]; Laravel's hashing configuration uses bcrypt with 12 rounds unless `BCRYPT_ROUNDS` says otherwise; Spring Security's `BCryptPasswordEncoder` uses strength 10 when created through the delegating encoder (Section 14.3.1 shows what a step of the cost means). The dangerous places are also the same in all three: raw SQL built by string concatenation (`DB::raw`, `FromSqlRaw`, native queries), templates that print unescaped HTML (`@Html.Raw`, `{!! !!}`, `th:utext`), and CSRF protection switched off wholesale. Knowing the escape hatches of your framework is part of knowing the framework.

15.9 Choosing

The evidence now fits into a short procedure. It is written for Dimas's catering system, but it works for any project.

1. **Write down the constraints that cannot move.** Where must it run (shared PHP hosting, a company's Windows servers, a cloud container platform)? Must it connect to an existing system (a company-wide Java platform, an Active Directory login)? A constraint often decides alone: a client with a Windows Server estate and SQL Server leans to ASP.NET Core; a client who already pays for PHP hosting leans to Laravel; a bank whose other systems are Spring leans to Spring Boot.
2. **Count what the team already knows.** A framework the team knows well is faster and safer than a "better" one it is learning. With one developer, choose what that developer knows best, unless a constraint forbids it.
3. **Check the load.** Estimate the busiest minute with Little's law (Equation (15.2)). If all three are fast enough by a wide margin (the usual case), speed drops out of the decision. If not, measure your own workload as in Section 15.4.2.
4. **Check the calendar.** Pick a version with at least a year of support left, and plan the upgrades.
5. **Build one real feature as a spike.** Two days spent building the order form in the leading candidate reveal more than two weeks of reading.
6. **Write the decision down** with its reasons, so that the next person understands it and knows when to revisit it.

Table 15.6 applies the procedure to four realistic situations.

Table 15.6 Four situations and the framework the procedure suggests.

Situation	Likely choice	Main reason
Catering business, one developer who knows PHP, cheap hosting	Laravel	team knowledge and hosting
Hospital group with Windows servers and SQL Server	ASP.NET Core	existing platform and staff
Cooperative bank adding a service to its Spring-based systems	Spring Boot	fits the existing platform and its security rules
Start-up expecting heavy API traffic, team knows C# and Java	ASP.NET Core or Spring Boot, after a measurement	performance per server, strong typing

For Dimas the procedure is short. The catering business has one developer (him), who has now built the same application in all three frameworks and knows PHP best from his diploma; the family already rents PHP hosting for their website; the busiest minute is lunchtime orders by phone, a few requests per second. Laravel it is, with a note to revisit the decision if the business ever needs a mobile app with heavy traffic.

15.10 Using It for Real

Two habits from this chapter are worth keeping whatever you choose. First, keep the business rules in plain code that does not depend on the framework, as `LaundryRules` does in C# and Java and `PriceCalculator` with `OrderWorkflow` in PHP. The rules are the part that would have to be rewritten in a migration from one framework to another; if they are small, pure and tested, a migration is mostly routine work on the outer layers. Second, describe your API as a contract that is independent of the framework: URLs, fields, status codes, error bodies. Chapter 16 shows that one contract can be implemented three times and checked by one program; the same contract lets

a phone app or a partner's system work with your application no matter what it is built with, and lets you change the framework behind it one day without anyone noticing.

A decision is only useful if it can be found and understood later. Many teams keep a short *architecture decision record* for each important choice, one page in the project's repository with four headings: the context (what the project must do, for whom, where it runs), the options considered, the decision, and its consequences, including what would make the team reconsider. For the catering system Dimas's record would say: context, a twelve-person business ordering by phone, one developer, rented PHP hosting; options, the three frameworks of this book; decision, Laravel 13 on PHP 8.5; consequences, a yearly upgrade in the first quarter, and a review if a mobile app with heavy traffic is ever planned. Writing it takes twenty minutes, and it saves the next developer from repeating the whole discussion, or from undoing a good decision without knowing why it was made.

15.11 Profitina in Practice

Profitina itself was not built with any of the three frameworks of this book: it is a Python application with FastAPI, served by Uvicorn, with PostgreSQL in production. The reasons follow the procedure of Section 15.9. Its core work involves running AI models locally and processing text, where Python's libraries are strongest; the team knew Python well; and FastAPI gives typed request and response models, automatic validation and problem-style errors, the same ideas as the three frameworks here. The decision was written down together with its trade-offs, and the parts that must not depend on any framework (the scoring rules, the subscription rules) are plain functions with their own tests. Choosing well is less about the framework's name than about fitting the team, the work and the place.

15.12 Common Mistakes, and How to Avoid Them

- **Choosing by a benchmark chart.** A top-ranked result is the best entry of a framework on large hardware. Measure your own workload, and first check whether speed matters at all (Equation (15.2)).
- **Quoting numbers without source and date.** "Laravel is used by 9%" is meaningless without "of respondents to the Stack Overflow survey 2025". Always say where and when.
- **Measuring a cold application.** The first requests to .NET, the JVM and PHP are slow while code is compiled and cached. Warm up before measuring, and measure in Release or production mode.
- **Running the load generator on a saturated machine without saying so.** Our measurement shared two cores between server and client; that is fine for showing the method, not for comparing with a data-center benchmark.
- **Ignoring the support calendar.** Starting a project on a version that loses support in three months means an upgrade in the first release cycle.
- **Treating framework differences as the main risk.** The team's knowledge, the hosting and the quality of the code matter more than the framework for almost every business application.
- **Switching frameworks to fix a design problem.** Fat controllers and rules spread across files stay fat and spread in the new framework. Fix the layering first.

15.13 Chapter Summary

- After building the same application three times, the frameworks turn out to map concept for concept (**Table 15.2**); the real design differences are active record against unit of work, and how much is configured by convention.
- TechEmpower Round 23 (February 2025) is the last round of that benchmark, archived in March 2026; its Fortunes results rank ASP.NET Core, Spring and Laravel in that order (**Figure 15.1**), but they show the best tuned entries on very large hardware.
- Our own measurement of an ASP.NET Core endpoint showed throughput leveling off at the machine's limit while latency grew (**Figure 15.2**); Little's law (Equation (15.2)) checked it by hand.
- The same feature needs a similar number of lines in all three frameworks; Laravel is shortest for small features thanks to its conventions (**Table 15.3**).
- In the Stack Overflow Developer Survey 2025, ASP.NET Core was used by 19.7% of respondents, Spring Boot by 14.7% and Laravel by 8.9%; all three are admired by about half their users (**Figure 15.3**).
- Support calendars differ: .NET LTS for three years, Laravel yearly majors with two years of security fixes, Spring Boot minors every six months with about thirteen months of open-source support (**Table 15.4**, **Figure 15.4**).
- Security defaults are strong and similar in all three (**Table 15.5**); the risks are the escape hatches.
- A framework is chosen by constraints, team knowledge, load, calendar and a spike, and the decision is written down (**Table 15.6**).

15.14 Exercises

1. A friend says: "In Round 23 Laravel handled 16,800 requests per second in Fortunes, so my Laravel site on a shared hosting account can handle 16,800 visitors per second." Give three reasons why this does not follow.
2. A load test with 16 workers reports 12,000 requests per second and a p50 latency of 1.9 ms. Use Little's law to compute the average latency. Is the report plausible? What if it had reported a p50 of 3.5 ms?
3. The catering business expects 600 orders between 11:00 and 13:00, each order causing about 8 requests of 40 ms each. How many requests are in progress on average? Does this influence the choice of framework?
4. Map these Laravel pieces of Chapter 16 to their ASP.NET Core and Spring Boot equivalents: (a) `StoreCustomerRequest`; (b) `OrderResource`; (c) `Gate::define('delete-customers', ...)`; (d) the migration files; (e) the `{"data": ...}` wrapper that `JsonResource` adds.
5. Using **Table 15.4**, a team starts a project on October 1, 2026 and plans to release in March 2027 and maintain it until the end of 2028 without major work. For each framework, which version should it start on, and which upgrades must it plan?
6. Measure the PerfLab endpoint on your own computer with 1, 4 and 16 workers, and check each result with Little's law. Then change PerfLab to use `WebApplication.CreateBuilder` and the reflection-based JSON serializer and measure again. What changed, and why might the change be small?

15.15 Solutions to the Exercises

1. (a) The Round 23 number is the best Laravel entry, tuned by experts, not a typical application with authentication, logging and more queries. (b) It ran on a 56-core server with a 40 Gbit/s network; a shared hosting account gets a small share of one server. (c) Requests are not visitors: one page view loads the page and often several other resources, and visitors do not all arrive in the same second. A fourth reason: Fortunes reads one tiny table; a real page's database work usually costs much more.

2. $W = L/\lambda = 16/12000$ s ≈ 1.33 ms. An average of 1.33 ms with a median of 1.9 ms is *not* plausible: with long-tailed latency the average is normally above the median, and here it is well below it. Most likely the test measured throughput and latency over different periods, or some workers were idle. With a p50 of 3.5 ms the average would be far below the median, which is even less plausible. The law does not prove a report right, but it quickly shows when it is wrong.

3. 600 orders × 8 requests = 4,800 requests in 7,200 seconds, so $\lambda \approx 0.67$ requests per second; $L = \lambda W = 0.67 \times 0.04 \approx 0.027$. On average one request is in progress less than 3% of the time. Any of the three frameworks on the smallest server is more than enough; performance does not influence the choice.

4. (a) `CustomerRequest`, a record with data annotations in `Api/Dtos.cs`, in ASP.NET Core; `CreateCustomerRequest`, a record with Bean Validation constraints, in Spring Boot. (b) `OrderView` in both (a record in `Dtos.cs`, and a record with a factory method in the Spring `api` package). (c) `[Authorize(Roles = "OWNER")]` on the delete action in ASP.NET Core; `.requestMatchers(HttpMethod.DELETE, "/api/customers/*").hasRole("OWNER")` and `@PreAuthorize("hasRole('OWNER')")` in Spring Boot. (d) The EF Core model in `LaundryDbContext` (with `EnsureCreated`, or an EF Core migration generated from it) in ASP.NET Core; the entities with `ddl-auto=update`, or a Flyway script, in Spring Boot. (e) The generic record `DataView<T>` in ASP.NET Core and the record `DataView` in Spring Boot, which the controllers return around every single record.

5. .NET: start on .NET 10 LTS, supported until November 2028, which covers the whole period; no major upgrade is required, only monthly patch releases. Spring Boot: start on 4.1 (4.0 ends in December 2026), then plan to move to 4.2 (expected around November 2026) or later before July 2027, and to each following minor version roughly every six months; also keep Java 25 LTS. Laravel: start on Laravel 13 (bug fixes until September 30, 2027, security until March 17, 2028), plan the upgrade to Laravel 14 in 2027 and possibly to 15 before the end of 2028; PHP 8.5 is supported until the end of 2029.

6. The numbers depend on your computer; check each with $W = L/\lambda$ against the reported p50 and p99 as in Section 15.4.3. The standard builder adds more middleware and logging, and reflection-based JSON is slower than source-generated JSON, so throughput usually drops. The change may be small because on a small computer the load generator competes with the server for the processor, and because the work per request (one multiplication and a tiny JSON object) is so small that the fixed cost of HTTP handling dominates.

References

[1] TechEmpower, "Framework Benchmarks Round 23," blog post and results, Feb.–Mar. 2025. Accessed: Sep. 25, 2026. [Online]. Available: https://www.techempower.com/blog/2025/03/17/framework-benchmarks-round-23/

[2] TechEmpower, “Sunsetting the TechEmpower Framework Benchmarks,” GitHub issue #10932, Mar. 24, 2026. [Online]. Available: https://github.com/TechEmpower/FrameworkBenchmarks/issues/10932

[3] T. A. Pham, “Best popular backend frameworks by performance of throughput benchmark comparison and ranking in 2025,” DEV Community, 2025 (compiled from TechEmpower Round 23 Fortunes results). Accessed: Sep. 25, 2026. [Online]. Available: https://dev.to/tuananhpham/popular-backend-frameworks-performance-benchmark-1bkh

[4] Stack Overflow, “2025 Stack Overflow Developer Survey: Technology, Web frameworks and technologies,” 2025. Accessed: Sep. 25, 2026. [Online]. Available: https://survey.stackoverflow.co/2025/technology

[5] endoflife.date, “Microsoft .NET,” release and support dates. Accessed: Sep. 25, 2026. [Online]. Available: https://endoflife.date/dotnet

[6] endoflife.date, “Oracle JDK,” release and support dates. Accessed: Sep. 25, 2026. [Online]. Available: https://endoflife.date/oracle-jdk

[7] endoflife.date, “Spring Boot,” release and support dates. Accessed: Sep. 25, 2026. [Online]. Available: https://endoflife.date/spring-boot

[8] Broadcom, “Spring Support Policy.” Accessed: Sep. 25, 2026. [Online]. Available: https://spring.io/support-policy/

[9] endoflife.date, “PHP,” release and support dates. Accessed: Sep. 25, 2026. [Online]. Available: https://endoflife.date/php

[10] Laravel, “Release Notes: Versioning Scheme and Support Policy,” Laravel 13.x documentation; and endoflife.date, “Laravel.” Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/releases and https://endoflife.date/laravel

[11] .NET Foundation, “PasswordHasher.cs,” ASP.NET Core source code, `src/Identity/Extensions.Core/src`. Accessed: Sep. 25, 2026. [Online]. Available: https://github.com/dotnet/aspnetcore/blob/main/src/Identity/Extensions.Core/src/PasswordHasher.cs

Chapter 16
Evaluation and Discussion IV

This chapter brings the whole book together. Profitina Laundry is finished in all three frameworks: the same four entities (`Customer`, `Service`, `Order`, `Payment`), the same rules, the same JSON API with sign-in and roles, and the same public counter board. Like the earlier evaluation chapters, it is organized as problems, each worked through in the same order: understand the problem, work an example by hand, design on paper, build the solution, and study a tempting wrong solution. By the end of it you will be able to state an API contract precisely enough that a program can check it; verify three independent implementations against that one program; recognize the places where frameworks differ in their defaults (field names, validation answers, refusals) and bend each one to the contract; extend all three with the same feature and compare what it cost; and judge honestly what was run, what was only compiled, and what remains to be done.

16.1 Where We Left Off

Chapters 3 to 6 built the laundry in ASP.NET Core, Chapters 7 to 10 in Laravel, and Chapters 11 to 14 in Spring Boot. Chapter 12 wrote down the API that Raka's phone app talks to, as a contract of sixteen checks, and made the Spring version pass it the same way the Laravel version does. Chapter 14 added sign-in with tokens, the owner's role and tests. Chapter 15 compared the three frameworks from the outside.

