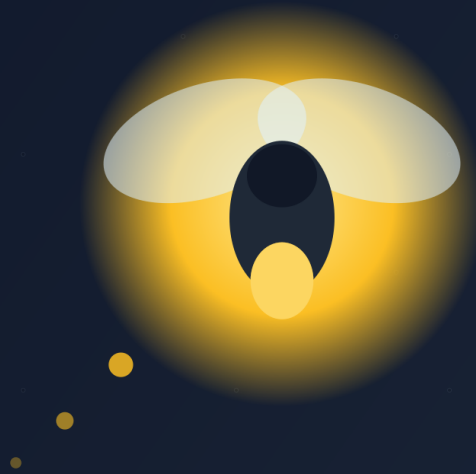


FIREFLY SOFTWARE SOLUTIONS INC.



LaraFly

by Example

Hexagonal PHP Microservices
on Laravel 13 with the Firefly Framework

LaraFly by Example

Hexagonal PHP Microservices on Laravel 13 with the Firefly Framework
Firefly Software Solutions Inc.

Copyright © 2026 Firefly Software Solutions Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this material except in compliance with the License. You may obtain a copy of the License at <https://www.apache.org/licenses/LICENSE-2.0>.

Unless required by applicable law or agreed to in writing, software and documentation distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First edition.

Every code listing in this book is drawn from the LaraFly framework's own source or from `samples/lumen`, the companion digital-wallet-and-ledger application shipped in the framework's monorepo, and is verified to lint cleanly against a real PHP 8.3+ toolchain — nothing here is illustrative pseudocode.

"LaraFly" is the working name for the Firefly Framework's PHP-on-Laravel port.

"Laravel" is a trademark of Taylor Otwell; this is an independent work and is not affiliated with or endorsed by the Laravel project.

Published by Firefly Software Solutions Inc.

For every PHP developer who has hand-wired a service container, a router, a validator, and an event dispatcher from four unrelated packages — and then done it again, slightly differently, on the next project.

This is the one way. Use it in good health.

Preface

Enterprise PHP has long meant assembling a service container from one package, a router from another, a validator from a third, and an event dispatcher from a fourth — each with its own configuration idiom, its own conventions, its own way of saying "inject this." Laravel solved a great deal of that fragmentation for web applications, but hexagonal, service-oriented PHP — attribute-driven dependency injection, declarative transactions, CQRS, event-driven architecture, method-level security, production observability — has stayed a matter of taste, folder-naming conventions, and hand-rolled glue.

LaraFly changes that. It brings the cohesive, convention-over-configuration experience Spring Boot gave the Java world to PHP 8.3+ and Laravel 13, built *on* Laravel rather than instead of it: the same Eloquent, the same service container, the same `artisan`, now organized by PHP 8 attributes into stereotyped components, a compiled dependency graph, and a deterministic boot pipeline.

This book teaches LaraFly **by example**. Every listing in these pages is not illustrative pseudocode — it is lifted from the framework's own shipped source or from `samples/Lumen`, a real, running digital-wallet-and-ledger application that ships in the framework's monorepo, compiles, boots, and passes its own test suite. What you read is what actually works.

Who This Book Is For

This book is for intermediate PHP developers who are comfortable with Laravel's basics — Eloquent models, routing, the service container, `artisan` — and want to see how a hexagonal, enterprise-shaped architecture is built *on top of* that foundation rather than instead of it. You need no prior exposure to Spring, CQRS, or event-driven architecture; each idea is introduced from first principles before the code that implements it.

Developers coming from Spring Boot, Micronaut, or Quarkus will feel especially at home. Wherever a LaraFly concept mirrors a Laravel-native one — a `#[Service]` next to a plain class bound in a service provider, a `#[RestController]` next to a plain Laravel controller — a **Laravel parity** callout draws the comparison explicitly, so you can map what you already know onto what is new.

What You Will Build

Every chapter advances **Lumen**, a digital-wallet-and-ledger service: a `Wallet` can be opened, deposited to, withdrawn from, and transferred between other wallets, protecting one invariant above all others — **the balance never goes negative** — and recording every state change as a domain event that a listener projects into an append-only ledger. It is a small system, but it is shaped exactly like a real one: a domain layer with no framework dependency, a hexagonal port and its Eloquent adapter, CQRS command and query handlers, domain events bridged to an event bus, method-level security on a sensitive operation, and a thin REST controller that holds no business logic of its own.

The journey starts gently. The **Quick Start** takes you from the bundled `firefly new` installer to a running, curl-able endpoint, previewing the stereotypes, the container, and the compiled boot path in miniature before any chapter asks you to reason about them. **Chapter 1** steps back and makes the case for the whole approach — what problem LaraFly solves and the pillars it stands on. **Chapter 2** opens the engine room: the dependency-injection container, the stereotype attributes, and the component scan that compiles your annotated classes into a cached, zero-reflection boot manifest. Later parts of this book — arriving in the chapters that follow — build outward from that foundation into configuration, HTTP, persistence, domain modelling, CQRS, event-driven architecture, security, and observability, always through the same `Lumen` codebase, always with code you can run.

By the time you have worked through the full book, you will have a mental model for every layer of a production LaraFly service, and a real, tested application to show for it.

How to Use This Book

Read sequentially. Each chapter builds on the vocabulary and the code of the one before it.

Type the listings yourself. Reading and typing code at the same time is how the patterns stick. Resist copy-pasting until you have written each listing at least once by hand.

Run it. Every listing in this book is real, and most of it lives, unchanged, in the framework's own `samples/lumen` package. Clone the framework's repository, and you can run the finished application — `composer test -- samples/lumen/tests` exercises the whole sample suite — and compare your own work against it at any point.

Each chapter closes with a short **Summary** of what you learned and, where it applies, a **Try it yourself** section that pushes one step further on your own.

Conventions in Brief

Typographic and structural conventions — code-listing style, callout types, and figure numbering — are demonstrated, with live examples, in the **Conventions** section that follows this preface.

The Companion Code

The finished `Lumen` application lives in the framework's monorepo at `samples/lumen`. It is the destination this book walks you toward: a layered project — `Domain`, `Application`, `Infrastructure`, `Web` — that the later chapters of this book grow, feature by feature. Use it to compare your own code, to catch up if a chapter moves faster than you expect, or simply to run the parts you are currently reading about.

Conventions

This page explains the typographic and structural conventions used throughout the book.

Code Listings

Real code is set in a plain fenced block tagged `php`, and **every one of those listings says where it comes from**. Directly above each fence sits an HTML comment — invisible in the rendered page, but the reason you can trust what follows it — and there are exactly two kinds:

`<!-- source: <path> -->` marks **framework code**: the block is a *verbatim* excerpt of that file in the LaraFly repository — the same lines, in the same order, at the same relative indentation. `tests/DocsCodeIsRealTest.php` compares every such block against the file it names on every run, so a listing cannot drift when the code moves; and the file it quotes is already held to the repository's own gates — PHPStan at level max, Pint, and the package's Pest suite. Nothing in one of these blocks is paraphrased, tidied for the page, or invented.

`<!-- illustrative: <why> -->` marks **your** code: the class a reader writes in their own application, which by definition cannot exist in this repository, so there is no file to compare it against. Those blocks are the ones linted with the real PHP CLI (`php -l`), and every `use Firefly\...` they import and every `#[Attribute]` they use is checked against the classes the framework really declares.

Both kinds are additionally checked for what they *assert*: every `firefly.*` configuration key, every `php artisan firefly:*` command and every `composer <script>` a listing names has to be one that really exists.

Elisions (`// ...`)

A `source:` excerpt may **cut** whole lines it does not need — an import list, a long docblock, the middle of a method. Every cut is marked with a line that is exactly `// ...` (or `# ...` in a file, such as YAML, whose comments start with a hash), and the comparison resumes after what it has already matched, so what remains is still, in order, what the file says. Three things a cut may never do, all refused by the guard: swallow the declaration it belongs to — leaving a `{` with nothing above it saying *what* is being declared — hollow a method out to an empty body, which would print a method that appears to do nothing, or leave a docblock standing on its own. A docblock is a claim *about* a declaration, so an excerpt that shows one shows its `/**`, its closing line, and at least one line of the class, method or constant it describes; starting partway down a docblock would print a paragraph of commentary about code the page never shows.

For example, this is the `Money` value object from `samples/lumen`, whole down to `zero()` and then cut:

```
<?php

declare(strict_types=1);

namespace Lumen\Domain;

use Firefly\Domain\ValueObject;
use Firefly\Domain\ValueObjectEquality;
use Firefly\Kernel\Exception\Business\ConflictException;

final readonly class Money implements ValueObject
{
    use ValueObjectEquality;

    public function __construct(public int $minorUnits, public Currency $currency) {}
}
```

```

    public static function zero(Currency $currency): self
    {
        return new self(0, $currency);
    }
    // ...
}

```

Inline code references within prose use `monospace` font, as in "the `#[Service]` attribute registers the class with LaraFly's container."

Callouts

Four callout styles appear throughout the book:

NOTE

Notes provide supplementary context or clarify a subtlety in the main text — worth reading, but not blocking.

TIP

Tips share a shortcut, idiom, or best practice that will save you time in real projects.

WARNING

Warnings flag a common mistake or a sharp edge that can cause hard-to-debug problems if ignored.

LARAVEL PARITY

Laravel-parity callouts map a LaraFly concept directly to its native Laravel equivalent — ideal if you are coming from stock Laravel rather than from Spring Boot or another convention-over-configuration framework.

Figures

Diagrams are captioned below the image and embedded as inline SVG, so they render crisply at any zoom level in both the screen and print editions. You will meet the first one in Chapter 2.

Contents

Quick Start

Build Lumen Step by Step.....	15
-------------------------------	----

PART I Foundations

1. Why LaraFly?	29
2. Dependency Injection & Auto-Configuration	38
3. Configuration, Profiles & Secrets	56
4. Your First HTTP API	72
4A. Documenting the API: OpenAPI 3.1 from the Manifests	101

PART II Modeling & Persisting the Domain

5. Persistence & the Repository Pattern	147
6. Domain-Driven Design	189

PART III Coordinating and Securing the Application

7. CQRS: Commands and Queries	219
8. Event-Driven Architecture and the Transactional Outbox	239
9. Transactions and the #[Transactional] Proxy	268
10. Security: Authentication and Authorization	299
10A. OAuth2 and OpenID Connect: Signing In, and Being the Provider	352

PART IV Observability, Testing & Delivery

11. Observability: Health, Metrics, and the Actuator.....	396
12. Testing LaraFly Applications.....	459

13. The CLI and the Zero-Reflection Cache.....	495
--	-----

Appendices

A. Laravel → LaraFly Cheat-Sheet	517
Glossary	527

Quick Start

QUICK START

Build Lumen Step by Step

Welcome. Before the deep dive, this chapter takes you from an *empty terminal* to a *running, curl-able* LaraFly application — installed, compiled, and served — in well under ten minutes. Every command and every listing here is real: it is exactly what `firefly new` generates, unmodified.

This is a *tour*, not the deep dive. Chapter 1 makes the case for the whole approach; Chapter 2 opens the engine room and explains, in depth, everything you meet here only in passing — the container, the stereotypes, and the compiled boot manifest. The goal of this chapter is momentum: by the end of it you will have a real service running, and a first, informal feel for what "convention over configuration" means in LaraFly.

 NOTE

Every listing in this chapter is copied verbatim from `skeleton/`, the project scaffold `firefly new` installs, and from `samples/lumen`, the fuller digital-wallet-and-ledger application this book builds toward over the chapters that follow.

Step 1 — Prerequisites

LaraFly is a PHP framework built on Laravel 13. You need:

- **PHP 8.3 or newer**, with the extensions Laravel itself requires (`mbstring`, `openssl`, `pdo`, `tokenizer`, `xml`, `ctype`, `json`).
- **Composer 2**, PHP's dependency manager.

Confirm both are available:

```
php --version  
composer --version
```

Step 2 — Install

The bundled `firefly new` installer supplies the application template to Composer:

```
composer global require fireflyframework/laraflly  
firefly new my-app  
cd my-app
```

`composer create-project` does more than copy files: it runs a `post-create-project-cmd` script that finishes setting the project up for you —

```
"post-create-project-cmd": [  
    "@php -r \"file_exists('.env') || copy('.env.example', '.env');\"",  
    "@php -r \"file_exists('database/database.sqlite') || touch('database/database.sqlite');\"",  
    "@php artisan key:generate --ansi",  
    "@php artisan firefly:cache"  
]
```

By the time that command returns, `.env` exists, a SQLite database file has been touched, `APP_KEY` is set, and — the step that matters most for this book —

`firefly:cache` has already compiled your application's manifests. You have not written a single line of PHP yet, and the zero-reflection boot path this whole framework is built around is already in place.

💡 THE BUNDLED INSTALLER

`composer global require fireflyframework/laraflly` gives you a `firefly` command on your `PATH`. `firefly new my-app` supplies the bundled skeleton to Composer, then runs `git init` and an initial commit for you — the LaraFly equivalent of `laravel new`.

What you just installed

The skeleton's `composer.json` requires the root library and records a CLI component requirement satisfied by that library:

```
"require": {  
    "php": "^8.3",  
    "firefly/cli": "*@dev",  
    "fireflyframework/laraflly": "*@dev",  
    "laravel/framework": "^13.0"  
}
```

`firefly/cli` gives you the `artisan firefly:*` commands you will use throughout this book. `fireflyframework/laraflly` is the **complete framework library**, which contains the whole Firefly family (container, context, config, web, data, cqrs, eda, security, validation, resilience, scheduling, observability, actuator, and more) in a single `require` line, so your own `composer.json` never has to enumerate them one at a time.

📦 LARAVEL PARITY

`firefly new` is LaraFly's counterpart to `laravel new` — and `fireflyframework/laraflly` is the counterpart of installing `laravel/framework` itself: one dependency line that brings in an entire, coherent stack rather than a collection of independently-versioned pieces.

Step 3 — A look inside

Open the generated project. Two things are worth noticing immediately, because they are the whole idea of this framework in miniature.

First, `bootstrap/providers.php` is empty — the whole file, not an excerpt of it:

```
<?php
declare(strict_types=1);

return [];
```

There is no service provider to register by hand. `firefly/cli` and `fireflyframework/laraflly`'s own providers are discovered automatically by Composer/Laravel package discovery — you never add a line here for a Firefly package.

Second, `routes/web.php` is almost empty too:

```
// Intentionally minimal: Firefly's WebServiceProvider registers the app's #[RestController]
routes from
// the compiled RouteManifest (see skeleton/app/Http/GreetingController.php). This file
exists because
// bootstrap/app.php's withRouting(web: ...) requires the path.
```

Routes are not declared here at all. As the comment says, they come from a **compiled RouteManifest** — built from attributes on your controllers, not from a routes file you maintain by hand. You will meet the class that comment points to in a moment.

The one file that does matter right now is `config/firefly.php`. The copy the skeleton ships is a long annotated **reference** — every `firefly.*` key the framework reads, grouped by the package that reads it, with the real default beside it — so the two blocks that matter today are shown here with the rest of the file cut away. Each `// ...` below stands for whole lines removed from the real file; the conventions page explains the marker:

```
<?php

declare(strict_types=1);
// ...
return [
// ...
    'scan' => [
        'paths' => [
            'App\\' => app_path(),
        ],
    ],
// ...
    'cache' => [
        'path' => base_path('bootstrap/cache/firefly'),
        'component_manifest' => base_path('bootstrap/cache/firefly/component.php'),
        'context_manifest' => base_path('bootstrap/cache/firefly/context.php'),
    ],
// ...
];
```

`scan.paths` tells LaraFly's component scanner which PSR-4 roots to look at — here, just your own `App\` namespace under `app/`. `cache.path` is where `firefly:cache` (which you already ran, in Step 2) wrote the compiled manifests it found there.

Step 4 — Meet the first bean

`app/GreetingProperties.php` is a small, typed configuration DTO — LaraFly's answer to Spring's `@ConfigurationProperties` :

```
<?php

declare(strict_types=1);

namespace App;

use Firefly\Config\Attributes\ConfigProperties;

/**
 * Binds the `greeting.*` configuration subtree onto this readonly DTO and registers it as a
 container
 * singleton, so it can be constructor-injected wherever GreetingProperties is requested.
 */
#[ConfigProperties('greeting')]
final readonly class GreetingProperties
{
    public function __construct(public string $salutation = 'Hello') {}
}
```

`app/GreetingService.php` is a plain PHP class carrying one attribute, `#[Service]` :

```
<?php

declare(strict_types=1);

namespace App;

use Firefly\Container\Attributes\Service;

/**
```

```

* A #[Service] stereotype: auto-registered as a singleton bean and resolved through the
container, so its
* GreetingProperties dependency is autowired.
*/
#[Service]
final class GreetingService
{
    public function __construct(private readonly GreetingProperties $properties) {}

    public function greet(string $name): string
    {
        return sprintf('%s, %s!', $this->properties->salutation, $name);
    }
}

```

Nothing here registers `GreetingService` with a container by hand, and nothing wires `GreetingProperties` into its constructor either. `#[Service]` is enough: the component scan finds the class, the container builds it, and because the constructor asks for a `GreetingProperties`, the container resolves and injects one automatically. This is **constructor dependency injection**, and it is the subject of the whole of Chapter 2.

Step 5 — Expose it over HTTP

`app/Http/GreetingController.php` puts a web edge on the service:

```

<?php

declare(strict_types=1);

namespace App\Http;

use App\GreetingService;

```

```

use Firefly\Web\Attributes\GetMapping;
use Firefly\Web\Attributes\PathVariable;
use Firefly\Web\Attributes\RestController;

/**
 * The sample Firefly slice: a #[RestController] whose routes are discovered by the
 * RouteScanner and served
 * from the compiled RouteManifest. GreetingService is autowired via constructor DI.
 *
 * '/' belongs to App\Http>WelcomeController, a #[Controller] that renders HTML – this one
 * returns a value
 * the ResponseFactory negotiates into JSON, which is the difference between the two
 * stereotypes.
 */
#[RestController]
final class GreetingController
{
    public function __construct(private readonly GreetingService $greetings) {}

    /** @return array<string, string> */
    #[GetMapping('/greetings/{name}', name: 'greetings.show')]
    public function show(#[PathVariable] string $name): array
    {
        return ['message' => $this->greetings->greet($name)];
    }
}

```

`#[RestController]` is itself a stereotype — it is built on the same `#[Component]` base as `#[Service]` — so `GreetingController` is autowired exactly like `GreetingService` was: the container builds it, sees the constructor wants a `GreetingService`, and supplies one. `#[GetMapping]` and `#[PathVariable]` are what the *route scan* reads, separately, to build the `RouteManifest` `routes/web.php`'s comment referred to.

Step 6 — Run it

Start the development server:

```
php artisan firefly:serve
```

`firefly:serve` is a thin wrapper: it calls `artisan serve` (or `octane:start`, if `laravel/octane` happens to be installed) — it does not reimplement anything of its own. In another terminal, hit the route you just read:

```
curl -s localhost:8000/greetings/Ada
```

```
{"message":"Hello, Ada!"}
```

"Hello" is `GreetingProperties`'s default `$salutation` — nothing in `config/greeting.php` overrides it yet, so the constructor default is what you see. Change that default, or bind `greeting.salutation` in your own config, and every response reflects it — with no code change to either the service or the controller.

`/` is not this controller's route, and the docblock above says which class owns it: `App\Http>WelcomeController`, a `#[Controller]` rather than a `#[RestController]`, whose `index()` returns a view. Open `localhost:8000/` in a browser and what comes back is the skeleton's HTML welcome page, not JSON — that difference between the two stereotypes is exactly what the docblock is there to warn you about.

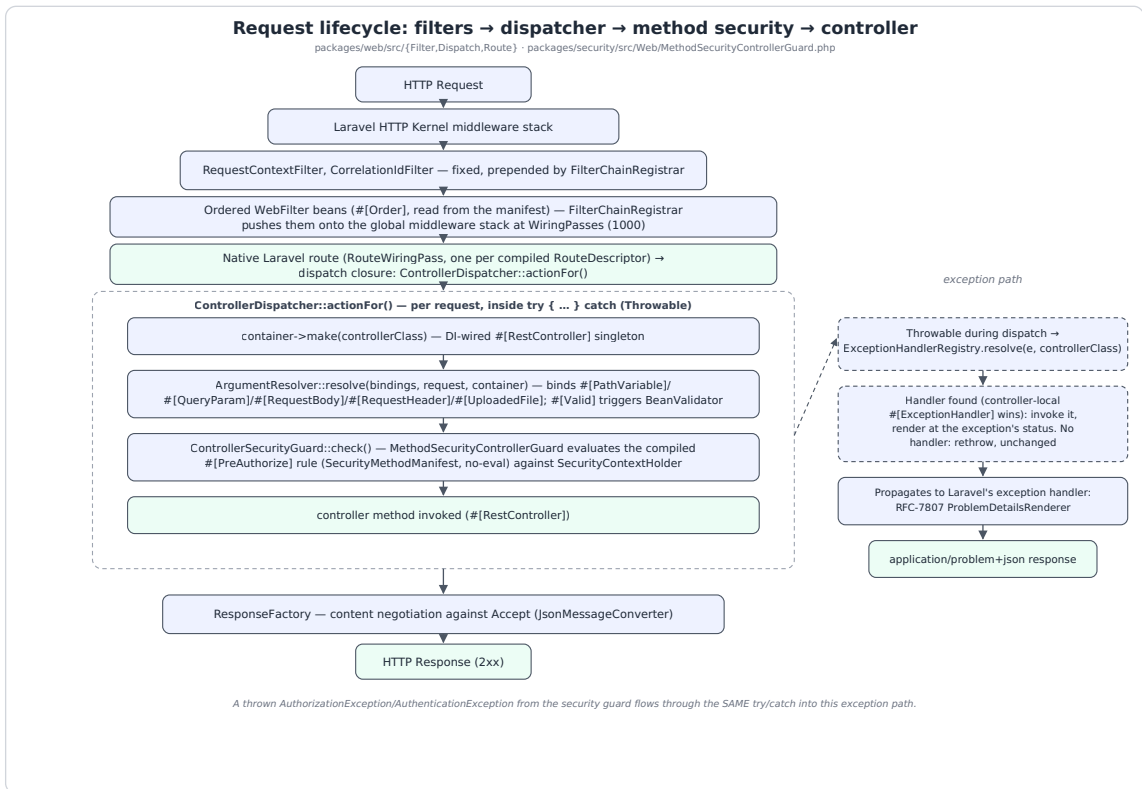


Figure 0.1 — A request travels through the web filter chain and the controller dispatcher before your handler method ever runs.

❗ TWO COMMANDS, ONE JOB

You already ran `firefly:cache` once, indirectly, during `composer create-project`. If you add or change an attribute — a new `#[Service]`, a new route — re-run `php artisan firefly:cache` to recompile the manifests, or `php artisan firefly:clear` to delete them and fall back to the (slower, reflection-based) development scan path. Chapter 2 explains exactly what each command produces.

Step 7 — A glimpse of where this is going

`GreetingService` is deliberately the smallest possible slice: one bean, one dependency, one route. The application this book actually builds, chapter by chapter, is bigger — **Lumen**, a digital-wallet-and-ledger service that already exists, fully built and tested, in the framework's own `samples/lumen` package. It exposes:

Method	Path	Purpose
POST	<code>/api/v1/wallets</code>	Open a wallet
POST	<code>/api/v1/wallets/{id}/deposit</code>	Deposit funds
POST	<code>/api/v1/wallets/{id}/withdraw</code>	Withdraw funds — secured
POST	<code>/api/v1/wallets/transfers</code>	Transfer funds between two wallets
GET	<code>/api/v1/wallets/{id}/balance</code>	Fetch just the balance
GET	<code>/api/v1/wallets/{id}/ledger</code>	Fetch the wallet's projected ledger

Every one of those endpoints is built from the same handful of ideas you just met in miniature — a stereotyped class, constructor injection, an attribute the framework reads at scan time — plus a great deal more: a hexagonal repository port and its Eloquent adapter, CQRS command and query handlers, domain events projected into a ledger, and method-level security on the withdraw endpoint. You can run its whole test suite right now, before reading another page, if you want to see it working:

```
composer test -- samples/lumen/tests
```

The rest of this book builds exactly that application with you, one concept at a time, starting with the idea you have already used twice without a full explanation: the container.

