Lecture 1 — From Web Programming to Frameworks

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 1

Lecturer: Irfan Subakti — yifana@gmail.com

Today we find out why frameworks exist at all. We follow one small laundry whose hand-written website began to hurt, name the eight jobs that every web request needs, build the same price list without a framework and with one, and set up the workbench for the whole semester. Book: Chapter 1, Sections 1.1–1.11.

What This Meeting Covers

1. The problem — Bu Wati's price list, twenty PHP files and one wish: one change in one place.

2. The idea — eight steps per request, library versus framework, Model–View–Controller.

3. The three frameworks — ASP.NET Core 10, Laravel 13 and Spring Boot 4, and why these three.

4. The workbench — VS Code, the .NET 10 SDK, PHP 8.5 with Composer, JDK 25 with Maven, SQLite and MySQL.

5. Without and with — the price list by hand in C#, PHP and Java, then with ASP.NET Core minimal APIs.

The running example of the whole course is Profitina Laundry: customers, orders, services and payments. The same features are built in each framework, so that at the end we can compare them fairly. Book: Chapter 1.

Where We Begin: Bu Wati's Price List

Profitina Laundry opened in Rungkut, Surabaya, in 2019. Today: three outlets, fourteen people, twelve services from Rp 8.000 per kilogram to Rp 45.000 per pair of shoes.

A student built the shop a website the way a first web course teaches: one PHP file per page, each opening the database, running a query and printing HTML with `echo`.

Then the shop grew. A new address had to be typed into five files and one was forgotten; a price changed on one page but not on another; a strange URL was printed back into the page unescaped.

When Bu Wati asked "can customers see whether their laundry is ready?", nobody dared to touch the code.

Her wish is modest and very common: one change should be made in one place; every page should look the same without copying; every value should be made safe automatically; every new feature should have an obvious home. That wish is exactly what a framework delivers. Book: Section 1.1.

Eight Steps for Every Request

1. Wait for a connection and read the request. **2. Route**: decide which code is responsible for this path and method.

3. Extract the variable parts of the path (the code `S0004`) and convert them. **4. Fetch** the data from the store.

5. Decide by the rules of the business. **6. Render** HTML or JSON, escaping every value from outside.

7. Respond with a status code, headers and bytes. **8. Survive**: log, and send a polite error page instead of crashing.

> **KEY INSIGHT.** Only steps 4 and 5 are about *laundry*. The other six are the same for a clinic, a school or a bank, and they are where most bugs live: a wrong `if` in routing, an id cut one character too early, a value printed without escaping.

Ask the room: which of the eight steps went wrong in each of Bu Wati's three incidents? The forgotten address is a missing shared layout (render), the stale price is data in two places (fetch), the strange URL is missing escaping (render). Book: Section 1.2.1.

Library or Framework: Who Calls Whom?

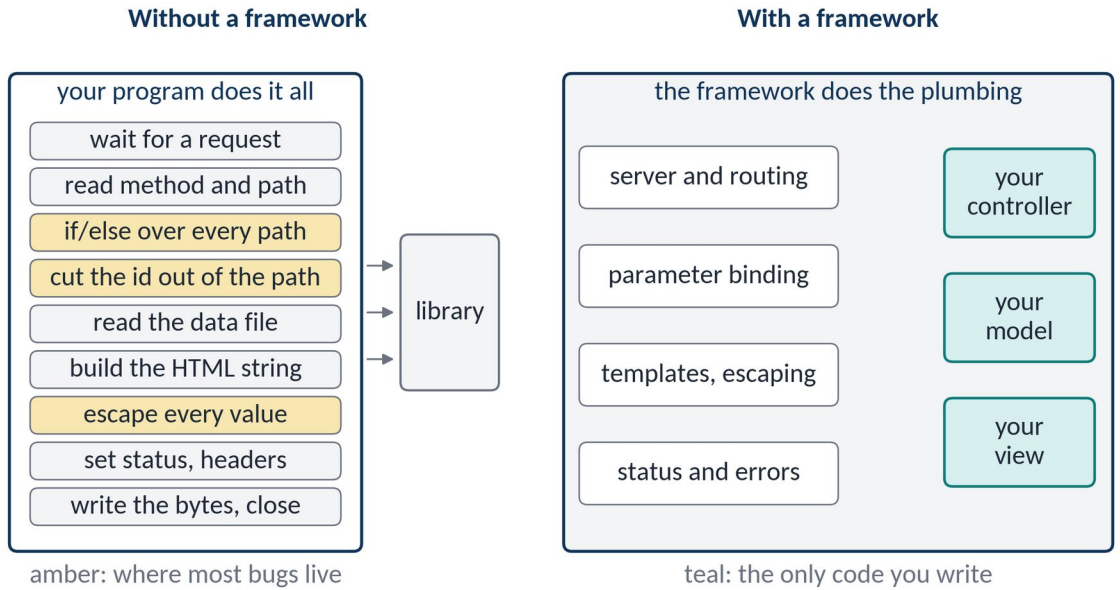


Figure 1.1 A library is called by your program; a framework calls your program. With a framework, the only code you write is the part that is about your business.

*Figure 1.1: a **library** is a set of ready-made routines that your program calls when it wants. A **framework** is an almost complete program with holes in it, and you fill the holes. Johnson and Foote described this in 1988: the control flow belongs to the framework. It is often called the*

Hollywood principle, "don't call us, we'll call you", or ***inversion of control****. Try it on the programs of today:* `HttpListener` *and* `fgetcsv` *are libraries our code calls; after* `app.Run()` *ASP.NET Core owns the loop and calls our handlers. Book: Section 1.2.2.*

Model, View, Controller

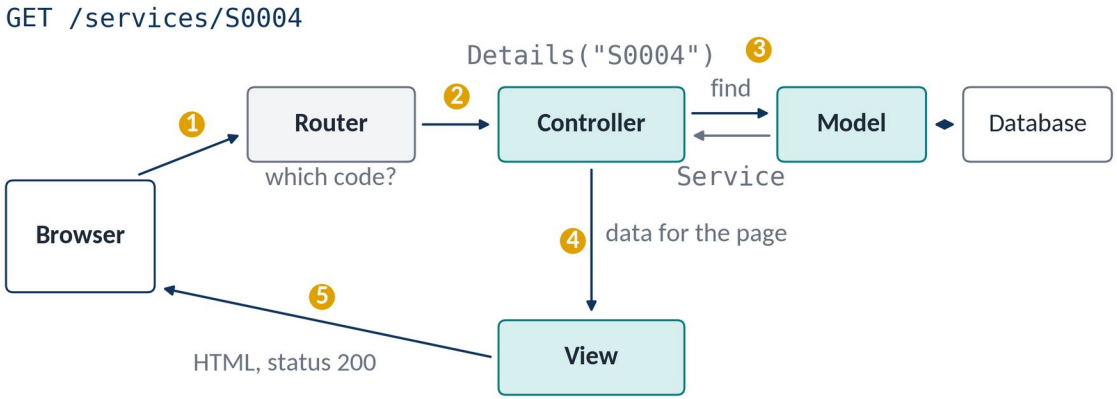


Figure 1.2 One request through Model–View–Controller: the router chooses the controller action, the controller asks the model and passes the result to a view, and the view's HTML is the response.

*Figure 1.2: the **model** holds data and business rules, the **view** turns data into HTML, the **controller** handles one request and connects the two. A **router** picks the controller action from the path and method. Rule of thumb for the whole course: keep controllers thin. Book: Section 1.2.3.*

What a Framework Gives, and What It Costs

Gives: a web server, routing, parameter binding, a template engine with automatic escaping, an ORM for the database, validation, logins, logging and testing tools.

Gives: a fixed place for every kind of code, so that a new team member knows where to look.

Costs: a learning curve, conventions you must follow, and “magic” that hides what happens until you understand it.

Costs: dependence on the framework’s release cycle; a major version every year or two means planned upgrades.

The best defense against the “magic” is to know the eight steps: when something fails, ask which step failed and read what the framework logs for it. Book: Section 1.2.4.

Why ASP.NET Core, Laravel and Spring Boot

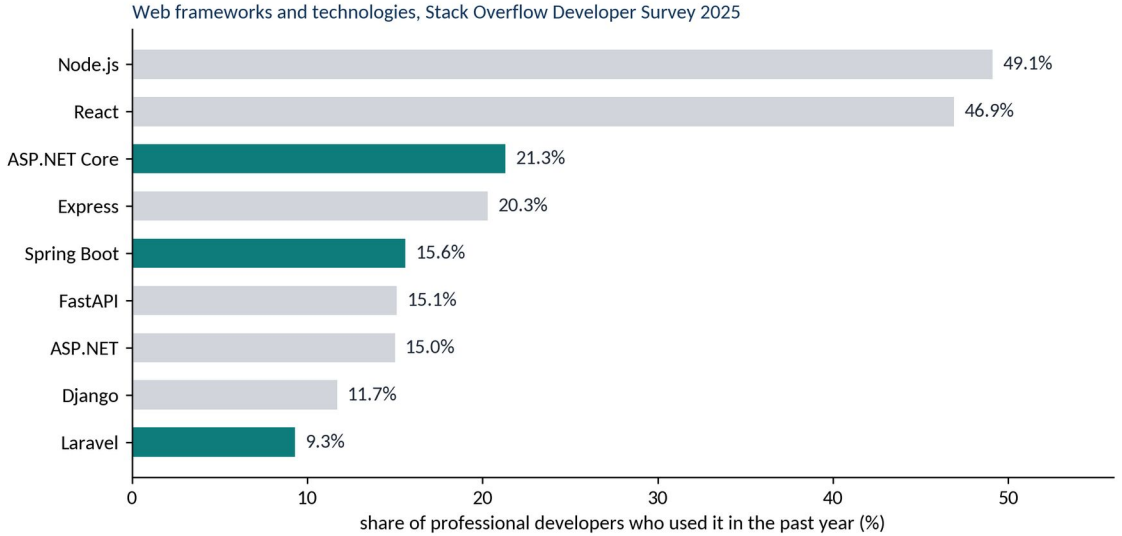


Figure 1.3 Web frameworks and technologies used by professional developers in the Stack Overflow Developer Survey 2025 (19,460 respondents to this question); the three frameworks of this book are in teal.

Figure 1.3: ASP.NET Core was used by 21.3% of the professional developers, Spring Boot by 15.6% and Laravel by 9.3%. Together they stand for the three big families of server-side work: the Microsoft world, PHP hosting and the Java enterprise. Book: Section 1.3.

The Three Side by Side

Table 1.1 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
Version (Sept. 2026)	10.0 LTS	13.x	4.1.1
Project tool	`dotnet` CLI	Composer, `php artisan`	Maven, Spring Initializr
Templates	Razor (`.cshtml`)	Blade (`.blade.php`)	Thymeleaf (`.html`)
ORM	Entity Framework Core	Eloquent	Spring Data JPA
Chapters	3–6	7–10	11–14

Table 1.1: every row becomes familiar during the semester. ASP.NET Core is by Microsoft, Laravel by Taylor Otwell, Spring Boot by VMware, now Broadcom; all three are open source. Packages come from NuGet, Packagist and Maven Central. Book: Section 1.3 and Section 1.9.2.

Setting Up the Workbench

VS Code with C# Dev Kit, PHP Intelephense and Laravel, the Extension Pack for Java and the Spring Boot Extension Pack; on a slow computer install only the ones for the framework of the week.

The .NET 10 SDK (not only the runtime): it brings `dotnet new`, `dotnet build`, `dotnet run`.

PHP 8.5 and Composer via `php.new` on macOS, Linux and Windows, then the `laravel` installer.

JDK 25 and Maven (or the Maven wrapper in each project); **SQLite** for development, **MySQL 9.7 LTS** for a shared server.

Open a new terminal after every installation, so that the new commands are found. Book: Sections 1.4.1–1.4.5.

Checking Everything

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

Each command must answer with a version. `dotnet --list-sdks` printing nothing means only a runtime was installed.

`java -version` answering 17 or 21 means an older JDK is first on the `PATH`: set `JAVA_HOME` and put its `bin` folder at the *front*.

The book's machine answered, for example, `10.0.112` for the SDK with `Microsoft.AspNetCore.App 10.0.12`. Book: Section 1.4.6.

The Price List Without a Framework

The same small site three times, reading `data/Service.csv`: C# on `HttpListener`, PHP on the built-in server, Java on `com.sun.net.httpserver`.

Each program does all eight steps by hand: 110 lines of C#, 66 of PHP and 108 of Java.

The PHP script is at once router, controller, model and view; the comments (1) to (6) in the C# file mark the jobs a framework would take over.

Each version made its own small decisions about trailing slashes, unknown methods and escaping, and they did not agree.

Show the three files side by side in the editor, and count with the class how many lines are about laundry. It is a handful. Book: Sections 1.5.1–1.5.3.

The Same Price List With ASP.NET Core

Listing 1.1 The heart of `aspnet/Program.cs`, the price list with ASP.NET Core minimal APIs (26 lines in full).

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

// ... services are read from ../data/Service.csv ...

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);
```

Listing 1.1: `MapGet` is the route, `{id}` is extracted and bound to the parameter `string id`, the list becomes JSON by itself, and `Results.NotFound` sets the status code. The project file uses the `Microsoft.NET.Sdk.Web` SDK and targets `net10.0`. Book: Section 1.6.1.

Running It

```
cd Chapter01/aspnet
dotnet run --urls http://127.0.0.1:4013
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}
```

Kestrel *is the web server built into ASP.NET Core; the JSON and the* `Content-Type` *header came from the framework. This program was built and run with .NET 10 for the book, and the output above is its real output. Book: Section 1.6.2.*

Four Programs, One Test Script

```
===== 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
```

The same five requests were sent to all four programs. The hand-written ones answered a `POST` with 411 (C#) or 405 (PHP, Java); the framework answered 405, consistently.

In JSON the `<b>` is harmless data; in HTML the hand-written programs had to escape it themselves, every single time.

The block above is the ASP.NET Core part of the real comparison run; the book prints all four. Book: Section 1.7.

A First Real Project in Each Framework

```
dotnet new mvc -o FirstMvc
laravel new first-laravel
curl -o first-spring.zip "https://start.spring.io/starter.zip?..."
```

`dotnet new mvc` creates a full MVC project with controllers, Razor views and a layout; `dotnet run` starts it.

`laravel new` creates a Laravel application; `php artisan serve` starts it. Spring Initializr gives a Maven project with the `web` and `thymeleaf` dependencies.

The Laravel and Spring projects need Packagist and Maven Central; run them on your own machine, following the chapter's README. Book: Section 1.6.3.

One More Route: Framework Versus Hand

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

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

`{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 `/services/under/cheap` no route matches and the framework answers 404.

The hand-written C# version needs its own `else if` branch with `StartsWith`, `int.TryParse`, a `StringBuilder` loop and a call to `Escape` for every description, about a dozen lines.

In an `if` chain a more specific test must come before a more general one; the router of a framework takes care of that ordering for us.

The output is from a real run of exactly this change for `/services/under/7000`: only ironing, at Rp 6.000, is under the limit. Ask the room which of the eight steps the constraint `:int` takes over: route and extract. Book: Exercises 1.3 and 1.4 with their solutions in Section 1.13.

Making It Better: What We Still Did by Hand

The data file — a CSV parsed with a regular expression. Chapter 5 moves it into SQLite through Entity Framework Core.

HTML pages — Chapter 3 builds them with controllers, Razor views, a shared layout and automatic escaping.

Forms and validation — Chapters 3 and 6. **Logins and roles** — Chapters 6, 10 and 14. **Automated tests** — Chapters 10 and 14.

Chapter 2 first builds a tiny framework of its own, about three hundred lines, with a router, a DI container, an ORM and a template engine. It is not for real use; it takes away the mystery. Book: Section 1.8.

Using It for Real

Start from the data file: a tailor lists “hemming trousers”, a small clinic its consultation fees, a fruit and durian orchard its fruits per kilogram by season. Keep the three columns and the programs work unchanged.

In practice the choice of framework is made by the people and the place: C# teams pick ASP.NET Core, agencies on shared PHP hosting pick Laravel, a Java enterprise picks Spring Boot.

All three can build everything in this course; Chapter 15 compares them with measurements after we have built the same features in each. Book: Section 1.9.

Common Mistakes

A runtime instead of the SDK; **an old terminal** opened before the installation; **two Javas** on the `PATH`.

The wrong folder: the programs read `../data/Service.csv` relative to where they start. **A busy port:** a previous run is still alive; stop it with **Ctrl+C**.

Thinking the framework is magic: ask which of the eight steps failed instead of making random changes.

> **WARNING.** HTML built by string concatenation must escape every outside value, every time. Let a template engine do it, and treat `Html.Raw`, `{!! !!}` or `th:utext` as places to review.

Each of these costs a beginner an afternoon; naming them costs a minute. Book: Section 1.10.

Summary: Five Things to Remember

1. Every request goes through eight steps: wait, route, extract, fetch, decide, render, respond, survive. Only fetch and decide are about the business.
2. A library is called by your code; a framework calls your code (Figure 1.1).
3. Frameworks organize code as Model–View–Controller with a router in front (Figure 1.2); keep controllers thin.
4. This course uses ASP.NET Core 10, Laravel 13 and Spring Boot 4 on VS Code, .NET 10, PHP 8.5, JDK 25, SQLite and MySQL.
5. By hand the price list took 110, 66 and 108 lines; with minimal APIs it took 26, and the framework made the small decisions consistently.

If only one sentence stays with you, let it be Bu Wati's: one change in one place. Every chapter that follows is a way of making that true for one more kind of change: a price, a page layout, a rule, a login, a test. When a framework feature seems strange, ask which of her four wishes it serves. Book: Section 1.11.

Next Time

Chapter 2 — Foundations Shared by Modern Frameworks. MVC, routing, ORM, dependency injection and templates, by building a tiny framework of our own.

Before then: install the workbench, run the four price-list programs, and work the chapter's exercises; the solutions are in the book.

A good warm-up exercise: add the route `/services/under/{max:int}` *to the minimal API. It takes two lines; the hand-written C# version needs a dozen. Book: Sections 1.12 and 1.13.*

Lecture 2 — Foundations Shared by Modern Frameworks

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 2

Lecturer: Irfan Subakti — yifana@gmail.com

Today we open the box. We name the four ideas that every modern web framework is built on (routing, dependency injection, object-relational mapping and templates), build each of them by hand in a tiny framework called MiniFrame, and then point at the same four ideas in ASP.NET Core, Laravel and Spring Boot. Book: Chapter 2, Sections 2.1–2.9.

What This Meeting Covers

1. The question — how does a framework call a function it has never seen?

2. The four foundations — routing, DI, ORM and templates along one request.

3. MiniFrame — all four in 226 lines of C#, with a demo that shows each foundation at work.

4. The real three — the same price list in Laravel and Spring Boot, and a table that translates between all of them.

5. What MiniFrame leaves out — and why real frameworks are large.

MiniFrame 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. Book: Chapter 2.

Where We Left Off

Chapter 1: the price list by hand took 66 to 110 lines; with ASP.NET Core minimal APIs, 26.

But *how*? When `MapGet("/services/{id}", (string id) => …)` receives `/services/S0004`, something splits the path, compares it with the template and passes `"S0004"` to a function it has never seen.

Bu Wati's nephew **Dimas** opened the three frameworks and found `[HttpGet("{id}")]`, `Route::get('/services/{id}', …)` and `@GetMapping("/{id}")`. Three syntaxes, one idea.

In each, a controller received a repository in its constructor, and no code anywhere created it. Again three syntaxes, one idea.

Without an answer a framework feels like magic, and magic is impossible to debug. This chapter names the four ideas once, builds each by hand, and then finds them in all three frameworks. Book: Section 2.1.

The Four Foundations Along One Request

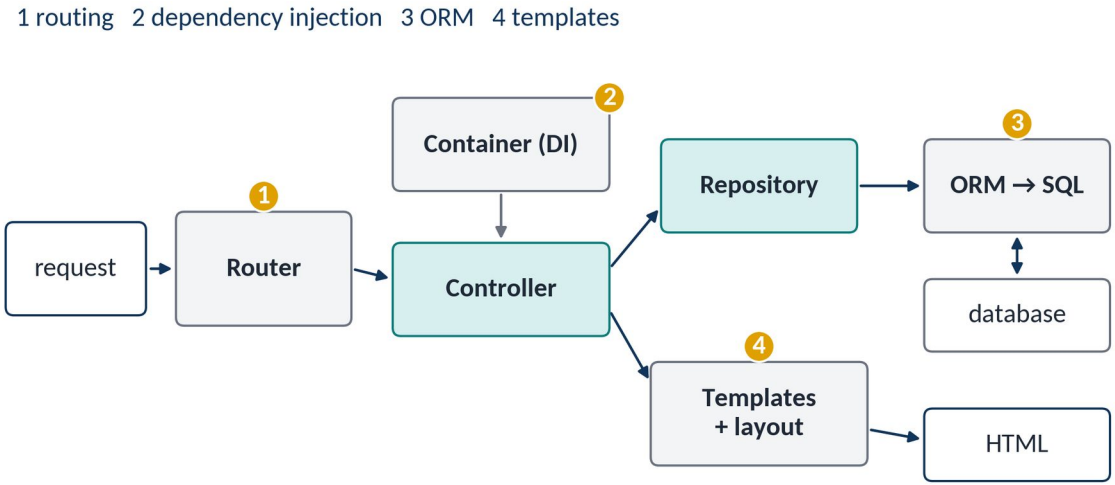


Figure 2.1 The four foundations along one request: routing chooses the action, dependency injection builds the controller, the ORM turns objects into SQL and rows into objects, and the templates turn objects into HTML.

Figure 2.1: the router (1) chooses the action; the container (2) builds the controller and everything it needs; the controller asks a repository for data, and the ORM (3) turns that into SQL and the rows into objects; the template engine (4) puts the data into a page, the page into the layout, and sends HTML back. Book: Section 2.2.

Four Definitions

> **DEFINITION.** A **route** pairs an HTTP method and a URL *template* such as `/services/{id}` with an action. A **router** keeps the routes in a route table, finds the one that matches the path and extracts its *parameters*. A **constraint** such as `{max:int}` restricts what a parameter may match.

Dependency injection: a class does not create what it depends on; it declares it in the constructor and receives it. A **DI container** builds the whole graph from *registrations* and decides each object's **lifetime**.

Object-relational mapper (ORM): reads a row into an object by matching column and property names, and turns a condition written in the language into SQL. A **migration** is one versioned change to the schema, in code.

Template engine: fills *holes* in a page with values, **escaped by default**, repeats blocks for lists, and puts every page into a shared **layout**.

Write the four words on the board and leave them there all semester. Every new framework feature we meet will land under one of them. Book: Section 2.2.

MiniFrame: a Tiny Framework in 226 Lines

`Router.cs` — a route table, template matching and parameter extraction (55 lines).

`Container.cs` — registrations, two lifetimes and constructor injection through reflection (45 lines).

`Mapper.cs` — rows to objects by name, and a C# condition to the text of an SQL `WHERE` clause (90 lines).

`Template.cs` — holes, loops and a layout, with escaping by default (36 lines). On top: `Laundry.cs` (the application) and `Program.cs` (registrations, routes and a small HTTP server).

MiniFrame is plain C# on .NET 10 with no packages, so it was built and run here; every MiniFrame output shown today is its real output. Book: Section 2.3.

Routing

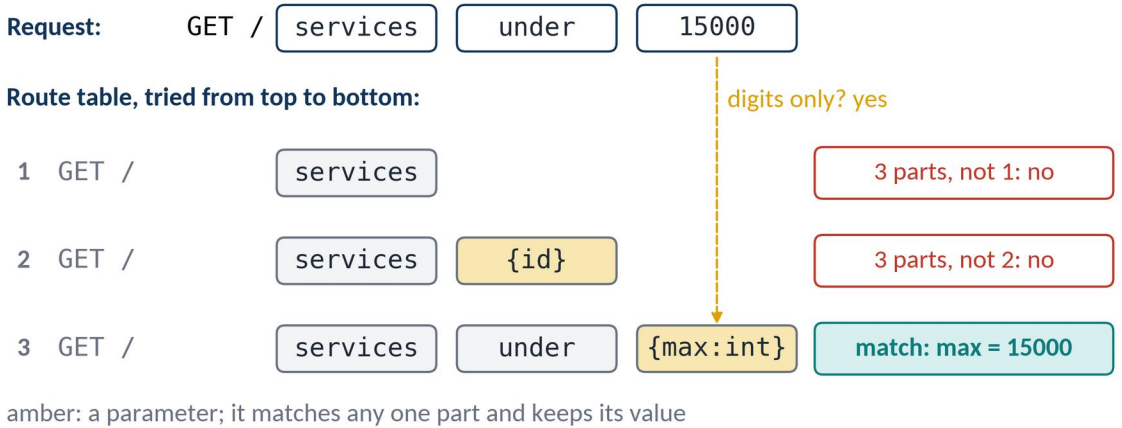


Figure 2.2 Matching `GET /services/under/15000` against the route table: the segment counts rule out the first two routes, and the third fits because its parameter `{max:int}` accepts digits.

Figure 2.2: the router splits the path at slashes, compares the number of segments, then compares each literal segment and checks each parameter against its constraint. The extracted values go into a small dictionary that the action receives. MiniFrame tries routes in order, so a general route registered before a specific one would swallow its requests. Book: Section 2.3.1.

Dependency Injection

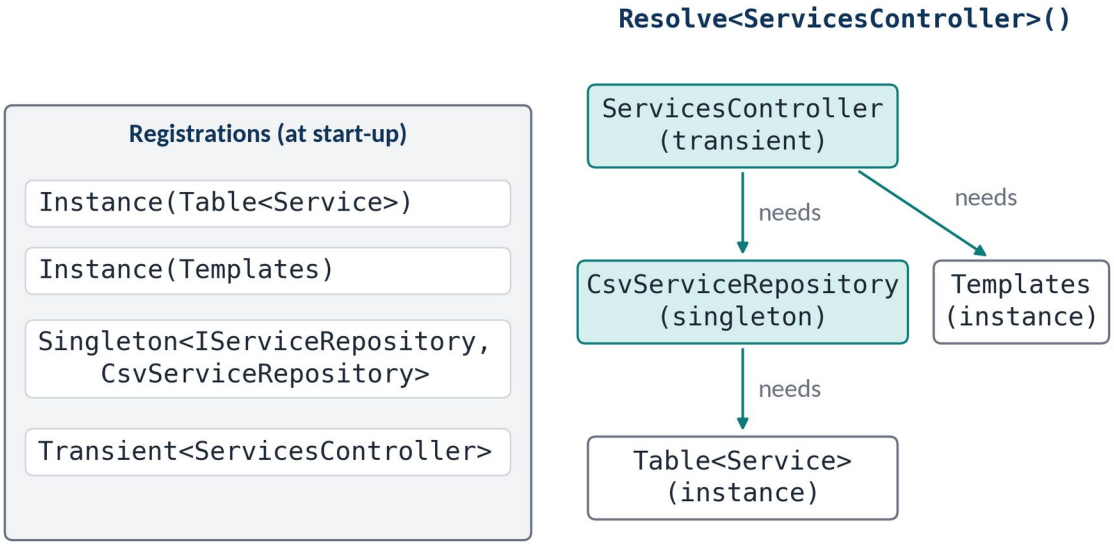


Figure 2.3 The registrations made at start-up, and the graph the container builds when a controller is requested. Only the transient controller is new on every request.

Figure 2.3: at start-up we register "for `IServiceRepository` use `CsvServiceRepository`, as a singleton" and "build `ServicesController` as a transient". When a controller is needed, the container reads its constructor by reflection, builds each parameter first, and calls it. Book: Section 2.3.2.

Lifetimes: May Two Requests Share This Object?

Singleton — one object for the whole application: a price catalog, a configuration.

Scoped — one object per request: a database context, the current user. MiniFrame leaves it out; all three real containers have it.

Transient — a new object every time it is asked for: a light controller.

> **WARNING.** A singleton that holds per-request data (the current user, a basket) mixes one visitor's data into another visitor's page. Ask of every registration: *may two requests share this object?*

Depending on an interface instead of a concrete class is what makes the implementation replaceable: a CSV today, SQLite tomorrow, a test double in Chapter 10, without touching the controller. Book: Section 2.3.2.

Object-Relational Mapping

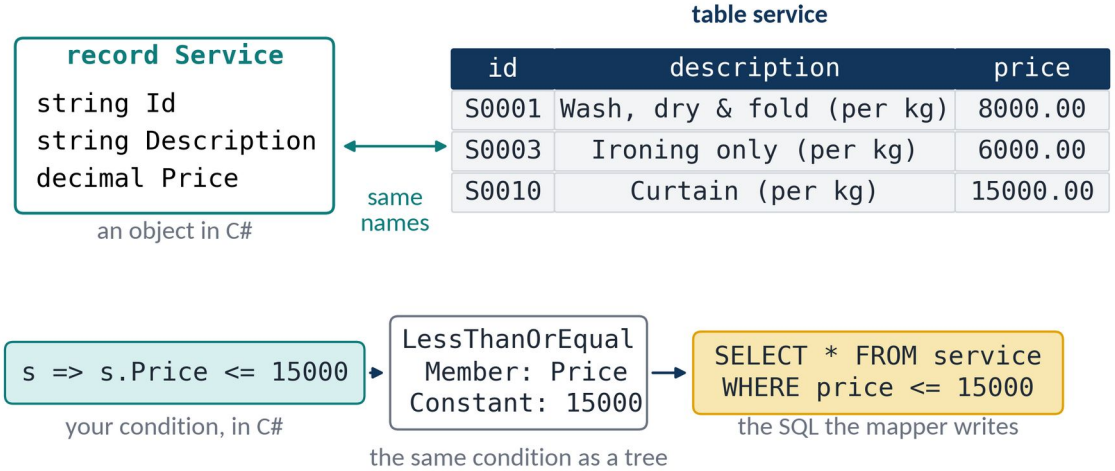


Figure 2.4 Above, a row of the table `service` becomes a `Service` object because column names equal property names. Below, the condition `s => s.Price <= 15000` is read as an expression tree and printed as SQL.

Figure 2.4: in C#, a lambda passed as `Expression<Func<Service, bool>>` is not compiled into code but kept as a tree of nodes that the mapper can walk: parameter, property `Price`, operator `<=`, constant 15000. Printing the tree gives `WHERE price <= 15000`. EF Core does exactly this, on a much larger scale. Book: Section 2.3.3.

Templates: Holes, Loops and a Layout

Listing 2.1 `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>
```

`{{ x }}` is an **escaped hole**: `<`, `>`, `&` and quotes become harmless text. `{{{ x }}}` is the **raw hole**, used only for the layout's `{{{ body }}}`.

`{{#each rows}}` repeats the block for every row; the shared `layout.html` holds the head and the header once for every page.

Listing 2.1 is the whole list page. Escaping is the default, and raw output is the exception that must be justified: that single rule removes a whole class of attacks. Book: Section 2.3.4.

From Model to Page

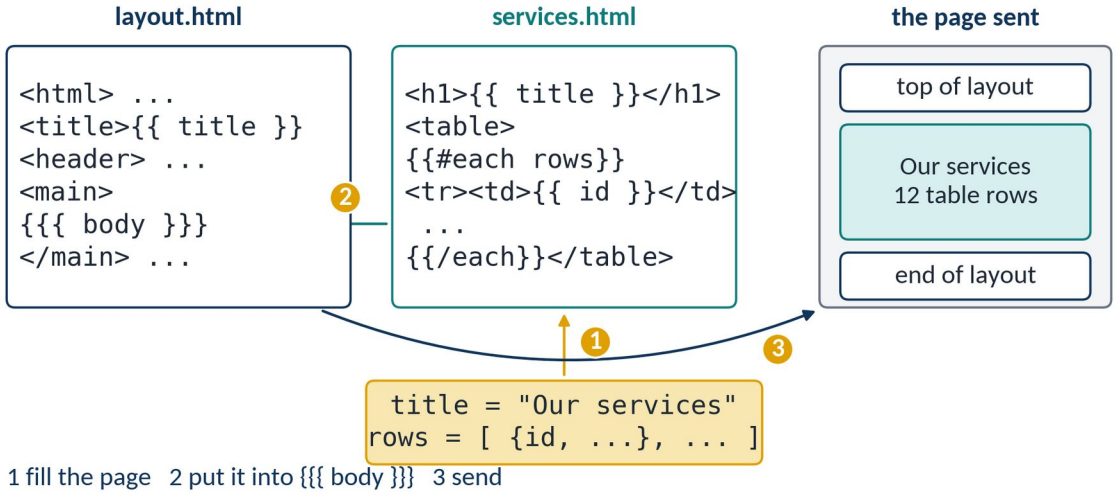


Figure 2.5 From model to page: the page template is filled with the model, the result is inserted into the layout, and the whole page is sent.

Figure 2.5: two passes. First the page template is filled with the model; then its HTML goes into the layout where `{{{ body }}}` *is. Razor's* `@RenderBody()`*, Blade's* `@yield` *and Thymeleaf's fragments are the same idea. Book: Section 2.3.4.*

The Application on Top

`Laundry.cs` holds the model `Service`, a repository and the `ServicesController`: no routing, no parsing, no SQL text, no HTML escaping.

`Program.cs` registers the classes, maps three routes, and either runs an offline demo (`--demo`) or starts a small HTTP server.

That split is the whole point: the application is only the part that is about laundry, exactly the picture of Chapter 1: the framework calls our code, and this time we built the framework ourselves.

Ask the room to find one line in `Laundry.cs` that would change if the shop moved from a CSV file to SQLite. There is none: the repository behind the interface changes, the controller does not. Book: Section 2.3.5.

Running the Demo

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

Every foundation reports what it did. The repository is built once (singleton), the controller on every request (transient). The book's full run also shows `/services/under/cheap` rejected by the `int` constraint with 404, and `/services/%3Cb%3E` answered with `<b>`, escaped. Book: Section 2.4.