Bu Wati now has one last request, and one decision. The request is a screen: a television by the counter in each outlet that shows which orders are in the queue and which are ready to collect, so that customers stop asking "is mine ready yet?". The decision is which of the three versions the outlets will run. Raka does not care which one, as long as his app keeps working. The cleanest way to keep that promise is to make all three versions keep the same contract, and to prove it with one program.

This chapter therefore has five problems.

- **Problem 16.1**, the contract with sign-in, roles and signing out, written as a program that checks any server.
- **Problem 16.2**, the rules of the shop, worked by hand and checked in three languages.
- **Problem 16.3**, the ASP.NET Core capstone, built from scratch on the contract and run for real.
- **Problem 16.4**, the Laravel capstone, completing the Chapter 10 project.
- **Problem 16.5**, the Spring Boot capstone, completing the Chapter 14 project.

The code is in `Chapter16`: `contract` for the checker, `aspnet`, `laravel` and `spring` for the three applications, and `runs` for the output of the runs made while writing this book.

16.2 How to Approach the Capstone

A capstone is where many small decisions meet, and the order of work matters more than in any single chapter. **Figure 16.1** shows the order used here: the contract first, then the rules, then the applications, and the same checker at the end of each.

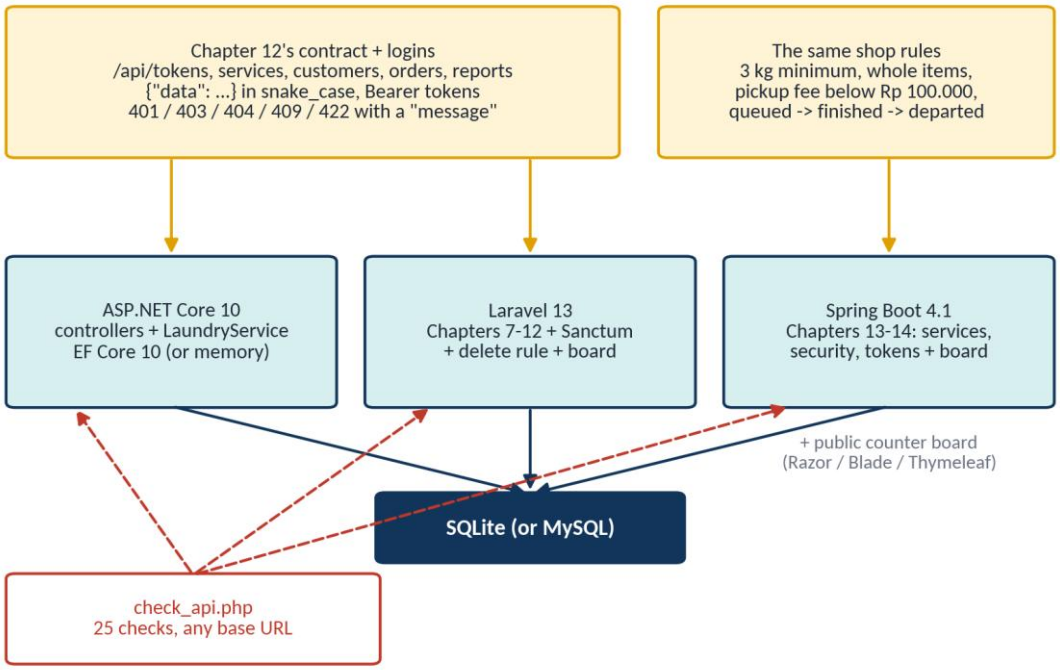


Figure 16.1 One contract, three implementations: the checker `check_api.php` sends the same 25 requests to each server and compares the answers with the contract.

Three habits carry the whole chapter.

Write the contract as a program. A contract in prose ("a bad order is refused") can be read in several ways; a check ("`POST /api/orders` with quantity 0 answers 422, and `errors` has the keys `customer_id`, `service_id`, `quantity` and `delivery_address`") cannot. A program also runs in a second and can be repeated after every change.

Keep the rules framework-free. The pricing and workflow rules are small, pure functions in each language (`LaundryRules` in C# and Java, `PriceCalculator` and `OrderWorkflow` in PHP). They are checked on their own, without a web server, before any application uses them.

Say what was run. The machine on which this book was written could build and run .NET applications that need only the shared framework, and it had PHP and a JDK, but it could not download packages from NuGet, Composer or Maven Central. Everything that could be run was run and its real output is shown; everything else was checked as far as that machine allowed, and each section says exactly how far.

So that such statements can be compared, the chapter uses four levels of evidence, from the weakest to the strongest. *Syntax-checked*: the file can be parsed by its language (`php -l`, the Java parser), so no typing mistake stops the program, but nothing is yet known about names and types. *Compiled*: all names, types and imports fit together, here partly against small stand-ins for libraries that could not be downloaded. *Run*: the program really ran and its output can be read, like the rules check and the ASP.NET Core capstone. *Checked against the contract*: a running server answered the checker's 25 requests correctly. Only the last level proves that Raka's app will work; the levels below it are steps towards it. Whenever the chapter states a result, it places it on one of these levels, and Section 16.8 summarizes them for all three frameworks. The habit is useful far beyond this book: a developer's report that something is "done" always deserves the question "to which level?"

16.3 Problem 16.1: The Contract

16.3.1 The problem

Write a program that checks whether a server keeps the Profitina Laundry API contract: the sixteen checks of Chapter 12 as a signed-in cashier, plus the doors of Chapter 14 (a wrong password, a missing token, the cashier's and the owner's rights, and signing out). The program must work against any of the three servers, use nothing but plain PHP, and leave behind data that does not break the next run.

16.3.2 Understanding it by hand

Before writing the program, walk through one session with Raka's app on paper. The cashier signs in and receives a token. She registers a new customer, Sri Rahayu, typing the phone as "0812-5566-7788"; the server keeps `081255667788`. She takes an order: 2.5 kg of wash, dry and fold at Rp 8.000, picked up from Jl. Kenanga 5. The rule of Chapter 8 gives

$$T = \left\lfloor \frac{8000 \cdot \max(250,300) + 50}{100} \right\rfloor + 10000 = 24000 + 10000 = 34000 \tag{16.1}$$

because the 3 kg minimum applies and the subtotal is below Rp 100.000, so the pickup fee is charged. The order is `queued` with a balance of Rp 34.000. Asking for `departed` now must fail, because a step would be skipped; `finished` succeeds and stamps `finished_at`; `departed` fails again, because nothing is paid. A payment of Rp 40.000 is refused ("Only Rp 34.000 is still owed."), one of Rp 34.000 by QRIS is accepted, and now `departed` succeeds. Any further payment is refused with 409, because the order has left the shop. If the payments on an order sum to P, the balance is

$$B = T - P, \qquad 0 \le B \le T \tag{16.2}$$

and every check in this session is a statement about T, B or the status. The cashier then tries to delete Sri's record and is refused with 403; the owner signs in and is refused with 409, because Sri has an order; a customer without orders she can delete. Finally the cashier signs out, and her token stops working.

That session contains every check. Written down, it is also the order in which the program runs them, because each step needs what the previous one created.

16.3.3 The contract

Table 16.1 summarizes the contract. The details, field by field, are those of Chapter 12; Chapter 14 explains the reasons behind them.

Table 16.1 The Profitina Laundry API contract after Chapter 16.

Part	Rule
Sign-in	`POST /api/tokens` with `email`, `password`, `device`: 201 `{"token", "user": {"name", "role"}}`; a wrong password is 422 on `email`
Doors	no token: 401 `{"message": "Unauthenticated."}`; a role that may not: 403 `{"message": "This action is unauthorized."}`
Data	JSON fields in snake_case; one record as `{"data": {...}}`; a page as `{"data": [...], "meta": {"current_page", "per_page", "total", "last_page"}}`
Values	statuses and payment methods in lower case; money in whole rupiah; times as `YYYY-MM-DDTHH:MM:SS` in the shop's time
Bad input	422 `{"message": "The given data was invalid.", "errors": {"field": ["..."]}}`, with every error at once
Not found, closed	404 `{"message": "No order 7."}`; 409 `{"message": "..."}` for a request the record's state forbids
Roles	only the owner deletes customers; a customer with orders cannot be deleted (409)

The demo accounts are part of the contract only through the command line: the Spring Boot and ASP.NET Core versions create `kasir1@profitina-laundry.example` and `owner@profitina-laundry.example`, while Chapter 10's Laravel project seeds `cashier@laundry.test` and `owner@laundry.test`. The checker therefore takes the four values as arguments, with the Spring and ASP.NET accounts as defaults.

16.3.4 The checker

Listing 16.1 is the whole program.

Listing 16.1 `check_api.php`: 25 checks of the contract, for any of the three servers.

```php
<?php

// The capstone contract as a program: Chapter 12's 16 checks, now with
// logins, roles and signing out. Plain PHP, no packages:
//   php check_api.php <base-url> [cashier-email cashier-password
//                                 owner-email owner-password]

$base = rtrim($argv[1] ?? 'http://127.0.0.1:8080', '/');
[$cashierEmail, $cashierPassword, $ownerEmail, $ownerPassword] = [
    $argv[2] ?? 'kasir1@profitina-laundry.example', $argv[3] ?? 'Kasir#2026',
    $argv[4] ?? 'owner@profitina-laundry.example', $argv[5] ?? 'Owner#2026'];
$token = null;
$passed = $failed = 0;

// Sends one request with the current token; returns [status, JSON body].
function call(string $method, string $path, ?array $body = null): array
{
    global $base, $token;
    $headers = ['Accept: application/json',
                'Content-Type: application/json'];
    if ($token !== null) {
        $headers[] = "Authorization: Bearer $token";
    }
    $context = stream_context_create(['http' => [
        'method' => $method,
        'header' => implode("\r\n", $headers),
        'content' => $body === null ? '' : json_encode($body),
        'ignore_errors' => true,          // give us the 4xx bodies too
    ]]);
    $raw = @file_get_contents($base . $path, false, $context);
    if ($raw === false) {
        fwrite(STDERR, "Cannot reach $base$path\n");
        exit(2);
    }
    preg_match('/^HTTP\/\S+ (\d{3})/', $http_response_header[0], $m);
    return [(int) $m[1], json_decode($raw, true)];
}

function check(string $what, bool $ok): void
{
    global $passed, $failed;
    $ok ? $passed++ : $failed++;
    printf("%s  %s\n", $ok ? 'PASS' : 'FAIL', $what);
}
```

Listing 16.1 (Continued) *check_api.php: 25 checks of the contract, for any of the three servers.*

```
function login(string $email, string $password): array
{
    global $token;
    $token = null;
    return call('POST', '/api/tokens', ['email' => $email,
        'password' => $password, 'device' => 'contract check']);
}

// ---- logging in ------------------------------------------------------
[$s, $j] = login($cashierEmail, $cashierPassword . 'x');
check('a wrong password -> 422, errors.email',
    $s === 422 && isset($j['errors']['email']));
[$s, $j] = call('GET', '/api/services');
check('no token -> 401 with a message',
    $s === 401 && is_string($j['message'] ?? null));
[$s, $j] = login($cashierEmail, $cashierPassword);
check('POST /api/tokens -> 201, a token for a cashier',
    $s === 201 && is_string($j['token'] ?? null)
    && ($j['user']['role'] ?? '') === 'cashier');
$token = $cashierToken = $j['token'] ?? '';

// ---- Chapter 12's checks, as a logged-in cashier ----------------------
$time = '/^\d{4}-\d\d-\d\dT\d\d:\d\d:\d\d$/';

[$s, $j] = call('GET', '/api/services');
$first = $j['data'][0] ?? [];
check('GET /api/services -> 200, S0001 first, 8000 per kg',
    $s === 200 && ($first['id'] ?? '') === 'S0001'
    && ($first['unit'] ?? '') === 'kg' && ($first['price'] ?? 0) === 8000);

$phone = '0812' . random_int(10000000, 99999999);   // new on every run
[$s, $j] = call('POST', '/api/customers', ['name' => 'Sri Rahayu',
    'phone' => substr($phone, 0, 4) . '-' . substr($phone, 4)]);
$customer = $j['data'] ?? [];
check('POST /api/customers -> 201, phone kept as digits only',
    $s === 201 && ($customer['phone'] ?? '') === $phone);
[$s, $j] = call('POST', '/api/customers', ['name' => 'Sri Rahayu',
    'phone' => $phone]);
check('the same phone again -> 422, errors.phone',
    $s === 422 && isset($j['errors']['phone']));
```

Listing 16.1 (Continued) *check_api.php: 25 checks of the contract, for any of the three servers.*

```
[$s, $j] = call('POST', '/api/orders', [
    'customer_id' => $customer['id'] ?? 0, 'outlet_id' => 'O01',
    'service_id' => 'S0001', 'quantity' => 2.5, 'pickup' => true,
    'delivery_address' => 'Jl. Kenanga 5, Surabaya']);
$o = $j['data'] ?? [];
check('POST /api/orders -> 201, total 34000 (3 kg + pickup), queued',
    $s === 201 && ($o['total'] ?? 0) === 34000
    && ($o['balance'] ?? 0) === 34000 && ($o['status'] ?? '') === 'queued');
check('times are YYYY-MM-DDTHH:MM:SS, finished_at is null',
    preg_match($time, $o['received_at'] ?? '') === 1
    && array_key_exists('finished_at', $o) && $o['finished_at'] === null);
$url = '/api/orders/' . ($o['id'] ?? 0);

[$s, $j] = call('POST', '/api/orders', ['quantity' => 0, 'pickup' => true]);
$fields = array_keys($j['errors'] ?? []);
check('a bad order -> 422, all of its errors at once',
    $s === 422 && is_string($j['message'] ?? null)
    && ! array_diff(['customer_id', 'service_id', 'quantity',
        'delivery_address'], $fields));

$move = fn (string $to) => call('PATCH', "$url/status", ['status' => $to]);
[$s, $j] = $move('departed');
check('queued -> departed skips a step: 422, errors.status',
    $s === 422 && isset($j['errors']['status']));
[$s, $j] = $move('finished');
check('queued -> finished: 200, finished_at set',
    $s === 200 && preg_match($time, $j['data']['finished_at'] ?? '') === 1);
[$s, $j] = $move('departed');
check('finished -> departed before payment: 422, errors.status',
    $s === 422 && isset($j['errors']['status']));

[$s, $j] = call('POST', "$url/payments", ['amount' => 40000,
    'method' => 'cash']);
check('paying more than the balance -> 422, errors.amount',
    $s === 422 && isset($j['errors']['amount']));
[$s, $j] = call('POST', "$url/payments", ['amount' => 34000,
    'method' => 'qris']);
check('paying the balance -> 201, balance 0, payments[0].method qris',
    $s === 201 && ($j['data']['balance'] ?? -1) === 0
    && ($j['data']['payments'][0]['method'] ?? '') === 'qris');
```