✓ What you built

You went from an empty terminal to a compiled, running, curl-able LaraFly application, and you met — without yet needing the full explanation — a `#[Service]`, a `#[ConfigProperties]` DTO, a `#[RestController]`, constructor dependency injection, and the `firefly:cache` compile step that turns all of it into a zero-reflection boot manifest.

In this Quick Start you...	Goes deep in
Installed via <code>firefly new</code>	Chapter 1 — Why LaraFly?
Saw <code>#[Service]</code> , <code>#[ConfigProperties]</code> , and <code>#[RestController]</code> register beans with no manual wiring	Chapter 2 — Dependency Injection & Auto-Configuration
Ran <code>firefly:cache</code> and saw routes served from a compiled manifest, not <code>routes/web.php</code>	Chapter 2 — Dependency Injection & Auto-Configuration

When you are ready for the *why* behind all of it, turn the page to Chapter 1.

Try it yourself

1. **Change the default salutation.** Edit `GreetingProperties` 's constructor default from `'Hello'` to something else, re-run `php artisan firefly:cache`, restart `firefly:serve`, and confirm the response changes with no other code touched.
2. **Add a third route.** Give `GreetingController` a `farewell(string $name)` method mapped to `#[GetMapping('/farewells/{name}')]` that returns `['message' => "Goodbye, {$name}!"]` directly — no new service needed yet.
3. **Read the Lumen sample's `README.md`** (`samples/lumen/README.md`) before Chapter 1 — it previews, in prose, everything the rest of this book builds in code.

PART I

Foundations

Why LaraFly?

By the end of this chapter you will understand the problem LaraFly exists to solve, the seven pillars its architecture rests on, and how each one builds on Laravel rather than replacing it — so that when Chapter 2 opens the dependency-injection container, you already know *why* it works the way it does.

The cohesion problem

Picture the first day of a new PHP microservice. Laravel itself answers most of the obvious questions — routing, the ORM, the service container, the queue — and answers them well. But the moment the service needs to be more than a CRUD-and-Blade application, a second layer of questions appears, and Laravel deliberately leaves them to you.

How do you keep business rules out of your Eloquent models and your controllers? Where does a command handler live, and how does it get dispatched? How do you guarantee a database write and the domain event it raises either both happen or neither does? How do you publish that event to other services without hand-rolling a queue job for every one? How do you lock down one specific operation to one specific role, independent of route middleware? How do you know, in production, whether the service is actually healthy?

Every team answers these questions differently — a folder-naming convention here, a base-repository class there, a home-grown "service" layer with no shared shape from one project to the next. Six months later, a second team starts a second service and

makes entirely different choices. Now two codebases share a language and a framework but nothing else: different testing strategies, different error-handling conventions, no shared mental model of how anything is wired together.

Laravel gives PHP a superb foundation. It does not, by itself, give you a shared architecture for the layer above that foundation.

Java developers solved the equivalent problem years ago with Spring Boot: one opinionated, cohesive framework, built *on* the JVM rather than replacing it, that makes the hard architectural decisions once and lets every team share the same answer.

LaraFly brings that same discipline to PHP, built on Laravel 13.

What LaraFly is

LaraFly is not a replacement for Laravel — it is a framework built *on* it, in the same way Spring Boot is built on the JVM and plain Java EE conventions. Every LaraFly application is a real Laravel application: the same Eloquent models, the same service container underneath, the same `artisan` command line, the same HTTP kernel and middleware pipeline. What LaraFly adds is a second, higher-level layer of PHP 8 **attributes** that describe *intent* — "this class is a service," "this method must run in a transaction," "this command handler requires the `ADMIN` role" — and a boot-time engine that reads those attributes once, compiles them into a manifest, and wires the resulting application deterministically.

You already saw a small piece of this in the Quick Start: `#[Service]` on a plain class was enough to make it injectable, with no manual binding. That single attribute is the smallest example of a pattern that repeats, at increasing depth, across the rest of this book: **declare what a class is, and let the framework work out what to do about it.**

⬆ LARAVEL PARITY

Nothing about LaraFly asks you to abandon what you already know. A `#[Service]` bean still ends up as an ordinary object the Laravel container resolves; a `#[RestController]` 's route still runs through Laravel's own HTTP kernel and middleware; `#[Transactional]` (Chapter 8) still calls Laravel's own `DB::beginTransaction() / commit() / rollBack()` underneath. LaraFly's attributes describe *what to wire*; Laravel still does the wiring's actual work.

Seven pillars

LaraFly's architecture rests on seven ideas. Each one is a real, independent package in the framework's monorepo — none of them requires the others — and each is the subject of one or more later chapters.

1. Attribute-driven DI and auto-configuration

`#[Service]`, `#[Repository]`, `#[Component]`, and `#[Configuration] / #[Bean]` mark a class or a factory method as a managed part of the application. A **component scan** — a one-time reflection pass — discovers every one of them and compiles the result into a cached PHP manifest. At runtime, nothing is scanned again: the container is built from that frozen manifest, which is what makes LaraFly safe and fast under long-running workers, not just classic PHP-FPM. This is the whole subject of Chapter 2.

2. `#[Transactional]` — declarative transaction demarcation

A method carrying `#[Transactional]` gets a generated proxy, built at compile time, that opens a real Laravel database transaction before the method runs and commits or rolls it back afterward — using manual `DB::beginTransaction() / commit() / rollBack()` rather than `DB::transaction($closure)`, specifically so a caught

exception can be committed-and-rethrown when your own rollback rules say it should be, instead of always rolling back. You write the business method; the proxy carries the transaction boundary.

3. CQRS — commands, queries, and a bus

Writes and reads travel two separate paths. A `#[CommandHandler]` handles one write intent; a `#[QueryHandler]` answers one read; both are dispatched through a bus rather than called directly. A `#[Transactional]` command handler gets its transaction semantics for free from the same proxy pillar 2 describes — CQRS adds no interception logic of its own.

4. Event-driven architecture, with a genuine outbox

An aggregate raises domain events describing what happened; after a commit, those events can be projected in-process or bridged out to `firefly/eda`'s broker-backed `EventPublisher` — in-memory for tests, a Laravel queue for async delivery, or a real message broker (Kafka, RabbitMQ, or Postgres `LISTEN / NOTIFY`) for cross-service delivery. The Postgres adapter in particular writes the outbox row in the *same* database transaction as the aggregate itself, so an event can never be recorded without the write it describes actually having committed.

5. Security

A Spring-Security-6-shaped principal model — an immutable `Authentication`, a `SecurityContext`, `GrantedAuthority` — backs a deny-by-default `HttpSecurity` URL configuration and method-level guards like `#[PreAuthorize]`, enforced at the CQRS bus and at controller dispatch, independent of route middleware.

6. Actuator — a production management surface

`/actuator/health`, `/actuator/info`, and friends, modelled directly on Spring Boot Actuator: a `HealthIndicator` SPI, liveness/readiness probe groups, and a route-registration pass that mounts these endpoints with zero code changes to your own application — configuration alone decides what is exposed.

7. Zero-reflection boot

Every one of the previous six pillars is discovered by reflection **exactly once** — at `firefly:cache` time, not at request time. The compiled manifests that scan produces are what a running application actually boots from. This is the thread that ties the whole framework together, and it is where Chapter 2 begins.

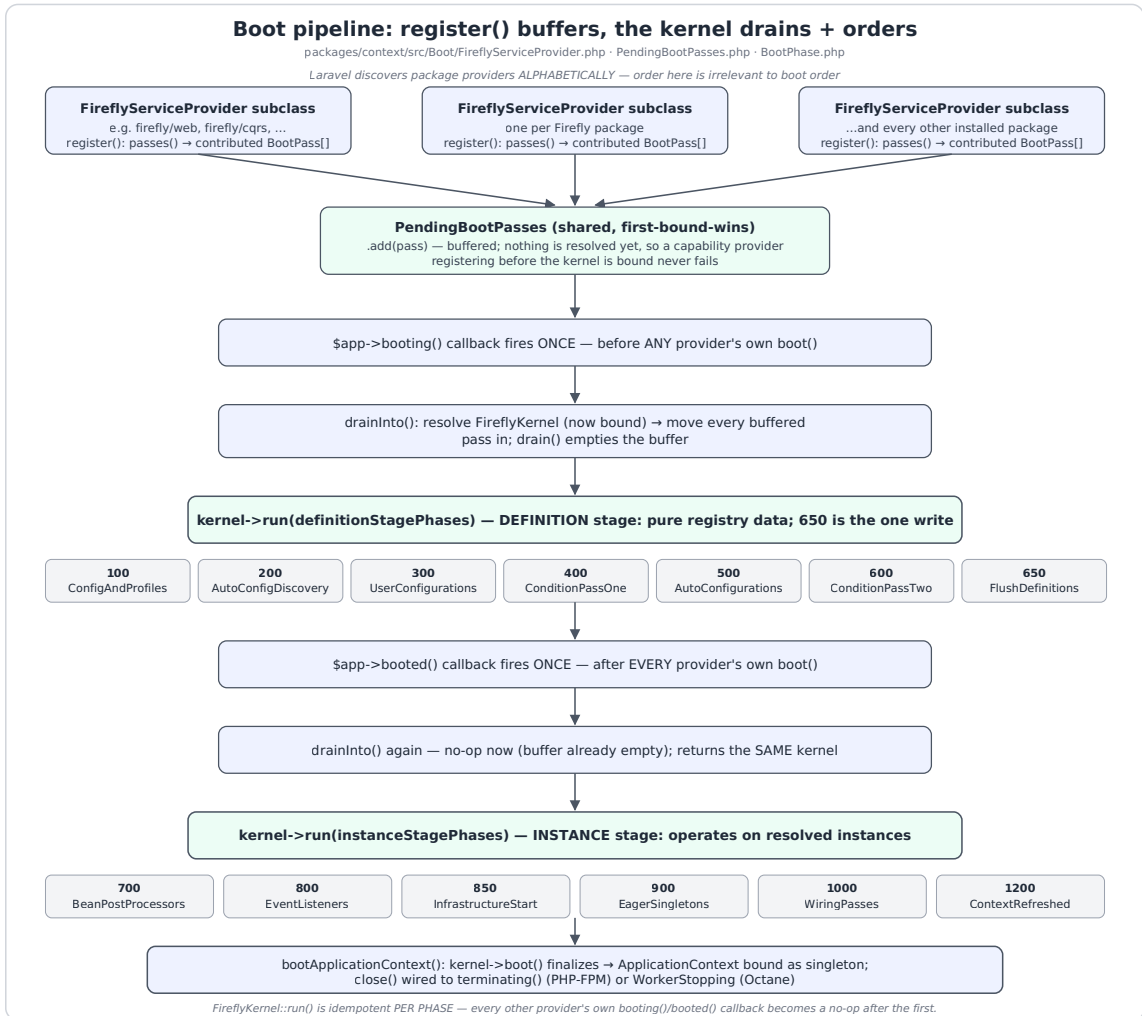


Figure 1.1 — Every Firefly package contributes a boot pass; the kernel alone decides the order they run in.

Why build Lumen

The rest of this book teaches these seven pillars by building one real application, **Lumen**, a digital-wallet-and-ledger service — the same one you glimpsed at the end of the Quick Start. A wallet can be opened, deposited to, withdrawn from, and transferred

between other wallets, and it protects exactly one rule above all others: **the balance never goes negative**. That single rule is small enough to hold in your head completely, which leaves your attention free for how each pillar is built, rather than for what the domain means.

Every listing you will read from here on is not illustrative — it is copied from the real `samples/lumen` package that ships in the framework's monorepo, and it lints, compiles, and passes its own test suite exactly as printed.

 NOTE

Lumen already exists, fully built, in `samples/lumen` — this book does not modify it. Instead, each chapter *explains* one more slice of it, in the order that makes the underlying ideas clearest, so that by the time you have read the whole book you understand every line of a real, production-shaped service.

✓ What you learned

Pillar	What it gives you	Where it's covered
Attribute-driven DI & auto-configuration	Stereotyped beans, constructor injection, a compiled manifest	Chapter 2
<code>#[Transactional]</code>	Declarative transaction boundaries via a generated proxy	Later chapters
CQRS	A command/query bus separating writes from reads	Later chapters
EDA + genuine outbox	Domain events bridged to in-memory, queue, or broker delivery	Later chapters
Security	Deny-by-default HTTP security + method-level guards	Later chapters
Actuator	A production health/info/management surface	Later chapters
Zero-reflection boot	One compile step, then a frozen manifest at runtime	Chapter 2

Try it yourself

1. **Read the pitch in one sitting.** Skim `samples/lumen/README.md` end to end. For each bullet point under "What the sample shows," name which of the seven pillars above it belongs to.
2. **Find the seam.** Open `skeleton/config/firefly.php` again and locate `scan.paths`. Every one of the seven pillars ultimately depends on that one array being correct — write, in your own words, what would happen if it pointed at the wrong namespace.
3. **Compare architectures.** If you have built a service in plain Laravel before, list three architectural decisions you made by convention (where commands live, how you handled transactions, how you secured one sensitive endpoint). You will revisit each one, built the LaraFly way, in a later chapter.

Dependency Injection & Auto-Configuration

By the end of this chapter you will know all four stereotype attributes, how constructor injection resolves dependencies, how to disambiguate between several implementations of one interface with `#[Primary]` and `#[Qualifier]`, how to control list ordering with `#[Order]`, how to inject configuration and expressions with `#[Value]`, what the three component scopes mean, and — tying all of it together — exactly what happens between `php artisan firefly:cache` and your application serving its first request.

NEW TERM: CONTAINER

A **dependency-injection container** is an object that knows how to build other objects, and hands you a fully-constructed instance instead of you calling `new` and wiring up its dependencies by hand. LaraFly's container is `Illuminate\Container\Container` — the very same container a plain Laravel application already uses — with one addition: PHP 8 attributes tell it *what* to build, so you rarely call `$container->bind(...)` yourself.

Stereotypes: declaring a bean

A **bean** is any object the container manages on your behalf — built once (by default) and handed out on request. You mark a class as a bean with a **stereotype** attribute. LaraFly ships four:

```
use Firefly\Container\Attributes\{Component, Configuration, Repository, Service};
```

`#[Component]` is the base attribute; `#[Service]`, `#[Repository]`, and `#[Configuration]` are all specialisations of it — literally, each one `extends` `Firefly\Container\Attributes\Component` in PHP. A component scan finds every one of them with a single reflection call:

```
$reflection->getAttributes(Component::class, ReflectionAttribute::IS_INSTANCEOF);
```

`ReflectionAttribute::IS_INSTANCEOF` is what makes this work: it matches `#[Component]` itself *and* every attribute that extends it, so the scanner never has to enumerate stereotypes by name. This is also why `firefly/web`'s `#[RestController]` — the attribute you met on `GreetingController` in the Quick Start — participates in the very same scan: it, too, extends `Component`.

You already met the simplest stereotype in the Quick Start:

```
<?php

declare(strict_types=1);

namespace App;

use Firefly\Container\Attributes\Service;

/**
 * A #[Service] stereotype: auto-registered as a singleton bean and resolved through the
 * container, so its
 * GreetingProperties dependency is autowired.
 */
#[Service]
final class GreetingService
{
    public function __construct(private readonly GreetingProperties $properties) {}
}
```

```

    public function greet(string $name): string
    {
        return sprintf('%s, %s!', $this->properties->salutation, $name);
    }
}

```

`#[Service]` is the stereotype for application/business logic — the LaraFly counterpart of Spring's `@Service`. There is nothing else to write: no `$this->app->bind(GreetingService::class, ...)` anywhere, no factory closure. The attribute is the whole registration.

`#[Repository]` on a hexagonal port

`#[Repository]` marks a persistence-facing bean, and it is where stereotypes start to show their real value: disambiguating between an interface and the class that implements it. `samples/lumen` defines the *port* — the interface the domain and application layers depend on — with no framework attribute on it at all, because an interface is never itself a bean:

```

<?php

declare(strict_types=1);

namespace Lumen\Infrastructure;

use Lumen\Domain\Wallet;

/**
 * The hexagonal PORT for wallet persistence: the domain/application layer depends on this
 * interface only, never on
 * Eloquent or any storage detail. `EloquentWalletRepository` is the sole adapter, auto-
 * bound by the framework's
 * nominal interface auto-binding (Firefly\Container's ComponentScanner/
 * ContainerRegistrar::wireInterfaces()).
 */

```



```

interface WalletRepository
{
    public function save(Wallet $wallet): Wallet;

    public function findById(string $id): ?Wallet;

    /** @return list<Wallet> */
    public function findByIdByOwnerId(string $ownerId): array;
}

```

The *adapter* — the class that actually talks to Eloquent — is what carries

`#[Repository]`:

```

<?php

declare(strict_types=1);

namespace Lumen\Infrastructure;

use Firefly\Container\Attributes\Repository;
use Firefly\Data\Repository\EloquentRepository;
use InvalidArgumentException;
use Lumen\Domain\Wallet;

/**
 * The Eloquent ADAPTER for the `WalletRepository` port, bound to the `Wallet` aggregate
 * model.
 *
 * // ...
 *
 * // ...
 * @extends EloquentRepository<Wallet>
 */
#[Repository]
final class EloquentWalletRepository extends EloquentRepository implements WalletRepository
{
    protected string $model = Wallet::class;
}

```

```

    public function save(object $entity): Wallet
    {
        if (! $entity instanceof Wallet) {
            throw new InvalidArgumentException(sprintf('%s::save() only accepts a %s.',
self::class, Wallet::class));
        }

        return parent::save($entity);
    }

    public function findById(mixed $id): ?Wallet
    {
        $found = parent::findById($id);

        return $found instanceof Wallet ? $found : null;
    }

    /** @return list<Wallet> */
    public function findByIdByOwnerId(string $ownerId): array
    {
        /** @var list<Wallet> $result */
        $result = $this->dispatchQuery('findByIdByOwnerId', [$ownerId]);

        return $result;
    }
}

```

Nothing in `samples/lumen` ever binds `WalletRepository` to `EloquentWalletRepository` by hand. When a bean's constructor asks for `WalletRepository`, the container needs to know which *concrete* class to hand it — and this is **interface auto-binding**: `ContainerRegistrar::wireInterfaces()` looks at every scanned component's implemented interfaces and, for each interface with exactly one implementing component (or exactly one marked `#[Primary]`, covered later in this chapter), binds the interface to it automatically.

📦 LARAVEL PARITY

In a plain Laravel application you would write `$this->app->bind(WalletRepository::class, EloquentWalletRepository::class);` yourself, usually in a service provider's `register()` method. `#[Repository]` plus interface auto-binding is exactly that binding, generated for you from the fact that only one scanned class implements the interface.

`#[Configuration]` and `#[Bean]` factory methods

Not everything you need to inject is a class you own. `#[Configuration]` marks a class as a **source of `#[Bean]` factory methods** — each method's return type becomes the bean's registered type, and its parameters are resolved and injected exactly like a constructor's. `#[Configuration]` is itself a `#[Component]`, so the configuration class is a managed bean too.

The framework's own packages are built out of these, and the shortest of them is the one `firefly/validation` uses to assemble the validator Chapter 4 will put to work:

```
#[Configuration]
#[Order(1000)]
final class ValidationAutoConfiguration
{
    #[Bean]
    #[ConditionalOnMissingBean(ValidationSettings::class)]
    public function validationSettings(Config $config): ValidationSettings
    {
        return ValidationSettings::fromConfig($config);
    }

    #[Bean]
    #[ConditionalOnMissingBean(Validator::class)]
    public function validator(Factory $factory, ValidationSettings $settings): Validator
    {
        return new IlluminateValidator($factory, $settings);
    }
}
```

```
    }
}
```

Read the two `#[Bean]` methods and you have the whole mechanism.

`validationSettings()` declares `ValidationSettings` as its return type, so `ValidationSettings` is the type its bean is registered under. `validator()` declares `Validator`, and its two parameters — Illuminate's own validation `Factory`, and the `ValidationSettings` the method above it produces — are resolved out of the container exactly as a constructor's parameters would be. Return type in, registration key out; parameter types in, injection out.

The two extra attributes belong to Chapter 3 and are named here only so they do not read as magic. `#[Order(1000)]` sorts this class *after* anything your application declares, and `#[ConditionalOnMissingBean]` makes each default step aside the moment an application binds a `Validator` of its own — the back-off rule that turns an ordinary `#[Configuration]` into an **auto-configuration**, which is how every Firefly package ships a working default you are free to replace.

Nothing about the shape changes when the class is yours rather than the framework's: a `#[Configuration]` under your own `app/` is scanned, ordered and wired by the same pass. It is the shape you will see again every time this book introduces a package that hands you a pre-built object rather than a class you construct yourself.

📦 LARAVEL PARITY

`#[Configuration]` + `#[Bean]` is the direct counterpart of a Laravel service provider's `register()` method calling `$this->app->singleton(SomeType::class, fn () => ...)` — except the *type* the closure returns is read from the method's own return-type declaration, so there is nothing to keep in sync by hand.

Constructor injection

NEW TERM: INJECTION

Dependency injection means a bean receives its collaborators through its constructor rather than constructing them itself or fetching them from a global. The container inspects a bean's constructor, resolves each parameter's type, and passes the built (or already-built) instance in.

You have now seen this twice without a full explanation. `OpenWalletHandler`, the command handler that opens a new wallet in `samples/lumen`, is the clearest example — its whole job is to receive the *port*, not the adapter:

```
<?php

declare(strict_types=1);

namespace Lumen\Application\Command;

use Firefly\Cqrs\Attributes\CommandHandler;
use Firefly\Data\Transaction\Attributes\Transactional;
use Lumen\Domain\Wallet;
use Lumen\Infrastructure\WalletRepository;

/**
 * Handles OpenWallet: mints a new wallet id, opens the aggregate, and persists it. The
 * handled command type is
 * inferred from the sole handle() parameter (HandlerScanner param inference) – no explicit
 * #[CommandHandler(...)].
 * // ...
 */
#[CommandHandler]
class OpenWalletHandler
{
    public function __construct(private readonly WalletRepository $wallets) {}
}
```

```

    #[Transactional]
    public function handle(OpenWallet $command): string
    {
        $id = 'wlt-'.bin2hex(random_bytes(8));
        $wallet = Wallet::open($id, $command->ownerId, $command->currency);
        $this->wallets->save($wallet);

        return $id;
    }
}

```

`OpenWalletHandler` 's constructor asks for `WalletRepository` — the *interface*, not `EloquentWalletRepository`. The container resolves that request in three steps: it looks up `WalletRepository` in the compiled manifest, finds the interface auto-bound to `EloquentWalletRepository` (the single scanned implementation), and builds — or reuses — that instance. `OpenWalletHandler` itself never imports `EloquentWalletRepository`, never mentions Eloquent, and would keep compiling unchanged if `samples/lumen` swapped in a different storage adapter tomorrow. That is the entire point of coding against a port: the handler's own source code is proof that it does not know, and does not need to know, what is on the other side of `WalletRepository`.

💡 **TIP: CONSTRUCTOR INJECTION IS THE ONLY KIND**

LaraFly does not support property or setter injection — every dependency a bean needs must be declared as a typed constructor parameter. This is deliberate: a class's constructor signature is a complete, honest list of everything it depends on, readable without opening the class body.