The Same Routes in Laravel

Listing 2.2 `routes/web.php` in Laravel: three routes, each pointing to a controller method; `whereNumber` is the constraint.

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

Listing 2.2: one central file, one line per route, a controller class and a method name. Laravel, like MiniFrame, is sensitive to order here: `under` comes before `{id}`. In ASP.NET Core and Spring the same routes are attributes or annotations on the controller's methods, with the constraints of Table 2.1. Book: Section 2.5.1.

The ORM in Spring Data: a Query From a Name

Listing 2.3 The heart of `ServiceRepository.java`: a Spring Data repository. The query is derived from the method's name.

```
// 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);
}
```

Three ways to write one query: an expression tree (EF Core, MiniFrame), a fluent builder `Service::where('price', '<=', $max)` (Eloquent), a method name (Spring Data).

The Eloquent model states only what differs from Laravel's conventions: table `service`, a string key, no timestamps.

Listing 2.3: Spring Data splits the name into `find`, `By`, `Price`, `LessThanEqual` and builds `WHERE price <= ?`. The entity `Service` is a class with `@Entity`, `@Table(name = "service")` and `@Id`. The Laravel and Spring code needs Packagist and Maven Central, so it was written against the official docs and checked for syntax here, not run. Book: Section 2.5.3.

Templates in Blade and Thymeleaf

Listing 2.4 Part of `resources/views/services/index.blade.php`: the list page, filling the layout's content slot.

```
@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
```

Listing 2.4: `{{ }}` is the escaped hole, `@foreach` the loop, `@extends('layouts.app')` with `@yield` the layout. In Thymeleaf the same page is plain HTML with attributes: `th:each="s : ${services}"` loops and `th:text="${s.description}"` is the escaped hole, so a designer can open the file in a browser. Book: Section 2.5.4.

One Feature, Three Frameworks

Table 2.1 The four foundations in MiniFrame and in the three frameworks of this book.

	MiniFrame	ASP.NET Core	Laravel	Spring Boot
Constraint	`{max:int}`	`{max:int}`	`whereNumber('max')`	`{max:\\d+}`
Injection	constructor	constructor	constructor	constructor
Row to object	`Table<Service>`	EF Core entity	Eloquent model	JPA `@Entity`
Escaped hole	`{{ x }}`	`@x`	`{{ $x }}`	`th:text`
Raw hole	`{{{ x }}}`	`@Html.Raw(x)`	`{!! $x !!}`	`th:utext`
Layout	`{{{ body }}}`	`@RenderBody()`	`@yield`	`th:replace`

Table 2.1 is the translation table of the course. Learn the idea once, and each syntax becomes a detail. Spring registers classes by scanning annotations such as `@Repository` and `@Service` instead of listing them. Book: Section 2.5.5.

Making It Better: What MiniFrame Leaves Out

Parameters in SQL. MiniFrame prints values into the SQL text (`WHERE id = '<b>'`); real ORMs send `WHERE id = @p0` and the value separately, so no value can change the meaning of a statement.

Route precedence. Real routers compile the table into a tree and rank templates, so the order of registration stops mattering.

Scoped lifetime and disposal, **concurrency** for thousands of simultaneous requests, **compiled templates**, and **migrations** (Chapters 5, 9 and 11).

Each omission is a lesson about why real frameworks are large: the idea fits in 226 lines, the safety and the speed do not. Book: Section 2.6.

Using It for Real: the Four Questions

Plan every new feature with four questions: **Which route? What does the controller need injected? Which query? Which template?**

For Bu Wati's "is my laundry ready?": `GET /orders/{code}`; an order repository; one row by its code; a small status page.

The same four answers fit any framework, and any unfamiliar framework can be read through them.

Try the four questions on a business the class knows: a clinic's queue number, a fruit and durian orchard's harvest list. The answers are short, and once they are written down the code almost writes itself in whichever framework the team uses. The same questions also help to read someone else's project: find the route, follow it to the controller, look at its constructor, then at the query and the view. Book: Section 2.7.

Common Mistakes

Routes in the wrong order in an order-sensitive router: the general one swallows the specific one. **`new` inside a class the container builds**: it quietly defeats the container.

The wrong lifetime. SQL by concatenation, even through an ORM's raw escape hatch (`FromSqlRaw`, `DB::raw`, native queries).

Switching escaping off to "fix" `&` on a page: the cause is double escaping. **Learning syntax instead of ideas.**

The lifetime mistake is the dangerous one, because it passes every test with one user and fails only when two customers use the site at the same moment. Beginners often learn `@GetMapping`, `Route::get` and `[HttpGet]` as three unrelated facts; learn the idea once. Book: Section 2.8.

Summary: Five Things to Remember

1. All three frameworks share MVC and four foundations: routing, DI, ORM, templates (Figure 2.1).

2. A router matches method and path against templates, applies constraints and passes the values to one action.

3. A container builds objects from registrations; singleton, scoped and transient say who shares them.

4. An ORM maps rows to objects by name and a query in the language to SQL, always with parameters.

5. A template engine fills escaped holes, repeats blocks and uses a layout; raw output must be justified.

MiniFrame did all four in 226 lines; its omissions explain the size of the real ones. Book: Section 2.9.

Next Time

Chapter 3 — ASP.NET Core MVC I. The project, the middleware pipeline, routing, controllers and Razor views, with a shared layout and tag helpers.

Before then: run MiniFrame's demo, then work the chapter's exercises, including the one that registers a general route before a specific one.

Book: Sections 2.10 and 2.11.

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

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 3

Lecturer: Irfan Subakti — yifana@gmail.com

Today Bu Wati gets the website her customers will actually see: a home page, the services, a price quote, the outlets and polite error pages. We build it as LaundryMvc with ASP.NET Core 10 and nothing but the shared framework that comes with the SDK, and we meet the host, the middleware pipeline, both kinds of routing, controllers, model binding and Razor views. Book: Chapter 3, Sections 3.1–3.12.

What This Meeting Covers

1. The wish list — Bu Wati's eight wishes on the back of an order ticket.

2. Host and pipeline — `Program.cs`: services first, then middleware.

3. The model — records, the price rules, view models, a form model, data behind an interface.

4. Controllers and routing — the conventional route, attribute routes, constraints, model binding.

5. Razor views — a layout, partial views, sections, tag helpers; then the running site, checked address by address.

LaundryMvc was built and run here with .NET 10; the screenshots are of our own running application and the status codes come from a real run of the chapter's `verify.sh`. Book: Chapter 3.

Bu Wati's Website

A **home page** with the three most asked-for services; the **list of services** and a page for each; the services **up to a price**.

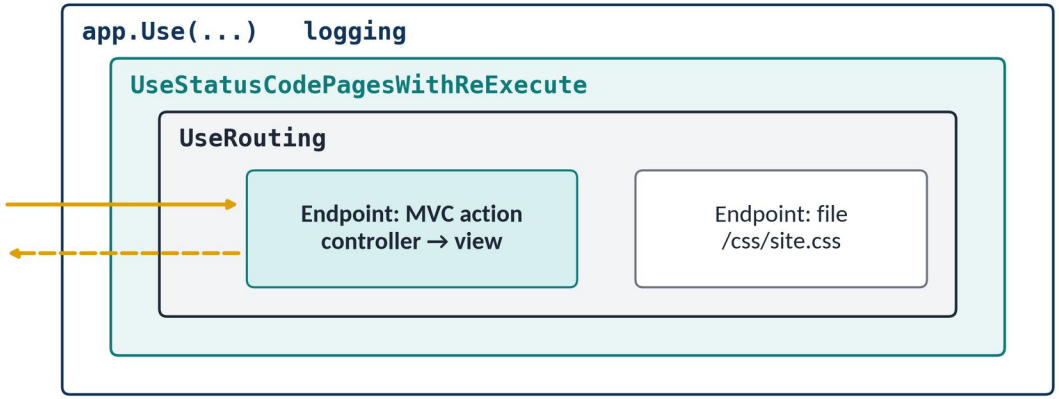
A **price quote** with the shop's rules: at least 3 kg for services charged by weight, items counted whole, pickup Rp 10.000 but free from Rp 100.000.

The **outlets**, an **about** page with opening hours, **no ugly error pages**, and for Dimas the service data as **JSON**.

Two rules learned the hard way: the price rules live in one place, and every page looks the same without copying a header.

Chapter 1 called the first rule "keep controllers thin"; Chapter 2 showed that the second is what a layout is for. The data are the same CSV exports as before, `data/Service.csv` and `data/Outlet.csv`; Chapter 5 moves them into a database. Book: Section 3.1.

The Host and the Pipeline

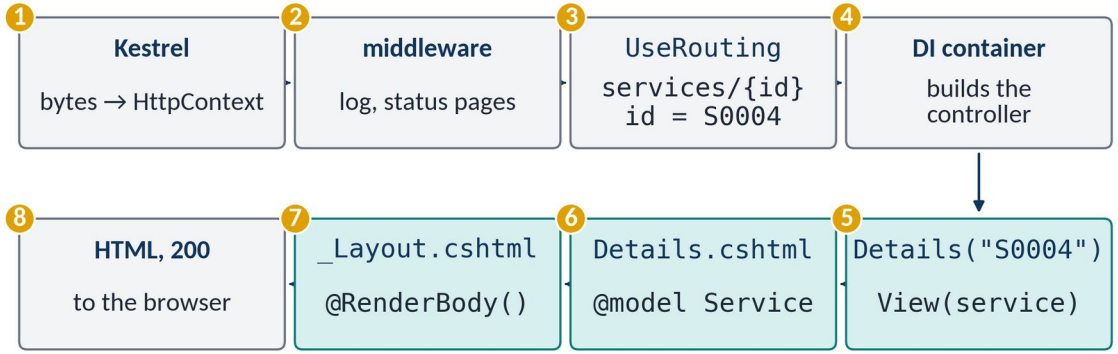


amber: the request goes in through every layer; the response comes back out through the same layers (dashed)

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.

*Figure 3.1: a **host** starts Kestrel, reads the configuration and sets up logging and the DI container. A **middleware** is a piece of code that receives the request, may do something, calls the next one, and may do something again on the way back. Middleware wraps middleware, like layers of an onion, so the order in which they are added is the order in which they act. Book: Section 3.2.1.*

One Request Through MVC



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.

Figure 3.2 is Chapter 2's picture made real. Kestrel turns bytes into an `HttpContext`; routing chooses `ServicesController.Details`; the container builds the controller with its catalog; the action returns a view; Razor fills it, puts it into the layout and writes HTML. Book: Section 3.2.2.

Creating the Project

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

The template gives `Program.cs`, the folders `Controllers`, `Models`, `Views` and `wwwroot`, `appsettings.json` and `Properties/launchSettings.json`.

We change little: the port becomes `http://localhost:4030` in the Development environment, and we add the folder `Data` and the CSV files.

`appsettings.json` sets the log levels: `Information` by default, `Warning` for the framework's own messages, so that our lines stand out.

*The **environment** (`Development`, `Production`) changes behaviour: in Development the detailed exception page is shown; in Production the friendly error page is. Book: Section 3.3.*

`Program.cs`: Services, Then Pipeline

Listing 3.1 `Program.cs`, shortened: the services are registered on the builder, then the pipeline is built on the application.

```
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) => { /* time and log the request */ });
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();
```

Listing 3.1: `AddControllersWithViews()` *registers everything MVC needs. The next two lines are MiniFrame's container in its real form: one* `CsvCatalog`*, built once by a function because its constructor needs a folder name, and one* `PriceCalculator` *that the container can*

build by itself. Both are singletons because both are safe to share. The first middleware wraps everything else, so it can time every request. Book: Section 3.4.

The Model: Data and Rules

Listing 3.2 The heart of `Models/Service.cs`: a service with its code, description and price, and three properties derived from the description.

```
// 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";
}
```

Listing 3.2: a C# `record` is a small immutable class with value equality. Beside it the model holds `Outlet`, `Money.Rp` (36000 becomes "Rp 36.000" in one place), and `PriceCalculator`, the shop's rules, which returns a quote together with an explanation of every rule it applied. Book: Sections 3.5.1 and 3.5.2.

View Models, a Form Model and an Interface

`HomeViewModel(Popular, ServiceCount, OutletCount)` — exactly what the home page shows, and nothing more.

`QuoteForm` — `ServiceId`, `Quantity`, `Pickup`, filled by **model binding**; its `Result` carries `[BindNever]`, so no URL can set it.

`ILaundryCatalog` — `Services()`, `FindService(id)`, `Outlets()`, `FindOutlet(id)`: what controllers may ask. `CsvCatalog` implements it on the two CSV files, read once.

The comment in the interface says it all: Chapter 5 swaps the CSV files behind it for a real database without touching a controller. That is dependency injection paying for itself. Book: Sections 3.5.3 and 3.5.4.

Conventional Routing

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

`/` means `Home/Index`; `/Outlets` means `Outlets/Index`; `/Outlets/Details/002` calls `Details("002")`. Matching ignores upper and lower case.

`OutletsController(ILaundryCatalog catalog)` receives its catalog through a **primary constructor**; `Details` returns `NotFound()` for an unknown code and `View(outlet)` otherwise.

A controller class must be public, not abstract, and end in `Controller`.

One pattern serves every controller that follows the convention: good for simple pages where the URL can mirror the code. Book: Section 3.6.1.

Attribute Routing: `ServicesController`

Listing 3.3 The first three actions of `Controllers/ServicesController.cs`: attribute routes and a constraint.

```
[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);
    }
```

Listing 3.3: the class prefix `services` is added to every action's template. `{max:int:min(1000)}` combines two constraints. `Under` reuses the view `Index`. The full controller adds `Quote` at `services/quote` and a JSON action at `/api/services/{id}`; a template starting with `/` ignores the prefix. Once a controller has attribute routes, the conventional route no longer reaches it. Book: Section 3.6.2.

What Actions Return, and Model Binding

Results: `View(model)`, `NotFound()` with or without a body, `Json(...)` or `Ok(...)`, redirects. The action says *what*; the framework does *how*.

Model binding fills parameters and form models from the route, the query string and the form: `/services/quote?serviceId=S0001&quantity=2.5&pickup=true` becomes a `QuoteForm`.

Conversion errors (`quantity=abc`) and the action's own checks (`AddModelError`) collect in **`ModelState`**; only when it is valid does `Quote` call the calculator.

Reading `Request.Query["quantity"]` by hand works, but it bypasses conversion, `ModelState` and the round trip of the form fields. Declare a parameter or a model class instead. Book: Sections 3.6.3 and 3.6.4.

Razor Views: Layout, Partial, Section

Listing 3.4 `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>
```

*Listing 3.4: `@` switches from HTML to C#, and every value it writes is **HTML-encoded** by default. `_ViewImports.cshtml` brings namespaces and tag helpers to every view; `_ViewStart.cshtml` gives every view the layout `_Layout`. The home page renders this partial once per popular service; the about page adds the opening hours to the layout's footer through a **section**. Book: Sections 3.7.1–3.7.3.*

Tag Helpers

`<a asp-action="Details" asp-route-id="@s.Id">` asks the *routing system* for the URL: `/services/S0004` for attribute routes, `/Outlets/Details/O02` for the conventional one. If a route changes, every link follows.

`<input asp-for="Quantity" />` writes the `name` and `id` the binder expects, puts the value back, and marks the field when `ModelState` has an error.

A `bool` becomes a checkbox plus a hidden `Pickup=false`, so an unticked box still sends a value. `<div asp-validation-summary="All">` lists every error.

Tag helpers are why the views contain almost no hand-written URLs. Book: Section 3.7.4.

Running It: the Home Page

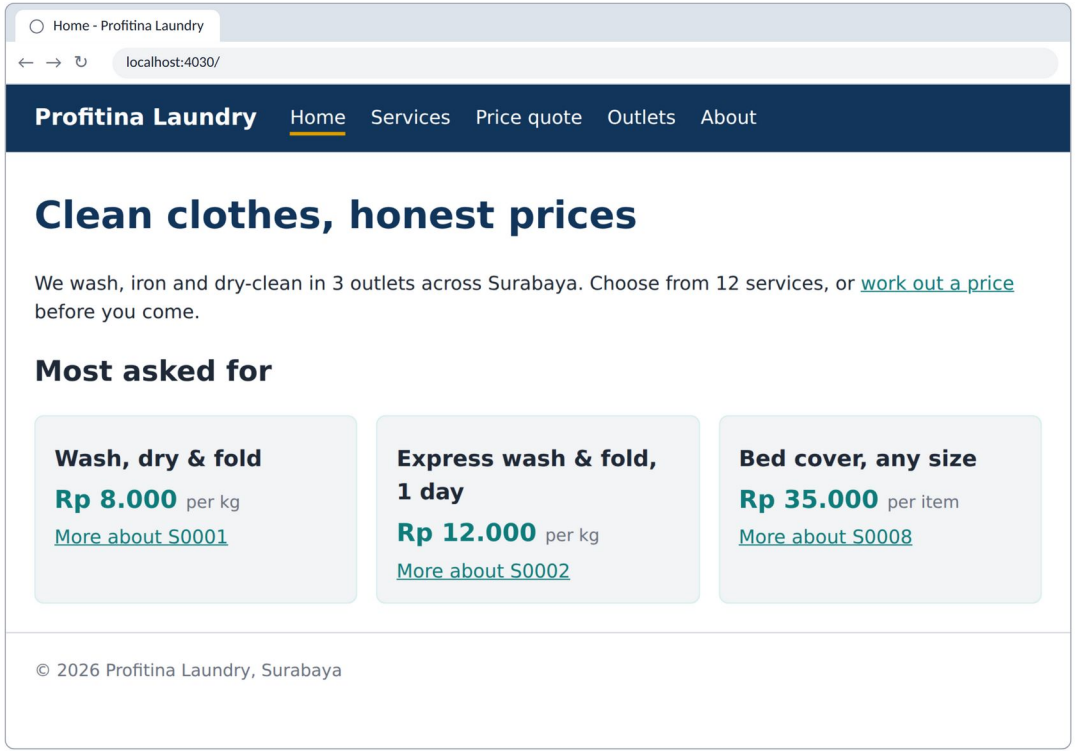


Figure 3.3 The home page of LaundryMvc: the layout's header and menu, the view's text with counts from the view model, and three cards drawn by the partial view `_ServiceCard`.

*Figure 3.3: start with `dotnet run` in the project folder, or with **Run** in VS Code, and open `http://localhost:4030`. Point out which parts come from the layout, which from the view and which from the partial. The services list at `/services` is the same layout with a table. Book: Sections 3.8.1 and 3.8.2.*

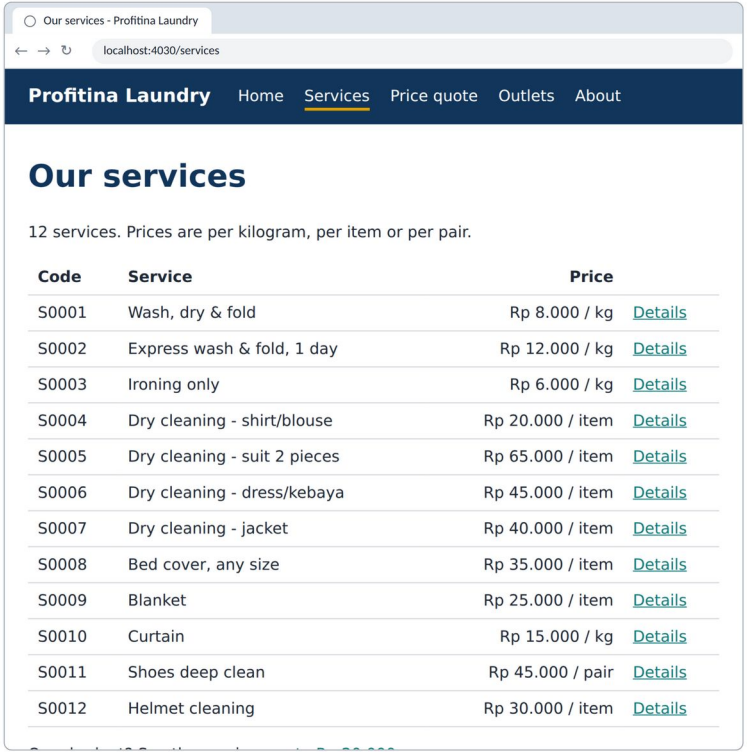


Figure 3.4 The list of services at `/services`, rendered by `Views/Services/Index.cshtml`.

A Quote, and a Quote That Went Wrong

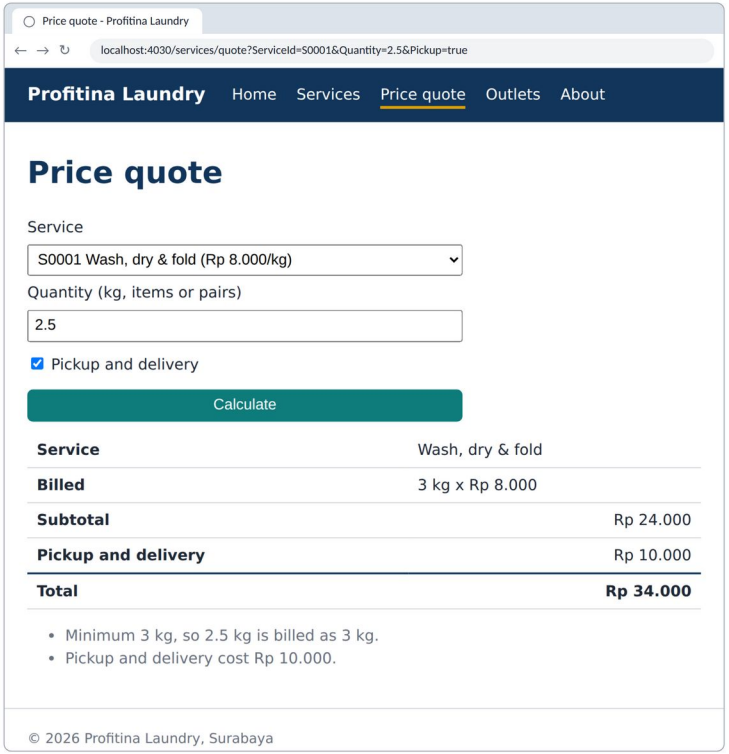


Figure 3.5 A price quote: model binding filled `QuoteForm` from the address, `PriceCalculator` applied the minimum weight and the pickup fee, and the view shows the result with its explanations.

Figure 3.5 and Figure 3.4: the quote for 2.5 kg of wash, dry and fold with pickup, worked by hand first: 3 kg billed, subtotal Rp 24.000, pickup Rp 10.000, total Rp 34.000. The site agrees with the hand calculation, and the form still holds the choices because the tag helpers wrote the bound values back. Each line of explanation comes from the calculator, not from the view. Book: Section 3.8.2.

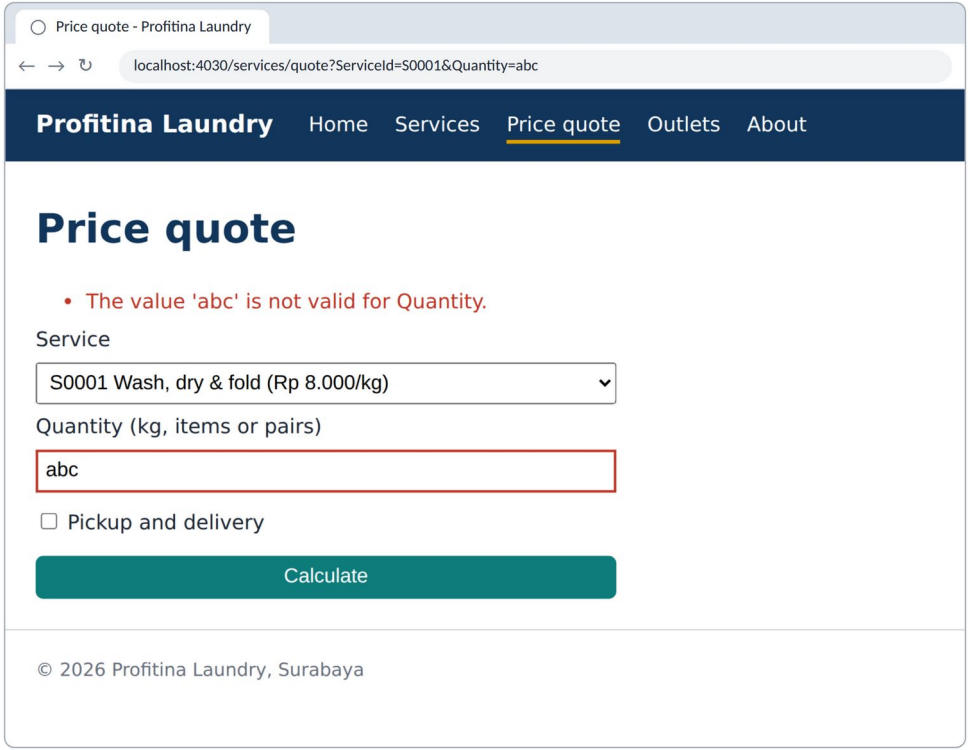


Figure 3.6 The quote form after `Quantity=abc`: the conversion error from model binding appears in the validation summary, and the field is marked.

Figure 3.6: nobody wrote code for this message; model binding put it in `ModelState` and the tag helpers showed it. Book: Section 3.8.2.

Every Address, Checked

Table 3.1 The status codes returned by the running LaundryMvc for addresses of this chapter, from `verify.sh`.

Status	Address	Why
200	`/`, `/Home`, `/home/index`	conventional route with defaults
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
404	`/Outlets/Details/009`	the action returned `NotFound()`
404	`/nothing/here`	no controller named `Nothing`
200	`/api/services/S0004`	absolute attribute route, JSON

Table 3.1: `verify.sh` builds the application, starts it and asks for every address with `curl`. Note `/services/quote`: the literal segment has precedence over the parameter `{id}`, whatever the order in the file. Book: Section 3.8.3.

A Friendly 404 That Keeps Its Status

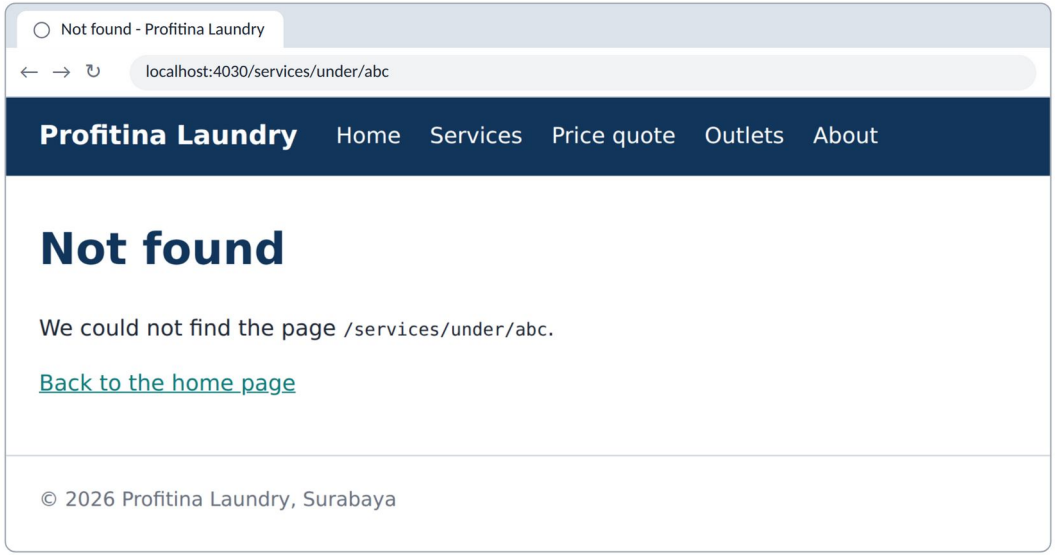


Figure 3.7 The friendly 404 page for `/services/under/abc`, produced by the status-code middleware and `HomeController.Error`.

Figure 3.7: `UseStatusCodePagesWithReExecute("/error/{0}")` *runs the pipeline again for* `/error/404`*, and* `HomeController.Error` *sets* `Response.StatusCode = code`*. People see a page in the shop's style; browsers, search engines and monitoring tools still see 404. The JSON endpoint answers errors with JSON, so programs are never handed HTML. Book: Sections 3.8.2 and 3.9.2.*

The Log

```
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 /nothing/here -> 404 in 1 ms
```

Our first middleware wrote these lines during the real run. It sits outside the status-code middleware, so it logs the path the visitor typed and the resulting status, never the re-executed `/error/404`. The first request after start-up takes about 200 ms because views and catalog load on first use. Book: Section 3.8.4.

Making It Better: No Fat Controller

```
// 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 looks shorter, but 3, 100000 and 10000 now live where no JSON endpoint and no test can reach them, and the explanations would be rebuilt in the view.

The day the counter screen of Chapter 5 needs the same rule, it gets copied, and the copies drift apart.

The test for any rule: could a JSON endpoint and a unit test use it without an HTTP request? If not, it is in the wrong place. Still missing: a database (Chapter 5), validation attributes and logins (Chapter 6). Book: Section 3.9.

Using It for Real, and Common Mistakes

Your own business on LaundryMvc: replace the two CSV files and the rules in `PriceCalculator`; controllers and views change little.

Mistakes: a controller the framework cannot find; a view that cannot be found (`Detail` is not `Details`); mixing conventional and attribute routes on one controller.

Mistakes: middleware in the wrong order; reading the query string by hand; business rules in a controller or a view.

> **WARNING.** `@Html.Raw` switches off Razor's encoding. Use it only for HTML your own program produced and trusts, never for anything a user typed.

When a view is not found, the error message lists every place it searched; compare the action's name with the file name letter by letter. Write the pipeline from the outside in: logging and error handling first, then status pages, then routing. Book: Sections 3.10 and 3.11.

Worked Exercise: Moving the Logger Inside

```
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
```

Place `app.Use(...)` for logging *after* `UseStatusCodePagesWithReExecute`: now it is inside, and it sees both the original request and the re-executed `/error/404`.

Every wrong address now appears twice in the log. Nothing is broken for the visitor; the log is only harder to read and every error is counted double.

These are the real lines from the run in the book. The lesson is the onion of Figure 3.1: what a middleware sees depends on which layers are outside it. Book: Exercise 3.4 and its solution in Section 3.14.

Summary: Five Things to Remember

1. `Program.cs` builds a host, registers services, builds the pipeline and runs Kestrel, in that order (Figure 3.1).

2. Middleware wraps middleware; each acts before and after the rest, so order matters.

3. The model holds data and rules; data access sits behind `ILaundryCatalog`, registered as a singleton.

4. Controllers are reached by the conventional route or by attribute routes with prefixes, constraints and precedence; model binding fills parameters and records errors in `ModelState`.

5. Razor encodes by default, shares a layout, reuses partials, and tag helpers build URLs from routes.

The running site matched the hand calculations and every route answered as predicted; the only surprise, "2 item", became an exercise. Book: Section 3.12.

Next Time

Chapter 4 — Evaluation and Discussion I. A small real ASP.NET Core project, solved in full, using everything from Chapters 1 to 3.

Before then: run LaundryMvc and `verify.sh`, and work the chapter's exercises, including moving the logging middleware inside the status-code middleware.

Book: Sections 3.13 and 3.14.

Lecture 4 — Evaluation and Discussion I: The Profitina Laundry Showcase Site

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 4

Lecturer: Irfan Subakti — yifana@gmail.com

Today we solve five real problems with everything from Chapters 1 to 3: a showcase site, order tracking, a route table predicted by hand, a response-time header and a maintenance mode. For each we follow the same routine, look at the solution, run it, and then study the tempting wrong solution that a careful developer might still write. Book: Chapter 4, Sections 4.1–4.10.

What This Meeting Covers

1. The toolkit and the routine — six steps for any framework problem.

2. Problem 4.1 — the showcase site: five pages under `/laundry`, words in JSON, numbers from data.

3. Problem 4.2 — "Where is my laundry?": a `regex` constraint, a `GET` form, privacy.

4. Problem 4.3 — the route table by hand, checked against the router's own list.

5. Problems 4.4 and 4.5 — a response-time header and a maintenance mode: middleware done right.

The whole application, LaundryShowcase, was built and run with .NET 10; every status code, header and error message shown today is real output, including the crashes of the wrong solutions. Book: Chapter 4.

The Toolkit

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
4.2	Where is my laundry?	a `regex` constraint, a `GET` form, `RedirectToAction`, 404 pages
4.3	The route table by hand	route matching, precedence, case
4.4	A response-time header	middleware, `Response.OnStarting`
4.5	Maintenance mode	configuration, middleware order, status 503

Table 4.1 is a map: if middleware still feels vague, start with Problems 4.4 and 4.5; if routing does, start with 4.3. Naming the tool is already half of the work. Book: Section 4.1.

How to Approach a Framework Problem

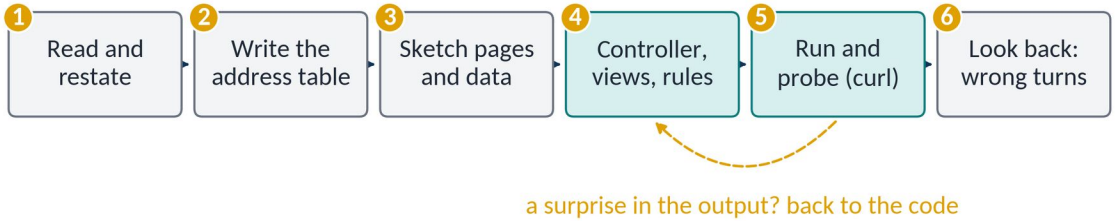


Figure 4.1 The six steps used for every problem in this chapter; the dashed arrow is the loop between running the program and fixing it.