Listing 16.1 (Continued) *check_api.php: 25 checks of the contract, for any of the three servers.*

```
[$s, $j] = $move('departed');
check('finished -> departed after payment: 200',
    $s === 200 && ($j['data']['status'] ?? '') === 'departed');
[$s, $j] = call('POST', "$url/payments", ['amount' => 1000,
    'method' => 'cash']);
check('a departed order is closed: 409 with a message',
    $s === 409 && is_string($j['message'] ?? null));

[$s, $j] = call('GET', '/api/orders?status=departed');
check('GET /api/orders?status=departed -> 200, data[] and meta',
    $s === 200 && array_is_list($j['data'] ?? null)
    && ($j['meta']['total'] ?? 0) >= 1
    && ($j['meta']['current_page'] ?? 0) === 1);
[$s, $j] = call('GET', '/api/orders/999999');
check('GET /api/orders/999999 -> 404 with a message',
    $s === 404 && is_string($j['message'] ?? null));

[$s, $j] = call('GET', '/api/reports/daily?date=' . date('Y-m-d'));
$r = $j['data'] ?? [];
check('GET /api/reports/daily -> 200, the counts add up',
    $s === 200 && ($r['orders_received'] ?? 0) >= 1
    && ($r['collected'] ?? 0) >= 34000
    && array_sum($r['by_status'] ?? []) === $r['orders_received']);

// ---- roles: only the owner deletes customers ----------------------------
$gone = '/api/customers/' . ($customer['id'] ?? 0);
[$s, $j] = call('DELETE', $gone);
check('a cashier deletes a customer -> 403 with a message',
    $s === 403 && is_string($j['message'] ?? null));
[$s, $j] = login($ownerEmail, $ownerPassword);
$token = $j['token'] ?? '';
check('the owner logs in -> 201, role owner',
    $s === 201 && ($j['user']['role'] ?? '') === 'owner');
[$s, $j] = call('DELETE', $gone);
check('the owner deletes a customer with orders -> 409',
    $s === 409 && is_string($j['message'] ?? null));
[$s, $j] = call('POST', '/api/customers', ['name' => 'Temporary',
    'phone' => '0813' . random_int(10000000, 99999999)]);
[$s] = call('DELETE', '/api/customers/' . ($j['data']['id'] ?? 0));
check('the owner deletes a customer without orders -> 204', $s === 204);
```

Listing 16.1 (Continued) *check_api.php: 25 checks of the contract, for any of the three servers.*

```
// ---- signing out --------------------------------------------------------
$token = $cashierToken;
[$s] = call('DELETE', '/api/tokens/current');
check('DELETE /api/tokens/current -> 204', $s === 204);
[$s] = call('GET', '/api/services');
check('the old token no longer works -> 401', $s === 401);

printf("\n%d passed, %d failed\n", $passed, $failed);
exit($failed === 0 ? 0 : 1);
```

`call` sends one request with PHP's built-in stream functions and HTTP context options [1], so the program needs no Composer package and runs with any PHP 8.1 or newer. The option `ignore_errors` makes PHP return the body of a 4xx answer instead of failing, and the status code comes from the first line of `$http_response_header`. Each `check` prints `PASS` or `FAIL` with a sentence that says what was expected, so a failure reads as a bug report. The exit code is 0 only if everything passed, so the checker can run automatically after every deployment.

Three details make the program repeatable. The new customer's phone number is random, so a second run does not trip over "The phone has already been taken." from the first. The customer that the owner deletes successfully is created for that purpose in the same run. And the order request also sends `outlet_id`, which the Laravel project of Chapter 10 requires and the other two ignore; an extra field is harmless, a missing one is not.

16.3.5 Reading a failing run

A checker earns its keep on the day it fails, so it is worth knowing how to read a failure. Each line names one expectation, and the lines run in the order of the session in Section 16.3.2, so the *first* `FAIL` is usually the one to study; later failures often follow from it. If the sign-in check fails, every later check runs without a token and most of them fail with 401: one cause, twenty symptoms. If `POST /api/orders` fails, the checks of the status steps and payments have no order to work on. Fix the first failure, run the checker again, and repeat.

Two kinds of first failure are common. A check that expects a field fails because the field has another name or shape: `customerId` instead of `customer_id`, `"QUEUED"` instead of `"queued"`, a date with fractions of a second. The fix belongs in the server's JSON settings or views, never in the checker. A check that expects a status fails because the server answered with a different one: 400 instead of 422, 500 instead of 404. Then the answer's body usually says why, and the quickest way to see it is to send the same request by hand with `curl -i`, which prints the status line, the headers and the body. The first run of the ASP.NET Core capstone, described in Section 16.5.8, followed exactly this path.

16.3.6 A tempting wrong solution

The tempting way to compare servers is to save the full JSON answer of one and compare the others with it, character by character. It fails at once, and for the wrong reasons: identifiers, times and the random phone differ on every run; one framework writes `4.5` where another writes `4.50`; the order of fields in an object is not meaningful in JSON. The opposite temptation is to check status codes only. Then a server that answers 422 with an empty body, or with `errors` keyed `customerId`, passes, and Raka's app shows no messages. The checker takes the middle way: it tests each field the app actually reads, by name and value, and nothing it does not.

16.4 Problem 16.2: The Rules in Three Languages

16.4.1 The problem

The price and the workflow must come out identical in C#, PHP and Java, to the rupiah, for every service in the price list, including the awkward cases: quantities with two decimals, the kilogram minimum, items rounded up, and the pickup fee at the boundary.

16.4.2 Understanding it by hand

Table 16.2 works nine cases by hand with Equation (16.1). The quantity is written in hundredths h; kilogram services bill $\max(h, 300)$, item services round h up to whole hundreds.

Table 16.2 Nine prices worked by hand.

Service, price	Quantity	Billed	Subtotal	Pickup	Total
Wash, dry & fold, 8.000	4.50 kg	450	36.000	no	36.000
Wash, dry & fold, 8.000	2.00 kg	300	24.000	no	24.000
Wash, dry & fold, 8.000	4.02 kg	402	32.160	no	32.160
Ironing only, 6.000	3.35 kg	335	20.100	no	20.100
Curtain, 15.000	4.40 kg	440	66.000	yes	76.000
Bed cover, 35.000	2 items	200	70.000	yes	80.000
Suit, 65.000	2 items	200	130.000	yes	130.000
Shirt, 20.000	5 items	500	100.000	yes	100.000
Shoes, 45.000	1.5 pairs	200	90.000	no	90.000

The last three rows are the boundaries. A suit order of Rp 130.000 is above Rp 100.000, so pickup is free. Five shirts come to exactly Rp 100.000, which is *not* below Rp 100.000, so pickup is free too; a program that wrote `<=` instead of `<` would charge Rp 10.000 here. Shoes are counted in whole pairs, so 1.5 pairs bill as 2.

16.4.3 The solutions

The three implementations have the same shape: turn the quantity into whole hundredths, compute in integers, and round half up with `+50` before dividing by 100. There is no floating-point number anywhere. The C# version is in the capstone (**Listing 16.3**); the PHP version is Chapter 8's `PriceCalculator`, and the Java version is Chapter 11's `LaundryRules`. Chapter 11's check program, copied into `Chapter16/spring/check`, prints the nine cases of **Table 16.2** and the workflow; it ran with the JDK alone on the machine where the book was written:

```
Prices (Problem 8.1)
  Wash, dry & fold              4.50 kg            36000
  Wash, dry & fold              2.00 kg            24000
  Wash, dry & fold              4.02 kg            32160
  Ironing only                  3.35 kg            20100
  Curtain                       4.40 kg   pickup   76000
  Bed cover, any size           2.00 item pickup   80000
  Dry cleaning - suit 2 pieces  2.00 item pickup  130000
  Dry cleaning - shirt/blouse   5.00 item pickup  100000
  Shoes deep clean              1.50 pair          90000
Workflow (Problem 8.2)
  Queued   owing  25000 -> Finished
  Finished owing  25000 -> Collect Rp 25.000 before handing over the laundry.
  Finished owing      0 -> Departed
  Departed owing      0 -> This order has already left the shop.
```

Every total matches **Table 16.2**. The C# rules produced the same numbers inside the running ASP.NET Core capstone (Section 16.5.6), where the contract's order came out at Rp 34.000 and the order of Sari Wulandari at Rp 36.000.

16.4.4 A tempting wrong solution

The tempting solution is `Math.round(price * Math.max(quantity, 3))` with `double`, or `round($price * max($qty, 3))` in PHP. It is shorter, and it is right for most inputs. It fails where binary fractions cannot represent a decimal exactly: 4.02 is stored as 4.0199999999999996..., and `8000 * 4.02` gives 32159.999999999996, which the float rounding happens to save here but which a truncating cast, a different rounding mode, or a slightly different price will not. Worse, the three languages round `.5` differently by default (C#'s `Math.Round` rounds half to even), so the

three servers would disagree by one rupiah in rare cases, the hardest kind of bug to find. Binary floating point, as standardized in IEEE 754 [2], is exact for integers but not for most decimal fractions; whole hundredths and integer arithmetic remove the whole class of problems.

16.5 Problem 16.3: The ASP.NET Core Capstone

16.5.1 The problem

Build the laundry API in ASP.NET Core 10 so that it passes all 25 checks, with bearer tokens in the Sanctum style, the owner's right to delete customers, the rules of Problem 16.2 and the public board. Store the data in SQLite through EF Core. Because the machine where this book was written cannot download EF Core, the application must also build without it, so that it can be run and checked for real.

16.5.2 Designing it on paper

Figure 16.2 is the data model, the same in all three frameworks apart from where the accounts live.

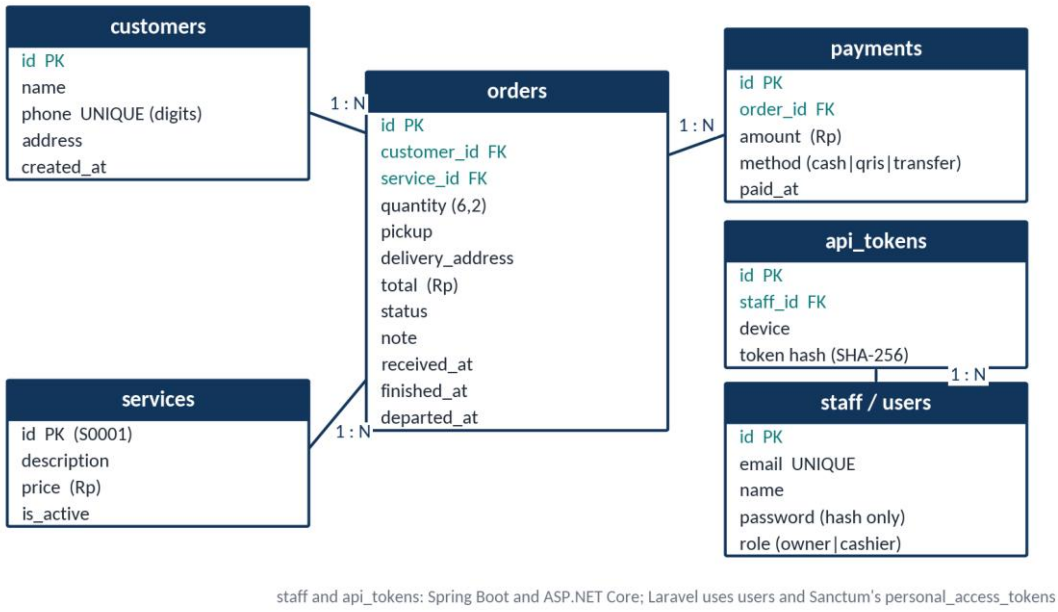


Figure 16.2 The capstone's tables: services from the price list (`S0001` to `S0012`), customers, orders with pickup, payments, staff and API tokens.

The layers are the ones of Chapter 13, in C# names. Controllers in `Api` read JSON and write JSON. `LaundryService` in `Application` applies every rule. Storage sits behind an interface, `ILaundryStore`, with two implementations: `EfLaundryStore` with EF Core and SQLite, and `MemoryLaundryStore` with lists in memory. A switch in the project file chooses one at build time (**Listing 16.2**).

Listing 16.2 `LaundryCapstone.csproj`: the `UseEf` switch decides whether EF Core is compiled in.

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

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <!-- true: SQLite through EF Core (the normal build).
         false: an in-memory store that needs no NuGet package. -->
    <UseEf Condition="'$(UseEf)' == ''">true</UseEf>
    <DefineConstants Condition="'$(UseEf)' == 'true'"
      >$(DefineConstants);USE_EF</DefineConstants>
  </PropertyGroup>

  <ItemGroup Condition="'$(UseEf)' == 'true'">
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite"
                      Version="10.0.*" />
  </ItemGroup>

  <ItemGroup Condition="'$(UseEf)' != 'true'">
    <Compile Remove="Data/Ef/**" />
  </ItemGroup>

  <ItemGroup>
    <Content Include="data/Service.csv" CopyToOutputDirectory="PreserveNewest" />
  </ItemGroup>

</Project>
```

With the default `UseEf=true`, the project references `Microsoft.EntityFrameworkCore.Sqlite` and defines the symbol `USE_EF`; with `-p:UseEf=false` it references no package at all, removes the `Data/Ef` folder from the build, and needs nothing but the .NET SDK. Everything else in the application is identical in both builds.

16.5.3 The domain and the storage

Listing 16.3 holds the rules of Problem 16.2 and the workflow.

Listing 16.3 `LaundryRules.cs`: price and workflow in integer arithmetic, as in Chapters 8 and 11.