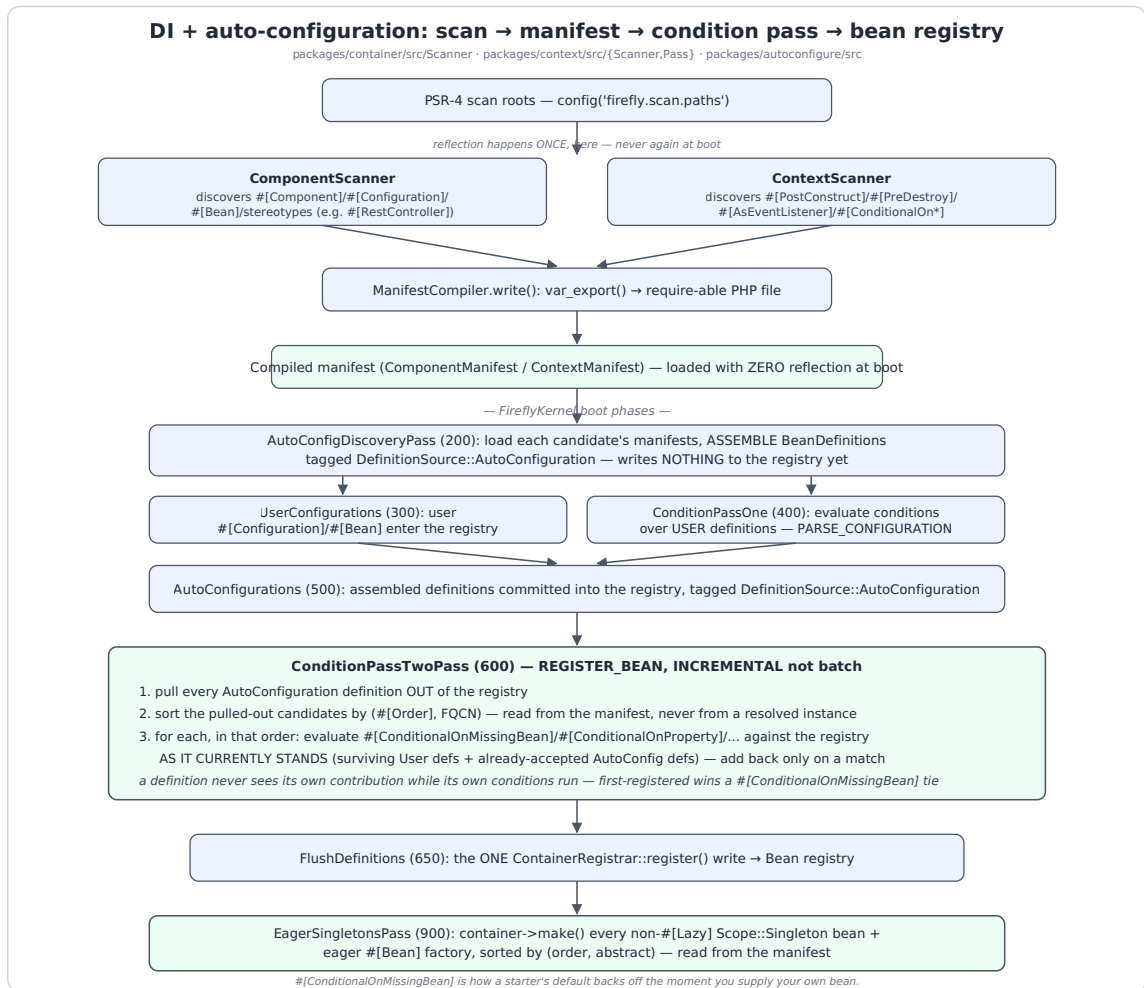


Figure 2.1 — Component scan to compiled manifest to condition pass to bean registry to eager singletons.

Scopes

Every component is a **singleton** by default: the container builds it once, and every resolution after the first returns the same instance. Choose a different lifetime with the `scope` argument any stereotype attribute accepts:

```

use Firefly\Container\Attributes\Service;
use Firefly\Container\Scope;

#[Service(scope: Scope::Transient)] // a new instance every time it is resolved
final class RequestId {}

```

`Scope` is a plain PHP enum with three cases:

```

enum Scope
{
    case Singleton;
    case Transient;
    case Scoped;
}

```

Case	Lifetime
<code>Singleton</code>	one instance for the life of the application — the default
<code>Transient</code>	a new instance on every resolution
<code>Scoped</code>	one instance per Laravel request or scope

Session scope is deliberately absent: it belongs to `firefly/session`, not to the container.

Use `Scope::Transient` for anything that must never be shared — a per-operation correlation id, a mutable builder. Use the default `Scope::Singleton` for everything else, which is almost everything: services, repositories, and configuration DTOs are all naturally shareable, and a singleton is cheaper to resolve.

⚠ WARNING

`Scope::Scoped` binds one instance per Laravel request under classic PHP-FPM. If you deploy under Octane, a scoped bean's lifetime is one *request*, not one *worker* — do not reach for it as a substitute for `Scope::Singleton` just because it sounds safer; the two have genuinely different lifetimes.

`#[Primary]` and `#[Qualifier]` : disambiguating implementations

Interface auto-binding, described earlier in this chapter, has a simple rule for the common case: exactly one scanned implementation of an interface gets bound to it automatically. But real applications often have *more* than one implementation of the same interface — a production adapter and a stub, or two genuinely different strategies.

`#[Primary]` and `#[Qualifier]` are how you tell the container which one you mean:

```
interface Greeter
{
    public function greet(): string;
}
```

With both beans registered, the container resolves the interface three different ways depending on what you ask for:

```
$container->get(Greeter::class);    // EnglishGreeter – the #[Primary] one
$container->getByName('spanish');    // SpanishGreeter – resolved by its bean name
$container->getAll(Greeter::class); // [EnglishGreeter, SpanishGreeter] – every
implementation
```

`#[Primary]` breaks the tie when a constructor asks for the bare interface: the container binds `Greeter::class` to whichever implementation carries `#[Primary]`, exactly the way `wireInterfaces()` bound `WalletRepository` to `EloquentWalletRepository` earlier — except now the choice is explicit instead of being the only option. `#[Qualifier('spanish')]` gives `SpanishGreeter` a *name* you can ask for directly, independent of type resolution, when you specifically need the non-primary one.

WARNING

With zero or with more than one `#[Primary]` implementation of the same interface, the container has no well-defined default — resolving the bare interface becomes ambiguous. Give every interface with multiple implementations exactly one `#[Primary]`, or resolve every implementation by its `#[Qualifier]` name instead.

`#[Order]`

`getAll(Greeter::class)` above returns every implementation — and it returns them **sorted by** `#[Order]`, lower first, exactly like Spring's `@Order` convention:

```
use Firefly\Container\Attributes\Order;
// ...
#[Service]
// ...
#[Order(10)]
```

The default order is `0` when the attribute is omitted, so anything you explicitly order with a positive number sorts after every un-ordered bean, and anything you order negatively sorts before them. `#[Order]` also targets `#[Bean]` factory methods, not just classes — the attribute works identically wherever a list of beans needs a deterministic sequence.

`#[Value]` injection

Sometimes what you need injected is not a bean at all, but a single scalar — a configuration value or a small computed expression. `#[Value]` targets one constructor parameter directly:

```
use Firefly\Container\Attributes\Value;

final class MailerConfig
{
    public function __construct(
        #[Value('${MAIL_HOST:localhost}')] public readonly string $host,
        #[Value('#{25 * 2}')] public readonly int $port,
    ) {}
}
```

Two expression forms are supported. `${NAME:default}` reads environment variable `NAME`, falling back to `default` when it is unset; `#{expr}` evaluates a small, sandboxed expression language — `#{25 * 2}` resolves to the integer `50`. (`firefly/config`, covered in a later chapter, extends the same resolution to read application configuration, not just the environment.)

NOTE

`#[Value]` targets exactly one constructor parameter — it is not a class-level stereotype, and a class carrying only `#[Value]`-annotated parameters still needs a stereotype attribute like `#[Service]` if you want the container to manage it as a bean in its own right.

From component scan to zero-reflection boot

Every stereotype, every `#[Bean]`, every `#[Primary]` / `#[Qualifier]` / `#[Order]` you have just read about is *reflection metadata* — inert until something reads it. That something is `ComponentScanner`, and it runs **exactly once**, when you invoke `firefly:cache`, never again on any request the running application serves.

`ComponentScanner::scan()` walks every PSR-4 root your `config/firefly.php` names under `scan.paths`, reflects each class it finds, and — for every class carrying a `#[Component]` -instanceof attribute — records its stereotype, its scope, whether it is `#[Primary]`, its `#[Order]`, its `#[Qualifier]` name, the interfaces it implements, and (for a `#[Configuration]` class) the list of its `#[Bean]` methods. The result is a plain array of `ComponentDescriptor`s — no objects, no closures, nothing that cannot be serialized.

`ManifestCompiler` takes that array and writes it to disk with nothing more exotic than PHP's own `var_export()`:

```
"<?php\n\ndeclare(strict_types=1);\n\n// Generated by firefly/container. Do not edit.\n\nreturn "\n\n    .var_export($rows, true)\n\n    .";\n";
```

The result — `bootstrap/cache/firefly/component.php` in your project — is an ordinary, `require`-able PHP file: a literal array, with no reflection, no attribute parsing, and no filesystem walk needed to read it back. `ContainerRegistrar::register()` loads that file and does the actual container wiring described throughout this chapter — binding each component under its own class, wiring interfaces (including the `#[Primary]` /sole-implementation rule), and registering every `#[Bean]` factory's output under its return type.

`php artisan firefly:cache` is what drives this whole pipeline for your application:

```
php artisan firefly:cache
```

```
firefly:cache - wrote 14 manifest(s) + 0 proxy(ies) to /path/to/my-app/bootstrap/cache/  
firefly
```

You already ran this once, indirectly — `firefly new` calls it for you in its `post-create-project-cmd` script, which is why the Quick Start's application booted correctly without you ever running the command by hand. From here on, any time you add or change a `#[Service]`, `#[Repository]`, `#[Configuration]`, or any other Firefly attribute, re-run `firefly:cache` to recompile the manifest; `php artisan firefly:clear` deletes the compiled cache and falls back to the slower, reflection-based development scan.

 TIP: FIREFLY:CACHE IS NOT JUST FOR CONTAINERS

The same command also compiles the manifests for every other pillar Chapter 1 introduced — routes, transactional proxies, CQRS handlers, event listeners, security rules, and more. Dependency injection is simply the first and most foundational one; the same "reflect once, freeze into a manifest, boot from the frozen copy" idea repeats for every one of them.

✓ What you learned

Concept	What it does
<code>#[Component]</code> / <code>#[Service]</code> / <code>#[Repository]</code> / <code>#[Configuration]</code>	Mark a class as a managed bean; the first three specialise the base attribute
Constructor injection	The container resolves and passes each typed constructor parameter automatically
<code>Scope::Singleton</code> / <code>Transient</code> / <code>Scoped</code>	One shared instance (default), a fresh instance per resolution, or one per request
<code>#[Primary]</code>	Breaks ties when several beans implement the same interface
<code>#[Qualifier]</code>	Names a specific bean for lookup independent of type resolution
<code>#[Order]</code>	Sorts a <code>getAll()</code> list of implementations, lower first
<code>#[Bean]</code> on a <code>#[Configuration]</code> method	Registers the method's return value under its return type
<code>#[Value]</code>	Injects a resolved environment value or expression into one constructor parameter
<code>firefly:cache</code>	Runs the component scan once and compiles it to a zero-reflection manifest

Try it yourself

1. **Add a second implementation.** Give `Lumen\Infrastructure\WalletRepository` a second, in-memory implementation of your own (in a scratch project — do not modify the shipped `samples/lumen` package) and mark it `#[Repository]` without `#[Primary]`. Run `firefly:cache` and observe what happens to interface auto-binding now that two implementations exist.
2. **Trace a constructor.** Open `OpenWalletHandler` again and, without looking anything up, write down every bean the container must resolve — directly or transitively — to build one instance of it.
3. **Read the compiled manifest.** After running `php artisan firefly:cache` in a skeleton app, open `bootstrap/cache/firefly/component.php` and find the entry for `App\GreetingService`. Match each array key back to a `ComponentDescriptor` field this chapter described.

Configuration, Profiles & Secrets

By the end of this chapter you will know how LaraFly's `Config` wraps Laravel's own configuration repository with fail-fast typed accessors, how `#[ConfigProperties]` binds a whole config subtree onto a typed, injectable DTO, how `#[Value]` resolves an individual scalar once `firefly/config` is installed, how active profiles are resolved and how a bean can be gated to one of them, and — closing the loop with Chapter 2 — exactly how a `#[ConfigProperties]` DTO travels from source attribute to a compiled, zero-reflection manifest.

NEW TERM: CONFIGURATION SUBTREE

A **subtree** is everything nested under one dotted key in Laravel's config array —

`mail.host`, `mail.port`, and `mail.tls` are all part of the `mail` subtree.

`#[ConfigProperties]` binds one whole subtree onto one DTO in a single step, instead of reading each key by hand.

Configuration is still Laravel's configuration

LaraFly does not replace Laravel's configuration system — it rides on top of it. Every `config/*.php` file, every `env('...')` call, every `.env` variable you already know from a plain Laravel application works exactly the same way in a LaraFly one.

`firefly/config`'s job is narrower and more specific: it gives you a **typed, fail-fast accessor** over that same repository, a way to **bind a whole subtree onto a DTO**, and a **config-backed resolver** for the `#[Value]` attribute Chapter 2 introduced.

You already met the smallest possible example of this in the Quick Start. `app/GreetingProperties.php`, generated by `firefly new`, is a real, shipped `#[ConfigProperties]` DTO:

```
<?php

declare(strict_types=1);

namespace App;

use Firefly\Config\Attributes\ConfigProperties;

/**
 * Binds the `greeting.*` configuration subtree onto this readonly DTO and registers it as a
 * container
 * singleton, so it can be constructor-injected wherever GreetingProperties is requested.
 */
#[ConfigProperties('greeting')]
final readonly class GreetingProperties
{
    public function __construct(public string $salutation = 'Hello') {}
}
```

This chapter explains, in depth, everything that one small file quietly does for you.

Typed access with `Config`

Before reaching for a DTO, it helps to see the layer directly underneath it.

`Firefly\Config\Config` wraps `Illuminate\Contracts\Config\Repository` — the very same config repository `config()` reads from in a plain Laravel app — with four typed getters: `string()`, `int()`, `bool()`, and `array()`. Each one is **fail-fast**: a

required key (no default supplied) that is missing, or a value that cannot be coerced to the requested type, throws `ConfigurationException` immediately, instead of handing you `null` or a silently wrong type three call frames later.

```
$config->string('mail.host');           // required – throws ConfigurationException if absent
$config->int('mail.port', 25);           // falls back to 25 when the key is absent
$config->bool('mail.tls', false);
$config->array('mail.recipients', []);
```

`Config` is registered as a container singleton, so it constructor-injects anywhere:

```
use Firefly\Config\Config;
use Firefly\Container\Attributes\Service;

#[Service]
final class MailerBootstrap
{
    public function __construct(private readonly Config $config) {}

    public function host(): string
    {
        return $this->config->string('mail.host', 'localhost');
    }
}
```

NOTE

`Config::has()` and `Config::get()` behave exactly like Laravel's own `Repository::has()` / `Repository::get()` — `Config` adds typed, fail-fast wrappers around them, it does not replace or reinterpret Laravel's own dot-notation resolution.

`string()` / `int()` / `bool()` / `array()` are enough for occasional reads. They do not scale past a handful of scattered calls: each one is an isolated lookup with no shared type information, a typo in a key string surfaces only when that exact line runs, and a service that needs five related settings ends up with five separate `$config->...()` calls sprinkled through its constructor body. `#[ConfigProperties]` solves exactly this.

`#[ConfigProperties]`: binding a subtree onto a typed DTO

`#[ConfigProperties(string $prefix)]` is a class-level attribute (`Firefly\Config\Attributes\ConfigProperties`). It names the dotted config prefix the class binds from; the framework reads that subtree, matches each key to a constructor parameter by name, coerces scalars to the declared type, and registers the resulting instance as a container singleton — so any bean that asks for the DTO type receives the bound configuration, fully typed, with no `config('...')` call anywhere in its own body.

Applying the same pattern to a business setting is just as short as

`GreetingProperties` was. Suppose Lumen wants a configurable daily transfer ceiling:

```
<?php
declare(strict_types=1);
// ...
use Firefly\Config\Attributes\ConfigProperties;

/**
// ...
*/
#[ConfigProperties('wallet')]
final readonly class WalletProperties
```

```
{
    public function __construct(
        public int $dailyTransferLimitMinor = 1_000_000,
        public string $defaultCurrency = 'EUR',
    ) {}
}
```

and the matching config file, `config/wallet.php`:

```
<?php

declare(strict_types=1);

return [
    'daily_transfer_limit_minor' => env('WALLET_DAILY_TRANSFER_LIMIT_MINOR', 1_000_000),
    'default_currency' => env('WALLET_DEFAULT_CURRENCY', 'EUR'),
];
```

`TransferHandler` (Chapter 4 and 6 build the rest of it) can now depend on `WalletProperties` directly:

```
use Firefly\Container\Attributes\Service;
use Lumen\Domain\WalletProperties;

#[Service]
final class TransferLimitGuard
{
    public function __construct(private readonly WalletProperties $properties) {}

    public function isWithinDailyLimit(int $amountMinor): bool
    {
        return $amountMinor <= $this->properties->dailyTransferLimitMinor;
    }
}
```

Nothing here calls `config('wallet.daily_transfer_limit_minor')`.

`WalletProperties` is a typed, self-documenting object — the constructor's own default values (`1_000_000`, `'EUR'`) double as living documentation of what the setting means when nothing overrides it, and a typo in a constructor parameter name is a PHP error at binding time, not a silently-`null` field discovered in production.

📦 LARAVEL PARITY

`#[ConfigProperties]` is LaraFly's counterpart to Spring Boot's `@ConfigurationProperties`. In a plain Laravel application the closest equivalent is writing your own DTO and populating it by hand from `config('wallet')` in a service provider's `register()` method; `#[ConfigProperties]` is that same idea with the boilerplate removed — the prefix is the only thing you declare, and the binder does the rest.

How binding actually works: `ReflectionConfigBinder`

The default binder, `Firefly\Config\Binder\ReflectionConfigBinder`, resolves one constructor parameter at a time from the bound subtree array:

```
final class ReflectionConfigBinder implements ConfigBinder
{
    // ...
    private function resolveParameter(string $class, ReflectionParameter $parameter, array
$config): mixed
    {
        $name = $parameter->getName();
        // ...
        if ($value === null) {
            if ($parameter->isDefaultValueAvailable()) {
                return $parameter->getDefaultValue();
            }
            if ($parameter->allowsNull()) {
                return null;
            }
        }
    }
}
```

```

    // ...
  }

  $type = $parameter->getType();
  if (! $type instanceof ReflectionNamedType) {
    return $value;
  }

  return $this->coerce($type, $value);
}
// ...
}

```

Three rules fall out of that method body directly. A missing key with a constructor default silently uses the default — exactly why `WalletProperties`'s constructor spells out `1_000_000` and `'EUR'` rather than leaving them required. A missing key with `no` default and a non-nullable type throws `ConfigurationException` immediately, at binding time — not the first time a handler happens to touch the field. And when a constructor parameter is itself typed as another class (not a scalar) and the subtree holds an array under that key, the binder **recurses**: it calls itself on the nested class and the nested array, so a `#[ConfigProperties]` DTO can compose smaller DTOs without any extra wiring.

Scalar coercion is equally mechanical — `int` / `float` / `bool` fields are coerced from whatever the config value actually is (useful when a value arrived from an environment variable as a string), and `string` accepts anything scalar:

```

final class ReflectionConfigBinder implements ConfigBinder
{
    // ...
    private function coerce(ReflectionNamedType $type, mixed $value): mixed
    {
        if ($type->isBuiltin()) {

```

```

        return match ($type->getName()) {
            'int' => is_numeric($value) ? (int) $value : $value,
            'float' => is_numeric($value) ? (float) $value : $value,
            'bool' => is_bool($value) ? $value : filter_var($value,
FILTER_VALIDATE_BOOLEAN),
            'string' => is_scalar($value) ? (string) $value : $value,
            // ...
        };
    }
    // ...
    $nested = $type->getName();
    if (is_array($value) && class_exists($nested)) {
        // ...
        return $this->bind($nested, $value);
    }
    return $value;
}
}

```

💡 THE BINDER IS A SEAM, NOT THE ONLY IMPLEMENTATION

`ReflectionConfigBinder` implements a one-method interface, `Firefly\Config\Binder\ConfigBinder`. An application (or a later Firefly package) can bind a richer binder over the container — one that understands enums, or value objects with their own validation — without touching a single `#[ConfigProperties]` class. The DTOs you write never know, or need to know, which binder actually populated them.

From attribute to compiled manifest — the same story as Chapter 2

`#[ConfigProperties]` follows the exact "reflect once, freeze into a manifest, boot from the frozen copy" shape Chapter 2 walked through for the container.

`ConfigPropertiesScanner::scan()` walks your PSR-4 roots, reflects every non-abstract class, and records a `ConfigPropertiesDescriptor` — just the class name and the prefix — for each one carrying `#[ConfigProperties]`:

```

foreach ($this->classesIn($prefix, $dir) as $class) {
    $reflection = new ReflectionClass($class);
    if ($reflection->isAbstract() || $reflection->isInterface()) {
        continue;
    }
    $attrs = $reflection->getAttributes(ConfigProperties::class);
    if ($attrs === []) {
        continue;
    }
    // ...
}

```

`ConfigManifestCompiler` writes that list to `bootstrap/cache/firefly/config-properties.php` with the same `var_export()`-based, zero-reflection technique the component manifest used. `php artisan firefly:cache` runs this scan as one step among the twelve scanner/compiler pairs it orchestrates — you never invoke `ConfigPropertiesScanner` yourself.

At boot, `Firefly\Cli\Boot\FireflyCacheServiceProvider` loads that compiled manifest and hands it straight to

`Firefly\Config\Registrar\ConfigRegistrar::register()`, which binds every descriptor's DTO as a container singleton in one pass:

```

foreach ($manifest->properties as $descriptor) {
    // ...
    $class = $descriptor->class;
    $prefix = $descriptor->prefix;
    // ...
    $this->container->singleton($class, static function () use ($binder, $class, $prefix, $config): object {
        // ...
        $subtree = $config->array($prefix, []);
        // ...
        return $binder->bind($class, $subtree);
    });
}

```



```
    });
}
```

This is the entire reason `GreetingProperties` and `WalletProperties` need no service-provider glue anywhere in your application: the compiled manifest already knows every `#[ConfigProperties]` class and its prefix, and `ConfigRegistrar` binds all of them, unconditionally, the moment the cache exists.

⚠ THE CACHE MATTERS HERE TOO

Exactly as Chapter 2 explained for beans, `#[ConfigProperties]` binding is driven from the **compiled** manifest — run `php artisan firefly:cache` after adding or renaming a `#[ConfigProperties]` class, the same way you would after adding a new `#[Service]`.

#[Value] from real configuration

Chapter 2 introduced `#[Value]` on a bare constructor parameter — `#[Value('${MAIL_HOST:localhost}')]` — and mentioned that `firefly/container`'s own resolver only understands environment variables and small expressions. Installing `firefly/config` upgrades that resolution without touching a single `#[Value]` site in your code:

```
use Firefly\Container\Attributes\Value;

final class Mailer
{
    public function __construct(
        #[Value('${mail.host:localhost}')] public readonly string $host,
    ) {}
}
```

`Firefly\Config\Value\ConfigValueResolver` implements the same `ValueResolver` port `firefly/container` defines, but resolves `${key:default}` against **application configuration first**, then the environment, then the literal default:

```
if (preg_match('/^\${{([^\:}]+)(?:{([^\}]+)*)}?\/}', $expression, $m) === 1) {
    $key = $m[1];
    if ($this->config->has($key)) {
        return $this->config->get($key);
    }
    $env = getenv($key);
    if ($env !== false) {
        return $env;
    }

    return array_key_exists(2, $m) ? $m[2] : null;
}
```

`#{expr}` still evaluates a small sandboxed expression, now with a `config('key')` function registered against it, so an expression can read configuration too — `#{config('mail.port') * 2}` is valid once `firefly/config` is installed.

`ConfigRegistrar::register()` binds `ConfigValueResolver` over the container's default:

```
$this->container->instance(ValueResolver::class, new ConfigValueResolver($this->config));
```

`firefly/container`'s own registrar only binds its `DefaultValueResolver` if none is **already bound** — so the config-backed resolver, once `firefly/config` registers it, always wins, and `#{mail.host:localhost}` now reads `config('mail.host')` first, the environment second, and only falls back to the literal `localhost` if neither is set.