*Figure 4.1: (1) read and restate the problem in your own words; (2) write the **address table**, including the addresses that must fail; (3) sketch pages and data on paper; (4) write thin controllers with the rules in services; (5) run and probe with a browser and `curl`; (6) look back for the tempting wrong solution. Step 2 is the one beginners skip, and it is the one that catches most mistakes. Book: Section 4.2.*

Problem 4.1: The Showcase Site

Bu Wati wants a site that presents the business to investors, partners and job candidates: **home**, **profile**, **company**, **services**, **collaboration**.

Every page lives under `/laundry`; all words that are not computed live in one file, `content/site.json`; every number on the pages is computed from the data.

The address table adds the edges a statement rarely lists: `/` redirects with 302 to `/laundry`; `/profile` and `/Showcase/Profile` must answer 404; `/laundry/Profile` answers 200 because matching ignores case.

Why a JSON file for words? Because Bu Wati changes the tagline more often than the code, and a text file can be edited without touching C#. Why computed numbers? Because "almost 3,000 orders a year" typed into a page is wrong the day after it is written. Book: Sections 4.3.1 and 4.3.2.

Designing It on Paper

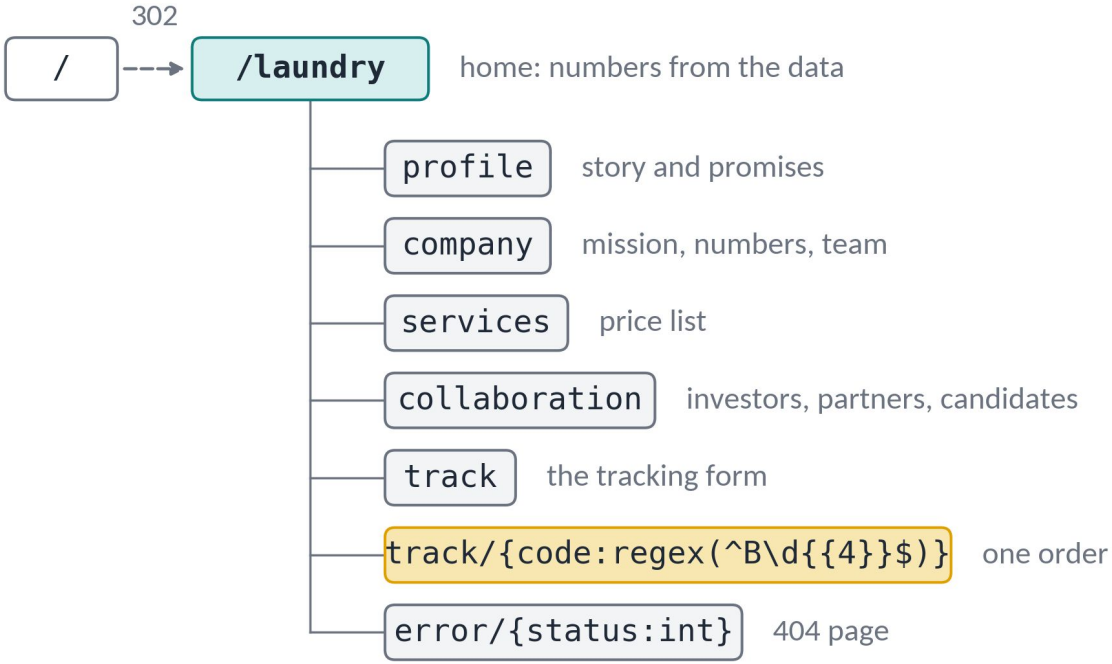


Figure 4.2 The site map of LaundryShowcase: every address is an attribute route of one controller under the prefix `/laundry`; the tracking route with a constraint belongs to Problem 4.2.

Figure 4.2: one `ShowcaseController` with `[Route("laundry")]`. `SiteContent` (the shape of `site.json` as C# records) and `LaundryData` (services, outlets and orders, loaded once, with a method `Facts()`) are registered as singletons. `_ViewImports.cshtml` uses `@inject` so that every view, including the layout, can read the site's content. Book: Sections 4.3.3 and 4.3.4.

The Solution: One Thin Controller

Listing 4.1 The first five actions of `Controllers/ShowcaseController.cs`: one attribute-routed controller for the whole showcase.

```
[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);
```

Listing 4.1: every action is one line: pick the data, pick the view. The content and the data arrive through the primary constructor, registered once in `Program.cs`, which also maps `/` to a redirect and the development-only route list. The views read the words from `SiteContent` and the numbers from the facts; no view contains a number typed by hand. Book: Section 4.3.4.

Running It: Numbers From Data

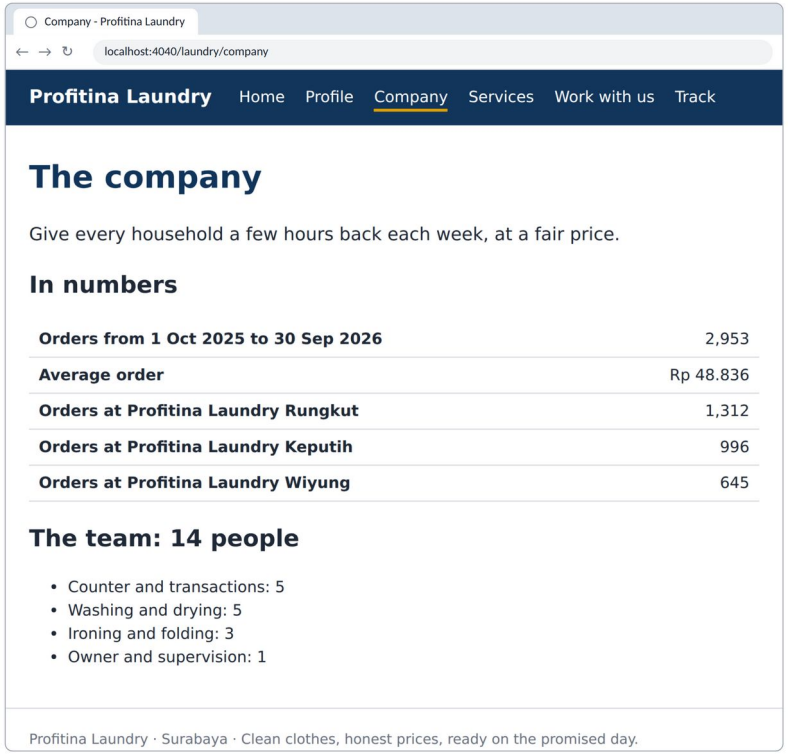


Figure 4.3 The company page: every number is computed by `LaundryData.Facts()` from the order data.

Figure 4.3: the mission and the team come from `site.json`; the numbers are computed at start-up. The home page takes its tagline from the JSON file and its counts from the data, and the collaboration page lists what the business offers and asks of each audience. Book: Section 4.3.5.

The Tempting Wrong Solution of 4.1

One leading slash: `[HttpGet("/profile")]` instead of `[HttpGet("profile")]` on the action.

A template that starts with `/` ignores the controller's prefix: the page now answers at `/profile`, and `/laundry/profile` answers 404.

There is no warning at all. Only the address table, checked against the running site, catches it.

This is why step 2 of the routine exists. The rows `/profile` and `/laundry/profile` of the address table reveal it at once. Book: Sections 4.3.6 and 4.3.7.

Problem 4.2: Where Is My Laundry?

A customer types the code on her ticket, such as `B0072`, and sees the status: queued, ready on a rack, or collected.

A public page shows **no personal data**: service, outlet, dates and amount, but never the customer's name or phone.

Worked by hand first: three orders from `Bill.csv`, one per status. The amount of B0072 checks: 7 kg at Rp 8.000 is Rp 56.000.

Anyone who guesses a code can open the page, so the page must be safe for a stranger to see. That privacy rule is part of the design, not an afterthought. Book: Sections 4.4.1 and 4.4.2.

The Design: a Constraint and a Redirect

The route `track/{code:regex(^B\d{{4}}$)}`: inside a route template the braces of the regular expression are **doubled**, because single braces mean “parameter”.

The form uses `method="get"`; its action redirects to `/laundry/track/B0072`, an address that can be bookmarked and shared.

Unknown codes and malformed codes both get the friendly 404 page, with status 404 kept.

The error view shows the address that was asked for, encoded by Razor, so even a strange code is harmless on the page. Book: Sections 4.4.3 and 4.4.4.

The Solution: Two Actions for Tracking

Listing 4.2 The tracking actions of `ShowcaseController` (the rest of `Track` looks up the service and outlet names for the view).

```
    [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() });
    }

    [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();
        }
```

Listing 4.2: `TrackForm` shows the empty form, or redirects to the clean address with `RedirectToAction`, so the URL generated from the route is always right. `Track` returns `NotFound()` for an unknown code, and the status-code middleware turns that into the friendly page. The view `Track.cshtml` shows the status in words, "Ready: collect it from rack A01.", and a table of facts. Book: Section 4.4.4.

Running It

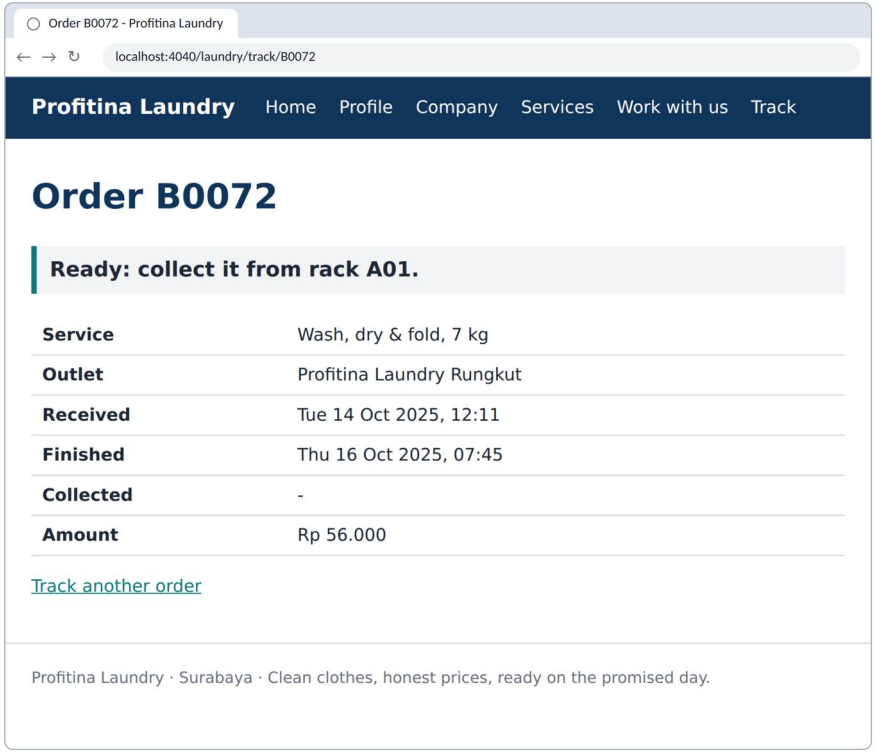


Figure 4.4 The tracking page for order B0072: ready on rack A01, with service, outlet, dates and amount, and nothing about the customer.

```
200  /laundry/track
302  /laundry/track?code=b0072
200  /laundry/track/B0072
200  /laundry/track/b0001
404  /laundry/track/B9999
404  /laundry/track/X12
404  /laundry/track/B00721
```

Figure 4.4: the form redirects with 302; a lower-case code matches too, because route constraints ignore case; a code of the right shape that does not exist, a wrong shape and a code one digit too long all get 404. Book: Section 4.4.5.

The Tempting Wrong Solution of 4.2

```
[HttpGet(@"track/{code:regex(^B\d{4}$)}")]     // WRONG: needs {{4}}
public string Track(string code) => $"order {code}";
```

```
Program.cs(14,28): warning ASP0017: Route issue: In a route parameter, '{' and '}' must be escaped with
'{{' and '}}'. (https://aka.ms/aspnet/analyzers)
```

The build warns with ASP0017, and if the warning is ignored the application stops at start-up with a `RoutePatternException`: no request is answered at all. Treat route warnings as errors. Book: Sections 4.4.6 and 4.4.7.

Problem 4.3: The Route Table by Hand

Predict, on paper, the status for twelve requests to LaundryShowcase, then check the predictions.

Two different reasons hide behind the 404s: **no route matched**, or **an action ran and returned `NotFound()`**.

The check: a development-only route `/_routes` lists every endpoint the router knows, taken from `EndpointDataSource`.

```
laundry                                  ShowcaseController.Home
laundry/services                         ShowcaseController.Services
laundry/track                            ShowcaseController.TrackForm
laundry/track/{code:regex(^B\d{{4}}$)}   ShowcaseController.Track
```

An excerpt of the real list. Ask the room to predict `/laundry/Profile` (200, case is ignored), `/laundry/track/X12` (404 from routing: the constraint fails) and `/laundry/track/B9999` (404 from the action: the order does not exist). The book's table predicts twelve requests, and every prediction held. Book: Sections 4.5.1–4.5.5.

The Tempting Wrong Solution of 4.3

```
[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}";
```

```
500  /laundry/track/B0072
     Microsoft.AspNetCore.Routing.Matching.AmbiguousMatchException: The request matched multiple
endpoints. Matches:
```

Parameter names do not distinguish routes; only literal segments and constraints do. The build warned (ASP0023) and the first matching request crashed with 500. Book: Sections 4.5.6 and 4.5.7.

Problem 4.4: A Response-Time Header

Listing 4.3 `Middleware/ResponseTimeMiddleware.cs`: the header is set in an `OnStarting` callback, just before the headers are sent.

```
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);
    }
}
```

Listing 4.3: HTTP sends headers before the body, so once the body has started, headers can no longer change. `OnStarting` *runs our code at exactly the last moment.* `curl -s -D - -o /dev/null http://localhost:4040/laundry` *showed* `X-Response-Time: 2 ms` *among the real headers. Book: Sections 4.6.1–4.6.5.*

The Tempting Wrong Solution of 4.4

```
    await next(context);
    context.Response.Headers["X-Response-Time"] =         // WRONG: too late
        $"{clock.ElapsedMilliseconds} ms";
```

```
System.InvalidOperationException: Headers are read-only, response has already started.
```

The page still arrives with status 200, the header is silently missing, and the log shows one exception per request. Only reading the log or looking at the headers with `curl -D -` catches it. Book: Sections 4.6.6 and 4.6.7.

Problem 4.5: Maintenance Mode That Keeps Its Style

A switch `Showcase:Maintenance` in `appsettings.json`, off by default, that can be turned on without a rebuild: `Showcase__Maintenance=true dotnet run`.

While on, every page answers **503** with `Retry-After: 3600` and a “Back soon” page, styled by the site’s own style sheet, which must therefore still be served.

The middleware sits **after** `UseRouting`, where the chosen endpoint is known, so it can let static assets through.

```
503  /laundry
503  /laundry/profile
200  /css/site.css
```

The double underscore in the environment variable stands for the colon of the configuration key; environment variables override `appsettings.json`. Book: Sections 4.7.1–4.7.5.

The Maintenance Page, and Its Wrong Twin

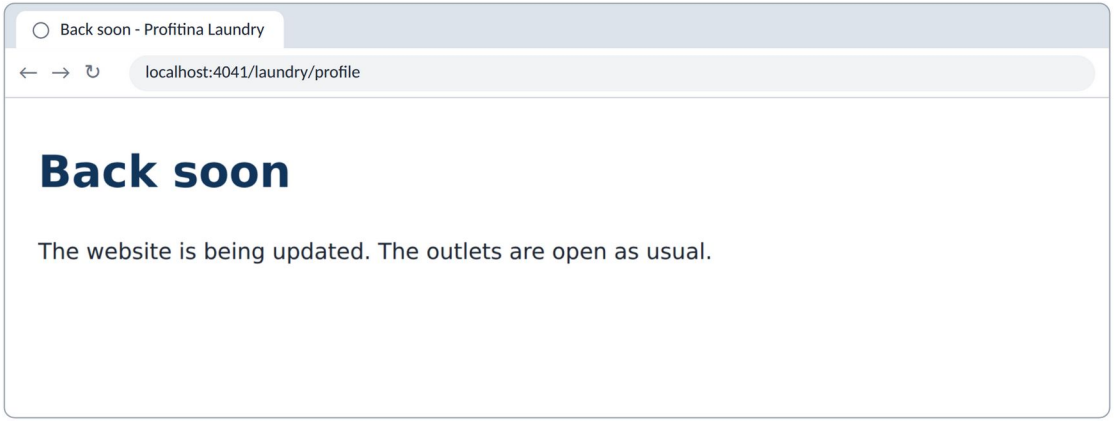


Figure 4.5 The maintenance page, answered with status 503 and styled by the site's own style sheet, which maintenance mode lets through.

Figure 4.5: the tempting wrong solution puts the check before `UseRouting`. There `context.GetEndpoint()` is always null, so nothing looks like an asset and the real run answered `503 /css/site.css Back soon`: an unstyled page. Probing the style sheet during maintenance catches it. Book: Sections 4.7.6 and 4.7.7.

Five Wrong Solutions, One Lesson

Table 4.2 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	wrong address, 404 at the right one	the address table
Single braces in `regex`	start-up exception	warning ASP0017
Two routes of one shape	500 `AmbiguousMatchException`	warning ASP0023
Headers after `await next`	header missing, exception logged	the log; `curl -D -`
Check before `UseRouting`	style sheet blocked too	probing `/css/site.css`

Table 4.2: two are caught by warnings, three only by probing the running application. Three more common mistakes: hard-coding numbers that come from data, leaving development tools such as `/_routes` on in Production (a Production run answered it with 404), and a friendly error page that forgets its status and answers 200. Book: Section 4.9.

Using It for Real

A showcase for your own business: replace `site.json` and the data files; the controller and views stay.

Tracking pages and personal data: show only what a stranger with the code may see; never names, phone numbers or addresses on a page reached by a guessable code.

For a tailor or a repair shop the same tracking page works with “ready for collection” in place of a rack number. A maintenance switch is useful to any small business on the day it moves its data or upgrades its server: customers see a polite page in the shop’s style, and monitoring tools see an honest 503. Book: Section 4.8.

Summary: Five Things to Remember

1. Follow the routine of Figure 4.1; the address table, with the addresses that must fail, catches most mistakes.

2. Words in a content file, numbers computed from data, both injected as singletons.

3. Double the braces of a regular expression inside a route template; keep 404 pages honest and public pages free of personal data.

4. Check predictions against the router's own list, on a development-only route; ambiguous routes crash with 500.

5. Set headers in `Response.OnStarting`; put endpoint-aware middleware after `UseRouting`.

Notice what the five problems have in common: none needed a new framework feature beyond Chapter 3. What they needed was care about order (routes, middleware, headers), about edges (addresses that must fail) and about honesty (status codes that tell the truth). That care is what separates a site that works in a demonstration from a site that works on a busy Monday. Book: Section 4.10.

Next Time

Chapter 5 — ASP.NET Core MVC II. Entity Framework Core and real CRUD: the data move into SQLite, and the counter finally remembers customers, orders and payments.

Before then: run LaundryShowcase, try each tempting wrong solution yourself, and work the chapter's exercises.

Book: Sections 4.11 and 4.12.

Lecture 5 — ASP.NET Core MVC II: Entity Framework Core and Real CRUD

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 5

Lecturer: Irfan Subakti — yifana@gmail.com

Today the shop gets a memory. We connect LaundryMvc to a SQLite database through Entity Framework Core 10: model classes, a `DbContext`, migrations, seed data, LINQ queries, and complete create-read-update-delete pages for customers, orders and payments. The catalog of Chapter 3 moves onto the database without one controller of Chapter 3 changing. Book: Chapter 5, Sections 5.1–5.17.

What This Meeting Covers

1. A shop without a memory — the carbon receipt book, and why the website cannot help yet.

2. The idea — tables and objects, design time and run time, LINQ, why SQLite.

3. The data — customers, services, orders, payments; the balance formula; LINQ first, without a database.

4. EF Core — packages, model classes, the `DbContext`, seeding, migrations.

5. Real CRUD — `CustomersController` action by action; orders, payments, the price list and the counter.

An honest note on what was run: the book's machine could not reach NuGet, so the EF Core web application was written with great care against the official documentation and checked, but not restored or run there. The console program `LinqDemo` was built and run, and its output today is real. On your computer, with internet access, everything runs as described. Book: Chapter 5 and its README.

A Shop Without a Memory

Seven in the morning at the Keputih outlet: a bag is weighed (4.5 kg), the customer's name and phone go onto a numbered carbon receipt, the bag goes into a blue basket.

By noon a customer asks whether her laundry is ready; the cashier flips through the carbon book and walks to the back to look.

In the evening the owner adds up the drawer and compares it, page by page, with the receipts.

The Chapter 3 site knows twelve services and three outlets from CSV files, read once at start-up. It cannot remember a single customer.

What the shop needs is the carbon book, the price board and the cash drawer inside the program: every customer, order and payment written down the moment it happens, found again in a second, and still there tomorrow. That place is a database. Book: Section 5.1.

Tables on One Side, Objects on the Other

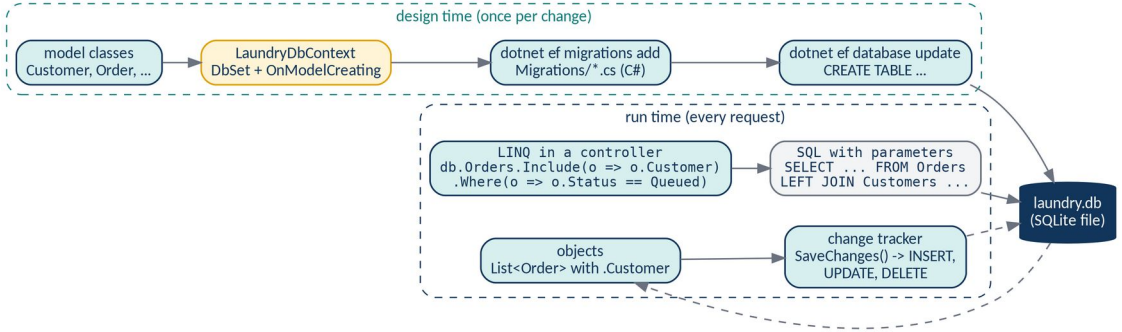


Figure 5.1 How EF Core connects the classes to the database: migrations at design time, LINQ translated into SQL and the change tracker at run time.

*Figure 5.1: two moments. At **design time** the command `dotnet ef` reads the classes and writes **migrations**, C# files that create or change tables. At **run time** a **LINQ** query on a `DbSet` is translated into SQL, the rows come back as objects, and the **change tracker** remembers what you changed so that `SaveChangesAsync` can write it. LINQ is the same language on a list in memory and on a database table. Book: Sections 5.2.1–5.2.3.*

Why SQLite

The whole database is one file, `laundry.db`, next to the program: nothing to install, nothing to start, easy to copy and to throw away.

It is a real SQL database with transactions, foreign keys and indexes, used in phones, browsers and countless desktop programs.

Moving to MySQL or SQL Server later means changing the provider package and the connection string; the model, the LINQ and the controllers stay.

For one outlet with three cashiers, SQLite is more than enough. When several outlets share one database over the network, a server database is the better choice. Book: Section 5.2.4.

Designing the Data

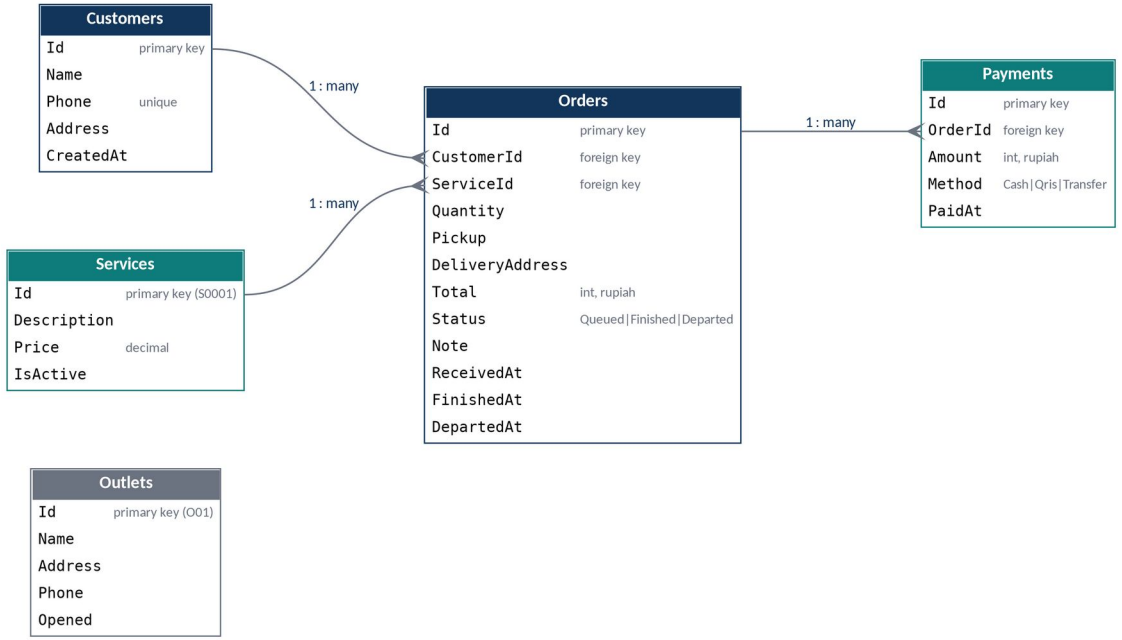


Figure 5.2 The Profitina Laundry database of this chapter: every order belongs to one customer and one service, and an order can have many payments.

Figure 5.2: five tables, `Services`, `Outlets`, `Customers`, `Orders` and `Payments`. A customer has many orders; an order belongs to one customer and one service and has many payments. Customers pay in parts, in cash, by QRIS or by transfer, so a payment is its own row. Book: Section 5.3.

The Balance Is Computed, Never Stored

$$\text{Balance} = \text{Total} - \sum_{p \in \text{payments}} \text{Amount}_p \tag{5.1}$$

Storing a `Paid` column *and* the payments would invite the two to disagree one day.

Equation (5.1) is computed by two read-only properties of `Order`: `Paid => Payments.Sum(p => p.Amount)` and `Balance => Total - Paid`.

Money is kept in whole rupiah as `int`; the order's `Total` is fixed when the order is created, so a later price change does not rewrite history.

Worked example from the book: order 2 cost Rp 36.000 and Rp 20.000 was paid, so Rp 16.000 is still owed. After two more payments of Rp 10.000 and Rp 6.000, `Paid` is 36000 and the balance is 0, and the payment form disappears from the order page. Book: Section 5.3 and Exercise 5.1.

LINQ First, Without a Database

```
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
```

The real output of `LinqDemo`, which runs the chapter's queries on lists in memory with the web app's own model classes. `Where` keeps what a lambda accepts, `OrderBy` sorts, `GroupBy` makes one group per service, `Select` shapes each group into a summary. The same queries, on a `DbSet`, become SQL. Learn LINQ here, where nothing can go wrong with a database. Book: Section 5.4.

Installing Entity Framework Core

```
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 SQLite provider does the work at run time; the Design package and the global tool `dotnet-ef` create migrations at design time.

`appsettings.json` gets a connection string, `"Laundry": "Data Source=laundry.db"`, and the log category `Microsoft.EntityFrameworkCore.Database.Command` at `Information`, so every SQL statement appears in the terminal.

The packages and the tool use the same version, 10.0.12, the current EF Core 10 release. Seeing the SQL is the best way to learn what EF Core does. Book: Section 5.5.

The Model Classes

Listing 5.1 Part of `Models/Order.cs`: an order with its foreign keys, status and payments; `Paid` and `Balance` implement the balance formula.

```
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 int Total { get; set; }               // rupiah, fixed at creation
    public OrderStatus Status { get; set; } = OrderStatus.Queued;

    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 5.1: plain C# classes. By ***convention*** `Id` *is the key,* `CustomerId` *is the foreign key to* `Customer`*, and* `Payments` *is a one-to-many relationship. The full class adds pickup, delivery address, a note and three timestamps.* `Service` *changes from Chapter 3's record into an entity class with setters and an* `IsActive` *flag. Book: Section 5.6.*

The DbContext

`LaundryDbContext` has five `DbSet` properties, one per table: `Services`, `Outlets`, `Customers`, `Orders`, `Payments`.

`OnModelCreating` adds what conventions cannot guess: maximum lengths, a unique index on `Customer.Phone`, an index on `Orders.Status`, enums stored as text with `HasConversion<string>()`.

Deleting a customer or a service must never delete orders: both relationships use `OnDelete(DeleteBehavior.Restrict)`.

EF Core's default for required relationships is cascade delete. Decide deliberately, per relationship, whether deleting the parent may delete the children; in a shop, orders are history and must stay. Book: Section 5.7.

Seeding, and Keeping Chapter 3's Promise

`UseSeeding` (EF Core 9 and later) runs whenever the database is created or migrated. `SeedData` loads services and outlets from Chapter 3's CSV files, then three sample customers with orders and payments; each step first checks that its table is empty.

`EfCatalog` answers the old `ILaundryCatalog` interface from the database.

`Program.cs` changes in two lines: `AddDbContext<LaundryDbContext>(…)` with SQLite and the seeding callbacks, and `AddScoped<ILaundryCatalog, EfCatalog>()` in place of the singleton `CsvCatalog`.

Not one controller of Chapter 3 changed. The catalog became scoped because it now uses a DbContext, which is scoped: one per request. A singleton holding a DbContext would break the moment two requests arrive together. Book: Section 5.8.

Migrations: From Classes to Tables

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

The first command writes `<timestamp>_InitialCreate.cs` with an `Up` method (create the five tables, keys, foreign keys and indexes) and a `Down` method that undoes it, plus a model snapshot.

The second applies it and records it in the table `__EFMigrationsHistory`.

Add `Rack` to `Order`, run `dotnet ef migrations add AddRackToOrders`, and EF Core compares the model with the snapshot: one difference, so `Up` adds one column and `Down` removes it.

> **KEY INSIGHT:** Migrations are to the database what Git commits are to the code: small, ordered, reviewable and stored in the repository, so every computer can be brought to exactly the same schema.

`dotnet ef migrations remove` deletes the last migration if it has not been applied yet; `dotnet ef migrations script` prints the SQL for a database administrator. Book: Section 5.9.

Real CRUD for Customers

Table 5.1 The main 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 …`
`Details(id)`	`GET /Customers/Details/3`	one `SELECT` with a `LEFT JOIN` to `Orders`
`Create(customer)`	`POST /Customers/Create`	a phone check, then `INSERT`
`Edit(id, input)`	`POST /Customers/Edit/3`	`SELECT`, phone check, `UPDATE`
`DeleteConfirmed(id)`	`POST /Customers/Delete/3`	`SELECT`, an orders check, `DELETE`

Table 5.1: every action that writes is a `POST`, and every `GET` only reads. The controller receives `LaundryDbContext db` through its primary constructor, one per request; it never creates, opens or disposes a connection. Values always travel as parameters such as `@search`, never pasted into the SQL. Book: Sections 5.10 and 5.10.1.

Create: Binding, Checking, Saving

Listing 5.2 The `Create` action for `POST` in `Controllers/CustomersController.cs`.

```
[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);

    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 });
}
```

*Listing 5.2: `[Bind]` lists exactly the fields a form may set; `[ValidateAntiForgeryToken]` rejects forms not produced by our own site. The phone check adds an error to `ModelState`; if anything is wrong the form comes back with the values. After `SaveChangesAsync` the database has chosen the `Id`, and the **Post/Redirect/Get** pattern sends the browser to the details page, so a refresh cannot insert twice. Book: Section 5.10.3.*

Read, Update, Delete

Read: list pages use `AsNoTracking()`; the details page uses `Include(c => c.Orders).ThenInclude(o => o.Service)` so that related rows are loaded in one query.

Update: load the tracked entity with `FindAsync`, copy the new values onto it, `SaveChangesAsync`: EF Core sends an `UPDATE` of only the changed columns.

Delete: a `GET` shows a confirmation page; the `POST` checks for orders first and refuses politely, and `Restrict` guards the database as well.

The search is case sensitive on SQLite (`instr`), so a search for `sari` does not find Sari Wulandari; Chapter 8 fixes that in Problem 8.4. A customer called `'; DROP TABLE Customers; --` is just a strange name, because the value travels as a parameter. Book: Sections 5.10.2, 5.10.4 and 5.10.5.

Orders, Payments, Prices and the Counter

OrdersController: new orders priced by Chapter 3's `PriceCalculator` (2 kg of wash, dry and fold is billed as 3 kg, Rp 24.000); the workflow Queued → Finished → Departed with timestamps; payments recorded on the order page.

PricesController: the owner changes prices and adds services with automatic codes; old orders keep their totals.

CounterController: four numbers for the counter (in the queue, ready for pickup, received today, money received today), each one SQL statement.

The pricing rule written once in Chapter 3 now prices real orders, exactly the reuse the thin-controller rule promised. Book: Section 5.11.

Running It on Your Computer

Run the two `dotnet ef` commands, then `dotnet run`, and open `http://localhost:4050`. The home page, services and outlets look as in Chapter 3, now from the database.

Try: add a customer twice with one phone number; create an order for 2 kg; record a payment of Rp 10.000; mark it finished and picked up; change a price; try to delete a customer with orders.

Watch the terminal: before each request's timing line, the SQL that EF Core sent, with parameters shown by name such as `@p0`.

These seven steps are the book's guided tour, and a good exercise to do in pairs: one clicks, the other reads the SQL aloud. Book: Section 5.12.

Making It Better

Read without tracking, write with tracking. Stay asynchronous: `ToListAsync`, `AnyAsync`, `SaveChangesAsync`.

Index what you search, when a real query is slow. **Watch the number of queries** and what each `Include` loads.

Mind the decimal separator: on a computer set to Indonesian, `4.5` from a number field can be read as 45.

The last point is a real trap on cashiers' computers set to Indonesian, where the comma is the decimal separator: keep the server culture at the default, or set `CultureInfo.DefaultThreadCurrentCulture` explicitly, and always test with a value that has decimals. Book: Section 5.13.