```
using System.Globalization;

namespace LaundryCapstone.Domain;

/// <summary>The shop's rules: the same arithmetic as Chapters 8 and 11.</summary>
public static class LaundryRules
{
    public const long MinimumKg = 3;
    public const long PickupFee = 10_000;
    public const long FreePickupFrom = 100_000;

    /// <summary>4.5 -> 450: the quantity in hundredths, exact.</summary>
    public static long Hundredths(decimal quantity) =>
        decimal.ToInt64(quantity * 100m);

    /// <summary>At least 3 kg for per-kg services; whole items otherwise.</summary>
    public static long Total(bool perKg, long price, decimal quantity,
                             bool pickup)
    {
        long h = Hundredths(quantity);
        long billed = perKg ? Math.Max(h, MinimumKg * 100)
                            : (h + 99) / 100 * 100;
        long subtotal = (price * billed + 50) / 100;         // half up
        long fee = pickup && subtotal < FreePickupFrom ? PickupFee : 0;
        return subtotal + fee;
    }

    /// <summary>The next status, or an exception that says why not.</summary>
    public static OrderStatus Advance(OrderStatus status, long balance)
    {
        OrderStatus next = status switch
        {
            OrderStatus.Queued => OrderStatus.Finished,
            OrderStatus.Finished => OrderStatus.Departed,
            _ => throw new InvalidOperationException(
                     "This order has already left the shop.")
        };
        if (next == OrderStatus.Departed && balance > 0)
            throw new InvalidOperationException(
                $"Collect {Rupiah(balance)} before handing over the laundry.");
        return next;
    }
```

Listing 16.3 (Continued) *LaundryRules.cs: price and workflow in integer arithmetic, as in Chapters 8 and 11.*

```
    /// <summary>36000 -> "Rp 36.000".</summary>
    public static string Rupiah(long amount) =>
        "Rp " + amount.ToString("#,0", CultureInfo.InvariantCulture)
                      .Replace(',', '.');

    /// <summary>Queued -> "queued": the word stored and sent as JSON.</summary>
    public static string Value(this OrderStatus s) =>
        s.ToString().ToLowerInvariant();

    public static string Value(this PaymentMethod m) =>
        m.ToString().ToLowerInvariant();
}
```

The entities are plain classes in `Domain/Entities.cs`. `Service` keeps the columns of the price list (`Id` such as `S0001`, `Description`, `Price`, `Active`) and derives `Name`, `PerKg` and `Unit` from the description, exactly as Chapter 11's Java class does, so "Wash, dry & fold (per kg)" becomes the name "Wash, dry & fold" and the unit `kg`. `Order` computes `Paid` and `Balance` from its payments. **Listing 16.4** is the interface the service uses.

Listing 16.4 `ILaundryStore.cs`: everything the application needs from storage.

```
using LaundryCapstone.Domain;

namespace LaundryCapstone.Data;

/// <summary>
/// Everything the application needs from storage. EfLaundryStore keeps it
/// in SQLite; MemoryLaundryStore keeps it in lists (a build without NuGet).
/// </summary>
public interface ILaundryStore
{
    Task<List<Service>> ActiveServicesAsync();
    Task<Service?> FindServiceAsync(string id);
    Task<List<Customer>> CustomersAsync();
    Task<Customer?> FindCustomerAsync(long id);
    Task<Customer?> FindCustomerByPhoneAsync(string phone);
    Task<bool> CustomerHasOrdersAsync(long customerId);
    Task<(List<Order> Items, int Total)> OrderPageAsync(
        OrderStatus? status, int page, int size);
    Task<Order?> FindOrderAsync(long id);        // with customer, service,
                                                 // and payments
    Task<Dictionary<OrderStatus, long>> CountByStatusAsync(
        DateTime from, DateTime to);
    Task<long> BilledAsync(DateTime from, DateTime to);
    Task<long> CollectedAsync(DateTime from, DateTime to);
    Task<Staff?> FindStaffAsync(string email);
    Task<bool> AnyServiceAsync();
    Task<ApiToken?> FindTokenAsync(long id);
    void Add(object entity);
    void Remove(object entity);
    Task SaveChangesAsync();
    Task EnsureCreatedAsync();
}
```

The interface speaks in the application's terms (`CustomerHasOrdersAsync`, `BilledAsync`), not in terms of tables or queries. `EfLaundryStore` implements each method with a LINQ query over `LaundryDbContext`; `MemoryLaundryStore` with the same query over lists. `Add`, `Remove` and `SaveChangesAsync` mirror EF Core's unit of work: in the EF build they are EF Core's own methods, in the memory build `SaveChangesAsync` hands out identifiers to new rows. The memory store is not a toy: it is what made the capstone runnable on a machine without packages, and it is a convenient stand-in for fast tests. But it also carries a risk that the chapter must name. Two implementations of one interface can disagree, for example about the order in which a query returns rows or about whether a deleted customer's orders still point at it, and a checker that passed against one of them says nothing certain about the other. That is why Section 16.5.6 states exactly which build was checked.

16.5.4 The service

Listing 16.5 is the heart of the capstone.

Listing 16.5 `LaundryService.cs`: customers, orders, status steps, payments and the daily figures, with every rule in one place.

```
using LaundryCapstone.Data;
using LaundryCapstone.Domain;

namespace LaundryCapstone.Application;

/// <summary>What a new order needs, from a form or from JSON alike.</summary>
public sealed record NewOrder(long? CustomerId, string? ServiceId,
    decimal? Quantity, bool Pickup, string? DeliveryAddress, string? Note);

/// <summary>Orders per status, money billed and collected on one day.</summary>
public sealed record DayFigures(DateOnly Day,
    Dictionary<OrderStatus, long> ByStatus, long Billed, long Collected)
{
    public long Received => ByStatus.Values.Sum();
}

/// <summary>The service layer: every rule of the shop is applied here.</summary>
public sealed class LaundryService(ILaundryStore store, TimeProvider clock)
{
    // Indonesia (WIB) is UTC+7 all year: no daylight saving time.
    private static readonly TimeSpan Wib = TimeSpan.FromHours(7);

    private DateTime Now()
    {
        var t = clock.GetUtcNow().ToOffset(Wib).DateTime;
        return t.AddTicks(-(t.Ticks % TimeSpan.TicksPerSecond));
    }

    public Task<List<Service>> ActiveServicesAsync() =>
        store.ActiveServicesAsync();

    public async Task<Service> ServiceAsync(string id) =>
        await store.FindServiceAsync(id)
        ?? throw new NotFoundException("service", id);

    public async Task<List<Customer>> CustomersAsync(string? phone)
    {
        if (phone is null) return await store.CustomersAsync();
        var found = await store.FindCustomerByPhoneAsync(Digits(phone));
        return found is null ? [] : [found];
    }
```

Listing 16.5 (Continued) *LaundryService.cs: customers, orders, status steps, payments and the daily figures, with every rule in one place.*

```
    public async Task<Customer> CreateCustomerAsync(string name, string phone,
                                                    string? address)
    {
        if (await store.FindCustomerByPhoneAsync(phone) is not null)
            throw new RuleException("phone",
                "The phone has already been taken.");
        var customer = new Customer
        {
            Name = name.Trim(), Phone = phone,
            Address = string.IsNullOrWhiteSpace(address) ? null : address,
            CreatedAt = Now()
        };
        store.Add(customer);
        await store.SaveChangesAsync();
        return customer;
    }

    public async Task DeleteCustomerAsync(long id)
    {
        var customer = await store.FindCustomerAsync(id)
                       ?? throw new NotFoundException("customer", id);
        if (await store.CustomerHasOrdersAsync(id))
            throw new ConflictException(
                $"{customer.Name} has orders and cannot be deleted.");
        store.Remove(customer);
        await store.SaveChangesAsync();
    }

    public Task<(List<Order> Items, int Total)> OrderPageAsync(
        OrderStatus? status, int page, int size) =>
        store.OrderPageAsync(status, Math.Max(page, 1), size);

    public async Task<Order> OrderAsync(long id) =>
        await store.FindOrderAsync(id)
        ?? throw new NotFoundException("order", id);
```

Listing 16.5 (Continued) *LaundryService.cs: customers, orders, status steps, payments and the daily figures, with every rule in one place.*

```
    /// <summary>Rules that need the database; empty means all is well.</summary>
    public async Task<Dictionary<string, List<string>>> CheckAsync(NewOrder o)
    {
        var errors = new Dictionary<string, List<string>>();
        if (o.CustomerId is long c && await store.FindCustomerAsync(c) is null)
            Add(errors, "customer_id", "The selected customer is invalid.");
        if (!string.IsNullOrWhiteSpace(o.ServiceId)
            && (await store.FindServiceAsync(o.ServiceId))?.Active != true)
            Add(errors, "service_id", "The selected service is invalid.");
        if (o.Pickup && string.IsNullOrWhiteSpace(o.DeliveryAddress))
            Add(errors, "delivery_address",
                "Where should the laundry be delivered?");
        return errors;
    }

    public async Task<Order> CreateOrderAsync(NewOrder o)
    {
        var errors = await CheckAsync(o);
        if (errors.Count > 0) throw new RuleException(errors);
        var service = (await store.FindServiceAsync(o.ServiceId!))!;
        var order = new Order
        {
            CustomerId = o.CustomerId!.Value, ServiceId = service.Id,
            Quantity = o.Quantity!.Value, Pickup = o.Pickup,
            DeliveryAddress = o.Pickup ? o.DeliveryAddress : null,
            Note = o.Note, ReceivedAt = Now(),
            Total = LaundryRules.Total(service.PerKg, service.Price,
                                       o.Quantity.Value, o.Pickup)
        };
        store.Add(order);
        await store.SaveChangesAsync();
        return await OrderAsync(order.Id);
    }

    /// <summary>One step; with a wanted status it must be the next one.</summary>
    public async Task<Order> AdvanceAsync(long id, OrderStatus? wanted)
    {
        var order = await OrderAsync(id);
        EnsureOpen(order);
        OrderStatus next;
        try { next = LaundryRules.Advance(order.Status, order.Balance); }
        catch (InvalidOperationException e)
        {
            throw new RuleException("status", e.Message);
        }
        if (wanted is not null && wanted != next)
```

Listing 16.5 (Continued) *LaundryService.cs: customers, orders, status steps, payments and the daily figures, with every rule in one place.*

```
            throw new RuleException("status", $"An order that is "
                + $"{order.Status.Value()} can only become {next.Value()}.");
        if (next == OrderStatus.Finished) order.FinishedAt = Now();
        else order.DepartedAt = Now();
        order.Status = next;
        await store.SaveChangesAsync();
        return order;
    }

    public async Task<Order> PayAsync(long id, long amount, PaymentMethod method)
    {
        var order = await OrderAsync(id);
        EnsureOpen(order);
        if (amount > order.Balance)
            throw new RuleException("amount",
                $"Only {LaundryRules.Rupiah(order.Balance)} is still owed.");
        var payment = new Payment
        {
            OrderId = id, Amount = amount, Method = method, PaidAt = Now()
        };
        order.Payments.Add(payment);
        store.Add(payment);
        await store.SaveChangesAsync();
        return order;
    }

    public async Task<DayFigures> DailyAsync(DateOnly day)
    {
        var from = day.ToDateTime(TimeOnly.MinValue);
        var to = from.AddDays(1);                  // [from, to): one whole day
        var counts = await store.CountByStatusAsync(from, to);
        var byStatus = Enum.GetValues<OrderStatus>()
            .ToDictionary(s => s, s => counts.GetValueOrDefault(s));
        return new DayFigures(day, byStatus, await store.BilledAsync(from, to),
                              await store.CollectedAsync(from, to));
    }

    /// <summary>Laundry that has left takes no more steps or money.</summary>
    private static void EnsureOpen(Order order)
    {
        if (order.Status == OrderStatus.Departed)
            throw new ConflictException(
                $"Order #{order.Id} has left the shop and is closed.");
    }
```

Listing 16.5 (Continued) *LaundryService.cs: customers, orders, status steps, payments and the daily figures, with every rule in one place.*

```
    public static string Digits(string s) => new(s.Where(char.IsAsciiDigit)
                                                  .ToArray());

    private static void Add(Dictionary<string, List<string>> errors,
                            string field, string message)
    {
        if (!errors.TryGetValue(field, out var list)) errors[field] = list = [];
        list.Add(message);
    }
}
```

Put it next to Chapter 13's `OrderService` and the correspondence is line by line: `NewOrder`, `CheckAsync` that returns every database rule as a dictionary, `CreateOrderAsync` that checks again, `AdvanceAsync` that refuses a closed order with a `ConflictException` and a wrong step with a

`RuleException`, `PayAsync` that refuses overpayment. The three exceptions are the same as in Spring (**Listing 16.6**).

Listing 16.6 `Exceptions.cs`: a broken rule, a missing record, a conflicting state.

```
namespace LaundryCapstone.Application;

/// <summary>Input that breaks a rule, field by field: 422.</summary>
public sealed class RuleException(Dictionary<string, List<string>> errors)
    : Exception("The given data was invalid.")
{
    public Dictionary<string, List<string>> Errors { get; } = errors;

    public RuleException(string field, string message)
        : this(new() { [field] = [message] }) { }
}

/// <summary>Asked for a record that does not exist: 404.</summary>
public sealed class NotFoundException(string what, object id)
    : Exception($"No {what} {id}.");

/// <summary>Not possible in the record's present state: 409.</summary>
public sealed class ConflictException(string message) : Exception(message);
```

Two details are specific to .NET. The time comes from `TimeProvider`, .NET's injectable clock, converted to Western Indonesian Time (UTC+7, with no daylight saving) and cut to whole seconds, so that the contract's time format holds. And the class uses a *primary constructor*, `LaundryService(ILaundryStore store, TimeProvider clock)`, C#'s short form for constructor injection.

16.5.5 The API, its errors and its doors

Listing 16.7 describes what comes in and what goes out.

Listing 16.7 `Dtos.cs`: request records with validation attributes, and the views in the contract's shape.

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;
using LaundryCapstone.Domain;

namespace LaundryCapstone.Api;

// ---- what comes in (validated before the service sees it) ----

public sealed record TokenRequest(
    [Required, EmailAddress] string? Email,
    [Required] string? Password,
    [Required, MaxLength(60)] string? Device);

public sealed record CustomerRequest(
    [Required, StringLength(80, MinimumLength = 2)] string? Name,
    [Required] string? Phone,
    [MaxLength(200)] string? Address);

public sealed record OrderRequest(
    [Required] long? CustomerId,
    [Required] string? ServiceId,
    [Required, Range(typeof(decimal), "0.1", "100",
        ParseLimitsInInvariantCulture = true), TwoDecimals] decimal? Quantity,
    bool Pickup,
    [MaxLength(200)] string? DeliveryAddress,
    [MaxLength(200)] string? Note);

public sealed record StatusRequest([Required] string? Status);

public sealed record PaymentRequest(
    [Required, Range(1000, long.MaxValue)] long? Amount,
    [Required] string? Method);

/// <summary>At most two digits after the decimal point, e.g. 4.02 kg.</summary>
public sealed class TwoDecimalsAttribute : ValidationAttribute
{
    public TwoDecimalsAttribute() : base("must have at most two decimals") { }

    public override bool IsValid(object? value) =>
        value is not decimal d || decimal.Round(d, 2) == d;
}

// ---- what goes out: the JSON of Chapter 12's contract ----

public sealed record DataView<T>(T Data);
```