LARAVEL PARITY

`#[Value('${mail.host:...}')]` reading application config first is the closest LaraFly equivalent to Spring's `@Value("${...}")` reading `application.yaml`. In a plain Laravel application the corresponding idiom is simply `config('mail.host')` called at the point of use — `#[Value]` gets you the same resolved value, but declared once on the constructor parameter, alongside the rest of the class's dependencies.

Profiles

A **profile** is a named environment — `local`, `testing`, `staging`, `prod` — and LaraFly gives you a small, focused vocabulary for reading which ones are currently active, independent of any one config key.

`Firefly\Config\Profile\ProfileResolver::resolve()` returns a `Profiles` value object, using a fixed precedence: the `FIREFLY_PROFILES_ACTIVE` environment variable (comma-separated, for more than one active profile) if set; otherwise the single Laravel `APP_ENV` value; otherwise the implicit `default` profile.

```
use Firefly\Config\Profile\ProfileResolver;

$profiles = (new ProfileResolver())->resolve();

$profiles->isActive('prod'); // bool
$profiles->all();             // list<string>
$profiles->isEmpty();        // bool
```

```
final readonly class Profiles
{
    // ...
    public function __construct(public array $active) {}
}
```

```

    public function isActive(string $profile): bool
    {
        return in_array($profile, $this->active, true);
    }
    // ...
}

```

LARAVEL PARITY

`FIREFLY_PROFILES_ACTIVE` falling back to `APP_ENV` means every existing Laravel application already has an active profile without adding a single environment variable:

`APP_ENV=production` in your `.env` file is already, today, LaraFly's `prod` profile.

`FIREFLY_PROFILES_ACTIVE` exists for the case Laravel's single `APP_ENV` cannot express on its own — activating more than one profile at a time.

Gating a component to a profile

Knowing which profiles are active becomes genuinely useful once a component's very presence in the container can depend on it. `#[ConditionalOnProfile]` — from `firefly/context`, the boot-engine package Chapter 2 mentioned in passing — gates a `#[Component]` (or a `#[Configuration]` / `#[Bean]`) on one or more profiles:

```

use Firefly\Container\Attributes\Component;
use Firefly\Context\Condition\Attributes\ConditionalOnProfile;

#[Component]
#[ConditionalOnProfile('local', 'testing')]
final class InMemoryMailer {}

```

`InMemoryMailer` only enters the container when `local` or `testing` is among the active profiles — the condition is a simple "any listed profile is active" match, evaluated by the same `ConditionEvaluator` that evaluates `#[ConditionalOnProperty]`,

`#[ConditionalOnClass]`, and `#[ConditionalOnMissingClass]`. On `prod`, the class is never even instantiated: it is filtered out of the `BeanDefinitionRegistry` before the container is built, not merely left unresolved.

📦 LARAVEL PARITY

`#[ConditionalOnProfile]` is the closest LaraFly gets to Spring's `@Profile`. In plain Laravel the nearest equivalent is an `if (app()->environment('local', 'testing'))` guard inside a service provider's `register()` method — `#[ConditionalOnProfile]` moves that same decision onto the class itself, so a component's environment-scoping is visible right next to its definition rather than in a separate provider file.

Keeping secrets out of files

Everything this chapter has shown reads through `env()` at some point —

`env('WALLET_DAILY_TRANSFER_LIMIT_MINOR', ...)` in `config/wallet.php`, `{mail.host:localhost}` falling through to the environment,

`FIREFLY_PROFILES_ACTIVE` itself. This is deliberate, and it is the same discipline any Laravel application should already follow: configuration that is safe to commit — ports, feature flags, default currencies — belongs in a `config/*.php` file; anything that grants access — database passwords, API keys, JWT signing secrets — belongs in `.env`, which is never committed, and is supplied differently (real files locally, injected secrets in CI/CD and production) per environment.

⚠️ NEVER COMMIT SECRETS

`#[ConfigProperties]` and `#[Value]` both ultimately read whatever your `config/*.php` files return — if a config file hardcodes a real credential instead of calling `env(...)`, that credential ships in your repository history. Keep every secret behind an `env()` call with no committed default, and inject the real value at deploy time.

✓ What you learned

Concept	What it does
<code>Config</code>	A typed, fail-fast accessor (<code>string()</code> / <code>int()</code> / <code>bool()</code> / <code>array()</code>) over Laravel's own config repository
<code>#[ConfigProperties(prefix)]</code>	Binds a config subtree onto a plain DTO, registered as a container singleton
<code>ReflectionConfigBinder</code>	The default <code>ConfigBinder</code> : matches constructor parameters by name, coerces scalars, recurses into nested DTOs
<code>ConfigPropertiesScanner</code> / <code>ConfigManifestCompiler</code>	Reflects <code>#[ConfigProperties]</code> classes once, at <code>firefly:cache</code> time, into a zero-reflection manifest
<code>ConfigRegistrar</code>	Binds every manifest entry as a singleton, and installs <code>ConfigValueResolver</code> over the container's default
<code>#[Value('\${key:default}')]</code>	With <code>firefly/config</code> installed, resolves against application config first, then the environment, then the default
<code>ProfileResolver</code> / <code>Profiles</code>	Resolves active profiles from <code>FIREFLY_PROFILES_ACTIVE</code> , falling back to <code>APP_ENV</code> , falling back to <code>default</code>
<code>#[ConditionalOnProfile(...)]</code>	Gates a component's presence in the container on one or more active profiles

Try it yourself

1. **Add `WalletProperties` for real.** Create `config/wallet.php` and `Lumen\Domain\WalletProperties` exactly as shown in this chapter (in a scratch copy of the project — do not modify the shipped `samples/lumen` package), constructor-inject it into a new service, and confirm `php artisan firefly:cache` followed by reading the injected DTO reflects a value you set via `WALLET_DAILY_TRANSFER_LIMIT_MINOR` in `.env`.
2. **Force a missing-key failure.** Remove the constructor default from one of `WalletProperties`'s parameters, delete the corresponding key from `config/wallet.php`, and observe `ConfigurationException` at the point the DTO is resolved — then restore the default and confirm the exception disappears.
3. **Gate a bean to a profile.** Add a `#[Component]` `#[ConditionalOnProfile('testing')]` class to your scratch project, run `php artisan firefly:cache` with `FIREFLY_PROFILES_ACTIVE` unset, and confirm the class never appears in the compiled component manifest; then re-run the cache with `FIREFLY_PROFILES_ACTIVE=testing` and confirm it does.

Your First HTTP API

By the end of this chapter you will know how `#[RestController]` and `#[RequestMapping]` turn a plain class into a routed HTTP surface, how each of the five parameter-binding attributes pulls one piece of a request into a typed method argument, how `#[Valid]` gates a request body on Bean-Validation constraints before your handler ever runs, and how every error — a validation failure, a missing wallet, a denied authorization check — renders as the same predictable `application/problem+json` shape. This chapter closes Part I by turning Lumen's wallet domain into a real, validated REST API.

NEW TERM: DISPATCH

Dispatch is the act of routing an incoming HTTP request to the right method on the right controller, with the right arguments already extracted and converted. In LaraFly this happens through native Laravel routes — there is no separate LaraFly router competing with Laravel's own.

`#[RestController]` and `#[RequestMapping]`

You already know, from Chapter 2, that `#[RestController]` is itself a `#[Component]` stereotype — it `extends Firefly\Container\Attributes\Component`, so a controller is discovered by the very same component scan as a `#[Service]` or a `#[Repository]`, and gets full constructor dependency injection for free. What Chapter 2 deferred is the *other* pass a `#[RestController]` participates in: a separate reflection walk, `RouteScanner`, that reads a different set of attributes on the same class to build the routing table.

Here is Lumen's whole wallet API — the running example for the rest of this chapter — in one real, shipped listing:

```
<?php

declare(strict_types=1);

namespace Lumen\Web;

use Firefly\CQRS\Command\CommandBus;
use Firefly\CQRS\Query\QueryBus;
use Firefly\Kernel\Exception\Business\ResourceNotFoundException;
use Firefly\Validation\Valid;
use Firefly\Web\Attributes.GetMapping;
use Firefly\Web\Attributes.PathVariable;
use Firefly\Web\Attributes.PostMapping;
use Firefly\Web\Attributes.RequestBody;
use Firefly\Web\Attributes.RequestMapping;
use Firefly\Web\Attributes.RestController;
use Lumen\Application\Command\Deposit;
use Lumen\Application\Command\OpenWallet;
use Lumen\Application\Command\Transfer;
use Lumen\Application\Command\Withdraw;
use Lumen\Application\Query\GetBalance;
use Lumen\Application\Query\GetLedger;
use Lumen\Domain\Currency;
use Lumen\Domain\LedgerEntry;
use Lumen\Web\Dto\AmountRequest;
use Lumen\Web\Dto\OpenWalletRequest;
use Lumen\Web\Dto\TransferRequest;

/**
 * The wallet REST surface: thin HTTP-onto-CQRS mapping, no business logic. Every method
 * builds a
 * command/query, dispatches it through the bus, and shapes the return array – the same rule
 * the sample's
 * CQRS handlers already enforce for the domain layer. A domain/business fault raised by the
```

```

bus (e.g. an
* unknown wallet's ResourceNotFoundException, an overdraft's ConflictException, or a denied
#[PreAuthorize]
* on withdraw) surfaces as CommandProcessingException/QueryProcessingException and renders
RFC-7807
* problem-details via the framework's global ProblemDetailsRenderer – no local
#[ExceptionHandler] needed.
*/
#[RestController]
#[RequestMapping('/api/v1/wallets')]
final class WalletController
{
    public function __construct(
        private readonly CommandBus $commands,
        private readonly QueryBus $queries,
    ) {}

    /** @return array{wallet_id: string} */
    #[PostMapping(status: 201)]
    public function open(#[Valid] #[RequestBody] OpenWalletRequest $body): array
    {
        /** @var string $id */
        $id = $this->commands->send(new OpenWallet($body->owner_id, Currency::from($body->cu
rrency)));

        return ['wallet_id' => $id];
    }

    /** @return array{wallet_id: string, balance_minor: int} */
    #[PostMapping('/{id}/deposit')]
    public function deposit(#[PathVariable] string $id, #[Valid] #[RequestBody] AmountReques
t $body): array
    {
        /** @var int $balance */
        $balance = $this->commands->send(new Deposit($id, $body->amount_minor));

        return ['wallet_id' => $id, 'balance_minor' => $balance];
    }
}

```

```

    /**
     * Debits `amount_minor` from the wallet. Guarded upstream at the bus: WithdrawHandler
     carries
     * #[PreAuthorize("hasRole('ADMIN') or hasRole('WALLET_OWNER')")] (S6), enforced by
     * SecurityCommandAuthorizer BEFORE the handler runs. With no principal in the
     SecurityContextHolder
     * the bus refuses the command with an AuthenticationException that renders as a 401
     problem-details
     * response (authenticate first); with a principal that is signed in but carries neither
     role it is an
     * AuthorizationException rendered as 403 – this endpoint is secured, not broken.
     *
     * @return array{wallet_id: string, balance_minor: int}
     */
    #[PostMapping("/{id}/withdraw")]
    public function withdraw(#[PathVariable] string $id, #[Valid] #[RequestBody] AmountReque
st $body): array
    {
        /** @var int $balance */
        $balance = $this->commands->send(new Withdraw($id, $body->amount_minor));

        return ['wallet_id' => $id, 'balance_minor' => $balance];
    }

    /**
     * Moves `amount_minor` from the source wallet to the destination wallet as one atomic
     unit of work
     * (TransferHandler's #[Transactional] boundary), then reads back both fresh balances
     for the response.
     *
     * @return array{source_wallet_id: string, destination_wallet_id: string,
     source_balance_minor: int, destination_balance_minor: int}
     */
    #[PostMapping("/transfers")]
    public function transfer(#[Valid] #[RequestBody] TransferRequest $body): array
    {
        $this->commands->send(new Transfer($body->source_wallet_id, $body->destination_walle
t_id, $body->amount_minor));
    }

```

```

    /** @var int $sourceBalance */
    $sourceBalance = $this->queries->ask(new GetBalance($body->source_wallet_id));

    /** @var int $destinationBalance */
    $destinationBalance = $this->queries->ask(new GetBalance($body->destination_wallet_id));

    return [
        'source_wallet_id' => $body->source_wallet_id,
        'destination_wallet_id' => $body->destination_wallet_id,
        'source_balance_minor' => $sourceBalance,
        'destination_balance_minor' => $destinationBalance,
    ];
}

/** @return array{wallet_id: string, balance_minor: int} */
#[GetMapping('/{id}/balance')]
public function balance([PathVariable] string $id): array
{
    /** @var int|null $balance */
    $balance = $this->queries->ask(new GetBalance($id));

    if ($balance === null) {
        throw new ResourceNotFoundException("Wallet {$id} not found");
    }

    return ['wallet_id' => $id, 'balance_minor' => $balance];
}

/** @return array{wallet_id: string, entries: list<LedgerEntry>} */
#[GetMapping('/{id}/ledger')]
public function ledger([PathVariable] string $id): array
{
    /** @var list<LedgerEntry> $entries */
    $entries = $this->queries->ask(new GetLedger($id));

    return ['wallet_id' => $id, 'entries' => $entries];
}
}

```

Four things about this class are worth pointing out before anything else. First, `#[RequestMapping('/api/v1/wallets')]` is class-level and prepends its path to every method mapping below it — `open()` maps `POST /api/v1/wallets`, `deposit()` maps `POST /api/v1/wallets/{id}/deposit`. Second, the controller's constructor asks for `CommandBus` and `QueryBus` — the same constructor-injection story Chapter 2 covered, applied to a controller instead of a service. Third, every method is a thin translation: build a command or query DTO, hand it to the bus, shape the array the framework will serialize as JSON. There is no business logic here — no overdraft check, no currency comparison — because that logic lives in the `Wallet` aggregate and its command handlers, which Chapters 5 and 6 build. Fourth, the docblock is not decoration: it tells you, in the same file where the routes live, exactly which failures each endpoint can produce and how they will be rendered — the subject of this chapter's final section.

LARAVEL PARITY

`#[RestController]` + `#[RequestMapping]` + the five verb-mapping attributes below are LaraFly's counterpart to Spring's `@RestController` + `@RequestMapping` + `@GetMapping` / `@PostMapping`. In a plain Laravel application the corresponding artifact is a `routes/api.php` entry pointing at a controller method; here, the route *is* the attribute, declared on the method it dispatches to, so there is exactly one place to look for both.

The verb-mapping attributes

Each method carries exactly one verb attribute, and all five implement the same `Mapping` marker interface, so `RouteScanner` discovers them polymorphically — with `ReflectionAttribute::IS_INSTANCEOF`, the same trick Chapter 2 used for stereotypes — rather than needing separate scanner code per verb:

Attribute	HTTP method
<code>#[GetMapping]</code>	GET
<code>#[PostMapping]</code>	POST
<code>#[PutMapping]</code>	PUT
<code>#[PatchMapping]</code>	PATCH
<code>#[DeleteMapping]</code>	DELETE

Every one of the five accepts the same three constructor arguments — a relative `path` (joined to the class-level base), an integer `status` (the default response status: 200 for the read verbs, and `status: 201` on `WalletController::open()` above, since opening a wallet is a creation), and an optional route `name`:

```
#[Attribute(Attribute::TARGET_METHOD)]
final class GetMapping implements Mapping
{
    public function __construct(
        public readonly string $path = '',
        public readonly int $status = 200,
        public readonly ?string $name = null,
    ) {}

    public function method(): string
    {
        return 'GET';
    }
    // ...
}
```

`PostMapping`, `PutMapping`, `PatchMapping`, and `DeleteMapping` are the same shape, each simply returning its own HTTP method string from `method()`.

Parameter binding: five attributes, two conventions

Chapter 2 explained how the container resolves a *bean*'s constructor. A controller *method*'s parameters are resolved differently — by `ArgumentResolver`, which reads one binding attribute per parameter and pulls exactly one piece of the request into it:

Attribute	Source	Notes
<code>#[PathVariable(name?)]</code>	route segment	always required; defaults to the parameter's own name
<code>#[QueryParam(name?, default?, required?)]</code>	query string	not required by default
<code>#[RequestBody]</code>	request body	decoded via the negotiated <code>MessageConverter</code> ; combine with <code>#[Valid]</code> to gate on validation
<code>#[RequestHeader(name?, default?)]</code>	HTTP header	not required
<code>#[UploadedFile(name?)]</code>	<code>multipart/form-data</code> file	a framework-neutral <code>Firefly\Web\Http\UploadedFile</code> (filename/mimeType/size + lazy <code>contents()</code> / <code>store()</code>)

`WalletController::balance()`, in the full listing above, shows the simplest of these: `#[GetMapping('/{id}/balance')]` paired with `#[PathVariable] string $id`, which reads the `{id}` route segment straight into a typed string parameter.

`open()` and `deposit()` show the combination you will use on every write endpoint — `#[Valid] #[RequestBody]` stacked on the same parameter, covered in full in the next section.

Two more parameter shapes need **no attribute at all**, and both are genuine conventions rather than omissions. A parameter typed as a class with no binding attribute — a service interface, for instance — resolves as a plain **container service**, exactly as `$container->make($type)` would; this lets a method reach for an extra collaborator without adding it to the controller's own constructor. A **scalar** parameter with no binding attribute defaults to a query parameter keyed by its own name, required unless the parameter itself has a default or is nullable.

Path and query values are coerced to the parameter's declared scalar type — `int`, `float`, or `bool` — and an uncoercible value, or a missing required one, raises `InvalidRequestException` (HTTP 400, with codes `MISSING_PARAMETER` / `TYPE_CONVERSION_ERROR`), rendered through the same problem-details pipeline this chapter ends with.

The request DTOs

Every write endpoint in `WalletController` accepts a small, real Pydantic-style — here, plain-PHP — DTO. All three live in `samples/lumen/src/Web/Dto/` and are pure constructor-promoted classes with per-property validation constraints:

```
<?php
declare(strict_types=1);

namespace Lumen\Web\Dto;
```



```

use Firefly\Validation\Constraint\CurrencyCode;
use Firefly\Validation\Constraint\NotBlank;

/**
 * Validated request body for `POST /api/v1/wallets`: an owner id and an ISO-4217 currency
 * code. `[Valid]` on the
 * controller parameter runs BeanValidator against these constraints BEFORE the DTO is
 * hydrated (ArgumentResolver),
 * so an invalid body never reaches WalletController::open() – it renders as a 422 RFC-7807
 * payload instead.
 */
final class OpenWalletRequest
{
    public function __construct(
        #[NotBlank]
        public readonly string $owner_id,
        #[NotBlank]
        #[CurrencyCode]
        public readonly string $currency,
    ) {}
}

```

```

<?php

declare(strict_types=1);

namespace Lumen\Web\Dto;

use Firefly\Validation\Constraint\Positive;

/**
 * Validated request body shared by `POST /api/v1/wallets/{id}/deposit` and `.../withdraw`:
 * a strictly positive
 * amount in minor units. Both Deposit and Withdraw take an existing wallet id from the path
 * and only the amount
 * from the body, so one DTO covers both endpoints.
 */

```

```

final class AmountRequest
{
    public function __construct(
        #[Positive]
        public readonly int $amount_minor,
    ) {}
}

```

```

<?php

declare(strict_types=1);

namespace Lumen\Web\Dto;

use Firefly\Validation\Constraint\NotBlank;
use Firefly\Validation\Constraint\Positive;

/**
 * Validated request body for `POST /api/v1/wallets/transfers`: both wallet ids travel in
 the body (a transfer is
 * not scoped under a single wallet's path) plus a strictly positive amount in minor units.
 */
final class TransferRequest
{
    public function __construct(
        #[NotBlank]
        public readonly string $source_wallet_id,
        #[NotBlank]
        public readonly string $destination_wallet_id,
        #[Positive]
        public readonly int $amount_minor,
    ) {}
}

```

`AmountRequest` doing double duty for both deposit and withdraw is a deliberate design choice, not laziness — both operations move a positive amount of money in minor units, and the wallet id already lives in the path, not the body, so a single shared DTO keeps the two endpoints' contracts identical without a shared base class.

`#[Valid]` → `BeanValidator` → 422

`#[Valid]`, from `firefly/validation`, is what turns those per-property attributes from inert metadata into an enforced gate. Stacking it onto a `#[RequestBody]` parameter — as every write method on `WalletController` does — tells the dispatcher: validate the *raw decoded body* against this DTO class's compiled constraint rules **before** constructing the DTO at all.

```
#[Attribute(Attribute::TARGET_PARAMETER | Attribute::TARGET_PROPERTY)]
final class Valid
{
    // ...
    public function __construct(public readonly ?string $each = null) {}
}
```

Two targets, and both matter. On a **parameter** it is the gate just described. On a **property** of a DTO it means *cascade* — validate the object that member holds, under dot-prefixed keys, so a class-typed `shipTo` reports its failures as `shipTo.postcode`, and a member typed `array` or `iterable` cascades into every element and reports `lines[0].sku`. That second case is the one that needs the constructor parameter the listing keeps: `#[Valid(each: Line::class)]` names the element class of a list member whose docblock does not already say it. Without either the attribute argument or a `@var list<Line>` / `@param list<Line> $lines` tag for `ConstraintScanner` to

read, a `#[Valid]` list is **refused at scan time** rather than silently skipped — an unvalidated collection that looks validated is exactly the failure this refusal exists to prevent.

The validation itself runs through `Firefly\Validation\Constraint\BeanValidator`, which delegates to a `Validator` port (`IlluminateValidator` by default) using rules a `ConstraintManifest` compiled ahead of time from each constraint attribute's `toRules()` method:

```
#[Attribute(Attribute::TARGET_PARAMETER | Attribute::TARGET_PROPERTY)]
final class NotBlank implements Constraint, HasMessage
{
    use MessageElement;

    public function __construct(public readonly ?string $message = null) {}

    public function toRules(): array
    {
        // required rejects null/''/[]; string constrains the type; regex:/\S/ requires a
        non-whitespace
        // char so an all-whitespace string is rejected (JSR-380 @NotBlank's trimmed-length
        > 0).
        return ['required', 'string', 'regex:/\S/'];
    }

    public function message(): ?string
    {
        return ConstraintMessage::resolve($this->message, 'must not be blank');
    }
}
```

```
#[Attribute(Attribute::TARGET_PARAMETER | Attribute::TARGET_PROPERTY)]
final class Positive implements Constraint, HasMessage
{

```

```

    use MessageElement;

    public function __construct(public readonly ?string $message = null) {}

    public function toRules(): array
    {
        return ['numeric', 'gt:0'];
    }

    public function message(): ?string
    {
        return ConstraintMessage::resolve($this->message, 'must be greater than 0');
    }
}

```

Every constraint attribute reduces, this way, to one or more plain Illuminate validation rules — `#[NotBlank]` rejects `null`, `''`, `[]`, and an all-whitespace string; `#[Positive]` requires a numeric value strictly greater than zero; `#[CurrencyCode]` (used on `OpenWalletRequest::$currency`) delegates to a dedicated `Currency` rule object rather than a rule string. On failure, `BeanValidator` throws the kernel's `ValidationException` — the same exception type this chapter's final section maps to HTTP 422 — carrying one `FieldError` per failed property. On success, the DTO is constructed from the **raw** decoded body, not from a validator-produced subset, so a property with no constraint attribute at all is never silently dropped.

NOTE

Constraints validate the raw request array, not the constructed DTO. This ordering matters: `#[Valid]` guarantees your handler method only ever receives a DTO whose constrained fields have already passed every check — there is no path where `WalletController::open()` runs with an `OpenWalletRequest` whose `owner_id` is blank.

LARAVEL PARITY

`#[Valid]` `#[RequestBody]` is LaraFly's counterpart to Spring's `@Valid` `@RequestBody` pair. In a plain Laravel application the closest equivalent is a Form Request class with a `rules()` method; `firefly/validation`'s per-property attributes move those same rules onto the DTO's own fields, so the constraint and the field it constrains are declared in the same place.