Common Mistakes

Forgetting `SaveChangesAsync()`: nothing reaches the database. **Changing the model without a migration**: since EF Core 9, `Migrate()` stops with "pending changes".

Navigation properties that are not loaded: without `Include`, `order.Payments` is empty and the balance is silently wrong.

A singleton DbContext; **cascade delete by accident**; **deleting through a link** (a `GET` must never change data).

Summing `decimal` on SQLite: EF Core cannot translate it and throws at run time; keep summed money in `int` columns.

The navigation-property mistake is the most dangerous, because nothing fails: the page shows a balance, and the balance is wrong. When a number looks suspicious, read the SQL in the terminal and check that the `JOIN` is there. Book: Section 5.16.

Summary: Five Things to Remember

1. EF Core maps classes to tables and objects to rows (Figure 5.1); conventions plus `OnModelCreating` describe the model.
2. The `DbContext` is registered with `AddDbContext` and injected, one per request.
3. Migrations are ordered, reversible C# files; `database update` applies them; `UseSeeding` fills an empty database.
4. LINQ on a `DbSet` becomes parameterized SQL when a method such as `ToListAsync` asks; `Include` loads related rows; `AsNoTracking` for reads.
5. Create adds and saves, update changes a tracked entity, delete is confirmed, checked and guarded; the balance is always computed (Equation (5.1)).

Book: Section 5.17.

Next Time

Chapter 6 — ASP.NET Core MVC III. Validation, Identity and deployment: rules on the model, logins and roles for the cashiers and the owner, and the site on a real server.

Before then: run `LinqDemo`, add the “three best customers” query of Exercise 5.3, and on your own computer run the web app and follow the guided tour.

Book: Sections 5.18 and 5.19.

Lecture 6 — ASP.NET Core MVC III: Validation, Identity and Deployment

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 6

Lecturer: Irfan Subakti — yifana@gmail.com

Today the laundry application gets rules and a lock, and leaves the owner's laptop. We add validation with data annotations and form classes, sign-in with ASP.NET Core Identity, two roles with a fallback policy that protects every page by default, and a deployment to a Linux server behind Nginx. Book: Chapter 6, Sections 6.1–6.14.

What This Meeting Covers

1. An open door — three small accidents in the first week.

2. Validation — attributes on the model, a rule of our own, rules about two fields, form classes.

3. Identity — users and roles in the shop's database, how passwords are stored, sign-in.

4. Authorization — who may do what, a fallback policy, named policies.

5. Deployment — environments, `dotnet publish`, migrations on the server, Nginx and systemd, a release checklist.

What was run here: the console program `ValidationDemo`, which uses the application's own rules and Identity's password hasher from the shared framework, was built, published and run, and its output today is real. The web application needs NuGet packages and was written against the official documentation; you run it on your own computer. Book: Chapter 6 and its README.

Where We Left Off: An Open Door

A new part-time cashier opened *Prices*, typed into the wrong field and saved *Ironing only* at Rp 60 per kg; nobody noticed until the evening.

A phone number was saved as `0812 0000 104`, with the letter O, so the WhatsApp message “your laundry is ready” never arrived.

An order with pickup and delivery was saved without a delivery address.

And the application still ran on the owner’s laptop, started by hand, stopped whenever the laptop went to sleep.

None of these is a programming error: the code did exactly what it was told. What was missing is a set of rules every request must pass, and a proper home for the application. The rules help the cashier, the customer, the driver and the owner. Book: Section 6.1.

The Idea: the Path of a Form Submission

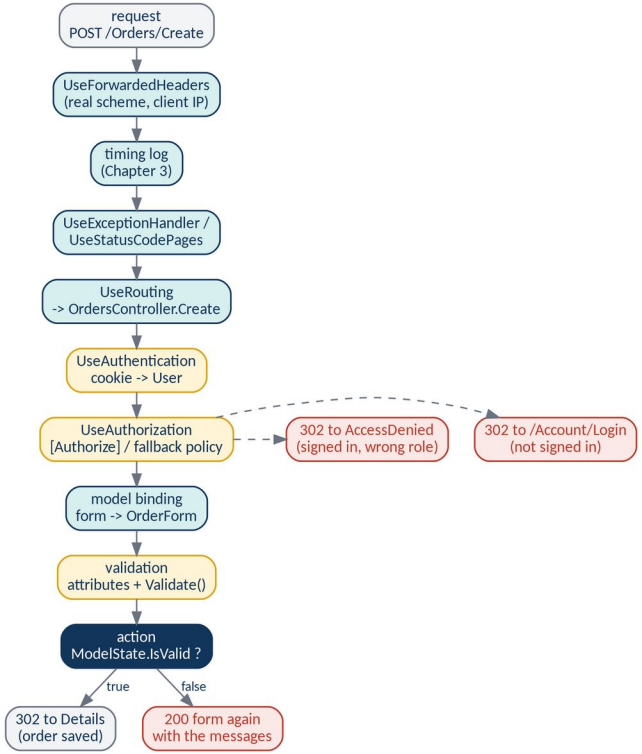


Figure 6.1 The path of a form submission through the Chapter 6 application: authentication and authorization before the controller, model binding and validation before the action, and the four ways the request can end.

*Figure 6.1: **authentication** establishes who the user is (from the cookie); **authorization** decides what the user may do. Only then does model binding fill the form class and validation fill `ModelState`. The request ends in one of four ways: sent to sign in, refused, form shown again with messages, or saved and redirected. Book: Section 6.2.*

Validation With Data Annotations

Listing 6.1 The first properties of `Models/Customer.cs` with validation attributes: a required name of 2 to 80 characters and a required Indonesian mobile number.

```
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; } = "";
```

Listing 6.1: rules are stated once, on the model, in words a cashier understands. `[Range]` keeps the price of a service between Rp 1.000 and Rp 1.000.000; `[RegularExpression]` requires the description to end in "(per kg)", "(per item)" or "(per pair)". `[Display]` gives the field the name shown in messages. Book: Section 6.3.1.

A Validation Attribute of Our Own

`IndonesianMobileAttribute` derives from `ValidationAttribute` and overrides `IsValid`: after removing spaces and dashes, the number must be a mobile number such as `0812…` or `+62 812…`.

A helper stores every number in one spelling: `+62 812-0000-0104` becomes `081200000104`, so searching and sending messages work.

The letter O in `0812 0000 1O4` fails, and the cashier sees the message at once instead of the customer never receiving hers.

Exercise 6.2 writes another one, `NotInFutureAttribute`, for dates: five lines, and every date property in the application can use it. Book: Section 6.3.2 and Exercise 6.2.

Running the Rules Without a Web Server

```
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.
5. Order: pickup without an address: INVALID
   [DeliveryAddress] Pickup and delivery needs a delivery address.
```

Part of the real output of `ValidationDemo`, which calls `Validator.TryValidateObject` on the application's own classes. Case 3 is exactly the price-list accident of the first week, now refused with two clear messages. Rules that can be run in a console can also be tested without a browser. Book: Section 6.3.3.

What MVC Does With the Rules

Model binding validates every bound object and records each failure in `ModelState`, with the property name and the message.

The action decides: `if (!ModelState.IsValid) return View(form);` shows the form again, with the values and the messages from `asp-validation-for` and the summary.

> **WARNING.** Validation does not stop the action by itself. An action that saves without asking `ModelState.IsValid` saves invalid data.

The controller adds the rules that need the database, such as "this phone number already belongs to another customer", with `ModelState.AddModelError`, so they appear in the same place. Book: Section 6.3.4.

Form Classes Instead of Entities

An entity bound directly to a form lets a crafted request set *any* property, such as `Total` or `Status`: this is called **overposting**.

`OrderForm` carries exactly what the new-order form may send (customer, service, quantity, pickup, delivery address, note) and the rules of that form.

“Pickup and delivery needs a delivery address” involves two fields, so `OrderForm` implements **`IValidatableObject`**, whose `Validate` method can `yield return` any rule.

The action copies from the form to a new entity and computes the total with `PriceCalculator`, so the price can never come from the browser. `PaymentForm` does the same for payments: between Rp 1.000 and Rp 10.000.000, method cash, QRIS or transfer. Book: Section 6.4.

Checks in the Browser, Too

The same attributes produce `data-val-*` attributes in the HTML; jQuery Validation and the unobtrusive adapter check them before the form is sent.

Browser checks are for comfort: the cashier sees the message without a round trip.

A rule checked only in JavaScript is not a rule: anyone can switch JavaScript off or send a request with `curl`. The server always checks again.

Rules about two fields and rules that need the database run only on the server, which is one more reason the server check is the real one. Book: Section 6.5.

Identity: Who Is at the Counter?

ASP.NET Core Identity stores users, password hashes, roles and lockout data; `LaundryDbContext` derives from `IdentityDbContext`, so users live in the same database as the orders.

`Program.cs` adds Identity with password rules and lockout, and a **fallback authorization policy**: every page requires a signed-in user unless it says `[AllowAnonymous]`.

Two roles, `Owner` (prices, deleting customers) and `Cashier` (orders and payments); `IdentitySeeder` applies migrations and creates roles and first accounts at start-up.

Demo accounts live only in `appsettings.Development.json` on the developer's computer, such as the owner `owner@profitina-laundry.example`; on the server the first owner is created once from environment variables set with `systemctl edit`. Identity also brings the sign-in cookie, lockout after repeated failures and security stamps, so none of that is written by hand. Should you write your own password storage? It is one of the few questions in this course whose answer is simply: no. Book: Sections 6.6.1, 6.6.2 and 6.6.4.

What Is Stored Instead of a Password

```
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
```

*The real output of the last case of `ValidationDemo`. The database never holds the password, only a **salted hash**: PBKDF2 with HMAC-SHA512, 100,000 iterations, a 16-byte random salt and a 32-byte key. The same password hashed twice gives different text, because the salt differs, so two users with the same password cannot be spotted. Checking recomputes the hash with the stored salt. Book: Section 6.6.3.*

Signing In and Keeping Doors Closed

Table 6.1 Who may use which pages after this chapter.

Page	Not signed in	Cashier	Owner
Home, services, quote, outlets, error pages	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

Table 6.1: `AccountController` *signs in with* `SignInManager`*, which also counts failures and locks the account for a while after too many.* `[Authorize(Roles = AppRoles.Owner)]` *restricts prices and deleting; a cashier who types the address sees a polite "Not allowed" page.* ***Policies*** *give a rule a name, so a controller says what it needs, not how it is checked. Book: Section 6.7.*

Hiding a Link Is Not Protecting a Page

The layout shows the *Prices* link only to the owner. That tidies the menu; it protects nothing.

Anyone can type `/Prices`. The attribute on the controller is what refuses the request.

Order in the pipeline: `UseRouting()`, then `UseAuthentication()`, then `UseAuthorization()`. In another order the rules see no endpoint or no user.

The fallback policy is the safe default: a new controller written next year is closed until someone opens it deliberately, instead of open until someone remembers to close it. Book: Sections 6.6.2 and 6.7.2.

Deployment: From Laptop to Server

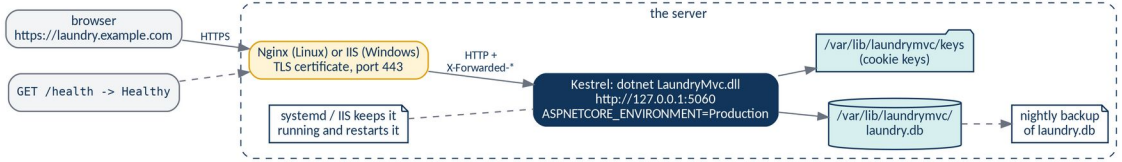


Figure 6.2 The laundry application on a Linux server: Nginx in front with the certificate, Kestrel behind it on the loopback address, systemd keeping it running, and the data folder with the database, the cookie keys and a nightly backup.

*Figure 6.2: **environments** choose settings: `appsettings.json`, then `appsettings.Production.json`, then environment variables, each layer overriding the one before. In Production the database and the cookie keys live in `/var/lib/laundrymvc`. `dotnet publish -c Release -o <folder>` produces the files to copy. Book: Sections 6.8.1 and 6.8.2.*

systemd Keeps It Running

Listing 6.2 The service section of `deploy/laundrymvc.service`: the systemd unit that keeps the app running.

```
[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
```

*Listing 6.2: systemd starts the app at boot, restarts it five seconds after a crash, and runs it as the unprivileged user `laundry`. Kestrel listens only on the loopback address; **Nginx** in front receives the public traffic, holds the TLS certificate from certbot and forwards with `X-Forwarded-For` and `X-Forwarded-Proto`. On Windows, IIS with the ASP.NET Core Module plays Nginx's part. Book: Sections 6.8.4 and 6.8.5.*

The Small Things That Make It Last

Migrations on the server before serving: the app applies them at start-up, or `dotnet ef migrations script` gives SQL to review.

Forwarded headers, or redirects point to `http://` and logs show every visitor as `127.0.0.1`. **Persisted Data Protection keys**, or every restart signs everyone out.

A **health endpoint** `/health` for monitoring; a **nightly backup** of the database file; a **release checklist** followed every time.

The checklist in the book starts with "copy the database file" for a reason: a migration that drops a column is not undone by `Down` in any way that brings the data back. Book: Sections 6.8.3, 6.8.6 and 6.8.7.

Using It for Real

Run it on your computer: sign in as the demo owner and the demo cashier, try the prices page as each, and try the three accidents of the first week.

For your own shop, clinic or orchard: write down the rules a new employee is told on the first day; each becomes an attribute, a form rule or a role.

A small business rarely needs more than two or three roles. Start with the fewest that separate “may change money rules” from “may serve customers”. Book: Sections 6.9–6.12.

Common Mistakes

Trusting the browser; **forgetting** `if (!ModelState.IsValid)`; **binding forms to entities**.

Middleware in the wrong order; **protecting pages one by one** instead of a fallback policy; **hiding links instead of protecting pages**.

Demo passwords in production; **losing the cookie keys**; **no forwarded headers behind a proxy**; **updating without a backup**.

Most of these mistakes are invisible on the developer's laptop: one user, one computer, no proxy, no restart. They appear on the first busy day on the server, which is why the release checklist and a rehearsal on a test server are worth an afternoon. Book: Section 6.13.

Summary: Five Things to Remember

1. Validation runs on the server for every request: attributes for single values, `IValidatableObject` for several, the controller for rules that need the database; all end in `ModelState`.
2. Form classes carry exactly what a form may send, which removes overposting by construction.
3. Authentication says who; authorization says what (Figure 6.1). Identity stores salted PBKDF2-HMAC-SHA512 hashes and locks out repeated failures.
4. A fallback policy protects every page by default; `[AllowAnonymous]` opens pages deliberately; roles restrict them.
5. Deployment: layered configuration, `dotnet publish`, migrations first, a reverse proxy with forwarded headers, persisted keys, health checks and backups (Figure 6.2).

Book: Section 6.14.

Next Time

Chapter 7 — Laravel I. The second framework: Artisan, routing, Blade and Eloquent, with the same laundry in PHP 8.5.

Before then: run `ValidationDemo`, add `NotInFutureAttribute` of Exercise 6.2, and on your computer sign in to the web app with both roles.

Everything we did in ASP.NET Core over four chapters has a counterpart in Laravel; keep the translation table of Chapter 2 at hand. Book: Sections 6.15 and 6.16.

Lecture 7 — Laravel I: Artisan, Routing, Blade and Eloquent

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 7

Lecturer: Irfan Subakti — yifana@gmail.com

Today we meet the second framework. We rebuild the staff pages of Profitina Laundry in Laravel 13 on PHP 8.5: Artisan, migrations written in PHP, Eloquent models, routes in one file, controllers with validation, and Blade templates. The design, the pages and the rules are those of Chapters 3 and 5; only where each piece lives, and what it is called, changes. Book: Chapter 7, Sections 7.1–7.16.

What This Meeting Covers

1. **The same shop, another language** — what carries over from ASP.NET Core, and what does not.
2. **The request lifecycle and the project** — front controller, middleware, router; where things live.
3. **Artisan and migrations** — the command line that writes code; tables written in PHP.
4. **Eloquent** — enums, models, relationships, casts; Eloquent and EF Core side by side.
5. **Routes, controllers, Blade** — `Route::resource`, validation, escaping, forms with CSRF tokens.

An honest note: Packagist could not be reached where the book was written, so no Laravel project could be created or run there. Every PHP file was checked with `php -l`, the Blade files were checked for balanced directives, and `tools/enum_demo.php` was run with plain PHP; its output today is real. You run the application on your own computer. Book: Chapter 7 and Section 7.11.

The Same Shop, Another Language

The owner's nephew **Raka** knows PHP. The Wiyung outlet might run its own copy of the staff pages on the cheap PHP hosting it already pays for.

His question: *how much of what you built in C# do I have to invent again?* Very little of the thinking, and all of the typing.

Same five tables, same pages, same workflow `queued → finished → departed`, same price rules: 3 kg minimum, whole items, Rp 10.000 pickup below Rp 100.000.

Learning a second framework is mostly learning where things live. Seeing the same feature in two frameworks is the fastest way to understand what a framework is, and Chapter 8 then solves the same problems in both. Book: Section 7.1.

One Request Through Laravel

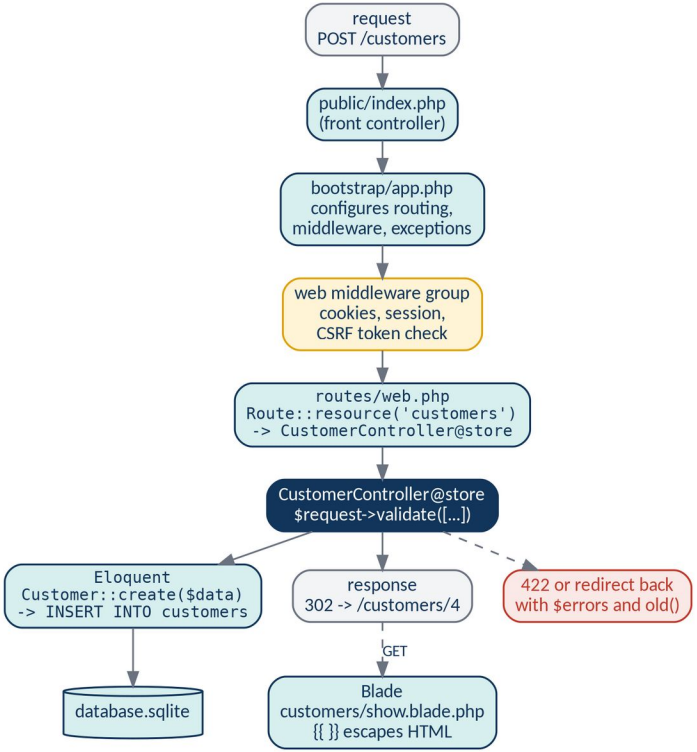


Figure 7.1 One request through the Laravel application: the front controller, the web middleware group, the router, the controller, Eloquent and Blade.

*Figure 7.1: every request enters through one **front controller**, `public/index.php`. The **web middleware group** handles cookies, the session and the CSRF token. The router picks a controller method; the controller asks **Eloquent** for data and returns a **Blade** view. It is Chapter 3's pipeline with different names. Book: Section 7.2.*

Installing and Creating the Project

```
composer create-project laravel/laravel profitina-laundry "^13.0"
cd profitina-laundry
php artisan serve --port=4070
```

Laravel 13 (March 2026) runs on PHP 8.3 to 8.5; **Composer** installs it and its packages from Packagist.

A new project uses **SQLite** by default, in `database/database.sqlite`; settings live in `.env`.

Where things live: `routes/web.php` (routes), `app/Http/Controllers`, `app/Models`, `database/migrations`, `resources/views`, `public/` (the only folder the web may reach).

The book's table maps each folder to its ASP.NET Core counterpart: `routes/web.php` for the routes in `Program.cs` and attributes, `resources/views` for `Views`, `.env` and `config/` for `appsettings*.json`, `public/` for `wwwroot/`, `bootstrap/app.php` for much of `Program.cs`. Book: Section 7.3.

Artisan, the Command Line That Writes Code

`php artisan make:model Customer -m` — a model and its migration; `make:controller CustomerController --resource --model=Customer` — a controller with the seven CRUD methods; `make:enum` — a backed enum.

`php artisan migrate`, `migrate:fresh --seed` — run migrations, or drop everything and rebuild with sample data.

`php artisan route:list --except-vendor` — the router's own list; `php artisan tinker` — a console inside the application.

Artisan plays the part of `dotnet new` and `dotnet ef` together. `route:list` is the Laravel twin of the `/_routes` page of Chapter 4. Book: Section 7.4.

Migrations: Tables Written in PHP

Listing 7.1 Part of `2026_09_01_000003_create_orders_table.php`: orders with their foreign keys, status and timestamps.

```
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->unsignedInteger('total');         // rupiah, fixed at creation
    $table->string('status', 12)->default('queued')->index();
    $table->timestamp('received_at');
    $table->timestamp('finished_at')->nullable();
    $table->timestamps();
});
```

Listing 7.1: in EF Core the migration was generated from the classes; in Laravel it is written by hand with the schema builder, and the model follows the table. Migrations run in the order of their file names. `foreignId(...)->constrained()` creates the foreign key; `restrictOnDelete` keeps orders when someone tries to delete a customer, while payments use `cascadeOnDelete` and go with their order. Book: Section 7.5.

Enums and Money, Run With Plain PHP

Listing 7.2 The heart of `app/Enums/PaymentMethod.php`: cash, QRIS and transfer.

```
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',
        };
    }
}
```

```
Queued    -> Finished
Finished  -> Departed
Departed  -> (end)
enum(App\Enums\OrderStatus::Finished)
NULL
QRIS is stored as "qris"
Rp 36.000 Rp 1.250.000
```

*Listing 7.2: a **backed enum** stores its string value in the database. `OrderStatus` knows the workflow (`next()` gives the following status, `NULL` at the end), and `Money::rp` formats rupiah with `number_format($amount, 0, ',', '.')`, the same text as C#'s `Money.Rp`. The output is the real run of `tools/enum_demo.php`. Book: Section 7.6.1.*

Eloquent Models

Listing 7.3 The heart of `app/Models/Customer.php`: a customer and the relationship to the customer's orders.

```
#[Fillable(['name', 'phone', 'address'])]
class Customer extends Model
{
    public function orders(): HasMany
    {
        return $this->hasMany(Order::class);
    }
}
```

*Listing 7.3: Eloquent is an **active record**: each model knows its table (`customers`, by convention) and objects save themselves. The attribute `#[Fillable]` lists the columns a request may fill in one go. `Order` adds `casts()` (quantity as decimal, pickup as boolean, status as `OrderStatus`), relationships `belongsTo` and `hasMany`, and `paid()` and `balance()`, the balance formula of Chapter 5 in PHP. Book: Section 7.6.2.*

Eloquent and EF Core Side by Side

Table 7.1 Everyday operations in EF Core and in Eloquent.

Task	EF Core	Eloquent
filter and sort	`Where(...).OrderBy(c => c.Name)`	`where(...)->orderBy('name')`
run the query	`await query.ToListAsync()`	`$query->get()`
related rows	`.Include(o => o.Customer)`	`->with('customer')`
insert	`Add(c); await SaveChangesAsync()`	`Customer::create($data)`
update	change, then `SaveChangesAsync()`	`$customer->update($data)`
delete	`Remove(c); await SaveChangesAsync()`	`$customer->delete()`
exists	`AnyAsync(o => o.CustomerId == id)`	`$customer->orders()->exists()`

Table 7.1: EF Core has a unit of work (the change tracker, one `SaveChanges` for many changes); Eloquent saves each object when you ask it to. Both send parameters, never pasted values. Book: Section 7.6.3.

Seeding From the Chapter 3 Files

`DatabaseSeeder` reads the same `Service.csv` and `Outlet.csv` as Chapter 3, then creates sample customers, orders and payments.

`php artisan migrate:fresh --seed` rebuilds the database with the sample data in one command, which is perfect on a laptop.

> **WARNING.** `migrate:fresh` drops every table. On the shop's server use `php artisan migrate --force`, which only runs new migrations.

The same data in both applications means the same pages show the same numbers, which Chapter 8 uses to compare the two versions line by line. Book: Section 7.7.

Routing: One File for Everything

Listing 7.4 Part of `routes/web.php`: the staff pages and an extra action on orders.

```
// 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');
```

*Listing 7.4: every route has a **name**, and views build URLs with `route('customers.show', $customer)`, just as tag helpers asked the router in Razor. `Route::resource` declares the seven CRUD routes at once; `only` and `except` trim them. Book: Section 7.8.*

The Seven Resource Routes

Table 7.2 The seven routes of `Route::resource('customers', …)`.

Method and address	Method	Route name
`GET /customers`	`index`	`customers.index`
`GET /customers/create`	`create`	`customers.create`
`POST /customers`	`store`	`customers.store`
`GET /customers/{customer}`	`show`	`customers.show`
`GET /customers/{customer}/edit`	`edit`	`customers.edit`
`PUT /customers/{customer}`	`update`	`customers.update`
`DELETE /customers/{customer}`	`destroy`	`customers.destroy`

Table 7.2: ***route model binding*** *turns* `{customer}` *into a* `Customer` *loaded by its key, or a 404 if there is none, before the controller method runs. HTML forms can only send* `GET` *and* `POST`*, so* `PUT` *and* `DELETE` *travel as a hidden field written by* `@method('PUT')`*. Book: Section 7.8.*

Controllers: Validation and CRUD

Listing 7.5 The `store` method and the rules of `app/Http/Controllers/CustomerController.php`.

```
public function store(Request $request)
{
    $customer = Customer::create($request->validate($this->rules()));

    return redirect()->route('customers.show', $customer)
        ->with('message', "Customer {$customer->name} added.");
}

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'],
    ];
}
```

Listing 7.5: `$request->validate(...)` *returns only the validated fields. When a rule fails it does not return at all: it* ***redirects back*** *with the errors and the old input, so the controller has no* `if`*.* `Rule::unique(...)->ignore($customer)` *is the "phone already used" check of Chapter 5 in one line.* `with('message', …)` *flashes a message for the next page. Book: Section 7.9.1.*

Orders, Payments, Prices, Counter

`OrderController` prices new orders by the rules of Chapter 3 and moves them through the workflow with `advance`; `PaymentController` records a payment for an order.

`PriceController` is the owner's price list; `CounterController` is an invokable controller with four numbers for the counter.

Seeing the SQL: `DB::listen` in development prints every query, just as EF Core's command log did.

The pricing rules are the same numbers as Chapter 3's `PriceCalculator`, so an order costs the same in both applications. Book: Sections 7.9.2 and 7.9.3.

Blade: Escaping, Forms and Old Input

Listing 7.6 The form of `resources/views/customers/edit.blade.php`, with its CSRF token and method field.

```
<form method="post" action="{{ route('customers.update', $customer) }}"
      class="form">
    @csrf
    @method('PUT')
    @include('customers._form')
    <button type="submit">Save</button>
</form>
```

Listing 7.6: `{{ }}` escapes everything it prints. `@csrf` adds the token without which Laravel answers 419 Page Expired. The shared fields in `_form.blade.php` use `old('name', $customer->name)` to put back what the user typed after a failed validation, and `@error('name')` shows the message. The layout uses `@extends`, `@section` and `@yield`. Book: Sections 7.10.1 and 7.10.2.

Razor and Blade Side by Side

Table 7.3 Razor and Blade constructs for the same tasks.

Task	Razor	Blade
print, escaped	`@Model.Name`	`{{ $customer->name }}`
print, raw	`@Html.Raw(x)`	`{!! $x !!}`
layout	`_Layout`, `@RenderBody()`	`@extends`, `@yield`
partial	`<partial name="_Form" />`	`@include('customers._form')`
anti-forgery	tag helper, automatic	`@csrf`

Table 7.3: the ideas are those of Chapter 2's MiniFrame; only the spelling changes. The order page shows the workflow button, the payments and the payment form only while a balance remains. Book: Sections 7.10.3 and 7.10.4.

Running It 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
```

Create a fresh project, copy the chapter's files over it, build the database with the sample data, look at the routes, and start the server. Then repeat the guided tour of Chapter 5: add a customer twice with one phone number, create an order for 2 kg, record a payment, move the order through the workflow, change a price, try to delete a customer with orders. The same clicks should give the same numbers and the same messages as the ASP.NET Core version. If they do not, the difference is a bug in one of the two, and finding it is an excellent exercise. Book: Section 7.11.

Making It Better and Using It for Real

Switch on `Model::preventLazyLoading()` in development, so that a forgotten `with(...)` fails loudly instead of sending one query per row.

Set `'timezone' => 'Asia/Jakarta'` in `config/app.php`: with the default UTC, “today” starts at seven in the morning in Surabaya.

On cheap PHP hosting, point the web root at `public/` only; `.env` and the SQLite file must never be reachable.

These three settings take a minute and prevent three of the most common Laravel incidents. Book: Sections 7.12–7.14.

Common Mistakes

Mass assignment without `#[Fillable]`, or `$guarded = []` that lets a request set any column. **Forgetting** `@csrf`: 419.

`{!! !!}` **for user data**: cross-site scripting. **N+1 queries**: a loop reading `$order->customer->name` without `with('customer')`.

`migrate:fresh` **on the server**; **the web server pointed at the project folder**; `APP_DEBUG=true` **in production**; **the wrong time zone**.

Notice how many of these are Chapter 6's mistakes in new clothes: mass assignment is overposting, a missing `@csrf` *is a missing anti-forgery token,* `{!! !!}` *is* `Html::raw`*. The dangers are the same in every framework; only the names of the guards change. Book: Section 7.15.*

Summary: Five Things to Remember

1. Laravel 13 on PHP 8.3–8.5: Composer installs it, Artisan creates files and runs migrations, SQLite by default (Figure 7.1).

2. Migrations are written by hand and run in file-name order; foreign keys decide with `restrictOnDelete` or `cascadeOnDelete`.

3. Eloquent models are active records: `#[Fillable]`, `casts()`, relationships, computed values.

4. `routes/web.php` holds every route; `Route::resource` gives the seven CRUD routes; route model binding gives a model or a 404.

5. `$request->validate` redirects back with errors and old input; Blade escapes with `{{ }}` and protects forms with `@csrf`.

The Laravel and ASP.NET Core versions share the database design, the pages and the rules; they differ in where things live and how much is done by convention. Book: Section 7.16.

Next Time

Chapter 8 — Evaluation and Discussion II. The same feature set in ASP.NET Core and Laravel, problem by problem.

Before then: run `tools/enum_demo.php`, build the Laravel application on your computer, and work Exercises 7.3 (a price-quote route) and 7.5 (a migration that adds a rack column).

Book: Sections 7.17 and 7.18.

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

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 8

Lecturer: Irfan Subakti — yifana@gmail.com

Two applications now do the same things with the same data: the ASP.NET Core version of Chapters 5 and 6 and the Laravel version of Chapter 7. That is exactly when differences show. Today we walk through seven problems the owner raised in the weeks after both went live, and solve each in both frameworks. Book: Chapter 8, Sections 8.1–8.13.

What This Meeting Covers

1. The routine — six steps, and where each problem belongs.

2. Rules in plain classes — 8.1 one price rule, 8.2 the order workflow.

3. Queries — 8.3 the order list without N+1, 8.4 a search that finds Sari.

4. Limits and reports — 8.5 no overpayment, 8.6 the daily revenue report.

5. Schema change — 8.7 a rack for every finished bag; then all seven side by side.

What was run: two small check programs, `CoreCheck` in C# and `tools/core_check.php` in plain PHP, and the PHP demonstration of the float mistake. Their outputs today are real. The web applications could not be run where the book was written (no NuGet, no Packagist): the C# code was compiled against compile-only stand-ins for EF Core and every PHP file passed `php -l`. Book: Section 8.2.2.

Six Steps, Seven Problems

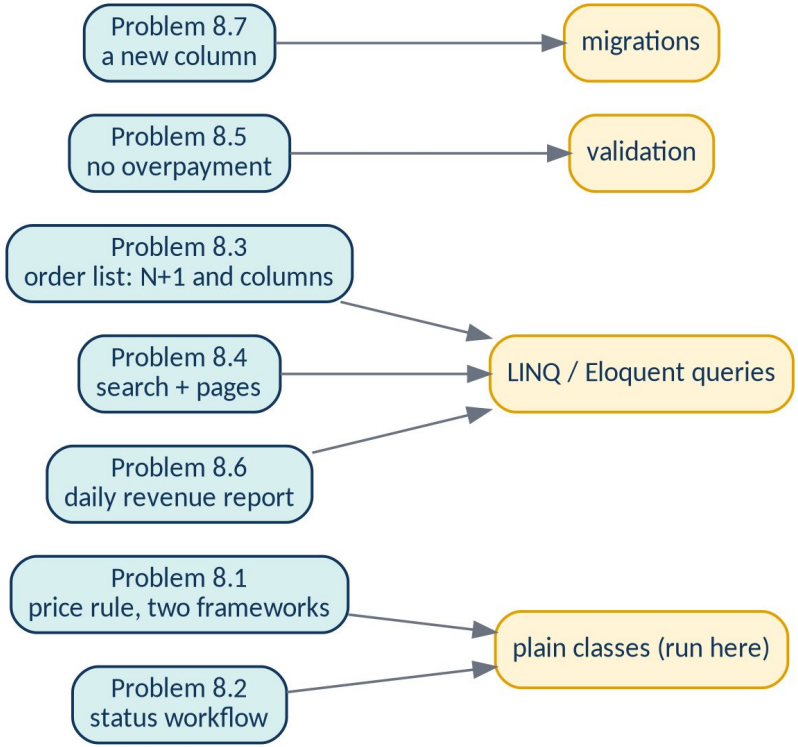


Figure 8.1 The seven problems of this chapter and the layer each one belongs to.

Figure 8.1: the six steps: state the problem in the shop's words; work an example by hand; decide the layer (plain rule, query, controller, view, schema); write it in one framework; port and compare; try to break it. Deciding the layer first is what keeps the two versions equal. Book: Section 8.2.1.

Problem 8.1: One Price Rule, Two Frameworks

The rules of Chapter 3: per-kg services at least 3 kg; per item or pair in whole units, rounded up; pickup Rp 10.000, free from a subtotal of Rp 100.000.

C# computes them in `PriceCalculator` with `decimal`. The Chapter 7 Laravel code computed them inline with PHP floats, and matched only because of `round`.

By hand: 4.5 kg at Rp 8.000 is Rp 36.000; 2 kg is billed as 3 kg, Rp 24.000; 4.02 kg is Rp 32.160; 1.5 pairs of shoes at Rp 45.000 are billed as 2, Rp 90.000.

The owner's worry is simple: a customer who uses both outlets must never see two prices for the same bag. Ask the room to compute 4.02 kg and 1.5 pairs by hand before showing any code; the hand answers are the test cases of the check programs. Book: Sections 8.3.1 and 8.3.2.

The Rule in Whole Numbers

$$b = \begin{cases} \max(h,\ 300) & \text{per kg} \\ 100 \cdot \lceil h/100 \rceil & \text{per item or pair} \end{cases} \tag{8.1}$$

$$S = \left\lfloor \frac{P \cdot b + 50}{100} \right\rfloor \tag{8.2}$$

The counter accepts at most two decimals, so every quantity is a whole number h of hundredths: 4.02 kg is 402. Equation (8.1) gives the billed quantity b in hundredths; Equation (8.2) gives the subtotal in rupiah for a price P, rounded half up. For 4.02 kg: $b = 402$ and $S = \lfloor 3216050/100 \rfloor = 32160$. Only multiplication, addition and whole-number division: no rounding surprises. Book: Section 8.3.2.

The Design: a Plain Class and a Check Program

The rule needs neither a request nor the database, so it belongs in a **plain class**: `App\Domain\PriceCalculator` in PHP, next to `PriceCalculator` in C#.

The PHP class receives the quantity as a *string*, as it comes 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; equal output proves equal rules.