Listing 16.7 (Continued) *Dtos.cs: request records with validation attributes, and the views in the contract's shape.*

```
public sealed record PageMeta(int CurrentPage, int PerPage, long Total,
                              int LastPage);

public sealed record PageView<T>(List<T> Data, PageMeta Meta);

public sealed record ServiceView(string Id, string Name, string Unit,
                                 long Price)
{
    public static ServiceView Of(Service s) => new(s.Id, s.Name, s.Unit,
                                                   s.Price);
}

public sealed record CustomerView(long Id, string Name, string Phone,
                                  string? Address)
{
    public static CustomerView Of(Customer c) => new(c.Id, c.Name, c.Phone,
                                                     c.Address);
}

public sealed record PaymentView(long Id, long Amount, string Method,
                                 string PaidAt)
{
    public static PaymentView Of(Payment p) =>
        new(p.Id, p.Amount, p.Method.Value(), Json.Time(p.PaidAt)!);
}

public sealed record OrderView(long Id, CustomerView Customer,
    ServiceView Service, decimal Quantity, bool Pickup,
    string? DeliveryAddress, long Total, long Paid, long Balance,
    string Status, string? Note, string ReceivedAt, string? FinishedAt,
    string? DepartedAt,
    [property: JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    List<PaymentView>? Payments)
{
    /// <summary>Without the payment list, like Laravel when not loaded.</summary>
    public static OrderView Of(Order o, bool withPayments = false) => new(
        o.Id, CustomerView.Of(o.Customer), ServiceView.Of(o.Service),
        o.Quantity, o.Pickup, o.DeliveryAddress, o.Total, o.Paid, o.Balance,
        o.Status.Value(), o.Note, Json.Time(o.ReceivedAt)!,
        Json.Time(o.FinishedAt), Json.Time(o.DepartedAt),
        withPayments ? o.Payments.Select(PaymentView.Of).ToList() : null);
}

public sealed record DailyReport(string Date, long OrdersReceived,
    long Billed, long Collected, Dictionary<string, long> ByStatus);
```

Listing 16.7 (Continued) *Dtos.cs: request records with validation attributes, and the views in the contract's shape.*

```
public sealed record UserView(string Name, string Role);

public sealed record TokenView(string Token, UserView User);

public static class Json
{
    /// <summary>The contract's one date-time format: 2026-09-25T08:15:00.</summary>
    public static string? Time(DateTime? t) => t?.ToString("yyyy-MM-dd'T'HH:mm:ss");
}
```

The request records use nullable types (`long?`, `decimal?`) with `[Required]`, so that a missing field is reported as missing rather than silently becoming 0. The views are records whose properties are written in C#'s PascalCase; the JSON options in `Program.cs` turn them into snake_case

(`DeliveryAddress` becomes `delivery_address`). `DataView<T>` and `PageView<T>` are the envelopes. **Listing 16.8** is the thin order controller.

Listing 16.8 `OrdersController.cs`: read JSON, call `LaundryService`, write JSON.

```
using LaundryCapstone.Application;
using LaundryCapstone.Domain;
using Microsoft.AspNetCore.Mvc;

namespace LaundryCapstone.Api;

/// <summary>Thin: read JSON, call LaundryService, write JSON.</summary>
[ApiController]
[Route("api/orders")]
public class OrdersController(LaundryService laundry) : ControllerBase
{
    [HttpGet]
    public async Task<IActionResult> Index(string? status, int page = 1)
    {
        OrderStatus? wanted = null;
        if (status is not null)
        {
            if (!Enum.TryParse<OrderStatus>(status, true, out var s))
                return Invalid("status", "The selected status is invalid.");
            wanted = s;
        }
        page = Math.Max(page, 1);
        var (items, total) = await laundry.OrderPageAsync(wanted, page, 20);
        return Ok(new PageView<OrderView>(
            items.Select(o => OrderView.Of(o)).ToList(),
            new PageMeta(page, 20, total, Math.Max(1, (total + 19) / 20))));
    }

    [HttpGet("{id:long}")]
    public async Task<DataView<OrderView>> Show(long id) =>
        new(OrderView.Of(await laundry.OrderAsync(id), withPayments: true));

    /// <summary>Every broken rule is reported at once, as Laravel does.</summary>
    [HttpPost]
    public async Task<IActionResult> Store(OrderRequest body)
    {
        var order = new NewOrder(body.CustomerId, body.ServiceId,
            body.Quantity, body.Pickup, body.DeliveryAddress, body.Note);
        var rules = await laundry.CheckAsync(order);
        if (!ModelState.IsValid || rules.Count > 0)
            return ApiErrors.Unprocessable(ModelState, rules);
        return StatusCode(201, new DataView<OrderView>(
            OrderView.Of(await laundry.CreateOrderAsync(order))));
    }
```

Listing 16.8 (Continued) *OrdersController.cs: read JSON, call LaundryService, write JSON.*

```
    [HttpPatch("{id:long}/status")]
    public async Task<IActionResult> UpdateStatus(long id, StatusRequest body)
    {
        if (!ModelState.IsValid) return ApiErrors.Unprocessable(ModelState);
        if (!Enum.TryParse<OrderStatus>(body.Status, true, out var wanted))
            return Invalid("status", "The selected status is invalid.");
        return Ok(new DataView<OrderView>(
            OrderView.Of(await laundry.AdvanceAsync(id, wanted))));
    }

    [HttpPost("{id:long}/payments")]
    public async Task<IActionResult> Pay(long id, PaymentRequest body)
    {
        if (!ModelState.IsValid) return ApiErrors.Unprocessable(ModelState);
        if (!Enum.TryParse<PaymentMethod>(body.Method, true, out var method))
            return Invalid("method", "The selected method is invalid.");
        var order = await laundry.PayAsync(id, body.Amount!.Value, method);
        return StatusCode(201, new DataView<OrderView>(
            OrderView.Of(order, withPayments: true)));
    }

    private ObjectResult Invalid(string field, string message) =>
        ApiErrors.Unprocessable(ModelState, new() { [field] = [message] });
}
```

`Store` merges the validation attributes' errors (`ModelState`) with the service's rules and answers once, like `OrderApiController.store` in Spring. That only works because of one line in `Program.cs`: `SuppressModelStateInvalidFilter = true`. By default, `[ApiController]` answers invalid input by itself, *before* the action runs, with status 400 and a problem-details document; the contract wants 422, Laravel's shape, and the service's rules in the same answer. **Listing 16.9** builds that answer and maps the service's exceptions.

Listing 16.9 `ApiErrors.cs`: every error in the contract's shape.

```
using System.Text.Json;
using LaundryCapstone.Application;
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace LaundryCapstone.Api;

/// <summary>Every error in the contract's shape: {"message", "errors"}.</summary>
public static class ApiErrors
{
    public const string Invalid = "The given data was invalid.";

    /// <summary>422 with the model's errors plus the rules' errors.</summary>
    public static ObjectResult Unprocessable(ModelStateDictionary model,
        Dictionary<string, List<string>>? more = null)
    {
        var errors = new Dictionary<string, List<string>>();
        foreach (var (key, entry) in model)
        {
            if (entry.Errors.Count == 0) continue;
            var list = errors.TryGetValue(Snake(key), out var l) ? l
                     : errors[Snake(key)] = [];
            list.AddRange(entry.Errors.Select(e => e.ErrorMessage));
        }
        foreach (var (key, messages) in more ?? [])
        {
            var list = errors.TryGetValue(key, out var l) ? l : errors[key] = [];
            list.AddRange(messages);
        }
        return new ObjectResult(new { message = Invalid, errors })
        { StatusCode = StatusCodes.Status422UnprocessableEntity };
    }

    /// <summary>"CustomerId" or "$.customer_id" -> "customer_id".</summary>
    public static string Snake(string key)
    {
        key = key.StartsWith("$.") ? key[2..] : key;
        return JsonNamingPolicy.SnakeCaseLower.ConvertName(key);
    }
}
```

Listing 16.9 (Continued) *ApiErrors.cs: every error in the contract's shape.*

```
/// <summary>Our exceptions as 422, 404 and 409 answers.</summary>
public sealed class ApiExceptionHandler : IExceptionHandler
{
    public async ValueTask<bool> TryHandleAsync(HttpContext http,
        Exception error, CancellationToken cancel)
    {
        (int status, object body) = error switch
        {
            RuleException e => (422, (object)new
            {
                message = ApiErrors.Invalid, errors = e.Errors
            }),
            NotFoundException e => (404, new { message = e.Message }),
            ConflictException e => (409, new { message = e.Message }),
            _ => (0, new { })
        };
        if (status == 0) return false;                  // a real bug: 500
        http.Response.StatusCode = status;
        await http.Response.WriteAsJsonAsync(body, cancel);
        return true;
    }
}
```

`ApiExceptionHandler` implements `IExceptionHandler`, ASP.NET Core's hook for exceptions that escape a controller [3]. It answers `RuleException`, `NotFoundException` and `ConflictException` with 422, 404 and 409 and returns `false` for anything else, so a real bug still becomes a 500 and is logged. The customer controller (**Listing 16.10**) shows the role check.

Listing 16.10 `CustomersController.cs`: the phone kept as digits, and `DELETE` for the owner only.

```
using System.Text.RegularExpressions;
using LaundryCapstone.Application;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace LaundryCapstone.Api;

[ApiController]
[Route("api/customers")]
public partial class CustomersController(LaundryService laundry)
    : ControllerBase
{
    [HttpGet]
    public async Task<DataView<List<CustomerView>>> Index(string? phone) =>
        new((await laundry.CustomersAsync(phone)).Select(CustomerView.Of)
                                                  .ToList());

    [HttpPost]
    public async Task<IActionResult> Store(CustomerRequest body)
    {
        // "0812-3456-7890" is kept as digits, then checked, as in Laravel
        string phone = LaundryService.Digits(body.Phone ?? "");
        if (body.Phone is not null && !MobilePhone().IsMatch(phone))
            ModelState.AddModelError("phone",
                "Write the phone as 08..., 10 to 13 digits.");
        if (!ModelState.IsValid) return ApiErrors.Unprocessable(ModelState);
        var customer = await laundry.CreateCustomerAsync(body.Name!, phone,
                                                         body.Address);
        return StatusCode(201, new DataView<CustomerView>(
            CustomerView.Of(customer)));
    }

    [HttpDelete("{id:long}"), Authorize(Roles = "OWNER")]
    public async Task<IActionResult> Destroy(long id)
    {
        await laundry.DeleteCustomerAsync(id);
        return NoContent();
    }

    [GeneratedRegex("^08[0-9]{8,11}$")]
    private static partial Regex MobilePhone();
}
```

`[Authorize(Roles = "OWNER")]` on `Destroy` is the ASP.NET Core form of Spring's URL rule; the service then refuses a customer with orders with 409. The phone is reduced to digits and checked with a regular expression that the compiler turns into code at build time (`[GeneratedRegex]`). The doors themselves are an authentication handler (**Listing 16.11**) and a token service (**Listing 16.12**).

Listing 16.11 `BearerTokenHandler.cs`: from a bearer token to a signed-in staff member, with the contract's 401 and 403.

```
using System.Security.Claims;
using System.Text.Encodings.Web;
using LaundryCapstone.Application;
using Microsoft.AspNetCore.Authentication;
using Microsoft.Extensions.Options;

namespace LaundryCapstone.Security;

/// <summary>"Authorization: Bearer 12|secret" -> the owner of token 12.</summary>
public sealed class BearerTokenHandler(
    IOptionsMonitor<AuthenticationSchemeOptions> options, ILoggerFactory logger,
    UrlEncoder encoder, TokenService tokens)
    : AuthenticationHandler<AuthenticationSchemeOptions>(options, logger, encoder)
{
    public const string SchemeName = "Bearer";

    protected override async Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        string header = Request.Headers.Authorization.ToString();
        if (!header.StartsWith("Bearer ", StringComparison.Ordinal))
            return AuthenticateResult.NoResult();       // stay anonymous
        var token = await tokens.FindAsync(header[7..].Trim());
        if (token is null) return AuthenticateResult.Fail("Unknown token.");

        Claim[] claims =
        [
            new(ClaimTypes.Name, token.Staff.Email),
            new(ClaimTypes.Role, token.Staff.Role.ToString().ToUpperInvariant()),
            new("token_id", token.Id.ToString())
        ];
        var principal = new ClaimsPrincipal(new ClaimsIdentity(claims,
                                                               SchemeName));
        var ticket = new AuthenticationTicket(principal, SchemeName);
        return AuthenticateResult.Success(ticket);
    }

    protected override Task HandleChallengeAsync(AuthenticationProperties p) =>
        Json(401, "Unauthenticated.");

    protected override Task HandleForbiddenAsync(AuthenticationProperties p) =>
        Json(403, "This action is unauthorized.");
```

Listing 16.11 (Continued) *BearerTokenHandler.cs: from a bearer token to a signed-in staff member, with the contract's 401 and 403.*

```
    private Task Json(int status, string message)
    {
        Response.StatusCode = status;
        return Response.WriteAsJsonAsync(new { message });
    }
}
```

Listing 16.12 `TokenService.cs`: Sanctum-style tokens, hashed with SHA-256 and compared in constant time.

```
using System.Security.Cryptography;
using System.Text;
using LaundryCapstone.Data;
using LaundryCapstone.Domain;
using Microsoft.AspNetCore.Identity;

namespace LaundryCapstone.Application;

/// <summary>
/// API tokens in the style of Laravel Sanctum: "id|secret" is shown once,
/// the table keeps only the SHA-256 of the secret.
/// </summary>
public sealed class TokenService(ILaundryStore store,
    IPasswordHasher<Staff> hasher, TimeProvider clock)
{
    public async Task<(string Token, Staff Staff)> IssueAsync(string email,
        string password, string device)
    {
        var staff = await store.FindStaffAsync(email);
        if (staff is null || hasher.VerifyHashedPassword(staff,
                staff.PasswordHash, password) == PasswordVerificationResult.Failed)
            throw new RuleException("email",
                "These credentials do not match our records.");
        string secret = Convert.ToBase64String(RandomNumberGenerator.GetBytes(30))
            .Replace('+', '-').Replace('/', '_');          // 40 characters
        var token = new ApiToken
        {
            StaffId = staff.Id, Device = device, TokenHash = Sha256(secret),
            CreatedAt = clock.GetLocalNow().DateTime
        };
        store.Add(token);
        await store.SaveChangesAsync();
        return ($"{token.Id}|{secret}", staff);
    }

    /// <summary>The token behind "id|secret", if the secret fits.</summary>
    public async Task<ApiToken?> FindAsync(string plain)
    {
        int bar = plain.IndexOf('|');
        if (bar < 1 || !long.TryParse(plain[..bar], out long id)) return null;
        var token = await store.FindTokenAsync(id);
        if (token is null || !CryptographicOperations.FixedTimeEquals(
                Encoding.ASCII.GetBytes(token.TokenHash),
                Encoding.ASCII.GetBytes(Sha256(plain[(bar + 1)..]))))
            return null;
        token.LastUsedAt = clock.GetLocalNow().DateTime;
```

Listing 16.12 (Continued) *TokenService.cs: Sanctum-style tokens, hashed with SHA-256 and compared in constant time.*

```
        await store.SaveChangesAsync();
        return token;
    }

    public async Task RevokeAsync(long id)
    {
        if (await store.FindTokenAsync(id) is { } token)
        {
            store.Remove(token);
            await store.SaveChangesAsync();
        }
    }

    private static string Sha256(string text) => Convert.ToHexStringLower(
        SHA256.HashData(Encoding.UTF8.GetBytes(text)));
}
```

These are Chapter 14's `BearerTokenFilter` and `TokenService` in C#. `AuthenticateResult.NoResult()` leaves a request without a token anonymous; the *fallback policy* in `Program.cs`, "require an authenticated user", then refuses it wherever no `[AllowAnonymous]` says otherwise, which is Spring's `anyRequest().authenticated()`. `HandleChallengeAsync` and `HandleForbiddenAsync` write the contract's 401 and 403 bodies. Passwords are checked with `PasswordHasher<Staff>`, the hasher of ASP.NET Core Identity, which is part of the shared framework and needs no package; it stores PBKDF2 with HMAC-SHA512 and 100,000 iterations [4]. **Listing 16.13** wires everything together.