`firefly/validation` ships sixteen financial-domain constraints beyond the general-purpose ones (`#[NotBlank]`, `#[NotEmpty]`, `#[NotNull]`, `#[Size]`, `#[Min]` / `#[Max]`, `#[Pattern]`, `#[Email]`, and more): `#[Iban]`, `#[Bic]`, `#[Swift]`, `#[Isin]`, `#[Cusip]`, `#[RoutingNumber]`, `#[Luhn]`, `#[CurrencyCode]`, `#[CountryCode]`, `#[LanguageTag]`, `#[UuidValue]`, `#[Phone]`, `#[PostalCode]`, `#[Percentage]`, `#[Money]`, and `#[DecimalScale]` — plus a `#[Rules]` escape hatch for any raw Illuminate rule string or `ValidationRule` object you need that no dedicated attribute covers yet.

Content negotiation

Once a handler returns, something still has to decide the wire format. `firefly/web` ships one `MessageConverter` today — `JsonMessageConverter`, matching `application/json` and any `+json` suffix type — chosen from the request's `Accept` header (parsed for q-values, with a header-order tiebreak) and falling back to JSON when nothing matches or the header is absent. Request bodies are read the same way, keyed off `Content-Type`. `MessageConverterRegistry` is an ordinary, guarded container binding, so an application can register additional converters — XML support is a deliberately deferred seam, not a shipped feature.

Errors that clients can trust

A well-designed API never leaks a raw stack trace, and never returns a bare 500 for something a caller could have avoided. `firefly/kernel`'s exception hierarchy is the backbone that makes this automatic: every framework exception extends `FireflyException`, which carries a stable `errorCode()`, an `httpStatus()`, a `category()`, and a `severity()` — so raising the right *kind* of exception is the only HTTP concern your handler code ever has to think about.

Exception	Code	Status
<code>Business\ResourceNotFoundException</code>	<code>RESOURCE_NOT_FOUND</code>	404
<code>Business\ConflictException</code>	<code>CONFLICT</code>	409
<code>Business\ValidationException</code>	<code>VALIDATION_ERROR</code>	422
<code>Security\AuthenticationException</code>	<code>AUTHENTICATION_FAILED</code>	401
<code>Security\AuthorizationException</code>	<code>ACCESS_DENIED</code>	403
<code>Infrastructure\ServiceUnavailableException</code>	<code>SERVICE_UNAVAILABLE</code>	503

`WalletController::balance()` raises the first of these directly, exactly the way any handler should:

```
use Firefly\Kernel\Exception\Business\ResourceNotFoundException;

throw new ResourceNotFoundException("Wallet {$id} not found");
```

Chapter 6's `Wallet` aggregate raises `ConflictException` for an overdraft or a currency mismatch, and `WithdrawHandler`'s `#[PreAuthorize]` (Chapter 6 briefly, security in full later) can deny a request before the handler body ever runs, wrapping an `AuthorizationException`. All three reach the same renderer.

The RFC-7807 renderer

`Firefly\Web\Exception\ProblemDetailsRenderer` is where every one of those exceptions actually becomes bytes on the wire:

```
final class ProblemDetailsRenderer
{
    // ...

    public function render(Throwable $e, Request $request): Response
    {
        // An absent settings object means the SAFE answer, not the open one – see the
        constructor.

        $disclose = $this->settings instanceof ErrorPageSettings && $this->settings->disclose;

        $typeUri = $this->settings instanceof ErrorPageSettings ? $this->settings->typeUri :
        ProblemType::BLANK;

        $correlationId = CorrelationIdFilter::of($request);
        $reference = TraceContext::referenceFor($request);
        $exception = ProblemMapper::toFireflyException($e, $disclose, $reference);

        // ...

        $problem = ErrorResponse::fromException(
            $exception,
            // ...

            instance: ProblemMapper::instanceFor($request),
```



```

        traceId: $reference,
        timestamp: (new DateTimeImmutable)->format(DateTimeInterface::ATOM),
        correlationId: $correlationId,
        // ...
        type: ProblemType::of($exception->errorCode(), $typeUri),
    );
    // ...
    $payload = $problem->toArray();

    $headers = [
        'Content-Type' => 'application/problem+json',
        CorrelationIdFilter::HEADER => $correlationId,
    ];

    $traceHeader = TraceContext::header();
    $traceId = TraceContext::traceId($request);
    if ($traceHeader !== '' && $traceId !== null) {
        $headers[$traceHeader] = $traceId;
    }

    // ...
    return new Response(
        $this->encode($payload, $reference),
        $exception->httpStatus(),
        $headers,
    );
}

// ...
}

```

`application/problem+json` is set here, on every response this renderer produces — that media type *is* the RFC-7807 contract, and a client is entitled to switch on it. Beside it travel two identifiers, and they are two on purpose. `X-Correlation-Id` carries the id `CorrelationIdFilter` minted or read at the edge of the request, and the document repeats that same value as `correlationId`, so a caller can match the body it is

holding to its own request log. The document's `traceId` is whatever `TraceContext::referenceFor()` answers — the request's W3C trace id when tracing is on and this request carries a valid one, and the correlation id when it does not — so the reference a person quotes is never empty, and switching tracing on is the only thing that changes which of the two it holds. Where there really is a trace id, it is echoed on a header of its own as well (`X-Trace-Id` by default; an empty header name turns that echo off). Both ids are passed *through* `ErrorResponse` rather than written onto the array afterwards, because the DTO's member list is what the published OpenAPI component is generated from — a member appended here would be one no generated client decodes.

The cut above the `return` hides the rest of the header work: the headers an `HttpExceptionInterface` already carries are copied onto the response — a `405`'s `Allow` among them — and a `503` additionally gains `Retry-After: 5`.

`$this->encode()` is the body, and it is total on purpose. A byte that is not valid UTF-8 — a driver message quoting a latin-1 column, a request header echoed into an extension member — is substituted rather than raised. Anything `json_encode` still refuses, such as an `INF` an application put in an extension at the throw site, falls back to a minimal document; so does anything an encoded object's *own* code throws, because encoding an object calls that code — a `jsonSerialize()`, an Eloquent accessor — and what it raises never becomes a `JsonException` at all, which is why the fallback catches `Throwable` and not just the JSON one. That fallback keeps every member the shape above defines — the status, the title, the code, the category, the severity, the sentence and the reference ids — because each of them is a string this class wrote and none of them can be what `json_encode` refused. It keeps the field errors too, names and sentences intact — a `422` that still says `category: validation` while carrying no `errors` member would send a generated client down the field-error arm with nothing to render — and it keeps every extension

member's *name*, rendering a value it cannot carry as the name of its type (`"balance": "App\Models\Balance"`) rather than deleting it, which is how `firefly/actuator`'s config endpoint has always answered the same question. Only two things actually go: an unreadable value, replaced by its type, and each field error's `rejectedValue`, which is dropped outright because that member's whole contract is *this is the value you sent* and the word `float` is not something anybody sent.

And the degradation is on the record. The throwable the fallback caught is an arbitrary application exception, so it is logged — through the optional `Psr\Log\LoggerInterface` the provider hands this renderer, with the document's own `traceId` in the context so the log line and the body a caller is holding join up, and inside a `try / catch` of its own so a log channel that is down too cannot throw out of the error handler. An error-handling subsystem that fails invisibly is the thing this whole surface exists to remove, and a renderer that quietly swallowed an exception while publishing a document nobody could tell from a healthy one would have been exactly that. A renderer that threw here, meanwhile, would fail while handling the failure, and the caller would receive no document at all.

What `render()` deliberately does **not** decide is which `FireflyException` an arbitrary throwable becomes. That rule lives one class along, in `Firefly\Web\Error\ProblemMapper`, because the HTML error page of Chapter 10 needs the identical answer and two copies of it would eventually hand a browser and an API client different codes for the same failure:

```
public static function toFireflyException(Throwable $e, bool $disclose = true, ?string
$reference = null): FireflyException
{
    return match (true) {
        $e instanceof FireflyException => $e,
        self::isExecutionTimeLimit($e) => self::executionTimeExceeded($e, $reference),
        $e instanceof MethodNotAllowedHttpException => self::methodNotAllowed($e),
```

```

    $e instanceof HttpExceptionInterface => new FireflyException(
        self::httpMessage($e),
        self::errorCode($e->getStatusCode()),
        $e->getStatusCode(),
        ErrorCategory::Framework,
        ErrorSeverity::Warning,
        $e,
    ),
    default => new FireflyException(
        $disclose && $e->getMessage() !== '' ? $e->getMessage() :
self::opaque($reference),
        'INTERNAL_ERROR',
        500,
        ErrorCategory::Internal,
        ErrorSeverity::Error,
        $e,
    ),
    };
}

```

Five arms covering four cases — the `405` has an arm of its own only so the verbs it permits can be lifted out of the router's sentence and into an `allowed` member and the `Allow` header — and every one of them ends back at the same `ErrorResponse::fromException(...)->toArray()` call, so the payload's shape never depends on which arm produced it.

A `FireflyException` — or one of its typed subclasses, like `ResourceNotFoundException` — is returned untouched and renders at its own `httpStatus()`, because its message was written by your application *for* the client; that is the whole point of the taxonomy. PHP's own execution-time limit is named separately and answers `503` with a `Retry-After` header, because "the server stopped this request after N seconds" is something a caller can act on and a bare `500` is not. A Laravel/Symfony HTTP exception — a URL matching no route, a verb a route does not accept — keeps its **real status code**, so an unmatched route still answers `404` and

never a misleading `500`; only the router's own wording is replaced with a sentence written for a person, and a `405`'s permitted verbs move into an `allowed` member and the `Allow` header where a client can read them without parsing English. Anything else is an accident, and `$disclose` — `firefly.web.problem.disclose`, default `false` — decides whether its message may be published at all: with it off the body carries a fixed sentence naming the same reference the document's `traceId` carries, and the real message stays in the log, which is where a `QueryException`'s SQL and its bindings belong.

Requesting a wallet that was never opened renders like this. Note `instance`: RFC 9457 §3.1.5 makes it a URI **reference**, and a relative reference resolves against the document's base URI — so the bare `api/v1/wallets/wlt-999` that `$request->path()` answers, served from `/api/v1/wallets/wlt-999`, would identify `/api/v1/api/v1/wallets/wlt-999`. LaraFly publishes the root-relative form, `ProblemMapper::instanceFor()`'s one job, and a client may compare it to the path it asked for. Note `type` too: RFC 9457 §3.1.1 says an absent `type` is `about:blank`, and LaraFly writes it out rather than leaving the reader to know that — point `firefly.web.problem.type-uri` at a base URI instead and the stable `code` derives a real, openable one.

```
{
  "status": 404,
  "title": "Not Found",
  "code": "RESOURCE_NOT_FOUND",
  "category": "business",
  "severity": "warning",
  "detail": "Wallet wlt-999 not found",
  "type": "about:blank",
  "instance": "/api/v1/wallets/wlt-999",
  "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
  "correlationId": "0f7c9b2e-6b43-4f5e-9a1d-2c8e5f0a91b7",
```

```

    "timestamp": "2026-06-07T10:30:00+00:00"
  }

```

A failed `#[Valid]` check on `POST /api/v1/wallets` — an empty `owner_id` — additionally carries an `errors` array, one entry per failed **constraint**:

```

{
  "status": 422,
  "title": "Unprocessable Entity",
  "code": "VALIDATION_ERROR",
  "category": "validation",
  "severity": "warning",
  "detail": "Validation failed",
  "type": "about:blank",
  "instance": "/api/v1/wallets",
  "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
  "correlationId": "0f7c9b2e-6b43-4f5e-9a1d-2c8e5f0a91b7",
  "timestamp": "2026-06-07T10:30:00+00:00",
  "errors": [
    {
      "field": "owner_id", "message": "must not be blank", "constraint": "NotBlank", "rejectedValue": ""
    }
  ]
}

```

Four members, and the pairing is deliberate: `message` is the sentence — Bean Validation's own wording for the constraint, or whatever `#[NotBlank(message: '...')]` supplied — and `constraint` is the **attribute's short name**, which is the member a client branches on when the sentence is written for a person and may be translated or reworded. `rejectedValue` is what the client actually sent, echoed back so a form can highlight the field without guessing. One entry appears per failed constraint, not per failed rule: `#[NotBlank]` compiles to three Laravel rules and still reports once.

A withdraw attempt is refused twice over, and the two refusals are **not the same status**. With no authenticated principal at all, `WithdrawHandler`'s `#[PreAuthorize]` — enforced at the bus, before the handler body runs — answers `401`: *authenticate first*.

```
{
  "status": 401,
  "title": "Unauthorized",
  "code": "AUTHENTICATION_FAILED",
  "category": "security",
  "severity": "warning",
  "detail": "Processing command [Lumen\\Application\\Command\\Withdraw] failed:
Authentication is required.",
  "type": "about:blank",
  "instance": "/api/v1/wallets/wlt-1/withdraw",
  "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
  "correlationId": "0f7c9b2e-6b43-4f5e-9a1d-2c8e5f0a91b7",
  "timestamp": "2026-06-07T10:30:00+00:00"
}
```

`403` is reserved for a principal who **is** signed in and still lacks `ROLE_ADMIN` or `ROLE_WALLET_OWNER` — *signing in would not help* — and that document names the authorities the expression wanted:

```
{
  "status": 403,
  "title": "Forbidden",
  "code": "ACCESS_DENIED",
  "category": "security",
  "severity": "warning",
  "detail": "Processing command [Lumen\\Application\\Command\\Withdraw] failed: You do not
have permission to do this.",
  "type": "about:blank",
  "instance": "/api/v1/wallets/wlt-1/withdraw",
}
```

```

    "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
    "correlationId": "0f7c9b2e-6b43-4f5e-9a1d-2c8e5f0a91b7",
    "timestamp": "2026-06-07T10:30:00+00:00",
    "requiredAuthorities": ["ROLE_ADMIN", "ROLE_WALLET_OWNER"]
}

```

Both are produced with no code change to `WalletController::withdraw()` at all: the bus wraps whichever security exception the evaluator raised in a `CommandProcessingException` that copies the cause's code, status and category, so the wire carries the cause's identity rather than a generic bus error. Chapter 10 returns to this pair — and to the two shipped tests that pin it — in full.

Handling an exception yourself

Before an exception reaches the generic renderer, LaraFly gives your application a chance to answer it directly. `#[ExceptionHandler(SomeException::class)]` marks a method as the renderer for one exception class (or any subclass) — either declared **on the controller that threw it**, or on a dedicated `#[ControllerAdvice]` bean for a **global** handler shared across every controller:

```

use Firefly\Web\Attributes\ControllerAdvice;
use Firefly\Web\Attributes\ExceptionHandler;
use Lumen\Domain\Exception\WalletFrozenException;

#[ControllerAdvice]
final class WalletExceptionAdvice
{
    #[ExceptionHandler(WalletFrozenException::class)]
    public function onFrozen(WalletFrozenException $e): array
    {
        return ['code' => 'WALLET_FROZEN', 'message' => $e->getMessage()];
    }
}

```


A controller-local handler always beats a global `#[ControllerAdvice]` one for the same exception; within whichever scope wins, the most-specific matching class in the hierarchy is chosen. A matched handler's return value is content-negotiated exactly like any other controller return, but rendered at the exception's own `httpStatus()` rather than the route's default. If nothing matches at either scope, the exception falls through to `ProblemDetailsRenderer` — so every unhandled error, however it started, still ends up as `application/problem+json`, never an uncaught framework error page.

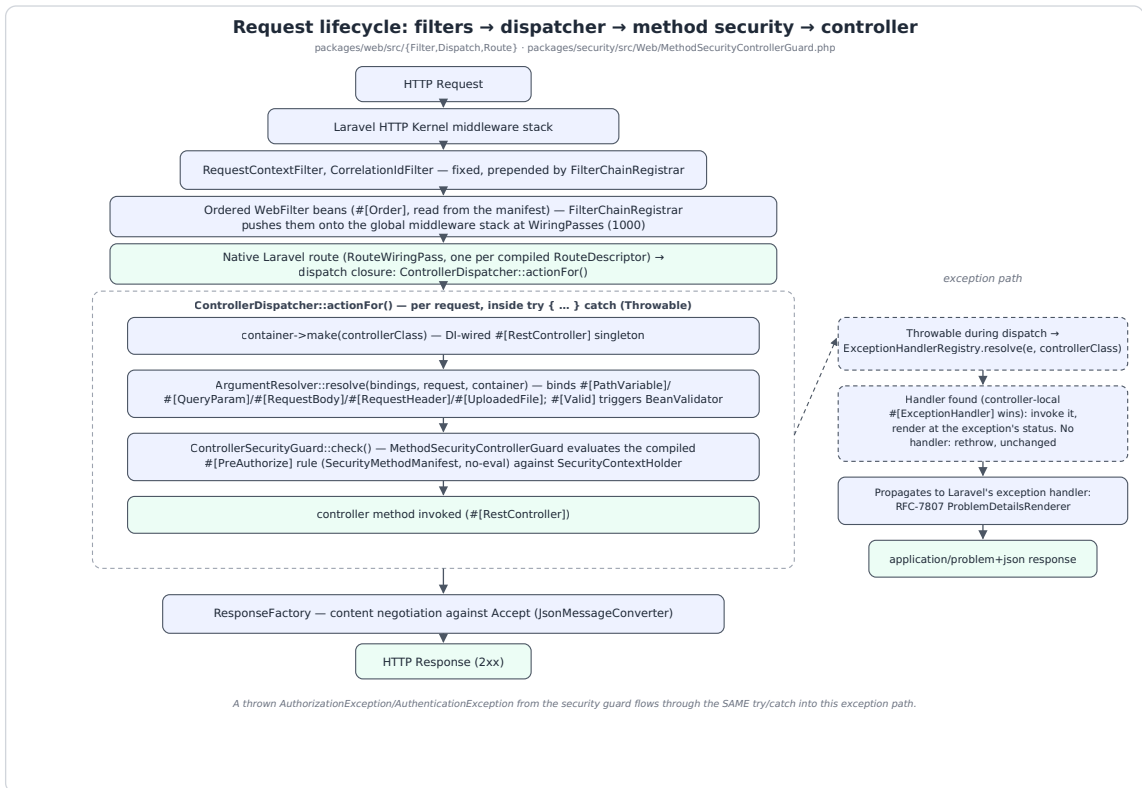


Figure 4.1 — A request travels through the web filter chain, route dispatch, argument binding and validation, and — on failure at any point — the same RFC-7807 renderer, before your handler method ever runs.

The compiled `RouteManifest`

You have now seen the same shape twice — Chapter 2's component manifest, this chapter's config-properties manifest — and routing follows it a third time.

`RouteScanner` is the one reflection site in `packages/web/src`: it walks `#[RestController]` classes, and for each verb-mapped method compiles a `RouteDescriptor` — the HTTP method, the joined path, the controller and method name, the default status, an optional route name, and a pure-array parameter-binding plan. `RouteManifestCompiler::var_export()` s the result to `bootstrap/cache/firefly/routes.php`; `RouteManifest::load()` reads it back with no reflection at all.

At boot, `RouteWiringPass` registers one **native Laravel route** per descriptor and hands each one a dispatch closure through `ControllerDispatcher` — so route listing, URL generation, and Laravel's own `route:cache` all apply to `WalletController`'s routes exactly as they would to any hand-written `routes/api.php` entry, because that is genuinely what they are, once `firefly:cache` has run.

✓ What you built

Part I is complete. Lumen now **boots** (Chapter 1), is **wired** (Chapter 2's constructor injection with zero glue code), is **configured** (Chapter 3's typed, profile-aware settings), and **serves** — a validated REST API at `/api/v1/wallets` with:

Piece	What it does
<code>#[RestController] + #[RequestMapping]</code>	Registers the controller as a bean and fixes its URL prefix
<code>#[GetMapping] / #[PostMapping] /...</code>	One verb attribute per method, compiled into a <code>RouteDescriptor</code>
<code>#[PathVariable] / #[QueryParam] / #[RequestBody] / #[RequestHeader] / #[UploadedFile]</code>	Five ways to pull one piece of the request into a typed parameter
<code>#[Valid]</code>	Gates a <code>#[RequestBody]</code> DTO on its compiled constraint rules, throwing <code>ValidationException</code> on failure
<code>FireflyException</code> hierarchy	Typed exceptions carrying their own error code, HTTP status, category, and severity
<code>ProblemDetailsRenderer</code>	Turns any exception into a consistent <code>application/problem+json</code> response
<code>#[ExceptionHandler] / #[ControllerAdvice]</code>	An escape hatch to render a specific exception yourself, locally or globally

Chapter 5 replaces nothing in `WalletController` — the controller you just read stays exactly as it is — and instead builds the repository layer the command handlers behind `CommandBus` / `QueryBus` persist through.

Try it yourself

1. **Add a `DELETE` endpoint.** In a scratch project (not the shipped `samples/lumen` package), add `#[DeleteMapping('/{id}', status: 204)]` to a new

`WalletController` method that sends a hypothetical `CloseWallet` command and returns `null`. Confirm the route compiles and that a `204` with no body is what a matching request produces.

2. **Trace a validation failure end to end.** Send `POST /api/v1/wallets` with `{"owner_id": "", "currency": "XYZ"}` against a running Lumen instance and read the resulting `application/problem+json` body. Match each `errors` entry back to the constraint attribute on `OpenWalletRequest` that produced it.
3. **Write a `#[ControllerAdvice]`.** Add a global exception handler for `Firefly\Kernel\Exception\Business\ConflictException` that returns a custom `{"code": "...", "hint": "..."} shape instead of the default problem-details payload, and confirm it applies to every controller in your scratch project, not just one.`

Documenting the API: OpenAPI 3.1 from the Manifests

By the end of this chapter you will know how `firefly/openapi` turns the `RouteManifest` and `ConstraintManifest` Chapter 4 just built into a valid OpenAPI 3.1 document with **no annotation dialect of its own** — how `firefly:openapi` makes that document a build artifact a CI job can diff, how each binding `kind` in the compiled route plan becomes a Parameter Object or a Request Body, how a `#[NotBlank]` or a `#[Positive]` you already wrote becomes a `pattern` or an `exclusiveMinimum`, why every DTO is registered once and reached by `$ref` rather than inlined, why every operation carries the same `problem+json` error component Chapter 4's renderer produces, and why a `#[Controller]` HTML route is left out of the document by default. It closes on the browser console the package serves over that document — three styles, of which the default is the **official Swagger UI served from your own origin**, with no npm step and no CDN request, and only one of the three ever talks to a third party.

NEW TERM: SPECIFICATION EXTENSION

OpenAPI 3.1 lets a document carry members whose names begin with `x-`, called **specification extensions**. Conforming tools must ignore them, so an extension can record something the standard vocabulary cannot express without making the document invalid. `firefly/openapi` uses exactly one, `x-firefly-constraints`, and this chapter shows what lands in it and why nothing is ever dropped silently instead.

Nothing to annotate

Every OpenAPI toolchain in PHP that predates this one asks you to write the document twice: once as code the server runs, and once as annotations, attributes or a YAML file that *describe* the code the server runs. The two drift the first time somebody adds a field in a hurry, and the drift is invisible — the document still validates, it just no longer matches the server.

`firefly/openapi` does not have that failure mode available to it, because it does not have a second source. Chapter 4 ended with `RouteManifest`: the compiled table of every `RouteDescriptor` the dispatcher serves from, carrying the verb, the path, the declared status, the route name, the controller class and method, and the per-parameter *binding plan*. Chapter 4 also introduced `ConstraintManifest`: the compiled rule list `BeanValidator` runs on a `#[Valid]` body. Those two artifacts, plus `firefly/kernel`'s `ErrorResponse`, are the entire input:

```
composer require fireflyframework/laraflly
```

The root library includes the OpenAPI component; generated LaraFly applications already have it. Boot the app and `GET /openapi.json` is served; `GET /openapi` renders a reference console over it. Nothing was annotated, and nothing can drift, because every fact in the document is read from the same compiled artifact the dispatcher reads.

`firefly:openapi`, and routes that are not attribute routes

The document is also a file you can commit:

```
php artisan firefly:openapi --output=docs/openapi.json
# writes the file, prints a summary line
php artisan firefly:openapi > openapi.json           # writes the raw document to stdout
```

The command exists so the document can be a **build artifact** rather than only a live endpoint. Committing the generated file is what lets a CI job diff it and fail a pull request that changed the public API without saying so, and what lets a front-end repository regenerate its typed client from a checked-in spec without booting the PHP application at all. It is also the only way to get a document out of a deployment that keeps `firefly.openapi.enabled` off in production.