```
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
  Shoes deep clean          1.50 pair          90000
```

Part of the real output, identical from `CoreCheck` and from `core_check.php`. Note the bed cover: Rp 70.000 plus the pickup fee, because the subtotal is under Rp 100.000. Book: Sections 8.3.3–8.3.5.

The Tempting Wrong Solution: Floats

```
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 real output of `tools/wrong_price.php`. 4.02 cannot be stored exactly as a binary float, so `8000 * 4.02` is 32159.999..., and casting with `(int)` loses a rupiah. `round` hides it here, but a rule that is right "because of `round`" is fragile. Money and quantities are safest as integers. Book: Section 8.3.6.

Problem 8.2: The Order Workflow in One Place

A customer left with her laundry without paying the Rp 16.000 she owed. New rule: *an order may not be marked as picked up while money is owed.*

The workflow queued → finished → departed moves into a plain class, `OrderWorkflow`, in both languages, with the new rule.

```
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.
```

The real output of both check programs. The tempting wrong solution in C#: load the order with `FindAsync`, as Chapter 5 did, and ask for `order.Balance`. `Payments` is not loaded, so `Paid` is 0 and every order is refused at pickup, paid or not. The fix is `Include(o => o.Payments)`; in Laravel, `sum('amount')` on the relationship. Book: Section 8.4.

Problem 8.3: The Order List Without N+1

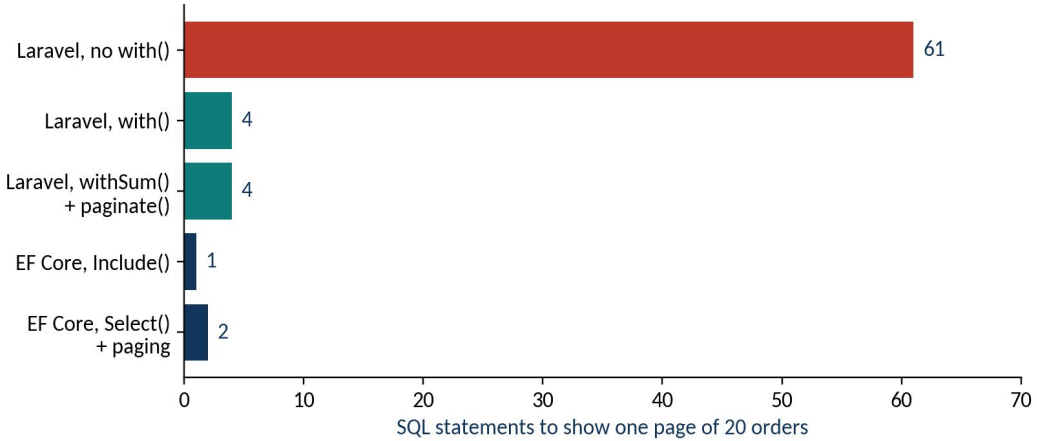


Figure 8.2 SQL statements needed to show one page of 20 orders, by the way the list is loaded; the counts follow from how each call is documented to load data.

Figure 8.2: in Laravel without eager loading, the view touches three relationships per order: 1 + 3 × 20 = 61 statements. With `with([...])`, as in Chapter 7: 4. In EF Core with three `Include`s: 1 statement, but with every column of four tables and every payment row, while the page needs seven values per order. Book: Section 8.5.2.

Page and Project in the Database

ASP.NET Core: `Select` into a small record `OrderRow` with exactly the columns of the list, and `PagedList` with `Skip` and `Take`: two statements, one for the count and one for the page.

Laravel: `select` the columns, `with` on the needed columns only, `withSum('payments', 'amount')`, and `paginate(20)`: four statements.

Keep the query an `IQueryable` until the last step; a `ToList` before `Where`, `Skip` or `Sum` reads the whole table into memory.

The same page, the same numbers, a fraction of the work. With thousands of orders the difference is the difference between a page that opens and a page that times out. Book: Sections 8.5.3–8.5.6.

Problem 8.4: A Customer Search That Finds Sari

The cashier types `sari`: the ASP.NET Core application finds nobody, the Laravel application finds Sari Wulandari. `Contains` becomes `instr`, case-sensitive on SQLite; `LIKE` is not.

The owner asks for the same search in both: ignore case, match name or phone, 15 customers per page, keep the search between pages.

C#: `EF.Functions.Like`, `OrderBy` then `ThenBy`, one shared `_Pager` partial. **Laravel:** a grouped `where`, `paginate(15)->withQueryString()`.

Know what your provider's string functions do: the same LINQ can mean different SQL on SQLite, SQL Server or MySQL. Book: Sections 8.6.1–8.6.5.

Two Tempting Wrong Solutions of 8.4

Ungrouped `orWhere`: `where name like ? or phone like ?` is right until someone adds "only this outlet": `and` binds more tightly than `or`, and every matching name appears, whatever its outlet. The closure `where(fn ($q) => ...)` adds the parentheses.

Paging without `OrderBy`: SQLite may return rows in a different order for page 1 and page 2, so a customer appears on both or on neither. EF Core even warns in the log.

Order by a unique column last, such as `Id`, so that the order is complete. Both mistakes pass every test with ten customers. Book: Section 8.6.6.

Problem 8.5: No Overpayment

A cashier typed Rp 60.000 instead of Rp 16.000, and the order showed a balance of −Rp 44.000. Rule: *a payment may not exceed what is still owed, and the message says how much that is.*

C#: compute the balance in one query on the server, then `ModelState.AddModelError`. **Laravel:** the rule `"max:{$balance}"`, built from the balance just read.

Wrong: comparing with the *total* (accepts Rp 36.000 on top of Rp 20.000 paid), or with a balance sent in a hidden field (anyone can edit it).

Limits that depend on stored data are computed on the server at the moment of the request, never taken from the form. The book also discusses two cashiers paying the same order in the same second: 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. Book: Section 8.7.

Problem 8.6: The Daily Revenue Report

Every evening the owner compares the drawer, the QRIS statement and the bank's transfer list with the system: *money received per day, split by payment method, for a chosen week.*

C#: `GroupBy` date and method in the database into `DailyRevenueRow`, then pivot in memory. **Laravel:** `selectRaw` with `groupByRaw`, then collections.

Days are ranges from midnight to midnight in the shop's time zone, `Asia/Jakarta`.

Wrong: grouping by `PaidAt` instead of its date gives one line per payment. Storing amounts as `decimal` would make `Sum` fail on SQLite in EF Core; the report works because Chapter 5 stored whole rupiah. And a Laravel application left on UTC counts the first seven hours of the day as yesterday. Aggregate in SQL, shape in memory. Book: Section 8.8.

Problem 8.7: A Rack for Every Finished Bag

Finished bags wait on numbered racks. A *rack* field such as `B-03` (a capital letter, a dash, two digits), optional, entered at the counter and shown on the order page.

C#: a property, a rule in `OrderForm`, `dotnet ef migrations add`. **Laravel:** a hand-written migration with `Schema::table`, `up` adds the column and `down` drops it.

Wrong: editing the migration that created the orders table. On a laptop `migrate:fresh` hides it; on the server the old migration is recorded as applied and is skipped, the column never appears, and the first order with a rack fails.

A migration that has left your computer is history. Schema changes are always new migrations. Book: Section 8.9.

The Problems Side by Side

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`	integer hundredths	same rule, proven equal
8.2 workflow	`OrderWorkflow` + `Include`	`OrderWorkflow` + `sum`	load what you compute from
8.3 list	`OrderRow`, `PagedList`	`withSum`, `paginate`	page and project in the database
8.4 search	`EF.Functions.Like`	grouped `where`	know your provider
8.5 payment	balance, `ModelState`	`"max:{$balance}"`	limits from the database
8.6 report	`GroupBy`, pivot	`selectRaw`, collections	aggregate in SQL
8.7 rack	`migrations add`	`Schema::table`	always a new migration

Table 8.1: read the last column aloud. None of the lessons belongs to one framework; each is a habit that makes any framework safe to use. Book: Section 8.10.

Common Mistakes

Porting code instead of the requirement; **computing from relationships that were not loaded**; **moving work to memory by accident**.

Paging without a complete order; **ungrouped `orWhere`**; **trusting numbers from the form**.

Grouping by a timestamp instead of its date, and the wrong time zone; **editing a migration that has already run**.

The first mistake is the root of the others. The Chapter 7 price code was a faithful port of the C# code, floats and all, and it matched only because of `round`. Port the rule, stated in the shop's words and worked by hand, and then prove equality on fixed cases with a check program. That habit would have caught Problems 8.1, 8.2 and 8.4 before any customer did. Book: Section 8.12.

Summary: Five Things to Remember

1. Six steps: state, work by hand, decide the layer, write, port and compare, try to break (Figure 8.1).

2. Rules that need neither web nor database live in plain classes, checked by console programs; integers for money (Equation (8.2)).

3. Lists are paged and projected in the database: 2 statements in EF Core, 4 in Eloquent, instead of 61 (Figure 8.2).

4. Limits come from stored data on the server; reports group in SQL by day in the shop’s time zone.

5. Schema changes are new migrations, never edits to old ones.

The two applications are now equal in behaviour, not only in appearance. That is the standard for any port between frameworks: the same inputs give the same outputs, and a program proves it. Book: Section 8.13.

Next Time

Chapter 9 — Laravel II. Migrations and seeders in depth, middleware, and form requests.

Before then: run `CoreCheck` and `core_check.php` and compare their output line by line; run `wrong_price.php`; work the chapter's exercises.

A good exercise for pairs: one student adds a new price rule in C#, the other in PHP, without looking; then both check programs must still print identical lines. Book: Sections 8.14 and 8.15.

Lecture 9 — Laravel II: Migrations, Seeders, Middleware and Form Requests

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 9

Lecturer: Irfan Subakti — yifana@gmail.com

Today the Laravel application grows up. Orders learn which outlet took them in, through a migration on a table that already holds data; a believable month of test data is made by factories and seeders; middleware closes orders that have left the shop; and form requests put the counter's checklists in one place. Book: Chapter 9, Sections 9.1–9.12.

What This Meeting Covers

1. The third outlet — four things that started to hurt at once.

2. Migrations in depth — batches, changing a table with data, the commands, four rules.

3. Factories and seeders — invented but plausible data, real data always, made-up data only where it belongs.

4. Middleware — the onion in plain PHP, a “before” and an “after” middleware, registering and attaching them.

5. Form requests — the checklist of one kind of request, and controllers that trust their input.

As in Chapter 7, Laravel could not be installed where the book was written; the steps were written from the code and the Laravel 13 documentation, every PHP file was checked, and the plain-PHP pipeline demo was run, so its output today is real. Book: Chapter 9 and Section 9.8.

Bu Wati's Third Outlet

Rungkut since 2019, Keputih since 2022, Wiyung since January 2026. Customers move between outlets, but the application does not know where an order was taken in.

The database: Dimas tries changes on his laptop, then on the shop's computer; a week later nobody knows which computer has which columns.

Test data: a counter page and an order list cannot be judged with three sample orders.

Money: a finished order was paid twice, in cash and by QRIS, and nothing stopped a payment on an order the customer had already taken home. Meanwhile the lists of validation rules kept growing inside the controllers.

Each pain has a Laravel tool: migrations as a logbook, factories and seeders for a believable month, middleware as checkpoints, form requests as the counter checklist. Book: Section 9.1.

The Idea: Checkpoints Before the Controller

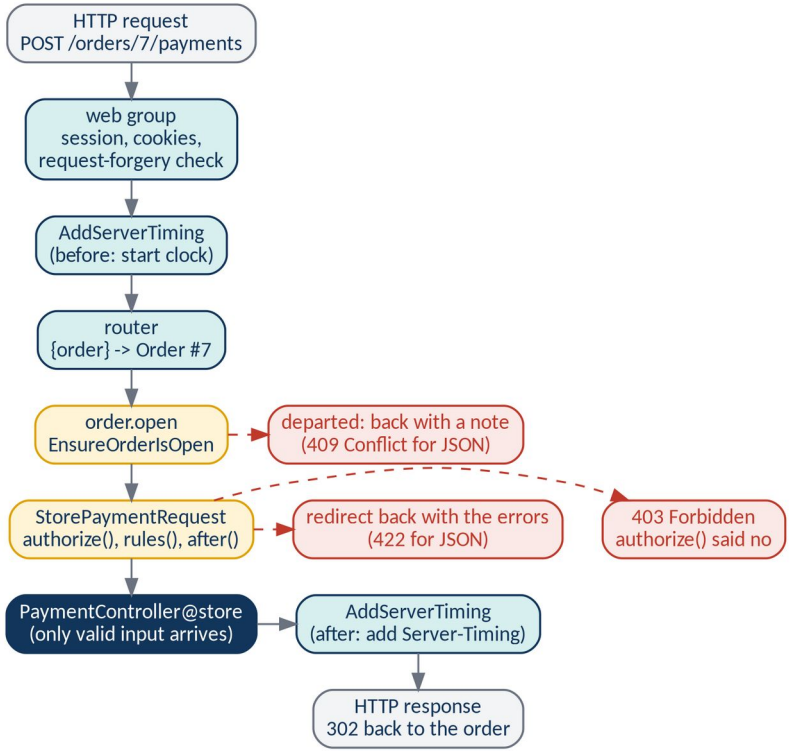


Figure 9.1 The path of one request, a payment on order 7, through middleware, route model binding and a form request; the red boxes are the places where the request stops before the controller.

Figure 9.1: middleware runs first, route model binding turns `{order}` into an `Order`, the form request checks the input, and only then does the controller run. Each red box can stop the request: a closed order, a missing order (404), invalid input (back to the form). The controller that remains is short and trusts what it receives. Book: Section 9.2.

What the Database Remembers

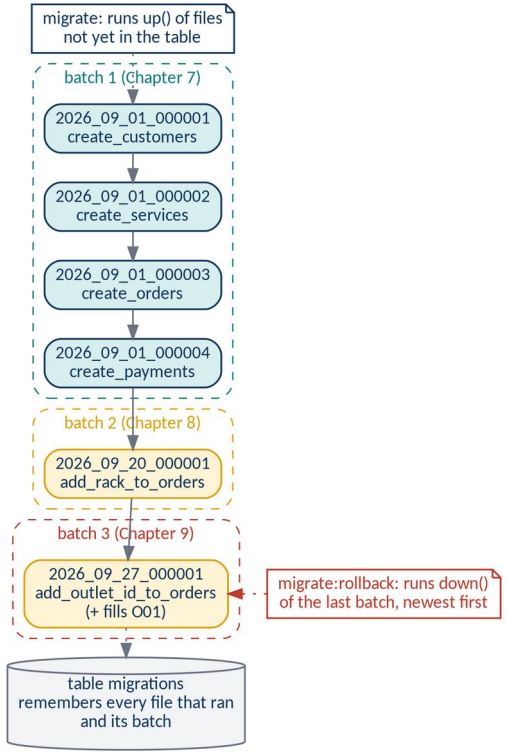


Figure 9.2 The migration history of the project: each run of `migrate` forms a batch, the `migrations` table records it, and `migrate:rollback` undoes only the last batch.

*Figure 9.2: a migration is a recorded, reversible change. The `migrations` table remembers which files ran and in which **batch**; `migrate` runs the missing files as one new batch, `migrate:rollback` runs `down()` of the last batch, newest first. This is why Dimas's two computers can be brought to the same state: each knows exactly what it has. Book: Section 9.4.1.*

Changing a Table That Already Holds Orders

Listing 9.1 The two methods of `2026_09_27_000001_add_outlet_id_to_orders_table.php`: a new column with a foreign key, and the old orders filled in.

```
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');
    });
}
```

*Listing 9.1: a required column cannot be added to a table with rows, so it is added **nullable**, then filled: every earlier order belongs to Rungkut, `001`. The data change uses `DB::table()`, not the `Order` model, because a model can change later in ways the old table cannot satisfy. `down()` drops the foreign key and the column: a true undo. Book: Section 9.4.2.*

The Commands, and Four Rules

Table 9.1 Some migration commands of Artisan and when to use each.

Command	Use it when
`php artisan migrate:status`	before and after every change
`php artisan migrate`	to apply your changes as one batch
`php artisan migrate --pretend`	to review the SQL on a real database
`php artisan migrate:rollback`	to undo the last batch
`php artisan migrate:fresh --seed`	on your own computer only

Table 9.1. The four rules, which hold for EF Core too: never edit a migration that has run anywhere but your own computer; make every `down()` a true undo and test it once; add columns nullable or with a default, then fill them; keep data changes in migrations small and written with the query builder. Book: Sections 9.4.3 and 9.4.4.

Factories: One Plausible Record

A **factory** describes one invented but plausible record: Faker for names and addresses, related factories or closures for foreign keys.

`OrderFactory` prices each order by the **real rule** in `afterMaking`, instead of inventing a total the application could never produce.

States name variations: `finished()`, `departed()`. `BusyMonthSeeder` gives each order a plausible history: seven in ten paid at drop-off, finished after two days, departed after five only if fully paid, because Chapter 8's workflow would allow nothing else.

The model needs the `HasFactory` trait. Random values where the rules decide create data that lies: a report built on invented totals shows numbers the real shop could never have. Book: Sections 9.5.1 and 9.5.2.

Seeders: Real Data Always, Made-Up Data Only Here

`PriceListSeeder` puts the real price list and outlets into every database with `updateOrCreate`, so it is safe to run again (**idempotent**).

`BusyMonthSeeder` invents forty customers and a month of orders that move through the workflow as real ones do.

`DatabaseSeeder` always calls the price list, and calls the busy month only on a developer's or a test computer, guarded by the environment.

> **WARNING.** A seeder that calls the demo seeders unconditionally puts invented customers into the shop's real system the first time somebody runs `db:seed` there.

A believable month makes the counter page, the order list with its pages and the daily report of Chapter 8 worth looking at. Book: Sections 9.5.3 and 9.5.4.

The Guard in `DatabaseSeeder`

Listing 9.2 `database/seeders/DatabaseSeeder.php`: real data always, made-up data only on a developer's or a test computer.

```
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
        }
    }
}
```

Listing 9.2: two lines carry the whole policy. `app()->environment('local', 'testing')` *is true on a developer's laptop and in the test runner, and false on the shop's computer, where* `APP_ENV=production`*. So* `php artisan db:seed` *is safe to type anywhere: the price list and the outlets are brought up to date, and nothing invented appears where the owner counts real money. The same idea protected the demo accounts in Chapter 6, which lived only in* `appsettings.Development.json`*. Book: Section 9.5.3.*

Middleware: the Onion, Without Laravel

```
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"}
```

The real output of `tools/pipeline_demo.php`: forty lines of plain PHP in which each layer is a closure receiving the request and `$next`, and `array_reduce` wraps the controller in the layers. For the departed order the inner layer never calls `$next`, so the controller does not run, yet the outer timing layer still finishes its work on the way out. This is Chapter 3's onion in PHP. Book: Section 9.6.1.

A “Before” Middleware for a Rule of the Shop

Listing 9.3 The `handle` method of `app/Http/Middleware/EnsureOrderIsOpen.php`: nothing may change an order that has left the shop.

```
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);
}
```

*Listing 9.3: an order that has left the shop is history. People get a note on the page they came from; programs get **409 Conflict**. Every path returns a response: its own, or the one from `$next`. The route parameter is available only because this is a route middleware; a global one runs before routing and would see `null`. Book: Section 9.6.2.*

An “After” Middleware, Registering and Attaching

`AddServerTiming` calls `$next` first, then adds a `Server-Timing` header with the time the rest of the pipeline took.

In `bootstrap/app.php`: `AddServerTiming` joins the `web` group; `EnsureOrderIsOpen` gets the short name (alias) `order.open`.

In `routes/web.php`, the routes that change an order, advance and payments, are grouped behind `Route::middleware('order.open')`.

Laravel’s built-in route middleware used in this book includes `auth`, `guest`, `throttle:5,1`, `can:manage-prices`, `verified` and `signed`. Group the routes once, and a new route added to the group next year is protected without anyone remembering. Book: Sections 9.6.3–9.6.5.

Form Requests: the Counter Checklist

A **form request** holds the checklist of one kind of request: `authorize`, `prepareForValidation` (tidy the input), `rules`, `messages`, `attributes` and `after`.

`StoreOrderRequest` tidies the input first (a comma in `4,5` becomes a dot, the rack is trimmed and upper-cased), then checks the fields and names them in the cashier's words.

The controller asks for it as a typed parameter; if anything fails, the controller never runs and the user goes back to the form with the messages.

It is Chapter 6's form class and Chapter 7's `$request->validate` joined into one class, found by name in `app/Http/Requests`. Book: Section 9.7.1.

A Rule That Needs the Order

Listing 9.4 The `after` method of `app/Http/Requests/StorePaymentRequest.php`: a check against what is still owed, run after the simple rules 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.');
            }
        },
    ];
}
```

Listing 9.4: the simple rules come first (amount an integer of at least 1.000, method one of the `PaymentMethod` cases); only when they pass does `after` load the order's payments and compare with the balance. That is Problem 8.5 in its proper home, and `PaymentController::store` shrinks to a few lines that use `$request->validated()`. Book: Sections 9.7.2 and 9.7.3.

Running It and Making It Better

On your computer: `migrate:status`, `migrate`, `migrate:rollback`, `migrate` again; then `migrate:fresh --seed` and look at a busy month.

Try to pay on a departed order, from the page and with a JSON request; try to pay more than the balance.

Look at the `Server-Timing` header in the browser's developer tools, under the network tab.

The exercises extend both tools: a migration that fills a new column row by row in chunks, and a `LogSlowRequests` middleware that takes a parameter, the number of milliseconds. Book: Sections 9.8, 9.9 and 9.14.

Using It for Real

Any business with more than one location needs “where did this happen?” on its records: a clinic with two branches, a shop with a stall at the market.

Rules like “a closed record cannot change” belong in middleware or policies, not in each controller: a paid invoice, a delivered parcel, a signed prescription.

The migration pattern of today (nullable, fill, tighten later) is how every real system grows without stopping. Book: Section 9.10.

Common Mistakes

Editing a migration that has already run; **a required column on a table with rows**; **models inside migrations**; **a `down()` that does not undo `up()`**.

Made-up data in the real database; **random values where the rules decide**.

Forgetting `return $next($request)`: every request gets an empty response. **A global middleware that needs a route parameter**.

`$request->all()` after validation: use `validated()` or `safe()`. **Database checks in `rules()` without guarding**: put them in `after()`.

The double payment of Section 9.1 is the story behind two of these tools: middleware closes the departed order, and the form request refuses more than the balance. Neither needed a line in the controller. Book: Section 9.11.

Summary: Five Things to Remember

1. Migrations are recorded, reversible changes in batches (Figure 9.2); a table with data gets a nullable column, filled with the query builder.
2. Factories describe one plausible record with the real rules; seeders put real reference data everywhere and made-up data only where it belongs.
3. Middleware wraps the controller like an onion; “before” may stop the request, “after” changes the response.
4. `EnsureOrderIsOpen` closes departed orders on every route that changes one: a note for people, 409 for programs.
5. A form request is the checklist of one request; the controller runs only when everything passed (Figure 9.1).

Compare with ASP.NET Core: migrations there are generated from the classes, here written by hand, but both keep a history table and both need the same four rules; middleware there is registered in `Program.cs`, here in `bootstrap/app.php`; form requests play the part of form classes plus `ModelState`. The ideas travel; only their addresses change. Book: Section 9.12.

Next Time

Chapter 10 — Laravel III. Authentication, API resources and testing.

Before then: run `tools/pipeline_demo.php` and add a third layer that logs the order number; on your computer, run the migration and roll it back once.

Book: Sections 9.13 and 9.14.

Lecture 10 — Laravel III: Authentication, API Resources and Testing

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 10

Lecturer: Irfan Subakti — yifana@gmail.com

Today the Laravel application learns who is asking. Staff sign in with a session, the owner alone may change prices, drivers' phones get an API with tokens from Laravel Sanctum, API resources decide exactly what the JSON contains, and automated tests check every rule, especially the refusals. Book: Chapter 10, Sections 10.1–10.12.

What This Meeting Covers

1. A price list anyone could change — and drivers who need an app, not a browser.

2. Users and roles — owner and cashier; a hand-written sign-in; routes and a gate.

3. An API for programs — Sanctum tokens, API routes, one way to create an order.

4. API resources — the JSON shape of an order, decided on purpose.

5. Testing — a unit test for the price rule, feature tests for sign-in and the API.

Laravel could not be installed where the book was written, so the code was written against the Laravel 13 and Sanctum documentation and checked with `php -l`; the tests are meant to be run by you, with `php artisan test`, on your own computer. Book: Chapter 10 and Section 10.8.

A Price List Anyone Could Change

One Saturday in September, Keputih charged Rp 6.000 per kilogram instead of Rp 8.000. A new part-time cashier had opened **Prices** to *look*, clicked **Edit** on the wrong line and saved.

The application did not ask who he was, so it could not stop him, and it could not say afterwards who had done it.

The same week Bu Wati hired two drivers. They need to see the orders to collect, mark a bag as delivered and record payments at the door, from a small phone app.

An app is not a browser: it cannot fill in Blade forms. It needs an **API**, a set of addresses that speak JSON, and a safe way to prove who is asking.

Two stories, one theme: the application must know who is asking and decide what they may do, for people at a screen and for programs on a phone. Book: Section 10.1.

Sessions for Browsers, Tokens for Programs

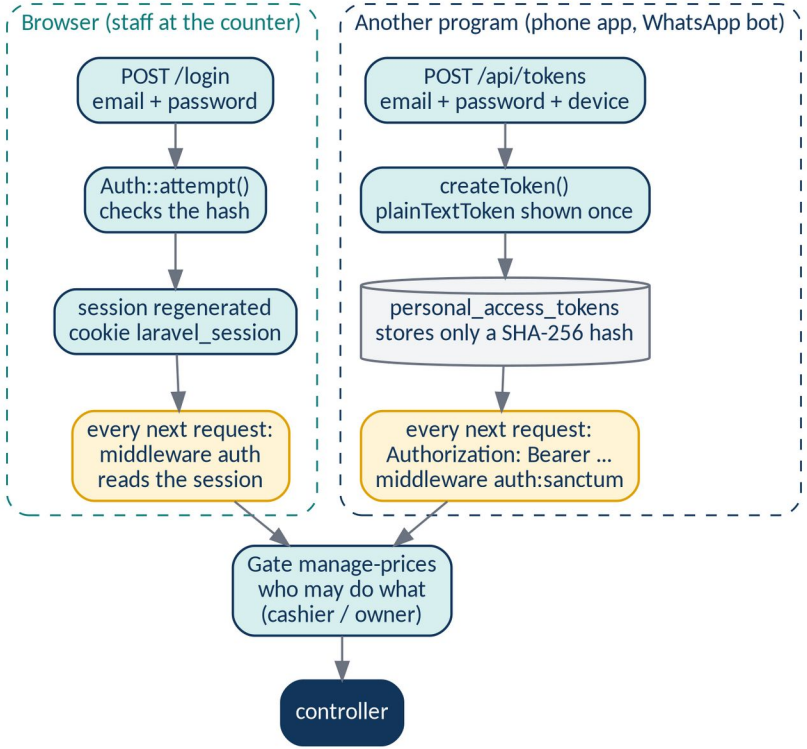


Figure 10.1 Two ways to prove who is asking: a browser signs in once and carries a session cookie; a program exchanges the password for a token and sends it with every request; both end at the same authorization check.

Figure 10.1: ***authentication*** *answers "who are you?",* ***authorization*** *"may you do this?". A browser signs in once and the session cookie travels with every request. A program trades e-mail and password for a* ***token*** *and sends it as* `Authorization: Bearer …` *with every call. Both roads end at the same check. Book: Sections 10.2.1 and 10.2.2.*

Users and Roles

`App\Enums\Role`: `Owner` and `Cashier`, a backed enum stored in a new `role` column; every user is a cashier unless told otherwise.

`User` has `#[Hidden(['password', 'remember_token'])]` and the `hashed` cast: a password is stored only as a hash, and never appears in JSON.

`UserFactory` makes cashiers by default and has an `owner()` state; `DatabaseSeeder` creates two demo accounts on developers' computers only.

The demo accounts, `cashier@laundry.test` and `owner@laundry.test` with the password `password`, follow Chapter 9's guard: they never reach the shop's real database. Book: Section 10.3.

The Sign-In Controller

Listing 10.1 The `store` method of `app/Http/Controllers/Auth/LoginController.php`: sign in.

```
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'));
}
```

Listing 10.1: `Auth::attempt` checks the password against the hash. One message for "no such user" and "wrong password", so a stranger learns nothing about which accounts exist. `regenerate()` gives a new session id after sign-in, closing the door to session fixation. `intended` returns the user to the page they first asked for. Sign-out is a `POST` that logs out, invalidates the session and regenerates the CSRF token. Book: Section 10.4.1.

Guarding the Routes, and a Gate

`routes/web.php` has four groups: **public** pages; **guest-only** sign-in, with `throttle`; **staff** pages behind `auth`; the **price list** behind `can:manage-prices`.

Listing 10.2 The one line in `boot()` of `app/Providers/AppServiceProvider.php`: the gate `manage-prices`, for the owner only.

```
// Only the owner changes the price list.
Gate::define('manage-prices', fn (User $user) => $user->isOwner());
```

*Listing 10.2: a **gate** is a named yes-or-no rule about the current user, the Laravel cousin of Chapter 6's named policy. Routes use it as the middleware `can:manage-prices`, and Blade as `@can('manage-prices')` to show the link only to the owner. Hiding the link is tidiness; the middleware is the protection. Book: Sections 10.4.2 and 10.4.3.*

The Refusals, Each With Its Code

Table 10.1 The refusals of this chapter's application: status code, meaning and an example.

Code	Meaning	Example
302 to `/login`	a guest asked for a staff page	opening `/counter` signed out
401	no valid session or token (JSON)	API call without a token
403	signed in, but not allowed	a cashier opening `/prices`
409	the order is closed	paying a departed order by API
419	CSRF token missing or old	a form left open overnight
422	the input breaks a rule	paying more than the balance
429	too many attempts	the sixth sign-in in a minute

Table 10.1: each refusal has its own code, and a program on a phone can act on each: ask for the password again on 401, show "not allowed" on 403, show the field messages on 422, wait on 429. Learn this table; it is the same in ASP.NET Core and Spring Boot. Book: Section 10.4.

An API for Programs: Sanctum Tokens

Listing 10.3 The heart of `TokenController::store` in `app/Http/Controllers/Api/TokenController.php`: a password in, a token out.

```
$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);
```

Listing 10.3: `php artisan install:api` *installs* ***Sanctum*** *and creates* `routes/api.php`. `POST /api/tokens` *with e-mail, password and a device name returns 201 and a token, shown once; Sanctum stores only its hash.* `DELETE /api/tokens/current` *revokes the token of this device, so a lost phone can be signed out alone. Book: Sections 10.5.1 and 10.5.2.*

The API Routes, and One Way to Create an Order

`routes/api.php`: tokens, customers, orders and payments, behind `auth:sanctum`, reusing the form requests and the `order.open` middleware of Chapter 9.

An **action class**, `CreateOrder`, is the one place where checked input becomes a priced order; the web counter and the API both call it.

`UpdateOrderStatusRequest` lets a program ask for a named status; Chapter 8's `OrderWorkflow` decides whether that is the allowed next step.

Without the action class, the API would grow its own copy of "create an order", and the two copies would drift apart exactly as the price rules did before Chapter 8. Book: Sections 10.5.3 and 10.5.4.

The Action Class `CreateOrder`

Listing 10.4 The class in `app/Actions/CreateOrder.php`: the one place where checked input becomes a priced order.