Listing 16.13 `Program.cs`: storage, JSON, errors, security and the start-up data, in order.

```
using System.Text.Json;
using LaundryCapstone.Api;
using LaundryCapstone.Application;
using LaundryCapstone.Data;
using LaundryCapstone.Domain;
using LaundryCapstone.Security;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
#if USE_EF
using LaundryCapstone.Data.Ef;
using Microsoft.EntityFrameworkCore;
#endif

var builder = WebApplication.CreateBuilder(args);

// ---- storage: SQLite through EF Core, or memory in a NuGet-free build ----
#if USE_EF
builder.Services.AddDbContext<LaundryDbContext>(o =>
    o.UseSqlite(builder.Configuration.GetConnectionString("Laundry")));
builder.Services.AddScoped<ILaundryStore, EfLaundryStore>();
#else
builder.Services.AddSingleton<ILaundryStore, MemoryLaundryStore>();
#endif
builder.Services.AddSingleton(TimeProvider.System);
builder.Services.AddScoped<LaundryService>();
builder.Services.AddScoped<TokenService>();
builder.Services.AddSingleton<IPasswordHasher<Staff>, PasswordHasher<Staff>>();

// ---- MVC + JSON in snake_case; we answer invalid input ourselves (422) ----
builder.Services.AddControllersWithViews()
    .AddJsonOptions(o => o.JsonSerializerOptions.PropertyNamingPolicy =
        JsonNamingPolicy.SnakeCaseLower)
    .ConfigureApiBehaviorOptions(o => o.SuppressModelStateInvalidFilter = true);
builder.Services.AddExceptionHandler<ApiExceptionHandler>();

// ---- security: bearer tokens, a login needed unless marked otherwise ----
builder.Services.AddAuthentication(BearerTokenHandler.SchemeName)
    .AddScheme<AuthenticationSchemeOptions, BearerTokenHandler>(
        BearerTokenHandler.SchemeName, null);
builder.Services.AddAuthorizationBuilder()
    .SetFallbackPolicy(new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser().Build());

var app = builder.Build();
```

Listing 16.13 (Continued) *Program.cs: storage, JSON, errors, security and the start-up data, in order.*

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    await Seeder.SeedAsync(services.GetRequiredService<ILaundryStore>(),
        services.GetRequiredService<IPasswordHasher<Staff>>(),
        services.GetRequiredService<LaundryService>());
}

app.UseExceptionHandler(_ => { });
app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();
app.MapGet("/", () => Results.Redirect("/board")).AllowAnonymous();

app.Run();
```

Read from top to bottom, `Program.cs`, the single start-up file of ASP.NET Core's minimal hosting model [5], is a list of every decision the capstone makes: which store, snake_case JSON, our own 422 answers, our exception handler, the bearer scheme, "signed in unless marked otherwise". The seeder loads the price list from `data/Service.csv`, the same file as Spring's, creates the two staff accounts with hashed passwords, and registers the three demo customers through `LaundryService`, so that even the demo data pass the service's rules.

16.5.6 Running it

On Windows, macOS or Linux with the .NET 10 SDK, open `Chapter16/aspnet/LaundryCapstone` in a terminal or in VS Code and start it with SQLite:

```
dotnet run --urls http://localhost:5000
```

Without access to NuGet, build the memory version instead:

```
dotnet run -p:UseEf=false --urls http://localhost:5000
```

The second command is the one that could run on the machine where the book was written (SDK 10.0.112, runtime 10.0.12). With the application listening, the checker ran in a second terminal:

```
php ../../contract/check_api.php http://localhost:5000
```

and printed:

```
PASS  a wrong password -> 422, errors.email
PASS  no token -> 401 with a message
PASS  POST /api/tokens -> 201, a token for a cashier
PASS  GET /api/services -> 200, S0001 first, 8000 per kg
PASS  POST /api/customers -> 201, phone kept as digits only
PASS  the same phone again -> 422, errors.phone
PASS  POST /api/orders -> 201, total 34000 (3 kg + pickup), queued
PASS  times are YYYY-MM-DDTHH:MM:SS, finished_at is null
PASS  a bad order -> 422, all of its errors at once
PASS  queued -> departed skips a step: 422, errors.status
PASS  queued -> finished: 200, finished_at set
PASS  finished -> departed before payment: 422, errors.status
PASS  paying more than the balance -> 422, errors.amount
PASS  paying the balance -> 201, balance 0, payments[0].method qris
PASS  finished -> departed after payment: 200
PASS  a departed order is closed: 409 with a message
PASS  GET /api/orders?status=departed -> 200, data[] and meta
PASS  GET /api/orders/999999 -> 404 with a message
PASS  GET /api/reports/daily -> 200, the counts add up
PASS  a cashier deletes a customer -> 403 with a message
PASS  the owner logs in -> 201, role owner
PASS  the owner deletes a customer with orders -> 409
PASS  the owner deletes a customer without orders -> 204
PASS  DELETE /api/tokens/current -> 204
PASS  the old token no longer works -> 401

25 passed, 0 failed
```

A second run against the same process also passed 25 of 25, which shows that the random phone numbers make the checker repeatable. Chapter 12's own checker, given a token, passed its 16 checks against the same server. A few orders were then taken by hand, and two answers were saved as they came. Sari Wulandari's order of 4.5 kg, read back with `GET /api/orders/5`:


```
{
    "data": {
        "id": 5,
        "customer": {
            "id": 1,
            "name": "Sari Wulandari",
            "phone": "081200000101",
            "address": "Jl. Kertajaya 12, Surabaya"
        },
        "service": {
            "id": "S0001",
            "name": "Wash, dry & fold",
            "unit": "kg",
            "price": 8000
        },
        "quantity": 4.5,
        "pickup": false,
        "delivery_address": null,
        "total": 36000,
        "paid": 0,
        "balance": 36000,
        "status": "queued",
        "note": null,
        "received_at": "2026-09-25T16:10:08",
        "finished_at": null,
        "departed_at": null,
        "payments": []
    }
}
```


and the refusal to hand over an unpaid bed-cover order of Rp 80.000 (two items plus the pickup fee, row six of **Table 16.2**):

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json; charset=utf-8

{
    "message": "The given data was invalid.",
    "errors": {
        "status": [
            "Collect Rp 80.000 before handing over the laundry."
        ]
    }
}
```

What was *not* run there is the EF Core build, since its package could not be downloaded. Its code is the same interface with LINQ queries, and the README asks you to run the checker against it first.

16.5.7 The counter board

The board is a Razor page that anyone in the shop may see (**Listing 16.14** and **Listing 16.15**).

Listing 16.14 `BoardController.cs`: the queue and the finished orders, without a login.

```
using LaundryCapstone.Application;
using LaundryCapstone.Domain;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace LaundryCapstone.Controllers;

/// <summary>
/// The counter board that faces the customers: order numbers and first
/// names only, so it may be shown without a login.
/// </summary>
[Route("board"), AllowAnonymous]
public class BoardController(LaundryService laundry) : Controller
{
    [HttpGet("")]
    public async Task<IActionResult> Index()
    {
        var (queued, _) =
            await laundry.OrderPageAsync(OrderStatus.Queued, 1, 100);
        var (finished, _) =
            await laundry.OrderPageAsync(OrderStatus.Finished, 1, 100);
        return View((queued, finished));
    }
}
```

Listing 16.15 `Views/Board/Index.cshtml`: order numbers and first names, refreshed every 30 seconds.

```
@model (List<Order> Queued, List<Order> Finished)
@{
    // "Sari Wulandari" -> "Sari": a public screen shows first names only
    static string First(string name) => name.Split(' ')[0];
}
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="refresh" content="30">
  <title>Counter board | Profitina Laundry</title>
  <style>
    body { font-family: system-ui, sans-serif; margin: 1rem; color: #1f2937; }
    h1 { color: #12355B; }
    .cols { display: flex; gap: 2rem; flex-wrap: wrap; }
    li { font-size: 1.3rem; margin: .3rem 0; }
    .ready { color: #0F7C7C; }
  </style>
</head>
<body>
  <h1>Profitina Laundry: counter board</h1>
  <div class="cols">
    <section>
      <h2>In the queue (@Model.Queued.Count)</h2>
      <ol>
        @foreach (var o in Model.Queued)
        {
          <li>#@o.Id @First(o.Customer.Name), @o.Service.Name</li>
        }
      </ol>
    </section>
    <section class="ready">
      <h2>Ready to collect (@Model.Finished.Count)</h2>
      <ol>
        @foreach (var o in Model.Finished)
        {
          <li>#@o.Id @First(o.Customer.Name)</li>
        }
      </ol>
    </section>
  </div>
</body>
</html>
```

`[AllowAnonymous]` opens this one controller in an application that is closed by default. Because the board hangs where every customer can read it, it shows the order number and the *first* name only: no surname, no phone, no address. This is *data minimization*: a public screen shows what it needs and nothing more. The page reloads itself every 30 seconds with a `meta refresh`, which needs no JavaScript. **Figure 16.3** is the board of the running capstone.

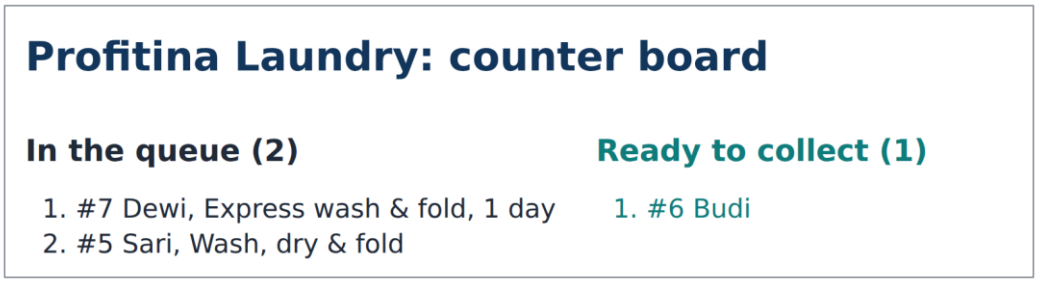


Figure 16.3 The counter board of the ASP.NET Core capstone as it ran on September 25, 2026: two orders in the queue, one ready to collect.

16.5.8 A tempting wrong solution

The first version of the request records looked like this:

```
public sealed record OrderRequest(
    [property: Required] long? CustomerId, ...
```

The `property:` target seems right: it puts the attribute on the generated property, where validation attributes normally live. The project built without a warning. But the first request to `POST /api/orders` failed with an `InvalidOperationException`: ASP.NET Core refuses validation attributes on the properties of a positional record and insists that they be placed on the constructor parameters, because it creates the record through its constructor [3]. The first run of the checker failed 14 of its checks for this one reason. The fix was to delete `property:`, as **Listing 16.7** shows.

The second surprise came right after. With the records fixed, a bad order was answered with `errors` keyed `CustomerId` and `Quantity`: `ModelState` names fields after the C# property, or after the JSON path (`$.customer_id`) when the JSON itself cannot be read, and the snake_case JSON options do not apply to those keys. The checker, which expects `customer_id`, reported it at once; `ApiErrors.Snake` now converts every key. Both mistakes compiled cleanly, looked correct in review, and were found only because a program checked the answers field by field. That is the strongest argument this chapter can make for a contract in code.

16.6 Problem 16.4: The Laravel Capstone

16.6.1 The problem

Complete the Laravel project of Chapter 10 so that it passes the same 25 checks and has the same board. Chapter 10 already has Sanctum tokens, the `owner` and `cashier` roles, and Chapter 12 already made its API match the contract; what is missing is the owner's delete and the board.

16.6.2 Designing it on paper

Laravel's conventions decide most of the design. Route-model binding turns `{customer}` in the URL into a `Customer` model, or a 404. A *gate* named `delete-customers` expresses the rule, and the `can:` middleware applies it to the route, answering 403 with Laravel's own "This action is unauthorized." The board is an invokable controller with a Blade view. The files in `Chapter16/laravel` are copied over the Chapter 10 project, which already contains Chapter 12's files; no migration is needed, because nothing new is stored.

16.6.3 The solution

Listing 16.16 is the API's route file with one new route.