Stdout mode is written with Symfony's `OUTPUT_RAW` flag, and that detail matters more than it looks: console output normally goes through Symfony's formatter, which treats `<...>` as markup. A `description` mentioning a generic type — anything carrying an angle bracket that reached the document from a config value — would either be swallowed or would throw on an unknown tag. The point of stdout mode is to pipe straight into a client generator, so the bytes must be exactly the bytes of the document. That is also why the confirmation line is printed **only** in `--output` mode, where stdout is not the document.

The HTTP routes — the spec, the console, and the console's own assets — are mounted natively on the Illuminate `Router` from a `BootPass`, not declared with `#[GetMapping]`:

```
final class OpenApiRouteRegistrar implements BootPass
{
    public function phase(): BootPhase
    {
        return BootPhase::WiringPasses;
    }
}
```

```

    public function order(): int
    {
        return 60;
    }

    public function run(BootContext $context): void
    {
        $container = $context->container;

        /** @var OpenApiProperties $properties */
        $properties = $container->make(OpenApiProperties::class);

        if (! $properties->enabled) {
            return;
        }

        /** @var Router $router */
        $router = $container->make('router');

        $router->get($properties->specPath, static fn (): mixed => $container-
>make(OpenApiSpecAction::class)())
            ->name('firefly.openapi.spec');

        if (! $properties->viewerEnabled) {
            return;
        }

        $router->get($properties->viewerPath, static fn (): mixed => $container-
>make(OpenApiViewerAction::class)())
            ->name('firefly.openapi.viewer');

        // The official Swagger UI files, served from this application's own origin rather
        than a CDN. Mounted
        // under the viewer path so moving the console moves its assets with it, and
        constrained to a single
        // path segment so the route cannot express a traversal in the first place -
        SwaggerAssets whitelists
        // and realpath-checks the name as well.
        $router->get($properties->viewerPath.'/assets/{file}', static fn (string $file):

```



```

mixed => $container->make(SwaggerAssetAction::class)($file))
    ->where('file', '[A-Za-z0-9._-]+')
    ->name('firefly.openapi.assets');
}
}

```

This is the same shape — and the same `BootPass` idiom — Chapter 11 will show you for the actuator's own routes, and it is chosen for two independent reasons. First, **an attribute route cannot be configurable**: `#[GetMapping('/openapi.json')]` bakes its literal into a compiled `RouteDescriptor` at `firefly:cache` time, so an operator could never move the spec off a path that collides with one of their own, and could never take it off a public surface without deleting the package. Second, an attribute route would enter the application's `RouteManifest` — and the generator reads that manifest, so **the package would document itself**. Registering natively keeps both problems out of existence: the paths come from config at boot, and they never appear in the spec they serve. Note that the assets route is mounted *under* the viewer path, so moving the console moves its stylesheet and scripts with it.

`firefly.openapi.enabled` (default `true`) is enforced *here*, on the routes, rather than on the beans — the generator and its collaborators are inert without routes, so gating the routes is the whole of the switch. Turning it off leaves every one of those paths genuinely unrouted, so they 404 through the router's own `NotFoundHttpException`, which Chapter 4's `ProblemDetailsRenderer` then renders as a proper `404` problem-details body rather than a `500`.

❗ THE ACTIONS ARE RESOLVED PER REQUEST, INSIDE THE CLOSURE

Building an `OpenApiSpecAction` at boot and capturing it in the route would freeze one `OpenApiGenerator` into the route for the process's lifetime — exactly the shape that breaks under Octane, where a later request's container is a different sandbox. `$container->make(...)` inside the closure is the rule, here and in every other framework package that mounts a native route.

From `RouteManifest` to Operation Objects

`OpenApiGenerator::generate()` is the entry point — it memoises a single private `build()` pass (`$this->document ??= $this->build()`), and `toJson()` wraps it for a file or an HTTP body. That pass walks the manifest once, skipping excluded routes, and hands each survivor to `OperationFactory`. Everything about the result is deterministic on purpose:

```
final class OpenApiGenerator
{
    // ...
    private const array VERB_ORDER = ['get', 'put', 'post', 'delete', 'options', 'head',
    'patch', 'trace'];
    // ...
    private function sortVerbs(array $item): array
    {
        $sorted = [];

        foreach (self::VERB_ORDER as $verb) {
            if (array_key_exists($verb, $item)) {
                $sorted[$verb] = $item[$verb];
                unset($item[$verb]);
            }
        }
    }
}
```

```

    ksort($item);

    return [...$sorted, ...$item];
}
// ...
}

```

Paths are sorted, verbs within a path are sorted into the canonical order the OpenAPI specification itself lists them in, and the schema registry sorts components by name. This is not tidiness for its own sake. Route discovery order depends on filesystem iteration, so an *unsorted* document would reshuffle itself between machines and turn every regeneration into an unreviewable diff — which is exactly what makes teams stop committing the generated file, which is what makes it go stale.

Inside an operation, the binding plan does the real work. `OperationFactory` dispatches on the same `kind` discriminator `ArgumentResolver` uses at request time:

```

    public function create(RouteDescriptor $route, string $operationId, SchemaRegistry
$registry, ApiDocs $docs): array
    {
        $doc = $docs->operation($route);

        $parameters = [];
        $body = null;
        $files = [];
        $validated = false;
        $rejectable = false;
        $notFound = false;

        foreach ($route->bindings as $binding) {
            // Claimed by a resolver: bound from somewhere other than the request, so
            neither a parameter nor
            // a reason for a 400 – the same first question ArgumentResolver::resolveOne()
            asks.
            if ($this->resolvers->resolverFor($binding) !== null) {

```

```

        continue;
    }

    switch ($binding['kind']) {
        case 'path':
            $parameters[] = $this->parameter($binding, 'path', true, $doc->parameters[$binding['key']] ?? null);
            $rejectable = $rejectable || $this->coercible($binding);
            $notFound = $notFound || isset($binding['pattern']);
            break;
        case 'query':
            $parameters[] = $this->parameter($binding, 'query',
            $binding['required'], $doc->parameters[$binding['key']] ?? null);
            $rejectable = $rejectable || $binding['required'] || $this->coercible($binding);
            break;
        case 'header':
            $parameters[] = $this->parameter($binding, 'header',
            $binding['required'], $doc->parameters[$binding['key']] ?? null);
            $rejectable = $rejectable || $binding['required'] || $this->coercible($binding);
            break;
        case 'file':
            $files[] = $binding;
            $rejectable = true;
            break;
        case 'body':
            $body = $binding;
            $rejectable = true;
            $validated = $validated || ($binding['valid'] && $binding['type'] !== null);
            break;
    }
}
// ...
if ($parameters !== []) {
    $operation['parameters'] = $parameters;
}

```

```

    if ($body !== null) {
        $operation['requestBody'] = $this->requestBody($body, $registry);
    } elseif ($files !== []) {
        $operation['requestBody'] = $this->multipartBody($files);
    }

    $operation['responses'] = $this->responseSet($route, $rejectable, $validated,
$notFound, $doc, $registry);
    // ...
    return $operation;
}

```

The two `// ...` cuts drop the parts the binding plan has no say in: the block that assembles the operation's human-facing prose — its `summary`, `description`, `deprecated` flag and `tags`, all four covered just below — and the operation-level `security` block, whose requirements come from the application's security rules rather than from the method signature. Everything else printed is what the *binding plan* decides. Reading the plan rather than re-reading the method signature is what makes the mapping unambiguous. `#[PathVariable]`, `#[QueryParam]` and `#[RequestHeader]` become Parameter Objects; `#[UploadedFile]` becomes a `multipart/form-data` part typed `format: binary`; `#[RequestBody]` becomes the Request Body Object; and the sixth `kind`, `service` — the no-attribute container-injected collaborator Chapter 4 introduced — is not part of the HTTP contract at all and never appears in the document. Deriving that list independently would have to re-decide every one of those cases and could disagree with the dispatcher; reading the plan cannot.

Four smaller decisions finish an operation:

- **Path template.** Laravel's optional-parameter spelling `{id?}` has no OpenAPI equivalent — a path parameter is required there, full stop — so the marker is stripped and the parameter stays required. Emitting two Path Items instead would

describe a router surface that does not exist and would double every such operation in a generated client.

- **operationId**. The route's **name** when it has one, otherwise derived as the controller's short name minus a trailing **Controller**, lower-cased, plus the method name: `Lumen\Web\WalletController::balance()` becomes `walletBalance`. Because **operationId** must be unique across the whole document — and a duplicate is the one flaw that makes most client generators abort rather than degrade — a repeat claim is suffixed (`walletBalance_2`) rather than allowed to overwrite.
- **tags**. The same short name, so every `WalletController` operation groups under "Wallet" in a viewer.
- **summary**. Split from the method name: `getBalance` reads as "Get balance". A method name is the only human-authored label a route carries — a `#[Mapping]`'s **name** is a Laravel route name, not prose — so it is the honest source.

Request bodies: one component per DTO, reached by `$ref`

Chapter 4's `OpenWalletRequest` is the running example again:

```
final class OpenWalletRequest
{
    public function __construct(
        #[NotBlank]
        public readonly string $owner_id,
        #[NotBlank]
        #[CurrencyCode]
        public readonly string $currency,
```

```
    ) {}
}
```

`POST /api/v1/wallets` binds it with `#[Valid] #[RequestBody]`, and the generated operation refers to it rather than repeating it:

```
{
  "requestBody": {
    "required": true,
    "content": {
      "application/json": {
        "schema": { "$ref": "#/components/schemas/OpenWalletRequest" }
      }
    }
  }
}
```

`SchemaRegistry` is what makes that `$ref` possible, and it solves two problems a naive "inline the schema at every use site" generator has. The first is **duplication**: a DTO used by six operations would be emitted six times, and every generated client would mint six structurally identical anonymous types with six different names. Registering once and referring by pointer is what makes `openapi-generator`, `orval` and `kiota` produce *one* named type per DTO — which is the whole point of generating the document in the first place. The second is **recursion**: a DTO with a `#[Valid] ?self $parent` member cannot be inlined at all, because the expansion does not terminate. The registry therefore reserves the component name *before* invoking the builder, so a nested call for the same class finds the name already taken and returns the pointer immediately, closing the cycle. A `$ref` cycle is legal and useful in a document where a flattened rule list would be infinite.

Component names are the class's short name, because that is what a human reads in a viewer and what a generator turns into a type name. Two DTOs sharing a short name across namespaces — `Order\Dto\Address` and `Billing\Dto\Address` — would collide and one would silently overwrite the other, so the **second** claimant of a name falls back to its dotted fully-qualified name: ugly, unambiguous, and rare. First claimant wins, so adding a second `Address` elsewhere in the app never renames the one that was already published.

Two properties of the emitted schema are worth stating explicitly because both are decisions rather than omissions. The member list is the **constructor's parameter list in declaration order**, because that is exactly what `ArgumentResolver` hydrates from — but a constraint keyed to a member with no constructor parameter is still documented, because `BeanValidator` validates the raw decoded array and so enforces it on input regardless. And **no `additionalProperties: false` is ever emitted**: the server genuinely ignores keys outside the constructor's list, so a spec claiming otherwise would make conforming clients reject requests the server would have served.

Constraints become schema keywords

Neither half of a DTO is enough on its own. The `ConstraintManifest` knows the *validation* contract but nothing about types, because a rule list is untyped by construction. The constructor knows the *type* contract — `?int`, a backed enum, a nested DTO, a default — but nothing about the constraints. Types-only would document `#[NotBlank] string $name` as an unbounded string; constraints-only would document `int $quantity` as a string. `DtoSchemaFactory` merges them, and `ConstraintSchemaMapper` maps the second half.

It maps the **compiled manifest**, never the `#[Constraint]` attributes. Reading the attributes back off the DTO would be the obvious route to "`#[Email]` → `format: email`", and it would document a validator that does not exist: the manifest has already applied `ConstraintScanner`'s Jakarta null contract, already expanded `#[Size]` into a first-party rule *object* rather than Laravel's polymorphic `min: / max:` strings, and already flattened one `#[Valid]` level into dotted keys. Generating from the attributes would re-derive all of that by hand and drift from it the first time a `toRules()` body changed. Generating from the manifest cannot drift, because the manifest *is* the contract.

Here is the real mapping, constraint by constraint, with the compiled rules in the middle column so you can see that the mapper is reading rules and not attributes:

Constraint	Compiles to	JSON Schema
<code>#[NotBlank]</code>	<code>required</code> , <code>string</code> , <code>regex:/\S/</code>	member added to <code>required</code> ; <code>type: string</code> ; <code>pattern: \S</code>
<code>#[NotEmpty]</code>	<code>required</code>	member added to <code>required</code>
<code>#[NotNull]</code>	<code>present</code> + a <code>NotNull</code> rule object	member added to <code>required</code> and <code>null</code> removed from the type union
<code>#[Min(n)]</code> / <code>#[Max(n)]</code>	<code>numeric</code> , <code>gte:n</code> / <code>lte:n</code>	<code>minimum</code> / <code>maximum</code>
<code>#[Positive]</code> / <code>#[Negative]</code>	<code>numeric</code> , <code>gt:0</code> / <code>lt:0</code>	<code>exclusiveMinimum: 0</code> / <code>exclusiveMaximum: 0</code>
<code>#[PositiveOrZero]</code> / <code>#[NegativeOrZero]</code>	<code>numeric</code> , <code>gte:0</code> / <code>lte:0</code>	<code>minimum: 0</code> / <code>maximum: 0</code>
<code>#[Email]</code>	<code>email</code>	<code>type: string</code> , <code>format:</code> <code>email</code>
<code>#[Pattern(re)]</code>	<code>regex:re</code>	<code>pattern</code> , with the PCRE delimiters and the no-op <code>D</code> / <code>u</code> flags stripped
<code>#[Digits(i, f)]</code>	<code>numeric</code> , an anchored <code>regex:</code>	<code>type: number</code> + <code>pattern</code>
<code>#[AssertTrue]</code> / <code>#[AssertFalse]</code>	<code>accepted</code> / <code>declined</code>	<code>type: boolean</code> + <code>const:</code> <code>true</code> / <code>const: false</code>

Constraint	Compiles to	JSON Schema
<code>#[Future]</code> / <code>#[Past]</code>	<code>date</code> , <code>after:now</code> / <code>before:now</code>	<code>type: string</code> , <code>format: date-time</code> (the <code>after:now</code> half is recorded, not mapped)
<code>#[Size(min, max)]</code>	a <code>Size</code> rule object	<code>minLength</code> / <code>maxLength</code> , or <code>minItems</code> / <code>maxItems</code> when the type is an array
<code>#[UuidValue]</code>	a <code>Uuid</code> rule object	<code>format: uuid</code> + the UUID pattern
<code>#[Phone]</code>	an <code>E164</code> rule object	<code>format: phone</code> + <code>pattern: ^\+[1-9]\d{1,14}\$</code>
<code>#[CurrencyCode]</code> / <code>#[CountryCode]</code>	<code>Currency</code> / <code>CountryCode</code> rule objects	<code>format: currency</code> + <code>^[A-Z]{3}\$</code> / <code>format: country-code</code> + <code>^[A-Z]{2}\$</code>
<code>#[LanguageTag]</code> / <code>#[PostalCode]</code>	rule objects	<code>format: bcp47</code> / <code>format: postal-code</code> , each with its pattern
<code>#[Iban]</code> / <code>#[Swift]</code> / <code>#[Bic]</code>	rule objects	<code>format: iban</code> / <code>swift</code> / <code>bic</code> , pattern withheld
<code>#[Cusip]</code> / <code>#[Isin]</code> / <code>#[Luhn]</code> / <code>#[RoutingNumber]</code>	rule objects	<code>format</code> only; the check digit is recorded, not mapped
<code>#[Percentage]</code>	a <code>Percentage</code> rule object	<code>type: number</code> , <code>minimum: 0</code> , <code>maximum: 100</code>

Constraint	Compiles to	JSON Schema
<code>#[Money]</code>	a <code>PositiveMoney</code> rule object	<code>type: number , exclusiveMinimum: 0 , multipleOf: 0.01</code>
<code>#[DecimalScale(n)]</code>	a <code>DecimalScale</code> rule object	<code>multipleOf</code> — scale 2 becomes <code>0.01</code>

Three rows in that table repay a closer look.

The pattern is withheld where the rule normalises first. `Iban` strips spaces and upper-cases before it matches; `Bic`, `Swift`, `Cusip` and `Isin` upper-case; `Luhn` and `RoutingNumber` strip separators. Publishing the post-normalisation pattern would reject payloads the server happily accepts, which is worse than under-specifying — so the `format` is emitted and the pattern is not.

`format` is an open vocabulary. In JSON Schema 2020-12 an unknown `format` is an annotation, not an error. IBAN, BIC, ISIN, CUSIP and E.164 have no registered format name, so self-describing ones (`iban` , `bic` , ...) are emitted rather than nothing.

Nullability is spelled the 3.1 way. OpenAPI 3.1 *is* JSON Schema 2020-12, which dropped 3.0's `nullable: true` keyword in favour of a type union. A nullable member is therefore `"type": ["string", "null"]`, and an `enum` additionally gains a `null` member — widening the type alone would leave `null` failing the enumeration, making the property undocumentable-as-null in practice.

The one rule that decides the whole merge is **first writer wins**, applied first to the declared type and then to the rules in declaration order — the same order the validator applies them in:

```

final class MapperState
{
    // ...
    public function keyword(string $keyword, mixed $value): void
    {
        if (! array_key_exists($keyword, $this->schema)) {
            $this->schema[$keyword] = $value;
        }
    }
    // ...
}

```

Seeding the declared PHP type *before* any rule is seen is why `#[Min(1)] int $quantity` stays `type: integer` instead of being widened to `number` by the `numeric` rule string `#[Min]` emits — a widening that would wrongly document `1.5` as acceptable.

Here is that whole pipeline on one real DTO. This is the generator's own test fixture, chosen because it deliberately spans every mapping route the generator has — a length-bounded string, a Jakarta null-contract nullable, an int with numeric bounds, a scaled decimal, a backed enum, a nested `#[Valid]` DTO, a PCRE pattern and a rule object:

```

final class CreateOrderRequest
{
    public function __construct(
        #[NotBlank] #[Size(max: 64)] public readonly string $reference,
        #[NotNull] #[Email] public readonly string $email,
        #[Min(1)] #[Max(999)] public readonly int $quantity,
        #[Positive] #[DecimalScale(2)] public readonly float $amount,
        public readonly Currency $currency,
        #[Valid] public readonly AddressPayload $shipTo,
        #[Pattern('/^[A-Z]{3}-\d{4}$/D')] public readonly ?string $coupon = null,
        #[UuidValue] public readonly ?string $idempotencyKey = null,
    ) {}
}

```

```
    ) {}
}
```

...and this is the component it generates, verbatim:

```
{
  "type": "object",
  "title": "CreateOrderRequest",
  "properties": {
    "reference": { "type": "string", "maxLength": 64, "pattern": "\\S" },
    "email": { "type": "string", "format": "email" },
    "quantity": { "type": "integer", "minimum": 1, "maximum": 999 },
    "amount": { "type": "number", "exclusiveMinimum": 0, "multipleOf": 0.01 },
    "currency": { "type": "string", "enum": ["EUR", "USD"] },
    "shipTo": { "$ref": "#/components/schemas/AddressPayload" },
    "coupon": { "type": ["string", "null"], "pattern": "[A-Z]{3}-\\d{4}$" },
    "idempotencyKey": {
      "type": ["string", "null"],
      "format": "uuid",
      "pattern": "^([0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})$"
    }
  },
  "required": ["reference", "email", "quantity", "amount", "currency", "shipTo"]
}
```

Every member of that object is traceable to something in the class above: the enum came from the backed enum's own cases, the `$ref` from `#[Valid]`, the two nullable unions from `?string`, the `multipleOf` from `#[DecimalScale(2)]`, and `required` from the presence rules — `coupon` and `idempotencyKey` are absent from it because nothing asserts their presence.

Lumen's own DTOs show the same machinery at a smaller scale, and one of them shows a case the table above cannot: a property carrying **two** patterns.

```
{
  "owner_id": { "type": "string", "pattern": "\\S" },
  "currency": {
    "type": "string",
    "format": "currency",
    "allOf": [{ "pattern": "\\S" }, { "pattern": "^[A-Z]{3}$" }]
  }
}
```

`#[NotBlank]` `#[CurrencyCode]` `string $currency` genuinely produces two patterns — `\S` from the non-blank rule and `^[A-Z]{3}$` from the currency rule — and JSON Schema has exactly one `pattern` slot per schema object. Collapsing them by keeping the last would silently drop the non-blank guarantee, so several patterns become an `allOf` of single-pattern subschemas instead. One pattern stays a plain `pattern`; the `allOf` appears only when it has to.

Nothing is dropped silently

Some constraints have no JSON Schema equivalent at all, and a few map only approximately. Dropping those quietly would produce a document that promises *less* validation than the server performs — a client would send a payload the spec calls valid and get a `422` back. The mapper's fall-through says what happens instead:

```
final class ConstraintSchemaMapper
{
  // ...
  private function applyObject(MapperState $state, ValidationRule $rule): void
  {
    switch (true) {
      // ...
      default:
```

```

    // A third-party ValidationRule. Its class name is the only thing about it
    that is knowable
    // without executing it, so that is what the extension records.
    $state->unmapped($rule::class);
  }
}
// ...
}

```

Everything unrecognised — `after:now` and `before:now`, `exists:` and `unique:`, a bare Luhn or CUSIP check digit, a bound that names another field (`gte:other_field`) rather than a number, and any third-party `ValidationRule` — is recorded under `x-firefly-constraints`. So is a pattern the mapper could only approximate: `D` and `u` are dropped as genuine no-ops, but any other flag — `i` above all, which ECMA-262 has no inline syntax for inside a pattern string — cannot be carried across, so the pattern is still emitted (it is the closest true statement available) *and* the original rule is recorded, so a reader can see that the published pattern is stricter than the server's.

Three properties, three outcomes:

```

{
  "deliverAfter": { "type": "string", "format": "date-time", "x-firefly-constraints": ["after:now"] },
  "slug": { "type": "string", "pattern": "^[a-z]+$", "x-firefly-constraints": ["regex:/^[a-z]+$<i>i</i>"] },
  "account": { "type": "string", "format": "iban", "x-firefly-constraints": ["iban:checksum"] }
}

```

Conforming tools ignore all three extensions and see a valid document. A human, or a generator you write yourself, can read the full truth.

Responses: one problem component, and an error set that is derived

Every operation ends at the same error component, and that component describes what LaraFly *actually* returns rather than what RFC 9457 describes in the abstract:

```
final class ProblemSchema
{
    // ...
    public const string MEDIA_TYPE = 'application/problem+json';
    // ...
    public static function response(): array
    {
        return [
            'description' => 'Error response in RFC 9457 problem+json form.',
            'content' => [self::MEDIA_TYPE => ['schema' => ['$ref' => self::REF]]],
        ];
    }
    // ...
}
```

The distinction matters because the two shapes differ. Chapter 4's

`ErrorResponse::toArray()` emits `status`, `title`, `code`, `category` and `severity` unconditionally, then `detail` / `type` / `instance` / `traceId` / `timestamp` only when non-null, then `errors` only when non-empty. So `code`, `category`, `severity` and `errors` are Firefly members on top of the RFC's five; `type` is written out on every document as `about:blank` — RFC 9457 §3.1.1 says an absent one *is* `about:blank`, and LaraFly makes Spring `ProblemDetail`'s choice of saying so, while `firefly.web.problem.type-uri` either names a base URI to derive a real type from the stable `code` or is set to `''` to drop the member; and `instance` is a **root-relative URI reference**, built by `ProblemMapper::instanceFor()`, rather than the bare request

path it carried until the RFC 9457 pass. Documenting the RFC's shape instead of this one would hand every generated client a decoder that silently drops the three members a caller actually branches on.