```
// 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(),
        ]);
    }
}
```

Listing 10.4: the input arrives already checked by a form request, from the web counter or from the API. The action looks up the service (404 if it vanished), computes the total with Chapter 8's integer `PriceCalculator`, and saves a queued order with the time of receipt. The total is never taken from the request: a phone app cannot send its own price. The web `OrderController::store` and the API

`OrderController::store` each shrink to a call of `handle` and a response in their own language, a redirect or a resource with 201. Book: Section 10.5.4.

API Resources: Deciding What an Answer Contains

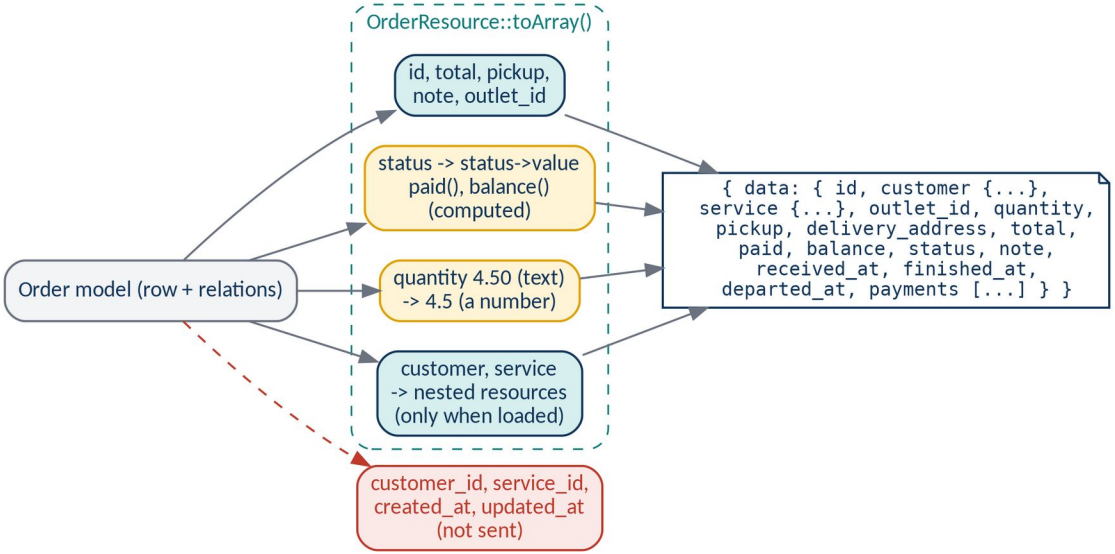


Figure 10.2 What `OrderResource` does with an order: fields passed through, converted or computed, relations nested when loaded, and internal columns left out.

*Figure 10.2: returning a model from an API makes the JSON follow the database, including columns nobody meant to publish. A **resource** lists every field on purpose: some passed through, some converted (quantity as a number, status as its string value), some computed (`paid`, `balance`), relations nested only when loaded, and internal columns left out. Book: Section 10.6.*

`OrderResource`

Listing 10.5 Part of `toArray` in `app/Http/Resources/OrderResource.php`: the JSON shape of an order.

```
return [
    'id' => $this->id,
    'customer' => new CustomerResource($this->whenLoaded('customer')),
    'service' => new ServiceResource($this->whenLoaded('service')),
    'quantity' => (float) $this->quantity,
    'total' => $this->total,
    'paid' => $this->paid(),
    'balance' => $this->balance(),
    'status' => $this->status->value,
    'received_at' => $this->received_at,
    'payments' => PaymentResource::collection(
        $this->whenLoaded('payments')),
];
```

Listing 10.5: `whenLoaded('customer')` includes the customer only if the controller loaded it with `with('customer')`, so a resource never triggers a query per order. A paginated collection adds `links` and `meta`, which the phone app uses to fetch the next page. Book: Section 10.6.

Testing: a Tireless Second Pair of Hands

Unit test: `PriceCalculatorTest` checks six price cases of the plain class, with no database and no web.

Feature tests send real requests through the whole application: `RefreshDatabase` gives each test a clean database; `actingAs($user)` and `Sanctum::actingAs` sign in without a form.

They check status codes, JSON paths and what is in the database afterwards.

`php artisan test` runs everything in `tests/Unit` and `tests/Feature`; `--filter=OrderApiTest` runs one class. When a test fails, the report names the method, the line of the assertion and what was expected. Book: Sections 10.2.4, 10.7.1 and 10.7.4.

Feature Tests for Signing In

Listing 10.6 Two methods of `tests/Feature/AuthTest.php`: guests and the owner's price list.

```
public function test_guests_are_sent_to_the_sign_in_page(): void
{
    $this->get('/counter')->assertRedirect('/login');
}

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();
}
```

Listing 10.6: the second test is the Saturday incident, written as a rule that is checked every time anyone changes anything. The full class also checks sign-in, a wrong password and an API token that opens the API and can be revoked. `OrderApiTest` checks prices, validation, the workflow, closed orders, payments and filters through the API. Book: Sections 10.7.2 and 10.7.3.

Running It as Each Person in the Story

Open `/counter`: the sign-in page. Sign in as the cashier: no **Prices** link, and typing `/prices` gives 403. Sign in as the owner: the link works.

A wrong password six times within a minute: the sixth attempt is refused with 429 before the password is even checked.

As the driver: ask for a token with `curl` (on Windows `curl.exe`), list the orders with it, add `?status=finished`.

Run `php artisan install:api` once, copy the chapter's files, `migrate:fresh --seed`, and start the server on port 4100, as the chapter's README describes. VS Code's REST Client extension is a friendly alternative to `curl`. Book: Section 10.8.

Making It Better and Using It for Real

Token abilities such as `orders:read` and `payments:write`, checked with `tokenCan`; **tokens that expire** via `expiration` in `config/sanctum.php`; `throttle:api` for the API; **policies** when one gate is no longer enough.

For real: one account per person, never shared; long passwords only their owners know; HTTPS everywhere outside your laptop; a token per device, revoked when the device is gone.

Customers' names, phone numbers and addresses are personal data under UU PDP 27/2022: sending only the needed fields in each resource and revoking access promptly are part of treating that data with care. And every time a mistake reaches the shop, write the test that would have caught it before you fix it. Book: Sections 10.9 and 10.10.

Common Mistakes

Checking the role in the view only; telling a stranger which part was wrong; forgetting `session()->regenerate()`; sign-out by GET.

Returning models instead of resources; `whenLoaded` without loading; storing or logging the plain token.

Tests that depend on each other or on your data; testing only the happy path.

Testing only the happy path is the quiet one. Most of the tests of this chapter check refusals: 401, 403, 409, 422, 429. Those are the rules that protect the shop, and they are exactly the ones that break silently, because a user who is wrongly allowed never complains. Book: Section 10.11.

Summary: Five Things to Remember

1. Authentication says who (failure 401), authorization says what (failure 403); a browser guest is redirected to sign in (Figure 10.1).

2. Roles as a backed enum; passwords only as hashes; a hand-written sign-in that regenerates the session; a gate for the price list.

3. Sanctum tokens: hashed, sent as bearer tokens, revoked per device; API routes behind `auth:sanctum`.

4. Resources decide the JSON field by field (Figure 10.2); an action class creates orders one way only.

5. Unit tests for plain classes, feature tests for the whole application with `RefreshDatabase`, `actingAs` and `Sanctum::actingAs`.

With this chapter the Laravel version has caught up with the ASP.NET Core version of Chapter 6, and gone one step further with an API and a test suite. Book: Section 10.12.

Next Time

Chapter 11 — Spring Boot I. The third framework: Spring Initializr, REST controllers and Spring Data JPA, in Java 25.

Before then: run the tests on your computer, break the gate on purpose and watch which test fails, and work the exercise that checks the price list on every route.

Book: Sections 10.13 and 10.14.

Lecture 11 — Spring Boot I: Initializr, REST Controllers and Spring Data JPA

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 11

Lecturer: Irfan Subakti — yifana@gmail.com

Today we meet the third framework. We build the Profitina Laundry API in Spring Boot 4.1 on Java 25: the same four tables, the same price list from the same `Service.csv`, the same rules and the same JSON field names as the Laravel API of Chapter 10. Along the way: the application context, starters and auto-configuration, JPA entities, Spring Data repositories and REST controllers. Book: Chapter 11, Sections 11.1–11.13.

What This Meeting Covers

1. Raka's question — would this look the same in Spring Boot?

2. The idea — a container that builds your objects, Boot's defaults, entities and repositories, controllers that speak JSON.

3. The project — Spring Initializr, `pom.xml`, the entry point, the settings.

4. The domain — four entities, statuses stored as agreed words, the rules in plain Java.

5. Repositories and controllers — queries from method names, records in and out, loading the price list.

What was run: Maven packages could not be downloaded where the book was written, so the Spring application was not started there. `LaundryRules.java` and the check program `RulesCheck.java` need only the JDK; they were compiled with `javac` and run, and their output today is real. Book: Chapter 11 and Section 11.8.

Raka's Question

Raka, who writes the drivers' phone app, studies informatics with Java on the back end. Reading the Laravel API he asked: *would this look the same in Spring Boot?*

Bu Wati has her own version: an accounting firm that keeps the books of several laundries in Surabaya wants to connect its system, and its system is written in Java.

Same four tables, same price list, same rules (3 kg minimum, whole items, pickup fee, free from Rp 100.000, laundry leaves only when paid), same JSON names.

Spring Boot 4.0 appeared in November 2025, 4.1 in June 2026; the book uses 4.1.1. It runs on Java 17 or newer; we use Java 25, the current long-term-support JDK. It embeds Tomcat 11, and under its data access sits Hibernate ORM 7. Chapter 12 then compares the two APIs field by field. Book: Section 11.1.

One Request Through Spring Boot

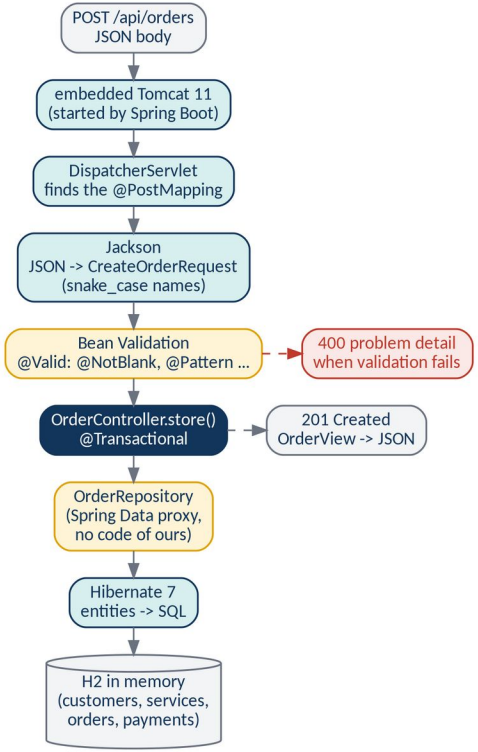


Figure 11.1 One `POST /api/orders` through a Spring Boot application: embedded Tomcat, the DispatcherServlet, Jackson, Bean Validation, the controller, a Spring Data repository, Hibernate and the database.

*Figure 11.1: Tomcat receives the bytes; the **DispatcherServlet** is Spring MVC's front controller and picks the method; **Jackson** turns the JSON body into a record; **Bean Validation** checks it; the controller calls a **Spring Data** repository; **Hibernate** writes the SQL. Same eight steps as Chapter 1, new names. Book: Section 11.2.4.*

The Idea in Plain Words

The application context is Spring's DI container: it scans the packages below `@SpringBootApplication`, creates **beans** from classes marked `@RestController`, `@Service`, `@Repository`, and injects them through constructors.

Boot adds sensible defaults switched on by what you depend on: put `spring-boot-starter-webmvc` in the build and a web server appears; add H2 and a data source appears. This is **auto-configuration**.

Spring Data JPA writes repository classes for you from interfaces; **Hibernate** writes the SQL.

Chapter 2's four foundations again: the router is the DispatcherServlet with `@GetMapping`, the container is the application context, the ORM is JPA with Hibernate, and there are no templates today because the API answers JSON; Thymeleaf comes in Chapter 13. Book: Sections 11.2.1–11.2.3.

The Same Jobs in Laravel and Spring Boot

Table 11.1 The same jobs in Laravel and in Spring Boot.

Job	Laravel	Spring Boot
start a project	`composer create-project`	Spring Initializr
dependencies	`composer.json`	`pom.xml` (Maven)
settings	`.env`, `config/*.php`	`application.properties`
the model	Eloquent model	JPA `@Entity`
queries	`Order::where(...)->get()`	repository methods
routes	`routes/api.php`	`@GetMapping`, `@PostMapping`
input rules	form requests	records with Jakarta Validation
JSON shape	API resources	response records (Jackson)
seed data	seeders	a `CommandLineRunner` bean

Table 11.1: one row differs in kind. Laravel has migrations from the start; here the schema comes from Hibernate's `ddl-auto`, fine for an in-memory database, while a real database needs Flyway or Liquibase later. Book: Section 11.2.

Creating the Project

Spring Initializr (start.spring.io, or the VS Code Spring Boot extension): Maven, Java 25, Spring Boot 4.1.1, dependencies for web, JPA, validation, H2 and tests.

`pom.xml` inherits from `spring-boot-starter-parent` 4.1.1, which fixes tested versions of every library, so our five dependencies need no version numbers.

`LaundryApplication.java` has `@SpringBootApplication` and a `main` that calls `SpringApplication.run`; everything else is found by scanning.

Every class must live in `com.profitina.laundry` or below. A controller in a sibling package is never scanned, and its URLs answer 404 with no error at start-up. Book: Section 11.3.

Settings

Listing 11.1 `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
```

Listing 11.1: H2 in memory is created at start and dropped at stop. `open-in-view=false` *means lazy relations can be read only inside a transaction, which forces us to load what we need on purpose.* `SNAKE_CASE` *makes* `customerId` *appear as* `customer_id`*, as in Laravel. Errors come out as* ***problem details****, a standard JSON shape. Book: Section 11.3.4.*

The Domain: Four Entities

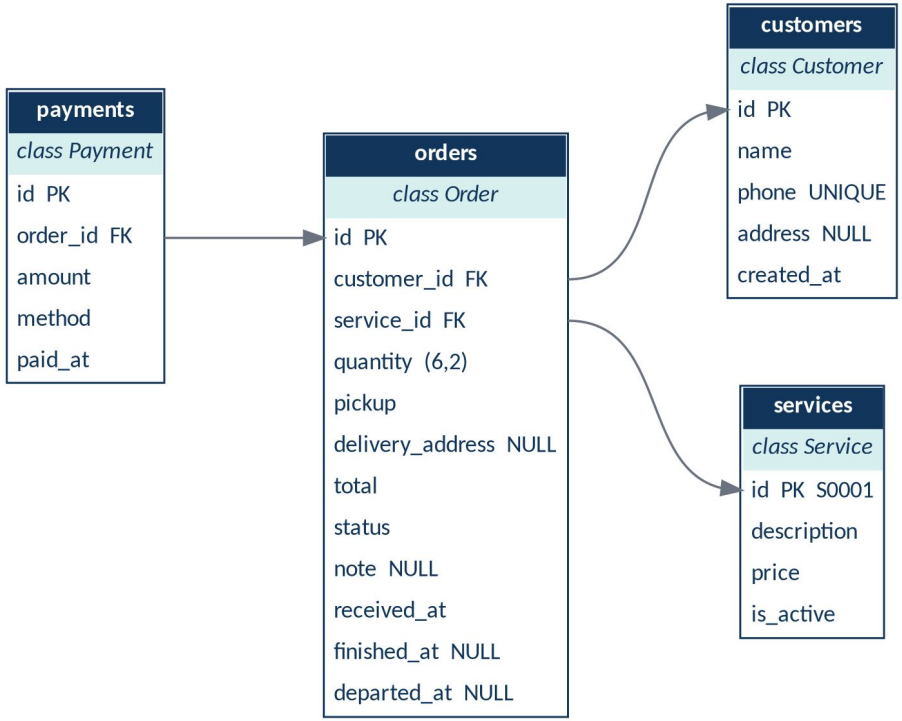


Figure 11.2 The four tables of Profitina Laundry and the entity classes that map them; arrows go from a foreign key to the primary key it refers to.

Figure 11.2: `Customer`, `Service`, `Order`, `Payment`. The table for orders is `orders`, because `ORDER` is an SQL keyword. `Service` is keyed by its own code such as `S0001`, so it has no `@GeneratedValue`; and its name collides with Spring's `@Service` annotation, so a file that uses both must write one in full, `@org.springframework.stereotype.Service`, as Chapter 13 does. Keeping the name is worth it: the same word means the same thing in all three versions of the shop. Book: Sections 11.4.1 and 11.4.3.

An Entity: `Customer`

Listing 11.2 The start of `Customer.java`: a customer, with a unique phone number.

```
@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;

    protected Customer() {                // for JPA only
    }
```

Listing 11.2: `@Entity` maps the class, `@Id` with `IDENTITY` lets the database number the rows, `@Column` states lengths and rules. Hibernate needs a no-argument constructor to build objects from rows; making it `protected` keeps our own code from using it. `Order` adds `@ManyToOne(fetch = LAZY)` to customer and service and `@OneToMany` to payments. Book: Sections 11.4.2 and 11.4.4.

Statuses Stored as the Other Frameworks Store Them

`EnumType.ORDINAL` stores 0, 1, 2 and breaks when a constant is inserted; `EnumType.STRING` stores `QUEUED`, which Laravel's database does not expect.

`OrderStatus` gives each constant its agreed word; `OrderStatusConverter`, an `AttributeConverter`, stores `queued`, `finished`, `departed` in every status column.

Money is `long` for whole rupiah; quantities are `BigDecimal`, never `double`.

The three applications can now read each other's data: the same words, the same integers. Book: Section 11.4.5.

The Rules, in Plain Java

```
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
  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.
```

Part of the real output of `RulesCheck`, compiled with `javac -d out -sourcepath src/main/java check/RulesCheck.java` and run with `java -cp out RulesCheck`. `LaundryRules` uses Chapter 8's arithmetic step for step: `movePointRight(2)` turns 4.5 into 450 hundredths with no rounding, `(h + 99) / 100 * 100` rounds items up, adding 50 before the division rounds half up. Compared byte for byte with `php tools/core_check.php`, the output is identical: three frameworks, two languages, one set of prices. Book: Section 11.4.6.

Repositories: Queries From Method Names

Listing 11.3 The interface in `OrderRepository.java`: pages of orders, filtered by status, and one order with everything it needs.

```
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);
}
```

Listing 11.3: no class to write. `JpaRepository` gives save, find, delete and count; each method name becomes a query: filter by status, order by date then id (a complete order, as Chapter 8 taught), one page. `@EntityGraph` loads the listed relations in the same query, the JPA version of `Include` and `with`. Book: Section 11.5.

Records In, Records Out

Out: `OrderView` is a Java **record** built from the entity, the counterpart of Laravel's `OrderResource`; `PageView` answers a page as `data` and `meta`, counting pages from 1 as Laravel does.

In: `CreateOrderRequest` is a record with Jakarta Validation annotations (`@NotNull`, `@NotBlank`, `@DecimalMin`, `@Digits`, `@Size`) and one rule about two fields, an `@AssertTrue` method: pickup needs an address.

Never serialize entities: Jackson may follow lazy relations outside a transaction and fail, or loop from order to payment to order.

Records are short, immutable and exactly as wide as the JSON. The entity is for the database, the record is for the wire. This is the third time we meet the idea: Chapter 6's form classes, Chapter 10's API resources, and now Java records. Each framework spells it differently; none of them lets the database decide what a stranger sees. Book: Sections 11.6.1 and 11.6.2.

The Order Controller

Listing 11.4 The `store` method of `OrderController.java`: create an order inside a transaction.

```
@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 11.4: `@RestController` with `@RequestMapping("/api/orders")`; repositories arrive through the constructor. `@Valid @RequestBody` binds and checks the JSON; a broken rule answers 400 with a problem-details body before the method runs, while a missing customer or service answers 422, and Chapter 12 settles which code is right. The `Order` constructor prices itself with `LaundryRules`, so no price comes from the request. The controller also lists (`?status=finished&page=2`), shows, and moves an order one step with `PATCH /api/orders/{id}/status`. Book: Section 11.6.3.

Loading the Price List, Running and Testing

`DemoData`, a `CommandLineRunner` bean, reads the real price list from `Service.csv` and adds the three customers of Chapters 5 and 7 with their orders.

On your computer: `mvn spring-boot:run` in the chapter's `spring` folder (or **Run** above `main` in VS Code); the first run downloads the libraries. Then `curl` the price list and create an order: 2.5 kg of wash, dry and fold with pickup answers a `total` of 34000.

`LaundryRulesTest` tests the price rule and the workflow with JUnit, without starting Spring: fast, and runnable on every change.

The same test cases as Chapter 8's check programs, now as JUnit tests that fail loudly. Book: Sections 11.7–11.9.

Making It Better and Using It for Real

Move from H2 in memory to a file or a server database; replace `ddl-auto` with **Flyway** migrations before any real data exists.

For the accounting firm: the same field names in both APIs mean one client program can talk to either version of the shop.

`ddl-auto=update` *adds columns but never removes or renames them and cannot move data: fine for a demonstration, dangerous for the shop's books. Book: Sections 11.10 and 11.11.*

Common Mistakes

Classes outside the application's package; a table named `order`; no no-argument constructor; eager relations everywhere.

Serializing entities directly; enums by position or by Java name; money in `double`.

Pages from 0 in the URL; forgetting `@Transactional` for lazy relations: `LazyInitializationException`**; `ddl-auto=update` for a real database.**

Several of these are Chapter 5 and Chapter 8 mistakes in Java clothes: eager relations are the N+1 problem, serialized entities are models returned without a resource, `double` money is Chapter 8's float. Book: Section 11.12.

Summary: Five Things to Remember

1. Spring Boot 4.1 on Java 25: starters, auto-configuration and embedded Tomcat 11; projects start at Spring Initializr (Figure 11.1).
2. The application context scans packages, creates beans and injects them through constructors.
3. JPA entities map the four tables (Figure 11.2); lazy relations, the table `orders`, converters for agreed words.
4. Repositories are interfaces; queries come from method names; `@EntityGraph` loads relations with the main query.
5. `@RestController` methods read JSON into validated records and answer records as snake_case JSON; `LaundryRules` prints exactly what Chapter 8's PHP check prints.

Three frameworks now answer the same questions about the same shop. The next chapter puts two of them side by side, request by request, and decides where they should agree. Book: Section 11.13.

Next Time

Chapter 12 — Evaluation and Discussion III. A Laravel and Spring Boot REST project, compared field by field.

Before then: compile and run `RulesCheck` with `javac` and compare it with `core_check.php`; on your computer start the API and create an order with `curl`; work the exercise on a customer's orders.

Book: Sections 11.14 and 11.15.

Lecture 12 — Evaluation and Discussion III: One REST API in Laravel and Spring Boot

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 12

Lecturer: Irfan Subakti — yifana@gmail.com

The rules of the Laravel API of Chapter 10 and the Spring API of Chapter 11 are the same down to the rupiah; the APIs themselves are not. Today we write one contract for both, solve seven problems that bring each API into line with it, and check any server against it with a small program. Book: Chapter 12, Sections 12.1–12.14.

What This Meeting Covers

1. Where the two APIs differ — envelopes, times, error shapes, status codes.

2. The contract — endpoints, the envelope, names, formats, refusals.

3. Seven problems — price list, times, one error shape, every error at once, moving orders, payments, the daily report.

4. Checking the contract — `check_api.php` and a real run.

5. Lessons — what a contract fixes, and the mistakes it prevents.

Neither Laravel nor Spring Boot could be installed where the book was written, but PHP could run. So the chapter includes a ***reference server****, `contract/reference/server.php`, about 240 lines of plain PHP with SQLite using the rule files of Chapter 8, and the checker's output today is a real run against it. On your computer the same checker runs against the Laravel and Spring applications. Book: Section 12.2.*

Where We Left Off: Same Rules, Different APIs

Envelopes: Chapter 10's resources wrap every answer in `data`; Chapter 11's controllers return bare objects and lists.

Times: Laravel's default is UTC with microseconds; Spring sends Java's local date-time.

Errors: Laravel answers a broken rule with 422 and a list of errors per field; Spring answers 400 with a problem-details body and Java field names.

Raka's phone app must work with either server, and the accounting firm's Java system will read whichever one Bu Wati runs. Every difference is a line of special code in the app, and a place for a bug. The cure is to decide each point once, write it down, and make both servers obey. Book: Section 12.1.

One Contract, Three Implementations, One Checker

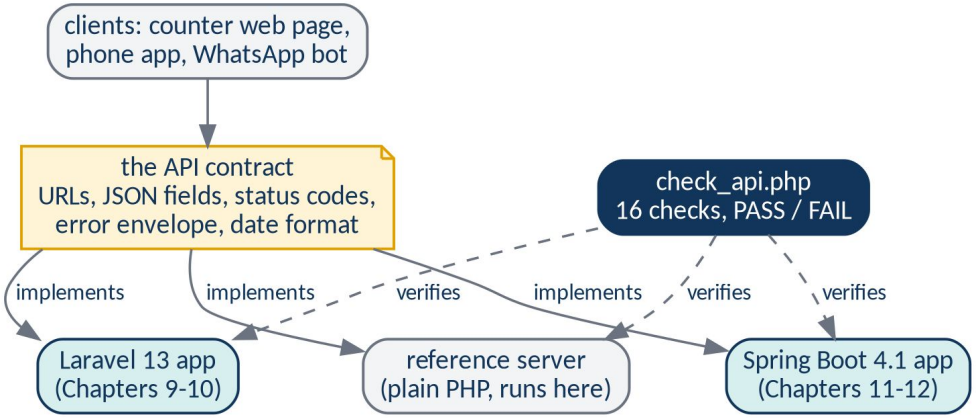


Figure 12.1 One contract, three implementations and one checker: the clients depend only on the contract, and `check_api.php` verifies every server against it.

*Figure 12.1: the clients (the drivers' app, the accounting firm) depend only on the **contract**, never on a framework. Each server is checked against the contract by the same program. The approach for every problem: state it, work an example by hand, compare what the two frameworks do today, change both to the contract, look for the tempting wrong solution. Book: Section 12.2.*

The Contract: Endpoints

Table 12.1 The endpoints of the Profitina Laundry API contract.

Method and URL	Success
`GET /api/services`	200, `data`: active services, by code
`POST /api/customers`	201, `data`: the customer
`GET /api/orders?status=&page=`	200, `data`: orders, `meta`: page numbers
`POST /api/orders`	201, `data`: the order
`GET /api/orders/{id}`	200, `data`: the order with `payments`
`PATCH /api/orders/{id}/status`	200, `data`: the order
`POST /api/orders/{id}/payments`	201, `data`: the order with `payments`
`GET /api/reports/daily?date=YYYY-MM-DD`	200, `data`: the day's figures

Table 12.1: besides the endpoints, the contract fixes the envelope `data`, snake_case names, lowercase status words (`queued`, `finished`, `departed`), whole rupiah, and one date-time format in shop time. Book: Section 12.3.

The Contract: Refusals and the Balance

Table 12.2 The refusals of the contract, and the body each one carries.

Status	When	Body
401	no valid token	`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`

$$balance = total - \sum_{p \in P} amount_p \tag{12.1}$$

Table 12.2: 400 is for a body that cannot be read at all, 422 for content that breaks a rule, 409 for a resource whose state forbids the request. Equation (12.1) is the balance of Chapter 5, now written into the contract so that both servers compute it the same way. Book: Section 12.3.

Problem 12.1: The Price List for Programs

Laravel had no API for the price list; Spring had one without the `data` envelope.

Laravel: `Api/ServiceController` answers the active services as a `ServiceResource` collection. **Spring:** `DataView`, a record with one field `data`, wraps the same list.

Tempting wrong: return the models, `Service::all()` or `services.findAll()`. It sends `description` and `is_active` (and in Laravel the timestamps), omits the computed `name` and `unit`, and ties every phone to today's columns.

The day the owner splits the description into two columns, every client of the model-shaped answer breaks. A resource or a record keeps the promise while the database changes. Book: Section 12.4.

Problem 12.2: One Way to Write a Time

The contract: `YYYY-MM-DDTHH:MM:SS`, in shop time, `Asia/Jakarta`.

`2026-09-25T01:15:00Z` and `2026-09-25T08:15:00` are the same moment and different strings; a client comparing them as text gets it wrong.

Laravel: an explicit format in the resources. **Spring:** `@JsonFormat` with a fixed pattern on the time fields of the records.

Tempting wrong: switching Laravel's time zone to UTC "to match Spring". The daily report would then cut the day at seven in the morning Surabaya time. Time zones are chosen for people, formats for programs; do not change the first to fix the second. Book: Section 12.5.

Problem 12.3: One Shape for Every Error

The contract's 422: a `message` and `errors`, an object from each field name, as the request spelled it, to a list of messages. Laravel already does this.

Spring: `ApiExceptionHandler`, a `@RestControllerAdvice`, turns validation errors, broken rules, status exceptions and unreadable bodies into the contract's shape.

It converts field names to snake_case: `serviceId` becomes `service_id`, so the client can put the message next to the right field.

Tempting wrong: rely on Jackson's snake_case setting of Chapter 11 and hope the error names follow. They do not: the names in a `FieldError` come from the validator, not from Jackson, and stay `serviceId`. The conversion must happen where the error body is built. Book: Section 12.6.

The Handler for Validation Errors

Listing 12.1 One method of `ApiExceptionHandler.java` (Spring): validation errors in the contract's shape.