Listing 16.16 `routes/api.php`: the new `DELETE` route, for the owner only.

```php
<?php

use App\Http\Controllers\Api\CustomerController;
use App\Http\Controllers\Api\OrderController;
use App\Http\Controllers\Api\PaymentController;
use App\Http\Controllers\Api\ReportController;
use App\Http\Controllers\Api\ServiceController;
use App\Http\Controllers\Api\TokenController;
use Illuminate\Support\Facades\Route;

// Every URL here starts with /api (Laravel adds the prefix).
Route::post('tokens', [TokenController::class, 'store'])
    ->middleware('throttle:5,1');

Route::middleware('auth:sanctum')->group(function () {
    Route::delete('tokens/current', [TokenController::class, 'destroy']);

    Route::get('services', [ServiceController::class, 'index']);
    Route::get('services/{service}', [ServiceController::class, 'show']);

    Route::get('customers', [CustomerController::class, 'index']);
    Route::post('customers', [CustomerController::class, 'store']);
    Route::delete('customers/{customer}',
        [CustomerController::class, 'destroy'])
        ->middleware('can:delete-customers');         // the owner only

    Route::get('orders', [OrderController::class, 'index']);
    Route::post('orders', [OrderController::class, 'store']);
    Route::get('orders/{order}', [OrderController::class, 'show']);

    Route::middleware('order.open')->group(function () {
        Route::patch('orders/{order}/status',
            [OrderController::class, 'updateStatus']);
        Route::post('orders/{order}/payments',
            [PaymentController::class, 'store']);
    });

    Route::get('reports/daily', [ReportController::class, 'daily']);
});
```

The routes also show Chapter 10's other doors: `throttle:5,1` limits sign-in attempts to five per minute, `auth:sanctum` protects everything else, and `order.open` answers 409 for a departed order before the controller runs. **Listing 16.17** defines the gate, and **Listing 16.18** the action.

Listing 16.17 `AppServiceProvider.php`: the gates of Chapter 10 and the new `delete-customers`.

```php
<?php

namespace App\Providers;

use App\Models\User;
use Illuminate\Support\Facades\Gate;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        //
    }

    public function boot(): void
    {
        // Only the owner changes the price list.
        Gate::define('manage-prices', fn (User $user) => $user->isOwner());

        // Only the owner deletes customers (Chapter 16).
        Gate::define('delete-customers', fn (User $user) => $user->isOwner());
    }
}
```

Listing 16.18 `CustomerController.php`: `destroy` refuses a customer with orders with 409.

```php
<?php

namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use App\Http\Requests\StoreCustomerRequest;
use App\Http\Resources\CustomerResource;
use App\Models\Customer;
use Illuminate\Http\Request;

class CustomerController extends Controller
{
    // GET /api/customers?phone=0812...  (the counter looks people up by phone)
    public function index(Request $request)
    {
        $phone = preg_replace('/[^0-9]/', '', (string) $request->query('phone'));
        $customers = Customer::query()
            ->when($phone !== '', fn ($q) => $q->where('phone', $phone))
            ->orderBy('name')
            ->paginate(20);

        return CustomerResource::collection($customers);
    }

    public function store(StoreCustomerRequest $request)
    {
        $customer = Customer::create($request->validated());

        return (new CustomerResource($customer))->response()
            ->setStatusCode(201);
    }

    // DELETE /api/customers/{customer}: the owner only (routes/api.php).
    public function destroy(Customer $customer)
    {
        if ($customer->orders()->exists()) {
            abort(409, "{$customer->name} has orders and cannot be deleted.");
        }
        $customer->delete();

        return response()->noContent();
    }
}
```

`abort(409, ...)` produces `{"message": "..."}` for a JSON request without any further code, which is exactly the contract's 409. **Listing 16.19**, **Listing 16.20** and **Listing 16.21** add the board.

Listing 16.19 `routes/web.php`: the board is a public route.

```php
<?php

use App\Http\Controllers\Auth\LoginController;
use App\Http\Controllers\BoardController;
use App\Http\Controllers\CounterController;
use App\Http\Controllers\CustomerController;
use App\Http\Controllers\HomeController;
use App\Http\Controllers\OrderController;
use App\Http\Controllers\OutletController;
use App\Http\Controllers\PaymentController;
use App\Http\Controllers\PriceController;
use App\Http\Controllers\ReportController;
use App\Http\Controllers\ServiceController;
use Illuminate\Support\Facades\Route;

// The public pages: anyone may see them.
Route::get('/', HomeController::class)->name('home');
Route::get('services', [ServiceController::class, 'index'])
    ->name('services.index');
Route::get('services/{service}', [ServiceController::class, 'show'])
    ->name('services.show');
Route::get('outlets', [OutletController::class, 'index'])
    ->name('outlets.index');
Route::get('board', BoardController::class)->name('board');   // Chapter 16

// Signing in and out.
Route::middleware('guest')->group(function () {
    Route::get('login', [LoginController::class, 'create'])->name('login');
    Route::post('login', [LoginController::class, 'store'])
        ->middleware('throttle:5,1');           // five tries a minute
});
Route::post('logout', [LoginController::class, 'destroy'])
    ->middleware('auth')->name('logout');

// The staff pages: only for signed-in staff.
Route::middleware('auth')->group(function () {
    Route::get('counter', CounterController::class)->name('counter');
    Route::resource('customers', CustomerController::class);
    Route::resource('orders', OrderController::class)
        ->only(['index', 'create', 'store', 'show']);
    Route::get('reports/daily', [ReportController::class, 'daily'])
        ->name('reports.daily');
```

Listing 16.19 (Continued) *routes/web.php: the board is a public route.*

```php
    Route::middleware('order.open')->group(function () {
        Route::post('orders/{order}/advance',
            [OrderController::class, 'advance'])->name('orders.advance');
        Route::post('orders/{order}/payments',
            [PaymentController::class, 'store'])
            ->name('orders.payments.store');
    });

    // The price list: the owner only.
    Route::resource('prices', PriceController::class)
        ->parameters(['prices' => 'service'])
        ->except(['show', 'destroy'])
        ->middleware('can:manage-prices');
});
```

Listing 16.20 `BoardController.php`: the queue and the finished orders.

```php
<?php

namespace App\Http\Controllers;

use App\Enums\OrderStatus;
use App\Models\Order;

// The counter board that faces the customers: order numbers and first
// names only, so it may be shown without a login.
class BoardController extends Controller
{
    public function __invoke()
    {
        $load = fn (OrderStatus $s) => Order::with(['customer', 'service'])
            ->where('status', $s)->orderByDesc('received_at')
            ->orderByDesc('id')->limit(100)->get();

        return view('board', [
            'queued' => $load(OrderStatus::Queued),
            'finished' => $load(OrderStatus::Finished),
        ]);
    }
}
```

Listing 16.21 `board.blade.php`: the same board as Razor's, in Blade.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="refresh" content="30">
  <title>Counter board | Profitina Laundry</title>
  <style>
    body { font-family: system-ui, sans-serif; margin: 1rem; color: #1f2937; }
    h1 { color: #12355B; }
    .cols { display: flex; gap: 2rem; flex-wrap: wrap; }
    li { font-size: 1.3rem; margin: .3rem 0; }
    .ready { color: #0F7C7C; }
  </style>
</head>
<body>
  <h1>Profitina Laundry: counter board</h1>
  <div class="cols">
    <section>
      <h2>In the queue ({{ $queued->count() }})</h2>
      <ol>
        @foreach ($queued as $o)
          <li>#{{ $o->id }} {{ strtok($o->customer->name, ' ') }},
              {{ $o->service->name }}</li>
        @endforeach
      </ol>
    </section>
    <section class="ready">
      <h2>Ready to collect ({{ $finished->count() }})</h2>
      <ol>
        @foreach ($finished as $o)
          <li>#{{ $o->id }} {{ strtok($o->customer->name, ' ') }}</li>
        @endforeach
      </ol>
    </section>
  </div>
</body>
</html>
```

`strtok($name, ' ')` returns the part before the first space, the first name, and Blade's `{{ }}` escapes it. The eager loading `with(['customer', 'service'])` loads the related rows in two extra queries instead of two per order. **Listing 16.22** tests the new behavior.

Listing 16.22 `CapstoneTest.php`: a cashier's delete, the owner's two deletes, and the board's privacy.

```
<?php

namespace Tests\Feature;

use App\Enums\Role;
use App\Models\Customer;
use App\Models\Order;
use App\Models\User;
use Database\Seeders\PriceListSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Laravel\Sanctum\Sanctum;
use Tests\TestCase;

// What Chapter 16 adds: owner-only deletes and the public counter board.
class CapstoneTest extends TestCase
{
    use RefreshDatabase;

    protected function setUp(): void
    {
        parent::setUp();
        $this->seed(PriceListSeeder::class);        // services and outlets
    }

    public function test_a_cashier_may_not_delete_a_customer(): void
    {
        Sanctum::actingAs(User::factory()->create());   // a cashier
        $customer = Customer::factory()->create();
        $this->deleteJson("/api/customers/{$customer->id}")
            ->assertForbidden()
            ->assertJson(['message' => 'This action is unauthorized.']);
        $this->assertModelExists($customer);
    }

    public function test_the_owner_deletes_only_customers_without_orders(): void
    {
        Sanctum::actingAs(User::factory()->create(['role' => Role::Owner]));
        $busy = Order::factory()->create()->customer;
        $this->deleteJson("/api/customers/{$busy->id}")->assertConflict();
        $idle = Customer::factory()->create();
        $this->deleteJson("/api/customers/{$idle->id}")->assertNoContent();
        $this->assertModelMissing($idle);
    }
```

Listing 16.22 (Continued) *CapstoneTest.php: a cashier's delete, the owner's two deletes, and the board's privacy.*

```
    public function test_the_board_shows_first_names_only(): void
    {
        $order = Order::factory()->create();
        $order->customer->update(['name' => 'Sari Wulandari']);
        $this->get('/board')->assertOk()
            ->assertSee('#' . $order->id . ' Sari')
            ->assertDontSee('Wulandari');
    }
}
```

`Sanctum::actingAs` signs a user in for one test without a token, the Laravel counterpart of `@WithMockUser`. The third test is worth copying into any project with a public screen: it asserts both what must be shown (`#12 Sari`) and what must not (`Wulandari`).

16.6.4 Running it

With PHP 8.3 or newer, Composer and the Chapter 10 project set up as its README describes, copy the files of `Chapter16/laravel` over it, then

```
php artisan migrate:fresh --seed
php artisan test
php artisan serve --port=8000
php ../contract/check_api.php http://localhost:8000 \
    cashier@laundry.test password owner@laundry.test password
```

(In Windows PowerShell, write the last command on one line, without the backslash.)

The first command rebuilds the SQLite database with the demo data, the second runs Chapter 10's tests and `CapstoneTest`, the third starts the server and the last runs the checker with Laravel's demo accounts. On the machine where the book was written, Composer could not download Laravel, so none of these commands could run there. Every PHP file of the chapter, the checker included, was checked with `php -l`, and the checker was run for real against the ASP.NET Core server. The Laravel code uses only features documented for Laravel 13 [6], [7].

16.6.5 A tempting wrong solution

It is tempting to write the role check inside the controller: `if (! $request->user()->isOwner()) abort(403);`. It works, but it hides the rule in one method where the next developer adding a delete for services will not see it, and it runs *after* route-model binding, so a cashier asking for customer 999 learns from the 404 that there is no such customer. The gate and the `can:` middleware put the rule where the routes are read, and run before the record is loaded.

16.7 Problem 16.5: The Spring Boot Capstone

16.7.1 The problem

Complete the Spring Boot project of Chapter 14 with the board. The owner's delete, the tokens and the contract's errors are already there.

16.7.2 The solution

Listing 16.23 and **Listing 16.24** are the board.

Listing 16.23 `BoardController.java`: the board reads its orders through `OrderService`.

```
package com.profitina.laundry.web;

import com.profitina.laundry.domain.OrderStatus;
import com.profitina.laundry.service.OrderService;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

/**
 * The counter board that faces the customers: order numbers and first
 * names only, so it may be shown without a login (SecurityConfig).
 */
@Controller
public class BoardController {

    private final OrderService orders;

    public BoardController(OrderService orders) {
        this.orders = orders;
    }

    @GetMapping("/board")
    public String board(Model model) {
        model.addAttribute("queued",
                orders.page(OrderStatus.QUEUED, 1, 100).getContent());
        model.addAttribute("finished",
                orders.page(OrderStatus.FINISHED, 1, 100).getContent());
        return "board";
    }
}
```

Listing 16.24 `board.html`: the same board in Thymeleaf.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="utf-8">
  <meta http-equiv="refresh" content="30">
  <title>Counter board | Profitina Laundry</title>
  <style>
    body { font-family: system-ui, sans-serif; margin: 1rem; color: #1f2937; }
    h1 { color: #12355B; }
    .cols { display: flex; gap: 2rem; flex-wrap: wrap; }
    li { font-size: 1.3rem; margin: .3rem 0; }
    .ready { color: #0F7C7C; }
  </style>
</head>
<body>
  <h1>Profitina Laundry: counter board</h1>
  <div class="cols">
    <section>
      <h2 th:text="'In the queue (' + ${queued.size()} + ')'">In the queue</h2>
      <ol>
        <li th:each="o : ${queued}"
            th:text="'#' + ${o.id} + ' ' + ${o.customer.name.split(' ')[0]}
                     + ', ' + ${o.service.name()}">#1 Sari, Wash</li>
      </ol>
    </section>
    <section class="ready">
      <h2 th:text="'Ready to collect (' + ${finished.size()} + ')'">Ready</h2>
      <ol>
        <li th:each="o : ${finished}"
            th:text="'#' + ${o.id} + ' ' + ${o.customer.name.split(' ')[0]}">
          #2 Budi</li>
      </ol>
    </section>
  </div>
</body>
</html>
```

The web chain of Chapter 14 must let the board through without a login, so `SecurityConfig` gains `"/board"` in the list of open pages: `.requestMatchers("/login", "/board", "/css/**", "/error").permitAll()` [8]. `${o.customer.name.split(' ')[0]}` calls Java's `String.split` from the template, and `th:text` escapes the result. The controller calls the service's existing `page` method, so the board uses the same queries, with the same entity graphs, as the counter pages.

16.7.3 Running it

With JDK 25 and Maven on Windows, macOS or Linux, from `Chapter16/spring`:

```
mvn test
mvn spring-boot:run
php ../contract/check_api.php http://localhost:8080
```

`mvn test` runs the 24 test runs of Chapter 14; the application listens on port 8080 with SQLite in the file `laundry.db`; the checker uses the default accounts. On the machine where the book was written, Maven could not download Spring, so these commands could not run there. Instead, every Java source file of the project, the tests included, was compiled with `javac` against small stand-ins for the Spring, Jakarta, Hibernate and JUnit types it uses, with no errors, and the rules ran for real (Problem 16.2). A compile check against stand-ins catches wrong names, types and imports; it cannot catch a wrong annotation attribute or a query that Hibernate rejects at start-up, which is why `mvn test` is the first command to run.

16.8 Comparing the Three Solutions

Table 16.3 puts the three capstones side by side, feature by feature.

Table 16.3 The three capstones compared.