The two enumerations are read straight off the kernel's own enums, so a case added in `firefly/kernel` appears in the spec on the next generation with no edit anywhere in `firefly/openapi`:

```
{
  "category": {
    "type": "string",
    "enum": ["business", "validation", "security", "infrastructure",
            "external", "framework", "plugin", "internal"]
  },
  "severity": { "type": "string", "enum": ["info", "warning", "error", "critical"] }
}
```

Which statuses an operation lists is **derived, not guessed**. Compare two real Lumen operations. `GET /api/v1/wallets/{id}/balance` takes one `string` path variable and no body:

```
{
  "200": { "description": "Successful response.",
    "content": { "application/json": { "schema": { "type": "object" } } } },
  "default": { "$ref": "#/components/responses/Problem" }
}
```

`POST /api/v1/wallets/{id}/deposit` takes the same path variable plus a `#[Valid]` `#[RequestBody]` `AmountRequest`:

```
{
  "200": { "description": "Successful response.",
```

```

    "content": { "application/json": { "schema": { "type": "object" } } },
    "400": { "$ref": "#/components/responses/Problem" },
    "422": { "$ref": "#/components/responses/Problem" },
    "default": { "$ref": "#/components/responses/Problem" }
  }

```

`400` appears exactly when the operation has something `ArgumentResolver` can reject *before* the controller runs — a body to decode and bind, an upload to validate, a required query or header the client may omit, or a non-`string` parameter that has to be coerced out of the wire's string. It is deliberately absent from `balance`: nothing about that request can fail binding, because a missing path segment does not match the route at all, and a documented `400` an endpoint cannot produce is noise a generated client turns into a dead error branch. `404` is inferred for a path binding with a pattern. `422` is inferred for typed request bodies carrying `#[Valid]`, matching the binding branch that calls `BeanValidator`; query parameters, services and untyped bodies do not activate it. And `default` covers everything the handler itself may raise — a `404` from a `ResourceNotFoundException`, a `409` from a `ConflictException`, a `403` from a denied `#[PreAuthorize]` — which cannot be enumerated from the route manifest without reading the controller's body, and which all render through the same `ProblemDetailsRenderer` anyway.

A `204`, or a `void / never` return, gets no content at all, because emitting a content map for a status that carries no body is exactly what a strict client generator turns into a phantom return type. The success body of everything else is the subject of the next section.

The success body: what an endpoint actually returns

Look again at the two operations above. Both success responses are `{"type": "object"}` — an object with no members.

That was every success response in every document this generator produced, and it is the one that matters most: a viewer renders it as a blank panel and `openapi-generator` turns it into `any`, so the single most useful sentence an API document contains — *here is what you get back* — was the one sentence missing, for every endpoint of every application.

The reasoning had been that a `@return array{...}` is comment text nothing else in the framework treats as binding. That had already stopped being true. `RouteScanner` reads `@param list<X>` to compile the table `ArgumentResolver` hydrates from, so a docblock type expression is exactly as binding as a declared type on the way *in*. And there is a stronger argument still: **PHPStan at level max already checks these expressions against the code on every build**, which is what makes reading them safe. An out-of-date `@return` is a failing gate, not a silent lie.

So the success body now comes from three sources, most specific first. Three methods of one of the package's own response fixtures show what each of them carries:

```
/**
 * A page of consignments.
 *
 * @return array{page: positive-int, size: positive-int, total: int, items:
list<Consignement>}
 */
#[GetMapping]
public function index(): array
{
    return ['page' => 1, 'size' => 20, 'total' => 0, 'items' => []];
}
```

```

/**
 * One consignment.
 *
 * @param non-empty-string $reference
 */
#[GetMapping('/{reference}')]
public function show([PathVariable] string $reference): Consignment
{
    return new Consignment($reference, new Money(0, Currency::Eur), []);
}

/**
 * The shipments on a consignment.
 *
 * @return list<Shipment> newest first
 */
#[GetMapping('/{reference}/shipments')]
public function shipments([PathVariable] string $reference): array
{
    return [];
}

```

`index()`'s `@return` expression is the first source, and it produces this schema — `positive-int` carrying its own `minimum`, and `list<Consignment>` resolving the short name through the controller's own imports into a component reference:

```

{
    "type": "object",
    "properties": {
        "page": { "type": "integer", "minimum": 1 },
        "size": { "type": "integer", "minimum": 1 },
        "total": { "type": "integer" },
        "items": { "type": "array", "items": { "$ref": "#/components/schemas/Consignment" } }
    },
    "required": ["page", "size", "total", "items"],
}

```

```
"additionalProperties": false
}
```

`show()` needs no docblock type at all — its declared return type is already a class, which is the second source — and `shipments()`, whose declared type is a bare `array`, is carried entirely by its `@return list<Shipment>`. In order, then:

1. The **@return type expression** — the only place a PHP `array` can say what is in it. Prose written after the type becomes the response `description`, which is the only response description anyone ever actually writes.
2. The **declared return type** — a class becomes a component `$ref`, a backed enum its value set, a scalar itself.
3. **Neither** — `type: object`, the old behaviour, kept as the *fallback* for a bare `array` return with nothing said about it. A `@return array<string, mixed>` parses fine and means nothing, so it is treated as saying nothing rather than allowed to suppress what the declared type knew.

A parser, not another regular expression

The package already had two regexes for the one shape it handled, `list<X>` and `X[]`, and they cannot be extended to the rest. `array{items: list<array<string, Order>>}` needs balanced `<>` and `{}` and a comma that separates only at the outer level. That is a grammar, and a grammar wants a parser — about two hundred lines of recursive descent in `DocType`, against the four transitive dependencies `phpstan/phpdoc-parser` would put in every application that installs this package.

Written	Becomes
<code>list<Order>, Order[], array<int, Order></code>	<code>type: array</code> with <code>items: {\$ref: Order}</code>
<code>array<string, Money></code>	<code>type: object</code> with <code>additionalProperties</code>
<code>array{a: int, b?: string}</code>	an object, <code>required: [a]</code> , <code>additionalProperties: false</code>
<code>array{a: int, ...}</code>	the same, open — the <code>...</code> is the only thing that lifts it
<code>array{int, string}</code>	<code>prefixItems</code> — a tuple
<code>'draft'\ 'sent'</code>	<code>type: string</code> with <code>enum</code>
<code>?Order</code>	<code>anyOf: [{ \$ref }, { type: null }]</code>
<code>non-empty-string, positive-int</code>	<code>minLength: 1, minimum: 1</code>
<code>never, callable</code> , an unresolvable name	nothing — the caller falls back to what it knew

A `?` on a shape **key** means "may be absent" and becomes `required`; a `?` on the **value** means "may be null". Conflating the two documents an omissible member as one a client must always send. Class names resolve through the imports of the file the expression was written in, because reflection does not expose a file's `use` statements — without that, only fully-qualified names would work, which is the one spelling nobody writes.

A returned class is built from its wire shape

`ResponseSchemaFactory` is not `DtoSchemaFactory`, and the difference is the point. That factory derives members from the **constructor** and rules from the `ConstraintManifest` — the right two sources for a payload the server binds and validates, and the wrong two on the way out. A response is never validated, and its members are what `json_encode` emits.

Which PHP spells two ways. A class implementing `JsonSerializable` serialises as whatever `jsonSerialize()` **returns**; everything else as its **public properties**. The skeleton's `App\Orders\Order` is the case that decides the design:

```
final readonly class Order implements JsonSerializable
{
    // ...

    /** The order's value, derived from its lines rather than stored beside them. */
    public function total(): float
    {
        return round(array_sum(array_map(static fn (OrderLine $line): float => $line->subtotal(), $this->lines)), 2);
    }

    /** @return array{id: int|null, customer: string, email: string, shipTo: Address, lines: list<OrderLine>, total: float} */
    public function jsonSerialize(): array
    {
        return [
            'id' => $this->id,
            'customer' => $this->customer,
            'email' => $this->email,
            'shipTo' => $this->shipTo,
            'lines' => $this->lines,
            'total' => $this->total(),
        ];
    }
}
```



```
    }
}
```

`total` is a derived **method**, not a property. Reflecting properties alone would publish five of the six members the API actually sends. The array shape states all six, PHPStan checks it against the method, and the generator reads it — delete the annotation and `total` silently disappears from the document while the API keeps sending it.

A declared shape only wins when it says something: `@return array<string, mixed>` on `jsonSerialize()` means "an object, members unknown", which is strictly less than the property list it would have suppressed, so it is ignored in favour of reflection.

One rule inverts on the way out. **Nullability is not requiredness here.** A response member is present or absent, and `?int $id` is always *present* and sometimes null — so response members stay **required** and nullable ones widen their type. The request side's rule would have told every client to expect an absence that never happens.

`#[ApiResponse]` takes a type expression too

```
/**
 * Books a consignment.
 *
 * @return array<string, mixed>
 */
#[PostMapping(status: 201)]
#[ApiResponse(status: 409, description: 'A consignment with that reference already
exists.', type: Consignment::class)]
#[ApiResponse(status: 202, description: 'Accepted for later booking.', type:
'list<Shipment>')]
public function book(): array
{
    return [];
}
```

(The empty body is not an omission: this is one of the package's own response fixtures, and the only things the generator reads are the signature, the docblock and the attributes.)

`type` is a full expression, not only a class or a scalar name, and a short name resolves through the controller's own imports — `Consignment::class` and the string `'list<Shipment>'` are handled by the same parser. Note also what the three statuses do together: `201` from `#[PostMapping]`, plus one `#[ApiResponse]` per alternative outcome, each with its own schema.

`#[Controller]` HTML routes are not operations

Chapter 4 introduced `#[RestController]` alongside its HTML sibling `#[Controller]`, and only the first is a JSON API. The generator honours that distinction by default:

```
private function excluded(RouteDescriptor $route, ApiDocs $docs): bool
{
    if ($route->html && ! $this->properties->includeHtml) {
        return true;
    }

    if ($docs->ignores($route)) {
        return true;
    }

    foreach ($this->properties->excludePathPrefixes as $prefix) {
        if (str_starts_with($route->path, $prefix)) {
            return true;
        }
    }
}
```

```
    return false;
  }
```

Read the name before the body: `excluded()` answers *leave this route out*, so every `return true` above is a route that does **not** reach the document. The first arm is the `#[Controller]` rule, the second is `#[ApiIgnore]`, and the third is the configured path-prefix list.

`#[Controller]` routes are excluded by default. Set `firefly.openapi.include-html` to `true` to include them. Response media still comes from the return contract: views and `Htmlable` values produce `text/html`, arrays produce JSON, and redirects carry `Location`. The stereotype itself does not determine response media.

The second half of that method is the blunt instrument for everything else:

`firefly.openapi.exclude` is a CSV of path prefixes — `'/internal,/admin'` — for routes that are JSON but are nobody's public API.

The viewer: three styles, and only one of them phones out

`GET /openapi` renders a browser console over the document. Which console is `firefly.openapi.viewer.style`, and the choice is a supply-chain decision dressed up as a preference:

style	Ships from	Third-party request at page view?
<code>swagger</code> (default)	your own origin, out of the <code>swagger-api/swagger-ui</code> composer package	no
<code>builtin</code>	inline in the response	no
<code>cdn</code>	<code>cdn.jsdelivr.net</code>	yes, on every view

An unrecognised value falls back to `swagger` rather than rendering a blank page — a typo in a config file should cost you nothing.

Why the default is the official Swagger UI, from your own origin

Every off-the-shelf viewer — Swagger UI, Redoc, Elements — is a bundled JavaScript application, and for years that left a PHP package exactly two options. Vendor a multi-megabyte minified bundle into the package's own git history, so every clone of every dependent project pays for it forever and the framework is pinned to a release train it cannot patch without cutting a release of its own. Or fetch it from a CDN on every page view, which is a supply-chain dependency and a data-protection question, and which renders *nothing at all* in the air-gapped and strict-CSP environments where an internal API console is most wanted.

There is a third option, and this package takes it. `swagger-api/swagger-ui` publishes its `dist` on Packagist under Apache-2.0, so **composer** can fetch and pin it — it is a hard `require` of `firefly/openapi`, so the files are already on disk in `vendor/` by the time you first hit the route — and `SwaggerAssetAction` serves those files from the application's own origin:

```
final class SwaggerAssetAction
{
```

```

    public function __construct(private readonly SwaggerAssets $assets) {}

    public function __invoke(string $file): SymfonyResponse
    {
        $path = $this->assets->path($file);
        $type = $this->assets->contentType($file);

        if ($path === null || $type === null) {
            return new Response('Not Found', 404, ['Content-Type' => 'text/plain; charset=UTF-8']);
        }

        $response = new BinaryFileResponse($path, 200, ['Content-Type' => $type]);
        $response->setPublic();
        $response->setMaxAge(31536000);
        $response->setImmutable();
        $response->setAutoEtag();

        return $response;
    }
}

```

You get the console byte-for-byte as Swagger publishes it — the full feature set, deep linking, try-it-out, the OAuth2 redirect popup — with no CDN request, no npm step, and nothing in this repository's history that a `composer update` could not replace. The long cache header is safe precisely because the bytes are immutable for a pinned version: composer changes them only when the pinned version changes, and the ETag changes with them.

❗ PATH TRAVERSAL IS DEFENDED BY A WHITELIST, NOT A SANITISER

Only seven basenames are servable at all — `swagger-ui.css`, `swagger-ui-bundle.js`, `swagger-ui-standalone-preset.js`, `oauth2-redirect.html`, two favicons and `index.css` — and each resolved path is `realpath()`-checked to be inside the `dist` directory. The route itself constrains `{file}` to `[A-Za-z0-9._-]+`, so it cannot even *express* a traversal. Matching against a fixed list rather than scrubbing the input is the deliberate choice: a whitelist cannot be defeated by an encoding trick that a sanitiser missed. Anything else is a plain `text/plain` 404 — not `problem+json`, because the caller here is a browser fetching a stylesheet, not an API client.

The `dist` directory is located by asking **Composer's own installed-versions metadata** for the package root, rather than walking up from `__DIR__`. The depth from `src/Web/` to `vendor/` differs between an installed package (`vendor/firefly/openapi/src/Web/`) and this monorepo (`packages/openapi/src/Web/`), so a relative walk would work in exactly one of them; the walk is kept only as a fallback for a runtime whose autoloader cannot answer.

And if the distribution is genuinely absent — a stripped `vendor/`, a phar, a non-composer runtime — `render()` falls back rather than serving a page whose assets 404:

```
final class ViewPage
{
    // ...
    public function render(string $specUrl, string $style, string $assetBase = ''): string
    {
        return match (true) {
            $style === 'cdn' => $this->swaggerUiFromCdn($specUrl),
            // ...
            $style === 'swagger' && $this->assets->available() => $this->swaggerUi($specUrl,
            $assetBase),
            default => $this->builtIn($specUrl),
        };
    }
}
```

```

    };
  }
  // ...
}

```

What `builtin` is for

A hand-written, dependency-free reference: one inline script, a few hundred bytes of CSS, one `fetch` of the spec route, and a palette that follows `prefers-color-scheme`. It does the two things a reader actually needs from a generated spec and that raw JSON does not give them — it groups operations by tag with verbs and paths visible at a glance, and it **resolves `$ref` pointers client-side**, so the reader sees a DTO's members rather than a pointer into `#/components/schemas`. Try-it-out, OAuth flows and code samples are deliberately absent; that is what `swagger` is for.

Choose it when the deployment's rule is *no third-party JavaScript in the response at all*, rather than merely *no third-party host*.

WHY THAT PAGE IS A NOWDOC

The built-in viewer embeds a JavaScript application, and a PHP **heredoc** interpolates variables. Every `$ref`, `$schema` and `$1` in that script was therefore read as a PHP variable — `$ref` silently became the empty string, and `$ref` resolution, the whole point of the page, stopped working. A nowdoc takes the script verbatim and the two real substitutions are made explicitly afterwards. It is the kind of bug that produces no error anywhere: the page renders, and simply shows pointers instead of schemas.

What `cdn` costs

```

// config/firefly.php
return [
    'openapi' => ['viewer' => ['style' => 'cdn']],
];

```

Every page view then loads Swagger UI from `cdn.jsdelivr.net`. The version is pinned exactly, and **no Subresource Integrity hash is claimed** — deliberately: a hash the framework cannot verify at release time is security theatre, and a wrong one would simply break the page.

Weigh the trade honestly. In exchange for a request to a third party on every view, a Content-Security-Policy that has to allow that host, and a console that renders nothing in an air-gapped deployment, you get... the same Swagger UI that `swagger` already served you from your own origin. The style is kept because it is what most tutorials show, and because some organisations genuinely prefer their bytes to come from a cache they already trust — not because it is the better default.

 VIEWER.CDN STILL WINS OVER VIEWER.STYLE

`firefly.openapi.viewer.cdn` (default `false`) is the older boolean spelling of this option, from before `style` existed. It still **forces** the CDN page and overrides `style`, so an application that set it keeps the behaviour it configured rather than being silently moved onto a different console by a framework upgrade. Prefer `style` in new configuration; delete `cdn` when you adopt it.

The viewer fetches the spec from the sibling route rather than having the document inlined into the page, so a regenerated spec shows up on a plain browser refresh, and so the two routes can be exposed independently — a deployment may well want the machine-readable document public and the console off, or the reverse. The spec URL is resolved through the `UrlGenerator` rather than concatenated, because an app mounted under a subdirectory or behind `APP_URL` would otherwise get a link that 404s from every page but the root, and a viewer whose only network call is wrong is a viewer that shows nothing at all.

Configuration, and securing the surface

Everything the document and its routes need lives under one config key. The block below is written as live PHP for readability; `skeleton/config/firefly.php` ships the same keys **commented out**, with each default beside it, because `firefly/openapi` is an optional package and an uncommented block would be a decision made for you:

```
<?php

declare(strict_types=1);

return [
    'openapi' => [
        'enabled' => true,           // master gate: off means every route is genuinely
unrouted
        'path' => '/openapi.json',  // spec route
        'viewer' => [
            'enabled' => true,
            'path' => '/openapi',    // assets are mounted under {path}/assets/{file}
            'style' => 'swagger',    // swagger (default) | builtin | cdn – see above
        ],
        'title' => 'Lumen Wallet API',
        'version' => '1.0.0',
        'description' => '',
        'servers' => ['https://api.example.test'], // bare URLs or OpenAPI Server Objects
        'exclude' => '/internal,/admin',          // CSV of path prefixes to leave out
        'include-html' => false,                  // include #[Controller] routes with
their return contract
    ],
];
```

`servers` accepts both spellings a real config file uses — a list of bare URL strings, and OpenAPI's own object form with a `description` — and an entry that is neither is dropped rather than emitted, because a Server Object with no `url` is invalid under the 3.1 schema and would poison an otherwise-good document.

A public deployment is secured the way any other route is. Chapter 10's `HttpSecurity` rules — ahead, in Part III — cover the spec and viewer paths with no code edge at all, because `HttpSecurityFilter` is a global middleware and runs for natively-registered routes exactly as it runs for your controllers:

```
<?php
declare(strict_types=1);

return [
    'security' => [
        'enabled' => true,
        'http' => [
            'enabled' => true,
            'rules' => [
                ['pattern' => 'openapi', 'access' => 'hasRole:DEVELOPER'],
                ['pattern' => 'openapi/*', 'access' => 'hasRole:DEVELOPER'],
                ['pattern' => 'openapi.json', 'access' => 'hasRole:DEVELOPER'],
            ],
        ],
    ],
];
```

Note the three patterns. `openapi` alone does not match `openapi/assets/swagger-ui.css`, and `openapi.json` is a separate literal — a rule set that covers the console but not its assets produces an authenticated page whose stylesheet answers 401, which is a worse outcome than either extreme.

The alternative, for a deployment that wants no documentation surface at all in production, is `enabled => false` plus a `firefly:openapi --output=` step in CI.

Replacing one piece of the pipeline

Every collaborator in the package — `OpenApiProperties`, `ConstraintSchemaMapper`, `DtoSchemaFactory`, `OperationFactory`, `OpenApiGenerator` and `ViewerPage` — is a `#[Bean]` behind `#[ConditionalOnMissingBean]`, the Chapter 2 mechanism. Teaching the generator about your own `ValidationRule` is therefore a short `#[Configuration]` in your application and never a fork:

```
#[Configuration]
final class ApiDocsConfiguration
{
    #[Bean]
    public function constraintSchemaMapper(): ConstraintSchemaMapper
    {
        return new HouseConstraintSchemaMapper; // teaches the generator your own
        ValidationRules
    }
}
```

i THE REFLECTION HERE IS REAL, AND IT IS NOT ON THE REQUEST PATH

`DtoSchemaFactory` reflects a DTO's constructor to learn its property types, which looks like a violation of the rule Chapter 13 will state in full: nothing on the cached request path reflects. It is not one. That reflection runs when `firefly:openapi` generates a file, or on a hit to the spec route — whose result the generator memoises for the life of the process — and never while dispatching an application request. It is the same category of work as `RouteScanner` and `ConstraintScanner`, both of which reflect at compile time only. The alternative, teaching `RouteScanner` to emit per-property types into every `RouteDescriptor`, was rejected because it would grow the compiled route manifest of every app for the benefit of one optional package.

📦 LARAVEL PARITY

Stock Laravel ships no OpenAPI support. The usual answers are a third-party package driven by its own annotation dialect (`zircote/swagger-php` 's `@OA\` blocks, or `vyuldashev/laravel-openapi` 's attribute classes) or a hand-maintained YAML file — both of which are a *second* description of the API, sitting beside the routes and the FormRequests that actually enforce it, and both of which drift. `firefly/openapi` has no dialect to learn because it has no second description: it reads the same `RouteManifest` the dispatcher dispatches from and the same `ConstraintManifest` the validator validates with. The closest analogue outside PHP is `springdoc-openapi`, and the closest thing in Laravel itself is `php artisan route:list` — accurate for the same reason, and for the same reason unable to tell you anything about a request body.

✓ What you learned

Concept	What it does
<code>OpenApiGenerator</code>	Assembles an OpenAPI 3.1 document from <code>RouteManifest</code> + <code>ConstraintManifest</code> ; memoised per instance, deterministically ordered so regenerations diff cleanly
<code>firefly:openapi</code>	Writes the document to <code>--output=</code> or raw to stdout, making the spec a committable build artifact a CI job can diff
<code>OpenApiRouteRegistrar</code>	Mounts <code>/openapi.json</code> , <code>/openapi</code> and <code>/openapi/assets/{file}</code> natively from a <code>BootPass</code> — configurable paths an attribute route could never have, and no self-documentation
<code>OperationFactory</code>	Maps each binding <code>kind</code> — <code>path / query / header / file / body</code> — to its OpenAPI shape; <code>service</code> bindings never appear
<code>SchemaRegistry</code>	One component per DTO, reached by <code>\$ref</code> : no duplicate generated types, and a reserved name closes a recursive <code>\$ref</code> cycle
<code>DtoSchemaFactory</code>	Merges declared constructor types with compiled constraints; emits no <code>additionalProperties: false</code> , because the server ignores extra keys
<code>ConstraintSchemaMapper</code>	Maps the compiled rule list — not the attributes — to keywords; first writer wins, so <code>#[Min(1)] int</code> stays <code>integer</code>

Concept	What it does
<code>x-firefly-constraints</code>	Records what JSON Schema cannot state (<code>after:now</code> , a checksum, a flagged PCRE, a third-party rule) instead of dropping it
<code>ProblemSchema</code>	The one shared <code>application/problem+json</code> response; documents Firefly's <code>code</code> / <code>category</code> / <code>severity</code> / <code>errors</code> , with the enums read off the kernel's own cases
Derived error set	<code>400</code> only when something is rejectable before the controller runs, <code>404</code> for path patterns, <code>422</code> for typed bodies under <code>#[Valid]</code> , <code>default</code> always
<code>DocType</code>	Compiles a PHPDoc type expression to a JSON Schema fragment — shapes, generics, tuples, literal unions, PHPStan pseudo-types — and returns <i>nothing</i> rather than guessing when it cannot read one
<code>ResponseSchemaFactory</code>	Builds a returned class from its WIRE shape: <code>jsonSerialize()</code> 's declared <code>@return</code> when there is one, public properties otherwise. Response members stay <code>required</code> and nullable ones widen their type
<code>#[ApiResponse(type:)]</code>	A full type expression (<code>'list<Shipment>'</code>), resolved through the controller's own imports
<code>\$route->html</code>	<code>#[Controller]</code> routes are excluded by default; <code>firefly.openapi.include-html</code> includes them with media derived from their return contract

Concept	What it does
<code>firefly.openapi.viewer.style</code>	<code>swagger</code> (default) <code>builtin</code> <code>cdn</code> . Only <code>cdn</code> makes a third-party request at page view; an unrecognised value falls back to <code>swagger</code>
<code>SwaggerAssets</code>	Serves the OFFICIAL Swagger UI from your own origin out of the <code>swagger-api/swagger-ui</code> composer package — seven whitelisted basenames, each <code>realpath()</code> -checked inside the dist directory
<code>ViewerPage::render()</code>	Falls back to <code>builtin</code> when the Swagger dist is absent, rather than rendering a console whose assets 404

Try it yourself

- 1. Generate Lumen's document and read it.** Run `php artisan firefly:openapi --output=openapi.json` in the sample, then open `/openapi` in a browser. Find `walletBalance` and confirm it has no `400` response, then find `walletDeposit` and confirm it has both a `400` and a `422` — and satisfy yourself, from this chapter's rules, why the two differ.
- 2. Make the spec a CI gate.** Commit the generated file, then add a job that regenerates it and runs `git diff --exit-code` over it. Change a DTO — add a `#[Size(max: 32)]` to `OpenWalletRequest::$owner_id` — and watch the job fail with a diff that names the exact schema keyword that changed.
- 3. Prove the default console makes no outbound request.** Open `/openapi` in the sample with the browser's network panel recording, and confirm every request is

same-origin: the page, `openapi/assets/swagger-ui.css`, the two bundles, and `openapi.json`. Then set `firefly.openapi.viewer.style` to `cdn`, reload, and watch `cdn.jsdelivr.net` appear in the same panel — that request is the entire difference, and it is what a strict CSP or an air-gapped host would block.