```
@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);
}
```

Listing 12.1: the class carries `@RestControllerAdvice`, so its methods handle exceptions thrown by every controller. This one collects each `FieldError` under its snake_case name, several messages per field if needed, and `unprocessable` builds the body `{"message": "The given data was invalid.", "errors": {...}}` with status 422, word for word what Laravel sends. Sibling methods answer a broken rule with the same shape, a `ResponseStatusException` with its code and a `message`, and an unreadable body with 400. One class, and every controller of the application speaks the contract's error language. Book: Section 12.6.4.

Problem 12.4: Every Error at Once

Spring's `@Valid` checked one-field rules first; the database checks (does the customer exist? is the service offered?) came in a second round.

A client that fixes the three errors it was shown and then receives a fourth feels cheated.

Spring: `CreateOrderRequest` keeps only one-field rules; the controller validates by hand, adds the two-field rule and the database checks to the **same map**, and answers once.

Tempting wrong: rename the `@AssertTrue` method so that its property becomes `deliveryAddress`. It collides with the real component and still leaves the database checks in a second round. Rules that need to see everything belong in code that can see everything. Book: Section 12.7.

Problem 12.5: Moving an Order, and Closed Orders

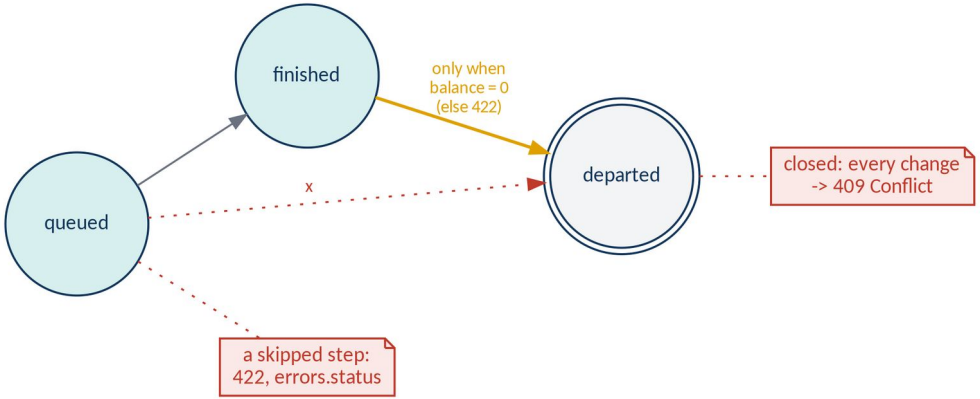


Figure 12.2 The order's workflow as the contract states it: one step at a time, departure only when the balance is 0, and a departed order closed to every change.

*Figure 12.2: a refused step, such as `queued` to `departed` or departing with money owed, is **422** with `errors.status`: the request broke a rule. A change to a departed order is **409**: the resource's state forbids any change. The client names the wanted status, which makes the request safe to repeat. Tempting wrong: a bare "advance", as the web button does; two drivers pressing "picked up" together could then move an order two steps. Book: Section 12.8.*

Problem 12.6: Payments

`POST /api/orders/{id}/payments` with `amount` and `method`; 201 with the order and its payments; more than the balance is 422 with `errors.amount`.

Spring: `CreatePaymentRequest` and `PaymentView`, with its time in the contract's format; each payment is saved at once with `payments.save`.

Tempting wrong: rely on the cascade at commit. The database is right, but the answer is written *before* the commit, and the client receives `"id": null`.

A person clicking through the app never sees that bug; a test that checks the fields the client uses catches it at once. Book: Section 12.9.

Problem 12.7: The Daily Report

$$\sum_{s \in S} n_s = N \tag{12.2}$$

`GET /api/reports/daily?date=…`: counts per status, the day's totals and payments, over the half-open day $[d, d+1)$ in shop time.

Every status present, even with zero orders, so that the counts per status add up to the number of orders, Equation (12.2), and no client has to guess a missing key.

Laravel: `Api/ReportController`. **Spring:** two new repository queries and a `LinkedHashMap` that lists every status in order.

Tempting wrong: `WHERE date(received_at) = ?`. *It reads naturally but applies a function to the column, so the database cannot use the index, and some databases interpret the stored time in the server's zone. The interval* `received_at >= :from AND received_at < :to` *is fast and exact. Book: Section 12.10.*

Checking the Contract: a Real Run

```
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  finished -> departed before payment: 422, errors.status
PASS  paying more than the balance -> 422, errors.amount
PASS  a departed order is closed: 409 with a message
PASS  GET /api/reports/daily -> 200, the counts add up

16 passed, 0 failed
```

Eleven of the sixteen lines of the real run of `contract/check_api.php` against the reference server. The checker takes a base URL, so the same program checks the Laravel server and the Spring server on your computer. It is the "check program" idea of Chapter 8, one level up: not one rule, but a whole API. Book: Sections 12.11.1 and 12.11.2.

Tests Inside Each Framework

Laravel: `ContractTest` checks the contract's new points with a fixed clock, so that times and the daily report are predictable.

Spring: `OrderApiTest` with MockMvc: the price rule, every error at once, the whole life of an order and the report.

Test the fields the client uses, not only the status code: an answer with the right status and `"id": null` still breaks the app.

The checker proves that a running server keeps the contract; the tests inside each framework prove it on every change, before anything runs on a server. Book: Section 12.11.3.

Using It for Real

Write the contract down before the second client arrives: the accounting firm, a new phone app, a partner outlet.

Publish it where clients can read it; version it when it must change; keep the checker in the repository and run it before every release.

A contract is also a promise to the people who build on your system: it lets them work without asking you what the server does today. Book: Section 12.12.

Common Mistakes

Letting frameworks choose the contract; comparing times as text across zones; mixing up 400, 409 and 422.

Error keys in the server's own names; errors in two rounds; missing keys for zero; checking only status codes in tests.

Every one of these mistakes is invisible inside one framework and obvious the moment a second server must answer the same client. That is the value of building the same API twice. Book: Section 12.13.

Summary: Five Things to Remember

1. A contract fixes endpoints (Table 12.1), envelope, names, words, money, time format and refusals (Table 12.2); clients depend only on it (Figure 12.1).

2. Resources and records, not models; one time format in shop time; time zones for people, formats for programs.

3. One error shape with the request's field names; every error at once.

4. 422 for a refused step, 409 for a closed order (Figure 12.2); save a payment before answering; every status in the report.

5. `check_api.php` checked the reference server: 16 passed, 0 failed; each framework has its own tests for the same points.

The phone app no longer needs to know which server it talks to. That is the whole point of the chapter, and the test of any API that more than one program will use. Book: Section 12.14.

Next Time

Chapter 13 — Spring Boot II. Layered architecture, validation and full CRUD pages with Thymeleaf.

Before then: run the reference server and the checker; on your computer, run the checker against the Laravel and the Spring servers; work the chapter's exercises.

Book: Sections 12.15 and 12.16.

Lecture 13 — Spring Boot II: Layered Architecture, Validation and Full CRUD with Thymeleaf

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 13

Lecturer: Irfan Subakti — yifana@gmail.com

Today the Spring Boot application gets pages for the counter, and to get them without copying rules we split it into layers: controllers that speak HTTP, services that do the shop's work inside transactions, repositories that speak SQL. The JSON API and the new Thymeleaf pages call the same services. Book: Chapter 13, Sections 13.1–13.15.

What This Meeting Covers

1. One class that does everything — why Chapter 12's controllers cannot simply gain pages.

2. Three layers, one direction — what each layer may know; transactions.

3. The service layer — three exceptions, `CustomerService`, `OrderService`, the report.

4. Forms and pages — form records with Bean Validation, Thymeleaf templates, Post/Redirect/Get.

5. Errors, SQLite, tests — one error advice per layer, a file database, services tested with Mockito.

What was run: the machine where the book was written could not download Spring's libraries. Every Java file was compiled against small stand-ins for the Spring, Jakarta and JUnit types it uses, with no errors, and the rules were compiled and run for real with `javac`; their output is the same as in Chapters 8 and 11. You run the application on your own computer. Book: Section 13.9.

One Class That Does Everything

Chapter 12's `OrderController.store` validates the request, looks up customer and service, checks the pickup address, collects errors, builds the order, saves it and wraps it in JSON: all in one method.

Now the counter wants **pages**: a queue, a form for a new bag, an order page with a button for the next step and a payment form.

Copy the logic into a page controller and there are two places for every rule. The day one changes and the other does not, the shop has two prices again.

It is Bu Wati's first wish from Chapter 1, "one change in one place", now inside one framework. Book: Sections 13.1 and 13.2.1.

Three Layers, One Direction

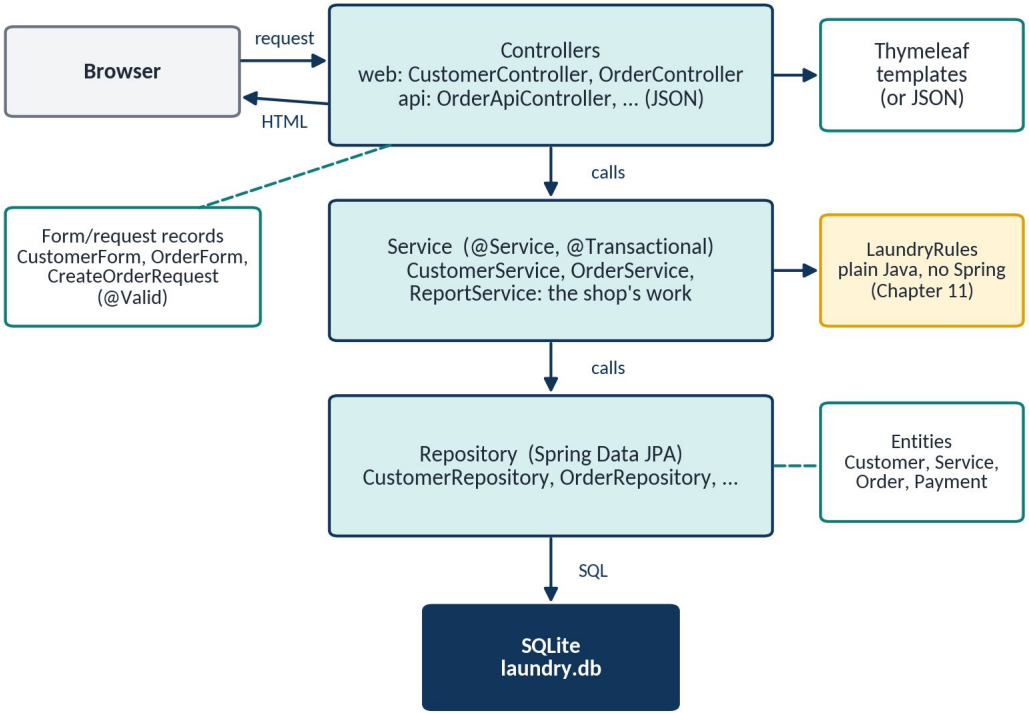


Figure 13.1 The layers of the Spring Boot laundry application: page and API controllers speak HTTP, services do the shop's work inside transactions, repositories speak SQL; the rules themselves are plain Java from Chapter 11.

*Figure 13.1: calls go **downwards only**. A controller calls a service; a service calls repositories and `LaundryRules`; nothing below knows anything above. That one direction is what lets two kinds of controller share one set of services. Book: Section 13.2.2.*

What Each Layer May Know

Table 13.1 What each layer may know, and what it must not know.

Layer	Knows about	Must not know about	Example
Controller	URLs, parameters, JSON, forms, redirects, templates	SQL, transactions, prices	`OrderController.advance`
Service	the shop's operations and rules, transactions	HTML, URLs, status codes	`OrderService.advance`
Repository	tables, queries	rules, HTTP	`OrderRepository.billedBetween`
Rules	numbers, statuses	everything else	`LaundryRules.advance`

Table 13.1: a quick test for any line of code: which column would it be in? A price in a controller, an HTTP status in a service, a rule in a repository: each is in the wrong place. ***Transactions*** *belong to services:* `@Transactional` *groups the operations of one service method so that they succeed or fail together, isolated from other transactions: two cashiers taking the last Rp 34.000 of one order at the same moment must not both succeed. Book: Sections 13.2.3 and 13.2.4.*

Where the Code Goes

`domain` — entities and `LaundryRules`; `repository` — Spring Data interfaces; `service` — the services and their exceptions.

`api` — Chapter 12's JSON controllers, moved and made thin; `web` — the new page controllers and form records.

`web.OrderController` and `api.OrderController` would both become the bean `orderController` and the application would not start: the API one is `OrderApiController`.

Packages make the layers visible in the file tree, and they let each error advice be limited to its own package, as we see shortly. Book: Section 13.3.

The Service Layer: Three Exceptions

`RuleException` — input that breaks a rule, with messages per field: 422 in the API, messages next to fields on a page.

`NotFoundException` — a record that does not exist: 404.

`ConflictException` — a request that the record's present state does not allow, such as paying a departed order: 409.

The service says what went wrong in the shop's words; each kind of controller decides how to tell its audience. That is how the service stays free of HTTP. Book: Section 13.4.1.

`OrderService`: Every Rule in One Place

Listing 13.1 Two methods of `OrderService.java`: taking money, and refusing a closed 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.");
    }
}
```

Listing 13.1: the class carries `@Transactional`, so each public method runs in one transaction. `create` first calls `check`, which returns every database rule as a map so that API and pages can report all problems at once, then checks again and saves. `advance` asks

`LaundryRules` for the next step. `pay` refuses more than the balance and saves the payment at once, the lesson of Problem 12.6. `now()` reads a `Clock` bean set to Western Indonesian Time. Book: Section 13.4.3.

The Money Rules, Unchanged

$$S = \left\lfloor \frac{p \cdot b + 50}{100} \right\rfloor, \qquad T = S + \begin{cases} 10000 & \text{pickup and } S < 100000 \\ 0 & \text{otherwise} \end{cases} \tag{13.1}$$

$$B = T - P \tag{13.2}$$

Equation (13.1) is the total of Chapter 8: subtotal S from price p and billed hundredths b, plus the pickup fee below Rp 100.000. Equation (13.2) is the balance: total T minus payments P. The layers moved code around; they did not change one rupiah. `ReportService` *computes the figures of one day for both the API and a page. Book: Sections 13.4.3 and 13.4.4.*

Thin API Controllers

`OrderApiController`: read JSON into a request record, call `OrderService`, write the answer as a record. Three lines per method.

`ApiExceptionHandler` turns the three exceptions and validation errors into the contract's shapes: 422 with `errors`, 404 and 409 with a `message`.

It is a `@RestControllerAdvice` limited with `basePackages` to `api`; without that limit it would answer page errors with JSON.

The API's contract from Chapter 12 does not change; only where its code lives changes, and the same checker can confirm it on your computer. Book: Section 13.5.

Validating Form Input

Jakarta Bean Validation constraints on **form records**: `@NotBlank`, `@Size`, `@Pattern`, `@NotNull`, `@DecimalMin`, `@Digits`, `@Min`.

`CustomerForm` has the same limits as the API's request; `OrderForm` knows how to become a `NewOrder` for the service; `PaymentForm` has amount and method.

A form record contains only the fields the form may change, so a hand-made request cannot set `id` or `createdAt`: no mass assignment.

`Forms.reject` turns the messages of a `RuleException` into errors on the named fields of the form, so "this phone number is already registered" appears right under the phone field. Book: Section 13.6.

The Customer Pages

Listing 13.2 The `create` method of `web/CustomerController.java`: a new customer from a page.

```
@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";
    }
}
```

*Listing 13.2: `@Validated` checks the form record; `BindingResult` must come immediately after it, or Spring shows an error page instead of the form. Constraint errors show the form again; the service's rule errors are added to the same `BindingResult`; success redirects with a **flash attribute** that survives exactly one redirect. The page controllers never contain a rule. Book: Section 13.7.1.*

Post/Redirect/Get

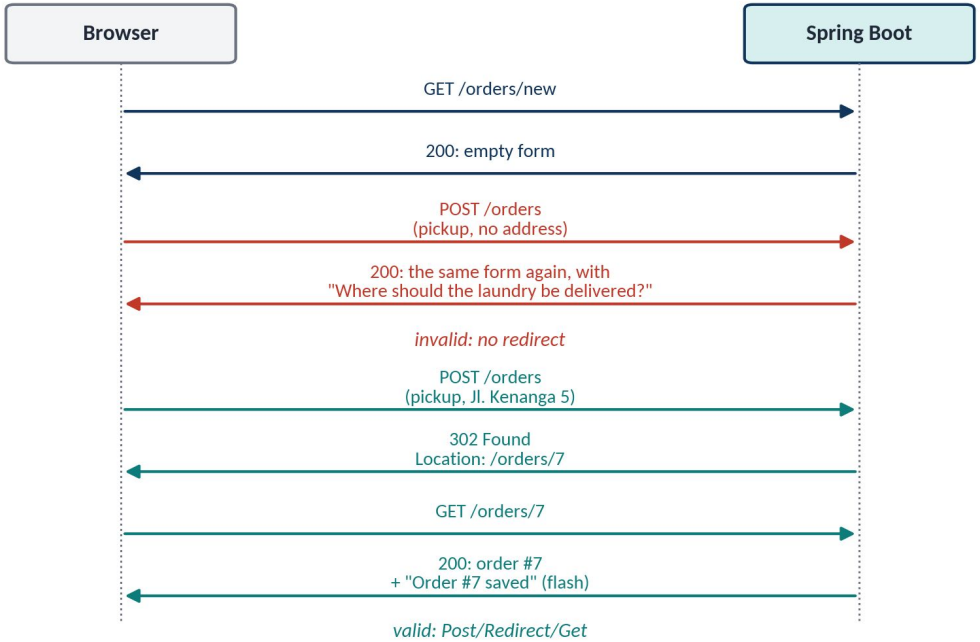


Figure 13.2 Post/Redirect/Get: an invalid form is answered with the same page and its messages; a valid one with a redirect, so that refreshing the page never takes the same order twice.

Table 13.2 The URLs of the customer pages.

URL	Does	Answers
`GET /customers?q=...`	list and search	the list page
`GET /customers/new`	show an empty form	the form page
`POST /customers`	create	redirect, or the form with errors
`GET /customers/{id}/edit`	show a filled form	the form page
`POST /customers/{id}`	update	redirect, or the form with errors
`POST /customers/{id}/delete`	delete	redirect with a message

Figure 13.2 and Table 13.2: HTML forms send only `GET` and `POST`, so update and delete are `POST`s to their own URLs. After a successful `POST` the browser is redirected; a refresh repeats the harmless `GET`, never the order. Book: Section 13.7.2.

Thymeleaf Templates

`fragments.html` holds the page head, the navigation and the flash messages; pages include them with `th:replace="~{fragments :: nav}"`.

`th:object="${form}"` binds a form; `th:field="*{phone}"` writes name, id and value; `th:errors="*{phone}"` shows its messages; `th:errorclass="invalid"` marks the field.

`th:text` escapes by default, as `{{ }}` in Blade and `@` in Razor do.

The order pages follow the same pattern: the queue filtered by status with pages, a form for taking in a bag, and an order page with its total, payments, balance and a button for the next step. Hiding the button when it does not apply is tidiness; the service checks the rule anyway. Book: Sections 13.7.3 and 13.7.4.

One Form for Adding and Editing

Listing 13.3 Part of `customers/form.html`: one form for adding and editing, with messages next to each field.

```
<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>
```

Listing 13.3: the same template serves both pages. When the controller puts a `customerId` in the model the form posts to `/customers/{id}` and the title says "Edit customer"; otherwise it posts to `/customers`. `@{...}` builds URLs relative to the application, so the site still works if it is later deployed under a sub-path. Messages that belong to no single field, such as a rule about two fields, appear at the top through `#fields.globalErrors()`. Compare with Razor's `asp-for` and Blade's `old()` with `@error`: three spellings, one idea. Book: Section 13.7.3.

One Place for Errors, SQLite, and Tests

`WebErrorAdvice`, limited to `web`, turns every `NotFoundException` on a page into `error/404.html` with status 404.

The data now live in **SQLite**, a file next to the project kept between starts; `DemoData` fills it on the first start only. The tests use a fresh H2 database in memory.

`OrderServiceTest` tests the service layer with **Mockito**-mocked repositories and a fixed clock: no database, no web, fast.

A fixed `Clock` makes "received today" and the daily report predictable in tests, the Java counterpart of Laravel's fixed clock in Chapter 12. Book: Sections 13.8–13.10.

Making It Better and Using It for Real

Keep adding features the same way: a service method first, then a page controller and an API method that call it.

For your own business: the counter pages of any shop, clinic or workshop follow this shape; only the entities and the rules change.

When a colleague asks where a rule lives, the answer is always “in a service”, and that answer is worth more than any single feature. Book: Sections 13.11–13.13.

Common Mistakes

Rules in controllers; **transactions on controllers**; **two beans with the same name**; **an error advice for everything**.

Binding forms to entities; **`BindingResult` in the wrong place**; **a page as the answer to a successful `POST`**.

`LazyInitializationException` in a template; **`@Transactional` on a private or self-called method** (Spring's proxy never sees it); **trusting hidden buttons**.

The proxy mistake surprises even experienced developers: `@Transactional` works because Spring wraps the bean in a proxy, and a call from one method of a class to another never passes through that proxy. Keep transactional methods public and call them from another bean, usually a controller. Book: Section 13.14.

Summary: Five Things to Remember

1. Controllers speak HTTP, services do the work in transactions, repositories speak SQL; calls go down only (Figure 13.1, Table 13.1).

2. API and page controllers call the same services; services throw `RuleException`, `NotFoundException`, `ConflictException`.

3. The money rules are unchanged (Equation (13.1), Equation (13.2)).

4. Form records with Bean Validation, `BindingResult`, Thymeleaf's `th:object` and `th:field`, and Post/Redirect/Get (Figure 13.2).

5. One error advice per layer; SQLite for data, H2 for tests; services tested with Mockito and a fixed clock.

Compare with the other two frameworks: ASP.NET Core put the price rules in `PriceCalculator` from Chapter 3, and Laravel grew an action class in Chapter 10. Spring's service layer is the same idea made into the backbone of the application. Book: Section 13.15.

Next Time

Chapter 14 — Spring Boot III. Spring Security, REST API design and automated testing.

Before then: run `RulesCheck` again; on your computer run the application, add a customer twice with one phone number, and follow an order from the form to departure.

So far anyone who can reach the Spring application can use every page and every endpoint, exactly the open door of Chapters 6 and 10. Chapter 14 closes it. Book: Sections 13.16 and 13.17.

Lecture 14 — Spring Boot III: Spring Security, REST API Design and Automated Testing

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 14

Lecturer: Irfan Subakti — yifana@gmail.com

Today the Spring Boot application gets its doors: staff accounts with hashed passwords, Sanctum-style tokens for the API, two security filter chains (one for the API, one for the pages), a second line of defense in the service, a look at the API's REST design, and automated tests on four levels. Book: Chapter 14, Sections 14.1–14.12.

What This Meeting Covers

1. The missing doors — what Laravel's API already refused, and Bu Wati's two wishes.

2. Accounts and passwords — never store a password; staff and roles; Spring Security's view of a user.

3. Tokens for the API — the token table, the token service, a filter, signing in and out.

4. Two filter chains — stateless API, session and CSRF for pages, followed request by request.

5. REST design and tests — methods, status codes, envelopes; the test pyramid in four levels.

As in Chapters 11 and 13, Spring's libraries could not be downloaded where the book was written, so the code was written against the Spring Security and Spring Boot documentation. On your computer `mvn test` *runs the chapter's tests. Book: Chapter 14.*

Where We Left Off: The Missing Doors

Chapter 12 made the Spring API speak Laravel's JSON; Chapter 13 put services beneath it and pages on top. One part was left out on purpose: the doors.

Laravel's API answers a request without a valid token with `401 {"message": "Unauthenticated."}`; the drivers' app signs in with `POST /api/tokens`.

Bu Wati's wishes: cashiers log in with their own accounts, so the shop can later see who took which order; deleting a customer, which removes a person's data, is for her alone.

Everything else a cashier does today, a cashier may keep doing. Good security starts from what people need to do, not from locking everything. Book: Section 14.1.

Never Store a Password

$$N = 2^c \tag{14.1}$$

Store a **salted, deliberately slow hash**. bcrypt with cost c performs N rounds of its expensive key setup, Equation (14.1), so each step of c doubles the work for the server and for any attacker alike.

Spring's `BCryptPasswordEncoder` uses $c = 10$ by default; Laravel's default is 12; ASP.NET Core Identity uses PBKDF2 with HMAC-SHA512 and 100,000 iterations.

`PasswordEncoderFactories.createDelegatingPasswordEncoder()` writes the algorithm's name into the stored value, so the shop can move to a stronger one later.

A cost that makes one login take a few tens of milliseconds is invisible to a cashier and makes guessing expensive. A fast hash such as SHA-256 of a human password can be reversed by guessing; never use one for passwords. Book: Section 14.3.1.

Staff and Roles

`Role`: `OWNER` and `CASHIER`, two roles are all the shop needs. `Staff`: a person who may log in, with an e-mail, a password hash and a role.

`StaffDetailsService` turns an e-mail into Spring Security's view of a user: name, hash and the role in Spring's spelling, such as `ROLE_CASHIER`.

`DemoStaffSeeder` creates the owner and one cashier, with hashed passwords and addresses ending in `.example`, a domain reserved for documentation.

The demo passwords are printed in the book and the README: the first thing a real shop does is change them. Book: Sections 14.3.2 and 14.3.3.

Tokens for the API

$$H = 30 \times 8 = 240 \text{ bits} \tag{14.2}$$

Tokens in the Sanctum style look like `id|secret`. The secret is 30 random bytes, Equation (14.2) bits: far beyond guessing.

`TokenService` issues, finds and revokes tokens; the table `ApiToken` keeps only the **SHA-256** of the secret, compared in constant time.

`TokenApiController`: `POST /api/tokens` gives a token (shown once), `DELETE /api/tokens/current` takes it back.

Why is a fast hash fine here and not for passwords? Because the secret is random with 240 bits, not a word a person chose: there is nothing to guess. The `id` part finds the row directly, and the hash of the secret is compared. The same design as Laravel Sanctum, so the phone app signs in the same way to both. Book: Sections 14.4.1, 14.4.2 and 14.4.4.

The Filter

`BearerTokenFilter` reads `Authorization: Bearer id|secret`, asks `TokenService` for the staff member, and puts an authenticated identity into the security context.

No header or a wrong token: it simply passes the request on without an identity, and the chain's rules answer 401.

It is created with `new` inside the API chain, not registered as a `@Component`; a component filter would run for every request, outside its chain.

A filter is Spring Security's middleware: the onion of Chapter 3 and Chapter 9 again, now with identity as its business. Book: Section 14.4.3.

Two Filter Chains

Listing 14.1 The start of the API chain in `SecurityConfig.java`: one chain for the API, one for the pages.

```
@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
```

Listing 14.1: the API chain has `@Order(1)` and matches `/api/**`; sign-in is open, deleting a customer needs `OWNER`, everything else needs any identity. No session, no CSRF. Its refusals are JSON: 401 "Unauthenticated." and 403 "This action is unauthorized.", Laravel's words. The web chain, `@Order(2)`, uses a login page at `/login` with the field `email`, a session and CSRF protection. `hasRole("OWNER")` adds the `ROLE_` prefix itself. Book: Section 14.5.1.

Following Requests by Hand

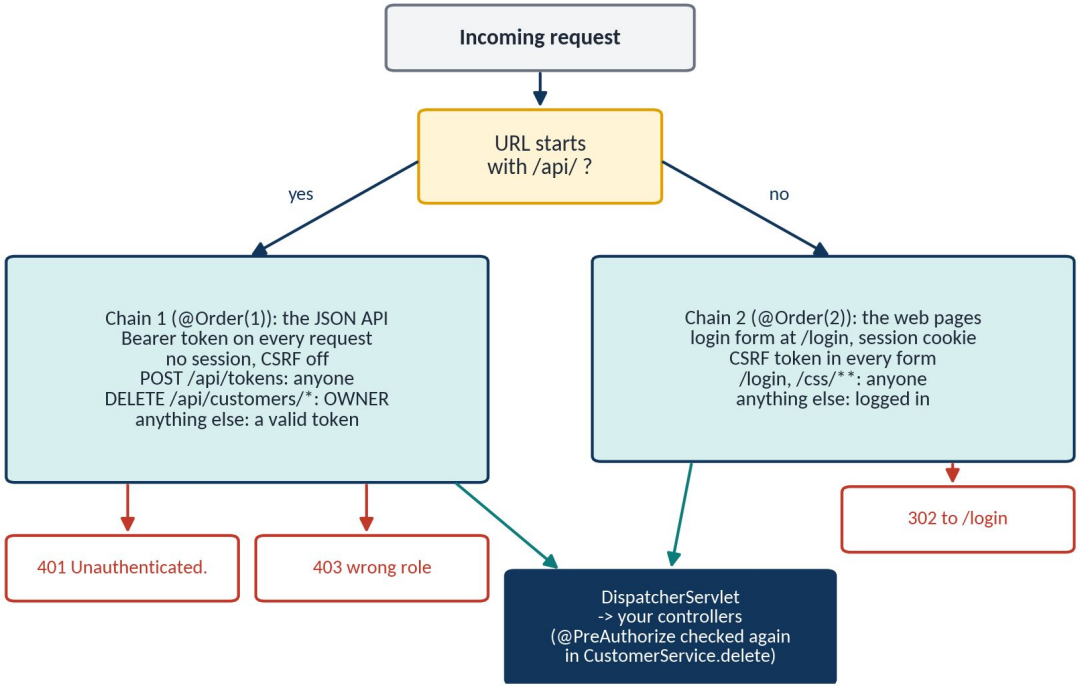


Figure 14.1 The two filter chains: requests under `/api/` go to the stateless chain with bearer tokens and JSON refusals; all others go to the chain with a login form, a session cookie and CSRF protection.

Table 14.1 Six requests followed through the two chains.

Request	Chain	Identity	Answer
`GET /api/orders`, no header	API	none	401 JSON
`GET /api/orders`, cashier's token	API	`ROLE_CASHIER`	200
`DELETE /api/customers/2`, cashier's token	API	`ROLE_CASHIER`	403 JSON
`DELETE /api/customers/2`, owner's token	API	`ROLE_OWNER`	204, or 409 with orders
`GET /orders`, no cookie	web	none	302 to `/login`
`POST /customers`, no CSRF field	web	a cashier	403

Figure 14.1 and Table 14.1: ask the room to predict each answer before revealing the column. The last row surprises people: the CSRF filter runs before any rule, so a cashier who is logged in still gets 403 for a form without its token. Book: Section 14.5.2.

CSRF: Why the Pages Need It and the API Does Not

A browser sends the session cookie with *every* request to the site, even one triggered by another site's page. A hidden form elsewhere could post to `/customers` as the logged-in cashier.

The CSRF token, a secret field the page itself contains, proves the form came from our page. `th:action` adds it automatically; a plain `action=` does not, and the form fails with 403.

The API uses no cookies: a bearer token is never sent automatically, so only a chain that uses no cookies may do without CSRF.

The same reason Laravel's web routes need `@csrf` and its API routes do not, and ASP.NET Core's forms carry an anti-forgery token. The logout is a form with `POST`, never a link. Book: Sections 14.5.3 and 14.5.4.

A Second Line of Defense in the Service

`CustomerService.delete` carries `@PreAuthorize("hasRole('OWNER')")`: deleting is for the owner, whoever calls the method.

The URL rule protects the API route; the service rule protects every future caller: a page, a scheduled job, a new endpoint someone adds next year.

This is **defense in depth**: two independent checks, so one mistake does not open the door.

The new `DELETE /api/customers/{id}` answers 204 on success and 409 if the customer has orders, because orders are the shop's history. Book: Section 14.5.5.

Designing the REST API

Table 14.2 The status codes of the API and when each is used.

Code	Meaning in this API
200 OK	a read or a status change succeeded
201 Created	a token, customer, order or payment was created
204 No Content	a sign-out or deletion succeeded
401 Unauthorized	no valid token
403 Forbidden	a valid token, but the role may not do this
404 Not Found	the record does not exist
409 Conflict	the record's state does not allow the request
422 Unprocessable Content	the input breaks a rule

*Table 14.2: nouns in URLs (`/api/orders/7/payments`), methods with defined meaning: `GET` is **safe** (changes nothing), `PUT` and `DELETE` are **idempotent** (repeating them changes nothing more), `POST` is neither. A small set of status codes, envelopes for data and pages. What the API still lacks is a version. While the app and the server are updated together, that is acceptable; once some drivers do not update the app, every change must be backward compatible: fields may be added, never renamed or removed. Book: Section 14.6.*

Automated Tests on Four Levels

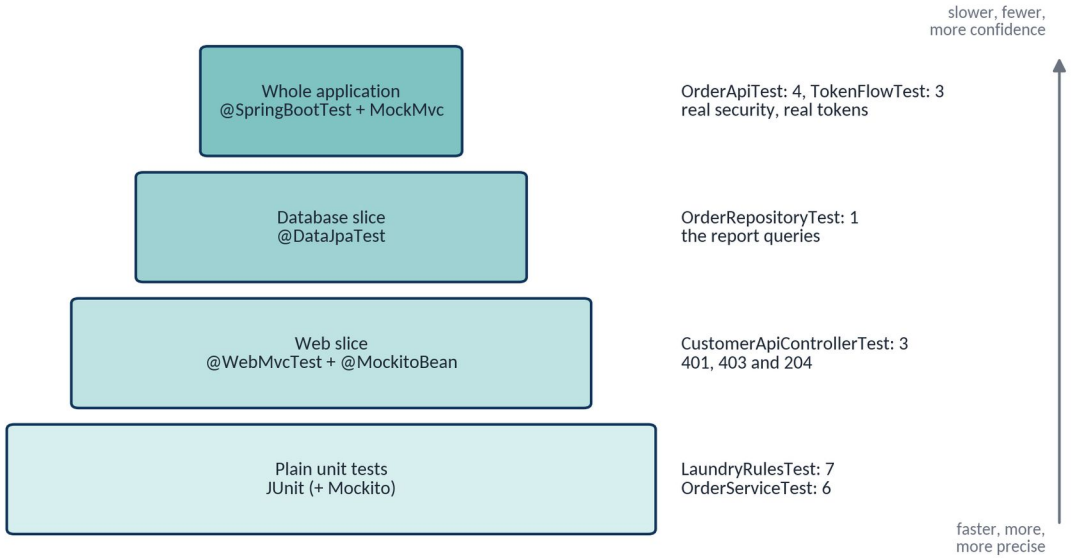


Figure 14.2 The four levels of tests in the Chapter 14 project, from plain unit tests at the bottom to tests of the whole application at the top.

*Figure 14.2: the **test pyramid**. Many fast tests at the bottom, few slow ones at the top. Each level starts only what it needs, catches its own kind of mistake, and misses others. Book: Section 14.7.1.*

What Each Level Catches

Table 14.3 The four levels of tests, with the 24 runs of this chapter.