Feature	ASP.NET Core 10	Laravel 13	Spring Boot 4.1
Token sign-in	own `TokenService` and `AuthenticationHandler`	Sanctum, built in	own `TokenService` and filter
401 and 403 bodies	written in the handler	Laravel's defaults	written in the filter chain
"Signed in unless marked"	fallback policy, `[AllowAnonymous]`	`auth:sanctum` group	`anyRequest().authenticated()`, `permitAll()`
Owner only	`[Authorize(Roles = "OWNER")]`	gate and `can:` middleware	URL rule and `@PreAuthorize`
422 with every error	switch off automatic 400, merge `ModelState` and rules	Form Requests, by default	`Validator` and rules merged in the controller
snake_case JSON	naming policy, plus converting error keys	by convention	Jackson setting
404 for a missing record	`NotFoundException` and handler	route-model binding	`NotFoundException` and advice
Board, first names only	`[AllowAnonymous]` controller, Razor	public route, Blade	`permitAll`, Thymeleaf
Storage	EF Core and SQLite, or memory	Eloquent and SQLite	JPA, Hibernate and SQLite
What was run here	built and run; 25/25 and 16/16	`php -l`	compiled against stand-ins; rules run

The pattern is clear. Laravel matches the contract with the least code, because the contract *is* Laravel's default shape: 422 with `errors`, the 401 and 403 messages, snake_case, the `data` envelope. The other two reach the same answers by changing defaults, and each change is a place where a mistake can hide, as Section 16.5.8 showed. Had the contract followed ASP.NET Core's defaults (400 with problem details, camelCase), the balance would have tipped the other way. The lesson for a real project is to choose the contract for the clients' sake first, and then to expect the frameworks that do not match it by default to need a small, well-tested layer of adaptation. Chapter 15's comparison of feature sizes (Section 15.5) counted the lines this chapter's two features cost in each framework; the differences were small.

What the table does not show is how alike the three services are. `LaundryService.CheckAsync`, `OrderService.check` and the rules inside Laravel's Form Request and actions ask the same four questions in the same order. The rules, the contract and the checker are the durable parts of the system; the frameworks are the replaceable parts.

How replaceable, in practice? Moving an outlet from one version to another means moving its data, not only its code. The four entities have the same columns in all three versions, with one difference that **Figure 16.2** points out: Laravel keeps its accounts in the `users` table and its tokens in Sanctum's `personal_access_tokens`, while the other two use `staff` and `api_tokens`. A move therefore copies services, customers, orders and payments table by table, keeping their identifiers so that order numbers printed on receipts stay valid, and asks the staff to set their passwords again, since the three frameworks store password hashes in different formats. Tokens are not copied at all: each phone simply signs in once more. The checker, run against the new server before the outlet opens, is the last step of such a move.

16.9 Bu Wati's Decision

With three versions that keep the same contract, the choice of which one runs the outlets no longer affects Raka's app, and it can be made with the procedure of Section 15.9 instead of by habit.

The constraints that cannot move. The shop already rents a small virtual server for its website, with 2 GB of memory. Any of the three runs on it; PHP hosting alone would also suffice for Laravel. Nothing in the shop has to connect to an existing Java or Windows platform.

What the people know. Dimas, who maintains the system during his holidays, knows PHP best and has used Laravel since Chapter 7. Raka writes the phone app and does not touch the server. Nobody in the family works with C# or Java daily.

The load. Three outlets take about 150 orders a day between them. Each order causes perhaps a dozen requests over its life (creation, lookups, two status steps, one or two payments, the board's reloads aside), so the busiest hour sees well under one request per second. By Little's law, with 30 ms per request, fewer than 0.03 requests are in progress on average. Speed drops out of the decision.

The calendar. Laravel 13 receives security fixes until March 17, 2028, .NET 10 until November 14, 2028, and Spring Boot 4.1 open-source fixes until July 31, 2027 (Section 15.7). Each is long enough for the first year.

The evidence of this chapter. The ASP.NET Core version is the only one that passed the checker on the machine where the book was written; the Laravel version is the one whose defaults match the contract and whose code for the new features was shortest; the Spring Boot version has the most complete counter pages (Chapter 13) and the richest tests (Chapter 14).

The procedure points to Laravel: the person who will maintain it knows it, it fits the hosting, and it needs the least adaptation to the contract. The decision is written down as an architecture decision record in the project, with the condition under which it would be revisited: if the shop ever grows to many outlets with a central server, or hires a developer who works in C# or Java, the other two versions are ready, and the checker will say within seconds whether a switch has broken anything. That last sentence is the real result of the capstone. The decision can be made on the grounds that matter to the shop, because the contract has made the three versions interchangeable for everyone outside the server.

16.10 Using It for Real

Before any of the three versions runs an outlet, a short list of work remains, and it is the same list whichever version Bu Wati chooses.

- **Run the untested builds.** Run `mvn test`, the Laravel tests and the EF Core build of ASP.NET Core, then the checker against each. Only the memory build of ASP.NET Core has passed it so far.
- **Change the demo passwords**, and create one account per cashier so that the records show who did what.
- **Serve over HTTPS** behind a reverse proxy, and back up the SQLite file every night to a different machine; test a restore once.
- **Migrations instead of automatic schema updates** before the first real data: EF Core migrations, Laravel migrations (already in place) or Flyway.

- **Run the checker after every deployment.** It takes two seconds and proves that Raka's app will keep working.

One more thing is often forgotten: knowing that the system is in trouble before a customer says so. All three frameworks log every request and every unhandled exception; send those logs to a file that is rotated daily, and check once a day for any 500 answer. A single 500 is an unknown bug, not bad luck. The same checker can be scheduled to run every morning before the shop opens, against the live server, with a test customer that the owner then deletes; if anything fails, a short message to Dimas's phone is enough to fix it before the first queue arrives.

The approach carries over to other businesses without change: a contract written as a checker, the rules in plain code with hand-worked examples, thin controllers, and one program that decides whether any implementation is acceptable.

16.11 Profitina in Practice

Profitina follows the same discipline with its own API. Its public endpoints are described by an OpenAPI document, and a test suite checks every endpoint's status codes and fields against it, so that the MCP server, the web dashboard and customers' scripts can rely on the same answers. Before each release the whole suite, well over a thousand tests, must pass, and a smaller set of checks runs against the live system after deployment, in the same spirit as `check_api.php`. When a new version of a library changes a default, such as a date format or the name of an error field, those checks are what notice it, long before a user does.

16.12 Common Mistakes, and How to Avoid Them

- **A contract only in prose.** Different readers implement different contracts. Write it as a program that checks field names and values.
- **Checkers that cannot run twice.** Fixed phone numbers or identifiers fail on the second run. Generate what must be unique.
- **Floating-point money.** Compute in whole units (hundredths of a kilogram, whole rupiah) with integers, as in Problem 16.2.
- **`<=` where the rule says "below".** Work the boundary by hand: five shirts at Rp 20.000 must not pay the pickup fee.
- **`[property:]` on validation attributes of positional records** in ASP.NET Core. Put them on the parameters.
- **Trusting the naming policy for error keys.** `ModelState` keys are not converted to snake_case; convert them yourself.
- **Personal data on a public screen.** Show order numbers and first names only, and test that the rest does not appear.
- **Role checks inside controller methods.** Declare them at the route (`can:`, `[Authorize]`, a URL rule) and, for important operations, again in the service.
- **Claiming more than was run.** Say what was built, what was run and what was only compiled.

16.13 Chapter Summary

- One contract (**Table 16.1**) and one checker (**Figure 16.1**, 25 checks) define the Profitina Laundry API for all three frameworks; the checker is plain PHP, repeatable, and reads the fields the app reads.
- The rules were worked by hand (Equation (16.1), Equation (16.2), **Table 16.2**) and produced the same totals in Java and C#, with integer arithmetic in hundredths.
- The ASP.NET Core capstone was built from scratch with a service layer, a storage interface with EF Core and memory implementations, token sign-in and the contract's errors; its memory build passed 25 of 25 checks and Chapter 12's 16.
- Two mistakes that compiled cleanly, `[property:]` on record parameters and PascalCase error keys, were found only by the checker.
- The Laravel capstone needed a route, a gate, a controller action and a board, because the contract is Laravel's default; the Spring Boot capstone needed the board and one security rule.
- Every result is placed on one of four levels of evidence (syntax-checked, compiled, run, checked against the contract), and only the last proves that the phone app will work; a move between versions moves data, not only code, and ends with the checker.
- The public board shows order numbers and first names only (**Figure 16.3**), in Razor, Blade and Thymeleaf.
- The comparison (**Table 16.3**) shows that the framework whose defaults match the contract needs the least adaptation, and that the rules, the contract and the checker are the durable parts of the system.

16.14 Exercises

1. Using Equation (16.1), compute by hand the totals of: (a) 2.99 kg of ironing only at Rp 6.000 with pickup; (b) 12.5 kg of wash, dry and fold at Rp 8.000 with pickup; (c) 3 blankets at Rp 25.000 with pickup; (d) 0.5 pairs of shoes at Rp 45.000.
2. Add two checks to `check_api.php`: `GET /api/customers?phone=` with the new customer's phone, written with dashes, finds exactly that customer; and `GET /api/services/S9999` answers 404 with a message.
3. The checker sends `outlet_id`, which only Laravel needs. What would happen if a future version of the ASP.NET Core capstone rejected unknown fields? Where would you change the design, and why?
4. Bu Wati wants the board to show, next to each finished order, how long it has been waiting, for example "2 h". Describe the change in all three frameworks, and say where the calculation belongs.
5. A colleague proposes to show the last four digits of the phone on the board, "so customers with the same first name can tell their orders apart". Argue for or against, and propose an alternative.
6. Write the Spring Boot counterpart of `CapstoneTest.test_the_board_shows_first_names_only` as a `@SpringBootTest` with `MockMvc`.
7. The ASP.NET Core capstone's memory build passed the checker, but the EF Core build did not run. List three kinds of bug that could exist only in the EF Core build, and how the checker would reveal each.

16.15 Solutions to the Exercises

1. (a) $h = 299$, per kg, billed $\max(299,300) = 300$; subtotal $\lfloor (6000 \cdot 300 + 50)/100 \rfloor = 18000$; below Rp 100.000 with pickup: $T = 28000$. (b) $h = 1250$, billed 1250, subtotal 100000; not below Rp 100.000, so $T = 100000$. (c) Items: $h = 300$, billed 300, subtotal 75000, with pickup $T = 85000$. (d) Pairs: $h = 50$ rounds up to 100, subtotal 45000, $T = 45000$.

2. After the customer is created:

```
[$s, $j] = call('GET', '/api/customers?phone='
    . substr($phone, 0, 4) . '-' . substr($phone, 4));
check('GET /api/customers?phone= finds the new customer',
    $s === 200 && count($j['data'] ?? []) === 1
    && ($j['data'][0]['id'] ?? 0) === ($customer['id'] ?? -1));
[$s, $j] = call('GET', '/api/services/S9999');
check('GET /api/services/S9999 -> 404 with a message',
    $s === 404 && is_string($j['message'] ?? null));
```

All three servers reduce the phone in the query to digits before searching, so the dashes do not matter.

3. The order check would fail with 422 (or 400) on `outlet_id`, and Raka's app, which may also send it, would break the same way. Two designs are sensible: make `outlet_id` part of the contract for all three servers (the shop has three outlets, so an order should record where it was taken), or remove it from the checker and from Laravel's request. The first is better, because it reflects the business; either way the decision belongs in the contract, not in one server.

4. Add the waiting time to what the board shows, computed from `finished_at` and the current time of the shop's clock: in the controller, or better in a small method of the service (`waitingSince`) so that it is testable with a fixed clock. The views print it: `@(...)` in Razor, `{{ }}` in Blade (Laravel's `diffForHumans()` does the formatting), `th:text` in Thymeleaf. The calculation belongs in the service or in a small helper, not in the template, because it depends on the clock.

5. Against: four digits of a phone number, next to a first name and the time a customer comes to the shop, help anyone who wants to identify or contact a stranger, and the benefit is small. The order number already distinguishes orders; printing it large on the receipt, and calling out numbers, solves the problem without showing any personal data.

6. In `src/test/java/com/profitina/laundry/web/BoardTest.java`:

```
@SpringBootTest
@AutoConfigureMockMvc
class BoardTest {

    @Autowired
    MockMvc mvc;

    @Test
    void theBoardShowsFirstNamesOnly() throws Exception {
        mvc.perform(get("/board"))                      // no login needed
           .andExpect(status().isOk())
           .andExpect(content().string(containsString("Sari")))
           .andExpect(content().string(
                   not(containsString("Wulandari"))));
    }
}
```

`DemoData` gives Sari Wulandari a finished order, so her first name appears in the list of orders ready to collect. `containsString` and `not` come from Hamcrest, which the test starter includes.

7. (a) A query that EF Core cannot translate to SQL, such as a method call inside `Where`: the checker's request fails with 500, and the check for that endpoint fails. (b) A missing `Include` of payments or of the service: the order comes back with an empty `payments` list or without a service name, and the payment check (`payments[0].method qris`) fails. (c) Times stored and read back with a different kind or precision in SQLite: the check for the time format fails. Each failure points to one endpoint and one field, which is where to look.

References

[1] The PHP Group, "HTTP context options" and "$http_response_header," PHP Manual. Accessed: Sep. 25, 2026. [Online]. Available: https://www.php.net/manual/en/context.http.php

[2] IEEE, "IEEE Standard for Floating-Point Arithmetic," IEEE Std 754-2019, 2019.

[3] Microsoft, "Model validation in ASP.NET Core MVC and Razor Pages" and "Handle errors in ASP.NET Core," ASP.NET Core 10 documentation. Accessed: Sep. 25, 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/mvc/models/validation

[4] Microsoft, "PasswordHasher Class," .NET API reference, and dotnet/aspnetcore source, `PasswordHasher.cs`. Accessed: Sep. 25, 2026. [Online]. Available: https://learn.microsoft.com/dotnet/api/microsoft.aspnetcore.identity.passwordhasher-1

[5] Microsoft, "ASP.NET Core fundamentals: Minimal APIs, controllers, and configuration," ASP.NET Core 10 documentation. Accessed: Sep. 25, 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/

[6] Laravel, "Routing: Route Model Binding and Middleware," Laravel 13.x documentation. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/routing

[7] Laravel, "Laravel Sanctum: API Token Authentication and Testing," Laravel 13.x documentation. Accessed: Sep. 25, 2026. [Online]. Available: https://laravel.com/docs/13.x/sanctum

[8] Broadcom, "Spring Security Reference: Servlet Applications," version 7.0. Accessed: Sep. 25, 2026. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/