Level	Starts	Runs	Catches	Misses
Unit	nothing but the class	13	wrong prices and rules	wiring, SQL, HTTP
Web slice	controllers, JSON, security	3	status codes, JSON, access rules	services, SQL
JPA slice	entities, repositories, H2	1	wrong queries and mappings	HTTP, rules
Whole application	everything, with H2	7	wiring, end-to-end flows	little, but slowly

*Table 14.3: the **web slice** (`@WebMvcTest`) starts controllers with the real security rules and a mocked service, so it checks that a cashier gets 403 on delete. The **JPA slice** (`@DataJpaTest`) checks the report queries against H2. `TokenFlowTest` signs in, uses the token, signs out and sees the token refused. `mvn test` runs them all and writes reports to `target/surefire-reports`. Book: Sections 14.7.2–14.7.4.*

A Web-Slice Test

Listing 14.2 One test of `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);
}
```

Listing 14.2: the class is `@WebMvcTest(CustomerApiController.class)` with `@Import(SecurityConfig.class)`, so the real chains run; `CustomerService` is a `@MockitoBean`. `@WithMockUser(roles = "CASHIER")` supplies an identity without a token. The test checks the status, the exact JSON message, and, with `verifyNoInteractions`, that the service was never even called: the refusal happened at the door. Sibling tests check 401 without a token and 204 for the owner. Book: Section 14.7.2.

Making It Better and Using It for Real

Idempotency keys for `POST`, so a phone that retries a payment does not pay twice; **token expiry** and a device list; **limit login attempts**.

Record who did what on orders and payments, now that accounts exist; **HTTPS** behind a reverse proxy, with `server.forward-headers-strategy=native`.

The first improvement matters most for drivers on weak mobile signals: a request that times out is often retried, and without a key the server cannot tell a retry from a second payment. Book: Sections 14.8–14.10.

Common Mistakes

Passwords with SHA-256 or in plain text; **a custom filter as a `@Component`**; **chains in the wrong order**; **CSRF disabled everywhere**.

Plain `action=` in a Thymeleaf form; **different answers for “no such e-mail” and “wrong password”**; **roles checked only in the template**; **`hasRole("ROLE_OWNER")`**.

Tests that break after security is added (give them `@WithMockUser` or a real token); **only whole-application tests**.

Two of these deserve a demonstration. Put the web chain first, without a matcher, and every API request is suddenly answered with a redirect to a login page that no phone can use. Write a Thymeleaf form with a plain `action=`, and a logged-in cashier gets 403 for no visible reason. Both take a minute to show and save an afternoon later. Book: Section 14.11.

Summary: Five Things to Remember

1. Missing identity is 401, a refusal 403. Passwords are salted, slow hashes (Equation (14.1)); tokens keep only the SHA-256 of a 240-bit secret (Equation (14.2)).

2. Two chains (Figure 14.1): a stateless API chain with bearer tokens and JSON refusals, a web chain with login, session and CSRF (Table 14.1).

3. Deleting a customer is checked at the URL and with `@PreAuthorize` in the service: defense in depth.

4. REST: nouns in URLs, safe and idempotent methods, a small set of status codes (Table 14.2).

5. Four levels of tests (Figure 14.2, Table 14.3), 24 runs, most of them at the bottom.

With this chapter the three versions of the shop stand level: each has rules in one place, a database with migrations or a path to them, pages, an API, logins, roles and tests. Book: Section 14.12.

Next Time

Chapter 15 — Comparing the Three Frameworks. Performance, productivity, ecosystem, and how to choose.

Before then: on your computer run `mvn test`, then break one rule on purpose (let a cashier delete) and see which level catches it; work the chapter's exercises.

Book: Sections 14.13 and 14.14.

Lecture 15 — Comparing the Three Frameworks: Performance, Productivity, Ecosystem, Choosing

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 15

Lecturer: Irfan Subakti — yifana@gmail.com

Today we step back from the laundry and compare the three frameworks we built it in. We line them up concept by concept, look at the evidence people usually quote (benchmarks, code size, surveys, support calendars, security defaults), measure one endpoint ourselves, and end with a short procedure for choosing. Book: Chapter 15, Sections 15.1–15.13.

What This Meeting Covers

1. Dimas's choice — a catering business, one developer, three opinions.

2. One application, three vocabularies — the same concepts under three names.

3. Performance — a public benchmark, our own measurement, and Little's law.

4. Productivity, ecosystem, calendars, security — each with its source and date.

5. Choosing — six steps, four situations.

The rule that runs through the chapter: say where each number comes from, when it was collected, and what it measures. The ASP.NET Core measurement was run for the book on September 25, 2026; the other numbers come from named public sources with their dates. Book: Chapter 15.

Where We Left Off: Dimas's Choice

Dimas finished his diploma in informatics and saw all three versions of the counter program running on the same computer.

A friend's family runs a small catering business with twelve staff and asks him to build their order system. He must pick one framework.

His classmates each have an opinion, each backed by a chart, a survey or a story.

The biggest surprise after building the same application three times is how similar the frameworks are. That is why the choice depends less on the frameworks than on the people and the place. Book: Section 15.1.

What to Compare, and What Evidence Each Question Needs

Table 15.1 What to compare, and what kind of evidence each question needs.

Question	Kind of evidence
Will it be fast enough?	benchmarks, and your own measurement of your own workload
How much work is a feature?	code size of the same feature, tools, learning curve
Can we hire and get help?	surveys, package registries, local job and hosting market
How long is a version supported?	official release and support calendars
Is it safe by default?	defaults for passwords, XSS, CSRF, SQL
Does it fit our team and servers?	what the team knows, where the app must run

Table 15.1: each kind of evidence has its 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. Book: Section 15.2.

One Application, Three Vocabularies

Table 15.2 The same concepts under three names (a selection).

Concept	ASP.NET Core 10	Laravel 13	Spring Boot 4.1
Routing	`[HttpGet("...")]` or `MapGet`	`routes/web.php`, `routes/api.php`	`@GetMapping`
Input validation	data annotations	Form Requests, `validate()`	Jakarta Bean Validation
ORM style	EF Core, unit of work	Eloquent, active record	JPA/Hibernate, repositories
Schema changes	EF Core migrations	Laravel migrations	Flyway or Liquibase
Templates	Razor	Blade	Thymeleaf
Login and roles	Identity, `[Authorize(Roles=...)]`	guards, Gates, `can`	filter chains, `@PreAuthorize`
Tests	xUnit, `WebApplicationFactory`	PHPUnit or Pest	JUnit 6, `@WebMvcTest`

*Table 15.2: concept for concept they map. The real design differences are two: **active record** (Eloquent: objects save themselves) against **unit of work** (EF Core, JPA: a context tracks changes and saves them together), and how much is configured by **convention**. The life of one request is the eight steps of Chapter 1 in all three. Book: Sections 15.3.1 and 15.3.2.*

The Same Endpoint Three Times

Listing 15.1 The quote endpoint in `PerfLab/Program.cs`, an ASP.NET Core minimal API.

```
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);
});
```

$$T = \left\lfloor \frac{p \cdot b + 50}{100} \right\rfloor + \begin{cases} 10000 & \text{pickup and subtotal} < 100000 \\ 0 & \text{otherwise} \end{cases} \tag{15.1}$$

Listing 15.1 computes Equation (15.1), the rule of Chapters 8 and 11: `GET /api/quote?price=8000&unit=kg&quantity=2.5&pickup=true` *answers a total of 34000. The book shows the same endpoint as a Laravel route and a Spring controller: a dozen lines each.* `CreateSlimBuilder`*, no logging and JSON generated at compile time keep the measurement about the work, not the extras. Book: Section 15.3.4.*

What a Public Benchmark Measures

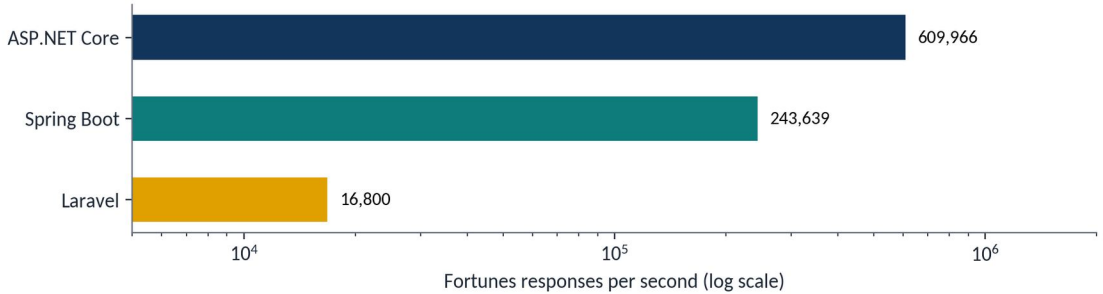


Figure 15.1 TechEmpower Round 23 (February 2025), Fortunes test: responses per second of the best ASP.NET Core, Spring and Laravel implementations, on a logarithmic scale.

Figure 15.1: Round 23 is the last round of the TechEmpower benchmark, archived in March 2026. Its Fortunes test ranks ASP.NET Core, Spring and Laravel in that order, but it shows the best tuned entry of each framework on very large hardware. Note the logarithmic scale. For a laundry, a catering business or a clinic, all three are far faster than the busiest minute will ever need. Book: Section 15.4.1.

Measuring It Yourself

```
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
```

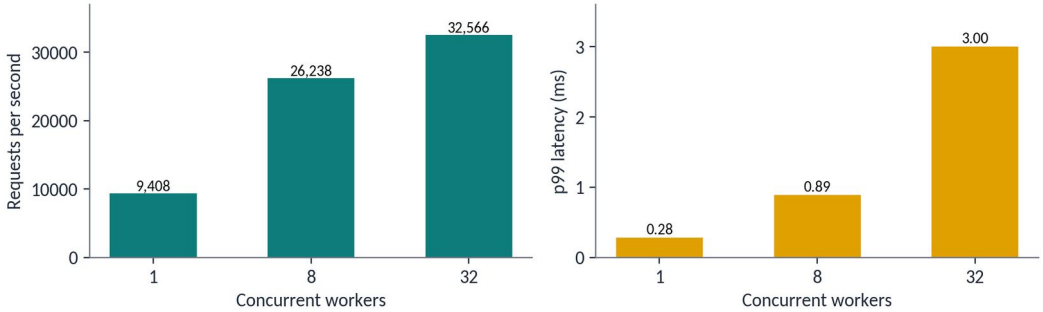


Figure 15.2 Our own measurement of the ASP.NET Core quote endpoint on a 2-vCPU virtual machine (September 25, 2026): throughput levels off after 8 workers while the p99 latency keeps growing.

Figure 15.2: the real output of `LoadGen`, a small load generator, against `PerfLab`. Throughput levels off at the machine's limit while latency keeps growing: more workers only wait longer. Server and client shared two cores, which is fine for showing the method and not for comparing with a data-center benchmark. Warm up first, and measure in Release mode. Book: Section 15.4.2.

Checking the Numbers With Little's Law

$$L = \lambda W \tag{15.2}$$

In a closed load test every worker has exactly one request in flight: L is the number of workers, λ the throughput, so the average latency is $W = L/\lambda$.

1 worker: $W = 1/9408$ s ≈ 0.106 ms. 8 workers: $8/26238$ s ≈ 0.305 ms. 32 workers: $32/32566$ s ≈ 0.983 ms.

Each average lies a little above the median (0.09, 0.28, 0.84 ms) and well below the p99: exactly what a long-tailed latency looks like.

Equation (15.2) is also the tool for step 3 of choosing: estimate the busiest minute. For Dimas's catering business the busiest minute is lunchtime phone orders, a few requests per second, and any of the three frameworks serves that on a small server. Book: Section 15.4.3.

Productivity: Lines for the Same Feature

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	about 12	about 17
Public board, controller and view	18 + 42	17 + 37, plus 1 route	21 + 35, plus 1 rule

Table 15.3: a similar number of lines in all three; Laravel is shortest for small features thanks to its conventions (a gate is one line). Lines are a rough measure: learning curve, tools and the team's familiarity matter as much. Start-up time and memory differ too, which matters on a small server or when paying per container. Book: Sections 15.4.4 and 15.5.

Ecosystem and Community

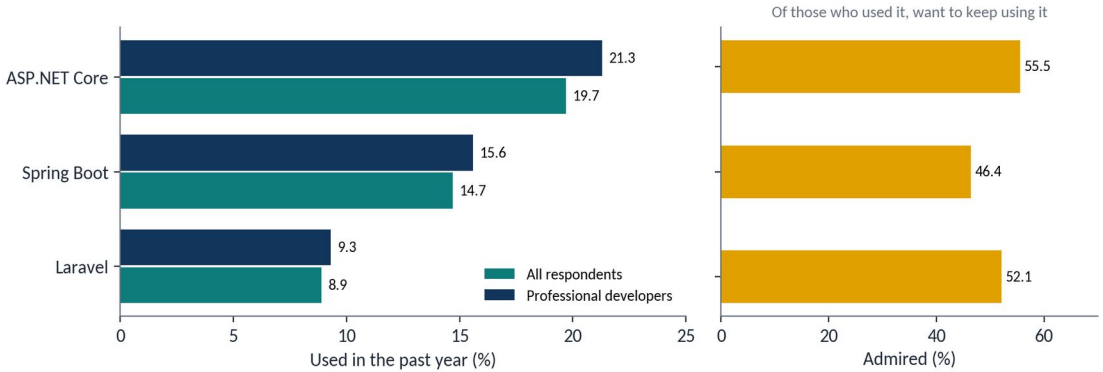


Figure 15.3 Stack Overflow Developer Survey 2025: share of respondents who used each framework in the past year, and share of its users who want to keep using it (“admired”).

Figure 15.3: ASP.NET Core was used by 19.7% of all respondents, Spring Boot by 14.7% and Laravel by 8.9%; about half the users of each want to keep using it. (Chapter 1 quoted slightly higher numbers for professional developers only; always say which population a number describes.) Beyond surveys: NuGet, Packagist and Maven Central, cheap PHP hosting everywhere in Indonesia, and the local job market. Book: Section 15.6.

Support Calendars

Table 15.4 Release and support dates (a selection), collected from endoflife.date on September 25, 2026.

Version	Released	Security fixes until
.NET 10 (LTS)	Nov 11, 2025	Nov 14, 2028
Java 25 (LTS, Oracle Premier)	Sep 16, 2025	Sep 30, 2030
Spring Boot 4.1 (open source)	Jun 30, 2026	Jul 31, 2027
PHP 8.5	Nov 20, 2025	Dec 31, 2029
Laravel 13	Mar 17, 2026	Mar 17, 2028

Table 15.4: .NET LTS versions get three years; Laravel ships a major version yearly with two years of security fixes; Spring Boot ships a minor version every six months with about thirteen months of open-source support. A project started on a version that loses support in three months needs an upgrade in its first release cycle. Book: Section 15.7.

The Calendar at a Glance

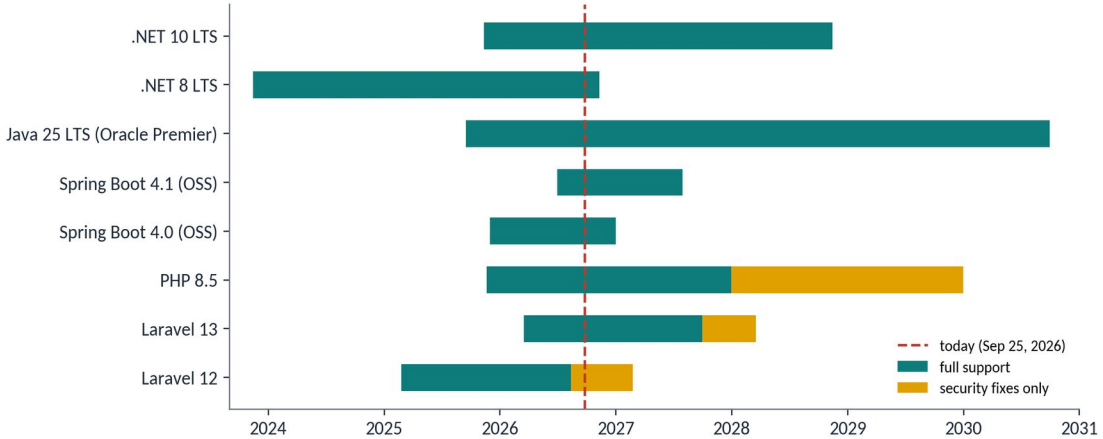


Figure 15.4 The support windows of the versions used in this book; the dashed line is September 25, 2026.

Figure 15.4: plan upgrades as part of the budget from the start. A small business that cannot afford a yearly upgrade should prefer the longest window it can get, or a hosting partner that does the upgrades. Book: Section 15.7.

Security by Default

Table 15.5 Security defaults compared (a selection).

Concern	ASP.NET Core	Laravel	Spring Boot
SQL injection	EF Core parameters	bound values	JPA binds values
XSS in templates	Razor encodes `@value`	Blade `{{ }}` escapes	`th:text` escapes
CSRF on forms	antiforgery tokens	`@csrf` token	token by Thymeleaf
Passwords	PBKDF2-HMAC-SHA512, 100,000	bcrypt, cost 12	bcrypt, cost 10

Table 15.5: the defaults are strong and similar in all three. The risks are the escape hatches: `Html.Raw`, `{!! !!}`, `th:utext`, raw SQL, disabling CSRF. A team that knows where its framework's escape hatches are is safer than a team that picked the "most secure" framework. Book: Section 15.8.

Choosing: Six Steps

1. **Constraints that cannot move:** where it must run, what it must connect to. **2. What the team already knows:** with one developer, choose what that developer knows best.
3. **The load:** estimate the busiest minute with Little's law; usually speed drops out. **4. The calendar:** at least a year of support left.
5. **A spike:** two days building one real feature in the leading candidate reveal more than two weeks of reading. **6. Write the decision down**, with its reasons.

A constraint often decides alone: a client with Windows servers and SQL Server, or a bank with a Java platform. Writing the decision down tells the next person why, and when to revisit it. Book: Section 15.9.

Four Situations

Table 15.6 Four situations and the framework the procedure suggests.

Situation	Likely choice	Main reason
Catering business, one PHP developer, cheap hosting	Laravel	team knowledge and hosting
Hospital group with Windows servers and SQL Server	ASP.NET Core	existing platform and staff
Cooperative bank with Spring-based systems	Spring Boot	fits the 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

Table 15.6: notice how rarely speed is the main reason. For Dimas's catering business the procedure points to what he and the hosting already have. Book: Sections 15.9–15.11.

Common Mistakes

Choosing by a benchmark chart; numbers without source and date; measuring a cold application; a saturated load generator, unmentioned.

Ignoring the support calendar; treating framework differences as the main risk; switching frameworks to fix a design problem.

The last one is common and expensive: fat controllers and rules spread across files stay fat and spread in the new framework. Fix the layering first; Chapters 3, 10 and 13 showed how in each framework. Book: Section 15.12.

Summary: Five Things to Remember

1. The frameworks map concept for concept (Table 15.2); the real differences are active record against unit of work, and convention.

2. Benchmarks show the best entries on large hardware (Figure 15.1); measure your own workload (Figure 15.2) and check it with Little's law (Equation (15.2)).

3. Similar code size, Laravel shortest for small features (Table 15.3); all three widely used and admired (Figure 15.3).

4. Calendars differ (Table 15.4, Figure 15.4); security defaults are similar, the escape hatches are the risk (Table 15.5).

5. Choose by constraints, team, load, calendar and a spike, and write the decision down (Table 15.1, Table 15.6).

For Dimas the procedure is short: one developer who knows PHP best, PHP hosting already rented, a busiest minute of lunchtime phone orders at a few requests per second. Laravel it is, recorded in a one-page architecture decision record (context, options, decision, consequences), with a note to revisit it if the business ever needs a mobile app with heavy traffic. Book: Sections 15.9 and 15.13.

Next Time

Chapter 16 — Evaluation and Discussion IV. The capstone: Profitina Laundry complete in all three frameworks.

Before then: run `PerfLab` and `LoadGen` on your own computer, check your numbers with Little's law, and write a one-page decision for a business you know.

Book: Sections 15.14 and 15.15.

Lecture 16 — Evaluation and Discussion IV: The Capstone

Course: Framework-Based Programming

Book: Irfan Subakti, *Framework-Based Programming: ASP.NET Core, Laravel and Spring Boot*, Chapter 16

Lecturer: Irfan Subakti — yifana@gmail.com

Today the whole course comes together. Profitina Laundry is finished in all three frameworks: the same four entities, the same rules, the same JSON API with sign-in and roles, and the same public counter board. We walk through five problems, one contract and its checker, the rules in three languages, and the three capstones, and end with Bu Wati's decision. Book: Chapter 16, Sections 16.1–16.13.

What This Meeting Covers

1. The last request and the decision — a board by the counter, and which version the outlets run.

2. Problem 16.1 — the contract, written as a checker of 25 requests.

3. Problem 16.2 — the rules in three languages, worked by hand.

4. Problems 16.3–16.5 — the ASP.NET Core, Laravel and Spring Boot capstones.

5. Comparing and deciding — the three solutions side by side, and Bu Wati's decision.

The chapter says exactly what was run, on four levels of evidence: syntax-checked, compiled, run, and checked against the contract. The ASP.NET Core capstone (memory build) was run here and passed the checker; the Laravel code was syntax-checked; the Spring code was compiled against stand-ins and its rules were run. Book: Section 16.2.

Where We Left Off

Chapters 3–6 built the laundry in ASP.NET Core, 7–10 in Laravel, 11–14 in Spring Boot; Chapter 12 wrote the API as a contract; Chapter 15 compared the frameworks.

The request: a television by the counter in each outlet, showing which orders are in the queue and which are ready, so customers stop asking “is mine ready yet?”.

The decision: which of the three versions the outlets will run. Raka does not care which, as long as his app keeps working.

The cleanest way to keep Raka’s app working, whichever version runs, is to make the contract so precise that a program can check it. That is Problem 16.1. Book: Section 16.1.

How to Approach the Capstone

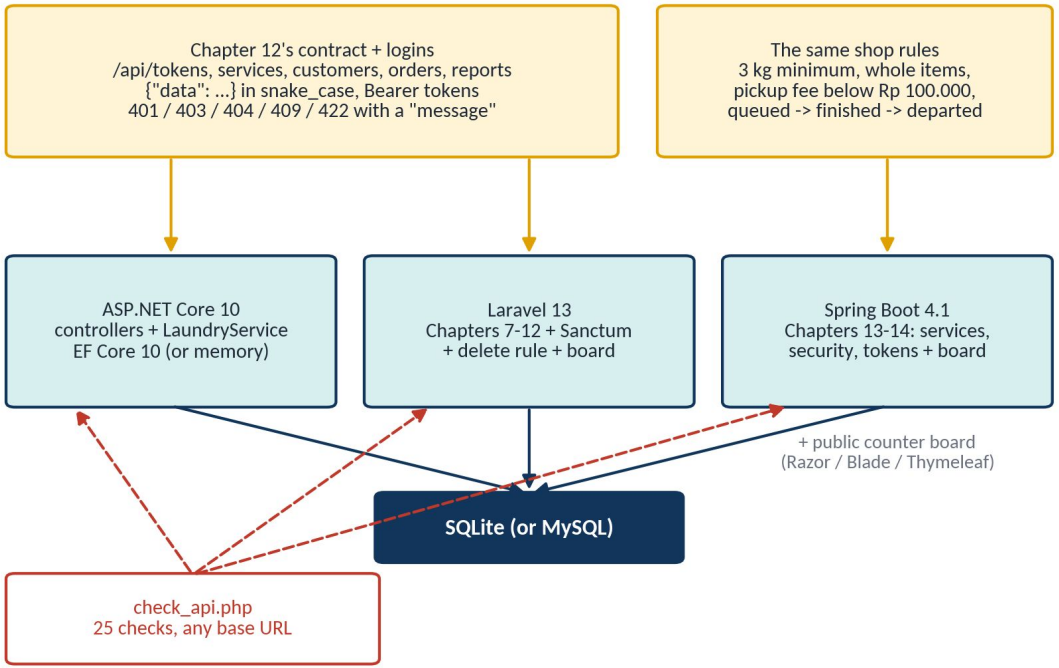


Figure 16.1 One contract, three implementations: the checker `check_api.php` sends the same 25 requests to each server and compares the answers with the contract.

Figure 16.1: the familiar routine: state the problem, work it by hand, design on paper, solve, run and check, look for the tempting wrong solution. Two rules are new: say what was run, and place every result on one of the four levels of evidence. Only the last level, checked against the contract, proves that the phone app will work. Book: Section 16.2.

Problem 16.1: The Contract as a Program

Chapter 12's sixteen checks, plus sign-in with a wrong password, a request without a token, a cashier's delete refused with 403, the owner's two deletes (409 with orders, 204 without), sign-out, and the old token refused.

The checker is plain PHP, runs against any base URL, generates what must be unique (a random phone), and reads the fields the app reads.

Reading a failing run: study the *first* `FAIL`; if sign-in fails, twenty later checks fail with 401 from one cause.

Two kinds of first failure are common: a field with another name or shape (`customerId` for `customer_id`, `"QUEUED"` for `"queued"`, a date with fractions of a second), and a different status (400 instead of 422). The fix belongs in the server, never in the checker. Book: Sections 16.3.1–16.3.5.

The Tempting Wrong Checker

Compare whole JSON answers character by character: fails for the wrong reasons. Identifiers, times and the random phone differ on every run; `4.5` against `4.50`; field order in JSON means nothing.

Check status codes only: a server that answers 422 with an empty body, or 201 with `"id": null`, passes and still breaks the app.

The right checker names each expectation: this status, this field, this value, this shape.

A contract only in prose is read differently by different people; a contract as a program is read the same way every time. Book: Section 16.3.6.

Problem 16.2: The Rules in Three Languages

$$T = \left\lfloor \frac{8000 \cdot \max(250, 300) + 50}{100} \right\rfloor + 10000 = 24000 + 10000 = 34000 \tag{16.1}$$

$$B = T - P, 0 \le B \le T \tag{16.2}$$

Equation (16.1) is the checker's own order worked by hand: 2.5 kg of wash, dry and fold is 250 hundredths, billed as 300, Rp 24.000, plus Rp 10.000 pickup because the subtotal is below Rp 100.000. Equation (16.2): the balance never goes below zero (no overpayment) or above the total. Book: Sections 16.4.1 and 16.4.2.

Nine Prices Worked by Hand

Table 16.1 Six of the nine prices worked by hand.

Service, price	Quantity	Billed	Subtotal	Pickup	Total
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
Curtain, 15.000	4.40 kg	440	66.000	yes	76.000
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

Table 16.1: look at the shirts. A subtotal of exactly Rp 100.000 pays no pickup fee, because the rule says "below". A `<=` in the code would charge Rp 10.000 wrongly; only a case on the boundary catches it. The Java and C# rules produced the same totals with integer arithmetic in hundredths. The tempting wrong solution, `Math.round(price * Math.max(quantity, 3))` with `double`, is right for most inputs and wrong where 4.02 is 4.0199999999999996. Book: Sections 16.4.3 and 16.4.4.

Problem 16.3: The ASP.NET Core Capstone

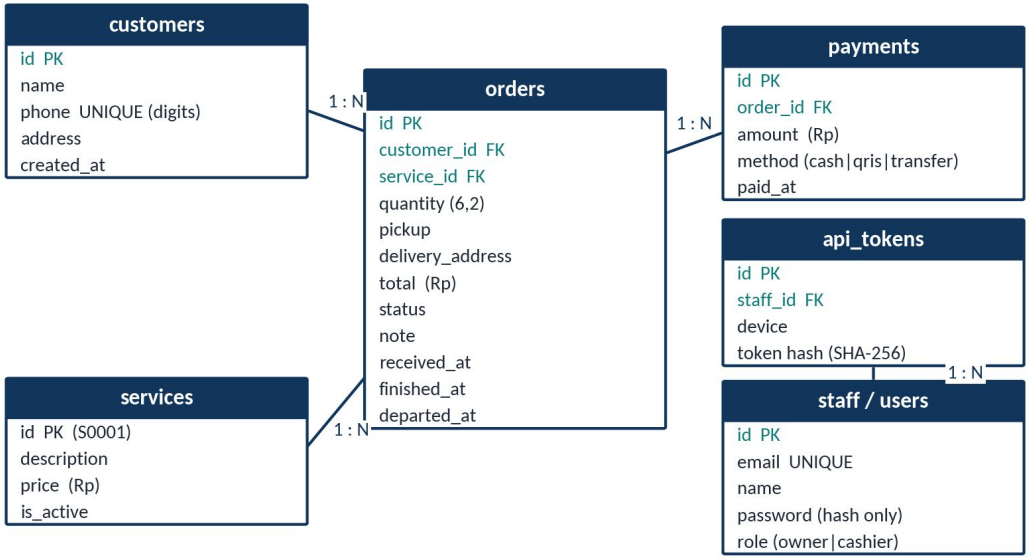


Figure 16.2 The capstone's tables: services from the price list (`S0001` to `S0012`), customers, orders with pickup, payments, staff and API tokens.

Figure 16.2: built from scratch with everything the course taught: `LaundryRules` in integer arithmetic, a storage interface `ILaundryStore` with two implementations (EF Core with SQLite, and memory), `LaundryService` with every rule, three exceptions, request records with validation attributes, `ApiErrors` for the contract's error shape, and a bearer-token handler with Sanctum-style hashed tokens. A switch `UseEf` in the project file decides whether EF Core is compiled in. Book: Sections 16.5.1–16.5.5.

Running It, and a Real Check

```
dotnet run -p:UseEf=false --urls http://localhost:5000
php ../../contract/check_api.php http://localhost:5000
```

```
PASS  a wrong password -> 422, errors.email
PASS  no token -> 401 with a message
PASS  POST /api/orders -> 201, total 34000 (3 kg + pickup), queued
PASS  a bad order -> 422, all of its errors at once
PASS  paying more than the balance -> 422, errors.amount
PASS  a cashier deletes a customer -> 403 with a message
PASS  the owner deletes a customer with orders -> 409
PASS  the owner deletes a customer without orders -> 204
PASS  the old token no longer works -> 401

25 passed, 0 failed
```

Nine of the 25 lines of the real run against the memory build, on SDK 10.0.112 with runtime 10.0.12; Chapter 12's sixteen checks passed too. The EF Core build could not be run where the book was written, since its package could not be downloaded; the README asks you to run the checker against it first. Book: Section 16.5.6.

The Counter Board

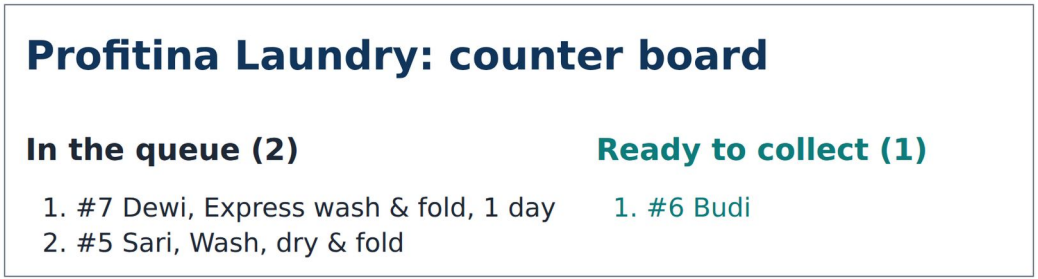


Figure 16.3 The counter board of the ASP.NET Core capstone as it ran on September 25, 2026: two orders in the queue, one ready to collect.

*Figure 16.3: a public page, `[AllowAnonymous]`, that refreshes every 30 seconds and shows **order numbers and first names only**. A television by the counter is seen by every customer in the shop: no phone numbers, no addresses, no amounts. The chapter tests that the rest does not appear, because personal data on a public screen is exactly the mistake nobody notices until a customer does. Book: Section 16.5.7.*

Two Mistakes Only the Checker Found

```
public sealed record OrderRequest(
    [property: Required] long? CustomerId, ...
```

It built without a warning. The first `POST /api/orders` failed: ASP.NET Core refuses validation attributes on the properties of a positional record and wants them on the constructor parameters.

With the records fixed, errors came back keyed `CustomerId` and `Quantity`: `ModelState` names fields after the C# property, and the snake_case JSON options do not apply to those keys.

Both compiled cleanly and would have passed a quick look in the browser. The checker, which expects `customer_id`, reported the second at once, and `ApiErrors` now converts the keys. This is why "compiled" is a lower level of evidence than "checked against the contract". Book: Section 16.5.8.

Problem 16.4: The Laravel Capstone

Laravel needed the least: the contract *is* Laravel's default shape. New: a `DELETE` route for the owner only, a gate `delete-customers`, `destroy` answering 409 for a customer with orders, and the board as a public route with a Blade view.

`CapstoneTest` checks a cashier's delete, the owner's two deletes, and the board's privacy.

Tempting wrong: `if (! $request->user()->isOwner()) abort(403);` inside the controller. 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 routes are read, and run before binding. Composer could not download Laravel where the book was written, so the code was syntax-checked with `php -l`; on your computer, `migrate:fresh --seed`, the tests, the server and then the checker with Laravel's demo accounts. Book: Section 16.6.

Problem 16.5: The Spring Boot Capstone

Spring needed the board and one security rule: `BoardController` reads its orders through `OrderService`, `board.html` is the same board in Thymeleaf, and `/board` is `permitAll()` in the web chain.

Everything else, the contract's errors, tokens and the owner-only delete, came from Chapters 12 to 14.

On your computer: `mvn test` runs the 24 test runs of Chapter 14, the application listens on port 8080 with SQLite, and the checker uses the default accounts. Where the book was written, every Java file was compiled against stand-ins and the rules were run. Book: Section 16.7.

Comparing the Three Solutions

Table 16.2 The three capstones compared (a selection).

Concern	ASP.NET Core	Laravel	Spring Boot
"Signed in unless marked"	fallback policy, `[AllowAnonymous]`	`auth:sanctum` group	`authenticated()`, `permitAll()`
Owner only	`[Authorize(Roles = "OWNER")]`	gate and `can:`	URL rule and `@PreAuthorize`
422 with every error	merge `ModelState` and rules	Form Requests, by default	`Validator` and rules merged
snake_case JSON	naming policy, plus error keys	by convention	Jackson setting
Board, first names only	Razor	Blade	Thymeleaf
What was run here	built and run; 25/25 and 16/16	`php -l`	compiled against stand-ins; rules run

Table 16.2: the framework whose defaults match the contract needs the least adaptation. The durable parts of the system are not the frameworks but the rules, the contract and the checker: they would survive a fourth framework unchanged. Book: Section 16.8.

Bu Wati's Decision

Constraints: a small rented virtual server with 2 GB; any of the three runs on it; nothing to connect to a Java or Windows platform.

People: Dimas maintains the system and knows PHP and Laravel best; nobody in the family works with C# or Java daily.

Load: about 150 orders a day across three outlets, well under one request per second in the busiest hour; by Little's law fewer than 0.03 requests in progress. **Calendar:** each version is supported long enough for the first year.

*The procedure points to **Laravel**: the person who maintains 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, with the condition to revisit it: many outlets with a central server, or 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. Book: Section 16.9.*

Common Mistakes

A contract only in prose; checkers that cannot run twice; floating-point money; `<=` where the rule says "below".

`[property:]` on validation attributes of positional records; trusting the naming policy for error keys; personal data on a public screen.

Role checks inside controller methods; claiming more than was run.

The last mistake is the one this book tried hardest to avoid: every chapter said what was built, what was run and what was only compiled. Say the same about your own work. Book: Section 16.12.

Summary: Five Things to Remember

1. One contract (Table 16.3) and one checker of 25 requests (Figure 16.1) define the API for all three frameworks.

2. The rules are worked by hand (Equation (16.1), Equation (16.2), Table 16.1) and computed in integers in every language.

3. The ASP.NET Core capstone (Figure 16.2) passed 25 of 25 checks; two clean-compiling mistakes were found only by the checker.

4. The public board shows order numbers and first names only (Figure 16.3), in Razor, Blade and Thymeleaf.

5. The framework whose defaults match the contract needs least adaptation (Table 16.2); the rules, the contract and the checker are the durable parts.

The contract itself comes next, the one artefact that every version of the shop must honour. Book: Section 16.13.

The Contract After Chapter 16

Table 16.3 The Profitina Laundry API contract after Chapter 16.

Part	Rule
Sign-in	`POST /api/tokens` with `email`, `password`, `device`: 201 with a token; a wrong password is 422 on `email`
Doors	no token: 401 “Unauthenticated.”; a role that may not: 403 “This action is unauthorized.”
Data	snake_case fields; one record as `{"data": {...}}`; a page with `data` and `meta`
Values	lower-case statuses and methods; whole rupiah; `YYYY-MM-DDTHH:MM:SS` in shop time
Bad input	422 with `message` and `errors`, every error at once
Not found, closed	404 with a `message`; 409 for a request the record’s state forbids
Roles	only the owner deletes customers; a customer with orders cannot be deleted (409)

Table 16.3: at the start of the course Bu Wati’s wish was “one change in one place”. This table, and the program that checks it, is that wish for the whole shop: one place where the promise to every client is written, in three frameworks at once. Book: Sections 16.3.3 and 16.14–16.15 for the exercises and their solutions.