4. **Delete an annotation and watch the document lose a member.** In the skeleton, remove the `@return array{...}` from `App\Orders\Order::jsonSerialize()`, regenerate, and find `total` gone from the `Order` schema while `GET /orders/1` still returns it. Put it back, then change `total: float` to `total: string` and run PHPStan: the gate that keeps the document honest is the one that fails.
5. **Watch a constraint fall through to the extension.** Add `#[Future]` to a `string` property on a request DTO, regenerate, and find the property's `x-firefly-constraints` array carrying `after:now` beside a perfectly ordinary `format: datetime`. Then add `#[Pattern('/^[a-z]+$|i')]` to another property and compare: the pattern *is* published, and the original rule is recorded beside it because the `i` flag could not survive the translation.

PART II

Modeling & Persisting the Domain

Persistence & the Repository Pattern

By the end of this chapter you will know the two ports every LaraFly repository implements, how `EloquentRepository` gives you full CRUD for the price of one `$model` assignment, how a derived-query method name compiles into a real Eloquent query with no body of your own, the `#[Query]` escape hatch for anything a name cannot express cleanly, how `Specification`s compose reusable predicates, how **query by example** turns a half-filled entity into a query with no predicate written at all, what the four method attributes (`#[Modifying]`, `#[Projection]`, `#[Lock]`, `#[EntityGraph]`) each say about a method, the `Page / Pageable / Sort` value objects that carry pagination end to end and the `Slice` that deliberately does not count, the `DataAccessException` family every driver error is translated into — and where that translation deliberately does not happen — and the domain-event bridge that ties a repository's `save()` call to Chapter 6's aggregate.

NEW TERM: DERIVED QUERY

A **derived query** is a repository method with **no body** — just a name like `findByOwnerId` — whose SQL the framework compiles by parsing the method name itself, at the moment it is first called. You write the name; the framework writes the `WHERE` clause.

The ports: `CrudRepository` and `PagingAndSortingRepository`

Every LaraFly repository ultimately implements two small interfaces from `firefly/data`. PHP has no runtime generics, so the port signatures are deliberately `object / mixed / array` — the `@template` docblock parameters give PHPStan the precise entity and id types back:

```
/**
 * // ...
 * @template TEntity of object
 * @template TId
 */
interface CrudRepository
{
    /**
     * @param TEntity $entity
     * @return TEntity
     */
    public function save(object $entity): object;

    /**
     * @param iterable<TEntity> $entities
     * @return list<TEntity>
     */
    public function saveAll(iterable $entities): array;

    /**
     * @param TId $id
     * @return TEntity|null
     */
    public function findById(mixed $id): ?object;

    /**
     * @return list<TEntity>
     */
}
```

```

    public function findAll(): array;

    /**
     * @param iterable<TId> $ids
     * @return list<TEntity>
     */
    public function findAllById(iterable $ids): array;

    /**
     * @param TId $id
     */
    public function existsById(mixed $id): bool;

    public function count(): int;

    /**
     * @param TEntity $entity
     */
    public function delete(object $entity): void;

    /**
     * @param TId $id
     */
    public function deleteById(mixed $id): void;

    public function deleteAll(): void;
}

```

Spring Data overloads a single `findAll(Pageable) / findAll(Sort)`; PHP has no method overloading, so `PagingAndSortingRepository` splits the two into distinct, cleanly-typed methods instead:

```

/**
 // ...
 * @template TEntity of object
 * @template TId

```

```

*
* @template-extends CrudRepository<TEntity, TId>
*/
interface PagingAndSortingRepository extends CrudRepository
{
    /**
     * @return Page<TEntity>
     */
    public function findPaged(Pageable $pageable): Page;

    /**
     * // ...
     * @return list<TEntity>
     */
    public function findSorted(Sort $sort): array;
}

```

You will not implement either interface directly.

`Firefly\Data\Repository\EloquentRepository` already implements both over `Model::query()`, and every concrete repository in your application extends *that* — the same relationship Chapter 2 described between an interface and its sole implementing bean, now one level lower in the stack.

LARAVEL PARITY

`CrudRepository` / `PagingAndSortingRepository` are LaraFly's counterparts to Spring Data's `CrudRepository` / `PagingAndSortingRepository`. If you have written a Spring interface `OrderRepository extends JpaRepository<Order, UUID>`, `EloquentRepository` is the same idea translated onto Eloquent: you get the full CRUD surface from a single `extends`, and add only the queries specific to your entity.

The port and the adapter: `WalletRepository`

Chapter 2 introduced the hexagonal split between a *port* your domain and application layers depend on, and the *adapter* that actually talks to storage. Persistence is where that split earns its keep. Here is the whole port, unchanged from Chapter 2:

```
<?php

declare(strict_types=1);

namespace Lumen\Infrastructure;

use Lumen\Domain\Wallet;

/**
 * The hexagonal PORT for wallet persistence: the domain/application layer depends on this
 * interface only, never on
 * Eloquent or any storage detail. `EloquentWalletRepository` is the sole adapter, auto-
 * bound by the framework's
 * nominal interface auto-binding (Firefly\Container's ComponentScanner/
 * ContainerRegistrar::wireInterfaces()).
 */
interface WalletRepository
{
    public function save(Wallet $wallet): Wallet;

    public function findById(string $id): ?Wallet;

    /** @return list<Wallet> */
    public function findByOwnerId(string $ownerId): array;
}
```

And here is the adapter — the class that extends `EloquentRepository` and does the real work:

```

<?php

declare(strict_types=1);

namespace Lumen\Infrastructure;

use Firefly\Container\Attributes\Repository;
use Firefly\Data\Repository\EloquentRepository;
use InvalidArgumentException;
use Lumen\Domain\Wallet;

/**
 * The Eloquent ADAPTER for the `WalletRepository` port, bound to the `Wallet` aggregate
 * model.
 *
 * Because this class `implements WalletRepository` (a real PHP interface) in addition to
 * `extends
 * EloquentRepository`, every interface method needs an EXPLICIT body – the framework's own
 * convention of leaning on
 * `EloquentRepository::__call()` (magic dispatch, documented via `@method` docblocks only)
 * does NOT satisfy an
 * `implements` contract; PHP fatals at class-load time if an abstract interface method is
 * left undefined.
 *
 * `save()`/`findById()` keep the PARENT's exact parameter type (`object`/`mixed`) and only
 * narrow the RETURN type to
 * `Wallet` – not the parameter to `Wallet`, even though that is what `WalletRepository`
 * declares. This is a real PHP
 * variance constraint, not a stylistic choice: overriding `EloquentRepository::save(object
 * $entity): object` with a
 * narrower parameter (`Wallet $wallet`) is an invalid override (contravariance requires the
 * override's parameter to
 * be the same type or WIDER than the parent's, never narrower) and fatals at class-load –
 * confirmed empirically
 * while building this adapter. Widening the implementation's parameter back to `object`/
 * `mixed` still satisfies
 * `WalletRepository`'s narrower `Wallet`/`string` parameters, because interface conformance
 * uses the same

```



```

* contravariant-parameter/covariant-return rule: an implementation may accept MORE than the
interface promises and
* return exactly what it promises. An `instanceof` guard narrows the parameter back to
`Wallet` before delegating to
* `parent::save()`, so the generic template parameter on the parent resolves to `Wallet`
and the return type lines
* up with no extra type-override annotation or PHPStan suppression comment needed.
*
* `findByOwnerId()` – a derived query with no parent counterpart, so no variance constraint
applies – delegates
* explicitly to the protected `dispatchQuery()` dispatcher (the same engine `__call()`
would have reached, just
* invoked directly so the method has a real body, per the `implements` requirement above).
*
* @extends EloquentRepository<Wallet>
*/
#[Repository]
final class EloquentWalletRepository extends EloquentRepository implements WalletRepository
{
    protected string $model = Wallet::class;

    public function save(object $entity): Wallet
    {
        if (! $entity instanceof Wallet) {
            throw new InvalidArgumentException(sprintf('%s::save() only accepts a %s.',
self::class, Wallet::class));
        }

        return parent::save($entity);
    }

    public function findById(mixed $id): ?Wallet
    {
        $found = parent::findById($id);

        return $found instanceof Wallet ? $found : null;
    }

    /** @return list<Wallet> */

```

```

    public function findByOwnerId(string $ownerId): array
    {
        /** @var list<Wallet> $result */
        $result = $this->dispatchQuery('findByOwnerId', [$ownerId]);

        return $result;
    }
}

```

Notice the `protected string $model = Wallet::class;` line — that single assignment is what tells `EloquentRepository` which model `query()` should run against, and it is the entire reason the full CRUD surface (`save`, `saveAll`, `findById`, `findAll`, `findAllById`, `existsById`, `count`, `delete`, `deleteById`, `deleteAll`, plus paging and sorting) shows up on `EloquentWalletRepository` for free. `$this->model::query()` is the one line at the bottom of every inherited method.

⚠ A PHP VARIANCE CONSTRAINT, NOT A DESIGN CHOICE

A class that both `extends EloquentRepository` and `implements WalletRepository` must give every interface method an explicit body — magic `__call()` dispatch alone does not satisfy an `implements` clause. And because `WalletRepository::save(Wallet $wallet): Wallet` narrows the parent's `save(object $entity): object`, overriding with the *narrower* parameter type is an invalid override in PHP (parameter types must stay the same or widen, never narrow); `EloquentWalletRepository::save()` keeps the parent's `object $entity` signature and narrows only the *return* type, guarding the actual `Wallet` requirement with an `instanceof` check inside the method body instead.

Derived queries: the method name is the query

`findById` above has no query logic in its own body — `dispatchQuery()` is the shared engine underneath both it and plain `__call()` magic dispatch. A repository method whose name follows the grammar below needs no body at all when called through `__call()`; `EloquentRepository::dispatchQuery()` parses the method name with `DerivedQueryParser::parse()` — pure string parsing, no reflection, the method name is the only input — and drives an Eloquent `Builder` accordingly.

The grammar

Piece	Values	Notes
Prefix	<code>findBy</code> / <code>countBy</code> / <code>existsBy</code> / <code>deleteBy</code>	<code>find</code> also accepts <code>findFirstBy...</code> (limit 1), <code>findTop{N}By...</code> (limit N), <code>findDistinctBy...</code> (adds <code>DISTINCT</code>) before the <code>By</code> .
Connector	<code>And</code> / <code>Or</code>	Splits predicate groups; only recognized before a new CamelCase field.
Operator	see below	Matched longest-first against the end of each predicate group; <code>Equals</code> (no token) is the implicit default.
Case modifier	<code>IgnoreCase</code> suffix	Routes the equality/LIKE family through <code>LOWER(col)</code> or <code>LOWER(?)</code> .
Ordering	trailing <code>OrderBy{Field}{Asc\ Desc}</code>	Chainable; direction defaults to <code>asc</code> if omitted.

Operators, in the exact longest-match-first order the parser walks:

Operator	SQL effect	Args consumed
<code>GreaterThanOrEqualTo</code>	<code>>=</code>	1
<code>LessThanOrEqualTo</code>	<code><=</code>	1
<code>GreaterThan</code>	<code>></code>	1
<code>LessThan</code>	<code><</code>	1
<code>Between</code>	<code>BETWEEN ? AND ?</code>	2
<code>NotLike</code>	<code>not like</code>	1
<code>Like</code>	<code>like</code>	1
<code>NotIn</code>	<code>NOT IN (?)</code>	1 (array)
<code>In</code>	<code>IN (?)</code>	1 (array)
<code>Containing</code>	<code>like '%value%'</code>	1
<code>StartingWith</code>	<code>like 'value%'</code>	1
<code>EndingWith</code>	<code>like '%value'</code>	1
<code>IsNotNull</code>	<code>IS NOT NULL</code>	0
<code>IsNull</code>	<code>IS NULL</code>	0
<code>Not</code>	<code>!=</code>	1
<code>True</code>	<code>= true</code>	0
<code>False</code>	<code>= false</code>	0
<code>(none)</code>	<code>Equals → =</code>	1

Each CamelCase field segment maps to a `snake_case` column, and every derived column is validated against a bare-identifier pattern before it can reach a query — the method name is untrusted input reachable through the public `__call`, so this validation is what keeps the `IgnoreCase` path's raw `LOWER(col)` fragment injection-free. Arguments bind to predicates in declaration order, left to right, advancing a cursor by however many values each operator consumes.

The framework's own `RecordRepository` fixture — the one `packages/data`'s derived-query suite runs against — shows the grammar at a glance. The `@method` tags are how PHPStan sees a typed return for what is, at runtime, dynamic `__call` dispatch:

```
/**
 // ...
 * @extends EloquentRepository<Record>
 *
 * @method list<Record> findByStatusAndAmountGreaterThan(string $status, int $amount)
 * @method list<Record> findTop2ByStatusOrderByAmountDesc(string $status)
 * @method bool existsByEmailIgnoreCase(string $email)
 * @method int countByStatus(string $status)
 * @method int deleteByStatus(string $status)
 * @method Page<Record> findByStatus(string $status, Pageable $pageable)
 */
#[Repository]
class RecordRepository extends EloquentRepository
{
    protected string $model = Record::class;
```

Not one of those six methods is declared anywhere in the class: `__call()` parses each name the first time it is used and the compiled manifest answers from then on.

`EloquentWalletRepository::findByOwnerId()` is the real, shipped version of exactly this pattern — the only difference is that it `implements WalletRepository`, so its body calls `dispatchQuery()` explicitly instead of relying on `__call()`, for the variance reason explained above. Either way, `findByOwnerId('alice')` compiles to `WHERE owner_id = ?` with no SQL written by hand.

💡 WHEN A NAME WOULD GET SILLY, USE `#[Query]`

Derived names read well up to two or three predicates. Past that, reach for the explicit-query escape hatch below instead of a fifty-character method name.

The `#[Query]` explicit-query escape hatch

A method can declare its SQL directly, bypassing the derived-query grammar entirely:

```
final class Query
{
    public function __construct(
        public string $sql,
        public bool $native = false,
    ) {}
}
```

```
/**
 * // ...
 * @return list<array<string, mixed>>
 */
#[Query('select * from records where email = :email order by amount asc')]
public function findByEmailRaw(string $email): array
{
    $rows = $this->dispatchQuery(__FUNCTION__, func_get_args());
}
```

```

    assert(is_array($rows));

    /** @var list<array<string, mixed>> $rows */
    return $rows;
}

```

The body is the same one line every declared repository method carries: hand the method's own name and its arguments to the shared dispatcher, which finds the method in the manifest's `#[Query]` set and runs the SQL. The `assert()` and the `@var` are there for PHPStan, which cannot see through `mixed`.

Named `:param` placeholders rewrite to positional `?` in appearance order, and the method's own arguments bind positionally. `#[Query]` methods are discovered by the same `TransactionalScanner` that compiles the `#[Transactional]` manifest (the next chapter's section on transactions covers that scanner in full), so a declared method's SQL resolves from the compiled manifest with zero per-request reflection — both the derived-query path and an explicit `#[Query]` method funnel through the single `dispatchQuery()` method shared by `__call()` and any method body that calls it directly.

Specification: composable, reusable predicates

Derived queries answer fixed questions. A **Specification** is a reusable predicate you can name once and compose freely at the call site — "wallets with at least this balance," combined with other conditions as needed:

```

interface Specification
{
    // ...
}

```

```

    public function toBuilder(Builder $query): Builder;
}

```

Specifications is the static factory — PHP interfaces cannot carry static factory bodies:

```

Specifications::allOf(...$specifications); // AND-folds left-to-right; zero-arg = match-all
Specifications::anyOf(...$specifications); // OR-folds left-to-right; zero-arg = match-all
Specifications::not($specification);
Specifications::where(fn (Builder $q) => $q->where('status', 'open'));

```

Applying a built specification runs through two repository methods every **EloquentRepository** already provides:

```

    public function findBySpecification(Specification $specification): array
    {
        $query = $this->reading(__FUNCTION__);

        return $this->translating(fn (): array => $this->narrow($specification->toBuilder($query)->get()->all()));
    }
    // ...
    public function findBySpecificationPaged(Specification $specification, Pageable $pageable): Page
    {
        $query = $this->reading(__FUNCTION__);

        return $this->translating(fn (): Page => $this->pageOf($specification->toBuilder($query), $pageable));
    }
}

```


Four helpers carry everything those two bodies do not spell out, and the split in their visibility is the point. `reading()` and `translating()` are **protected** — they are the seam a subclass repository reaches for, which is what makes the "annotate an override" recipe below available to you at all — while `pageOf()` and `narrow()` are **private** plumbing. `reading(__FUNCTION__)` opens the query and applies whatever `#[EntityGraph]` the compiled manifest holds *for this repository class and this method*, which is why annotating an override is the whole recipe. `narrow()` is the one the paged body above does not call: it keeps only the rows that really are `TModel` and hands back a `list<TModel>`. `pageOf()` counts over a clone of the builder and then windows it, so the count and the slice see the same predicate. And `translating()` wraps the terminal call so a driver failure leaves the repository as a typed `DataAccessException` rather than a raw `QueryException` — the subject of this chapter's last section.

A specification for wallets above a minimum balance, composed with a currency filter, reads exactly like the rule it expresses:

```
$rich = Specifications::where(
    fn (Builder $q) => $q->where('balance_minor', '>=', 100_000)
);
$inEur = Specifications::where(fn (Builder $q) => $q->where('currency', 'EUR'));
$richEurWallets = Specifications::allOf($rich, $inEur);
```

`$repo->findBySpecificationPaged($richEurWallets, Pageable::of(1, 20))` runs that composite predicate, counts the matches, sorts, and slices — returning a `Page` with no SQL of your own beyond the two `where()` closures.

Query by example

Sometimes the predicate you want is simply *"another row that looks like this one."*

Spring Data calls that an `Example`, and `firefly/data` ports it in the way that costs nothing to learn: **an `Example` is a `Specification`**. It has no separate call path, no special repository method it can only reach through, no rules of composition of its own. Everything the previous section showed applies to it unchanged.

```
final readonly class Example implements Specification
{
    private const string IDENTIFIER = '/^[A-Za-z_][A-Za-z0-9_]*$/D';
    // ...
    public static function of(object|array $probe, ?ExampleMatcher $matcher = null): self
    {
        $attributes = match (true) {
            $probe instanceof Model => [...$probe->getRawOriginal(), ...$probe->getDirty()],
            is_object($probe) => get_object_vars($probe),
            default => $probe,
        };
        // ...
        return new self($validated, $matcher ?? ExampleMatcher::matching());
    }
}
```

The **probe** is the thing that looks like what you want. It can be an Eloquent model with some attributes set, any plain object (its public properties), or a bare `column => value` array. A model probe contributes exactly the attributes that were set — `new Record(['status' => 'open'])` is a one-property probe, not a probe of every column in the table — because the attributes are read as the model's raw originals overlaid with its dirty set.

Every probe key is validated as a bare identifier right there in `of()`, and an invalid one is an `InvalidArgumentException` before any query is built. That is not defensive decoration: a key reaches a raw SQL fragment when case is ignored or a `LIKE` needs its `ESCAPE` clause, so it is checked at the only place where checking it once is enough — the same discipline the derived-query parser applies to field names.

Because an `Example` is a `Specification`, the repository methods you already know take it directly, and there are four convenience methods named the way a Spring Data reader expects:

```
$repo = new RecordRepository;

expect($repo->findByExample(Example::of(['status' => 'open', 'amount' => 150])))
->toHaveCount(1)
    ->and($repo->findByExample(Example::of(new Record(['status' => 'closed']))))
->toHaveCount(3)
    ->and($repo->countByExample(Example::of(['status' => 'open'])))>toBe(3)
    ->and($repo->existsByExample(Example::of(['status' => 'archived'])))>toBeFalse()
    ->and($repo->findByExample(Example::of([])))>toHaveCount(6);
```

An empty probe matches everything, which is the honest answer rather than an error: no properties means no predicate. `findOneByExample()` is the at-most-one variant — `null` for none, the row for one, and `IncorrectResultSizeDataAccessException` for more than one, because an ambiguous probe is a programming error and not a result:

```
expect($repo->findOneByExample(Example::of(['amount' => 150]))?->email)->toBe('b@x.test')
    ->and($repo->findOneByExample(Example::of(['amount' => 999])))>toBeNull()
    ->and(fn () => $repo->findOneByExample(Example::of(['status' => 'open'])))
->toThrow(IncorrectResultSizeDataAccessException::class);
```

The matcher carries the rules

A probe on its own says *what* to compare. An `ExampleMatcher` says *how*. It is an immutable value object whose every `with*` returns a new instance, so a matcher is safe to build once and share:

```
public function withIgnorePaths(string ...$paths): self
{
    return $this->copy(ignoredPaths: array_values(array_unique([...$this->ignoredPaths, ...
    $paths])));
}

/** No paths: every string property ignores case. With paths: only those. */
public function withIgnoreCase(string ...$paths): self
{
    return $paths === []
        ? $this->copy(ignoreCaseAll: true)
        : $this->copy(ignoreCasePaths: array_values(array_unique([...$this->
    ignoreCasePaths, ...$paths])));
}

public function withIncludeNullValues(): self
{
    return $this->copy(includeNullValues: true);
}

public function withStringMatcher(StringMatcher $matcher): self
{
    return $this->copy(defaultStringMatcher: $matcher);
}

public function withMatcher(string $path, GenericPropertyMatcher $matcher): self
{
    return $this->copy(propertyMatchers: [...$this->propertyMatchers, $path => $matcher]);
}
```

The rules those five combinators express, in full:

Rule	Default	How to change it
How properties fold together	<code>AND (matching() = matchingAll())</code>	<code>ExampleMatcher::matching Any()</code> for <code>OR</code>
A <code>null</code> probe value	skipped entirely	<code>withIncludeNullValues()</code> makes it <code>IS NULL</code>
A property you do not want compared	compared	<code>withIgnorePaths('id', 'created_at')</code>
How a string value is compared	<code>StringMatcher::EXACT (=)</code>	<code>withStringMatcher(String Matcher::CONTAINING)</code> and friends
Case sensitivity	case-sensitive	<code>withIgnoreCase()</code> for every string, <code>withIgnoreCase('email')</code> for one
One property that needs its own rules	follows the defaults above	<code>withMatcher('email', GenericPropertyMatcher::startsWith()- >caseSensitive())</code>

Only **string** values get string matching. An `int`, a `float` or a `bool` is always compared with `=`, whatever the default string matcher says — a `CONTAINING` matcher does not quietly turn `amount => 5` into `LIKE '%5%'`.

A per-path matcher beats the matcher-wide settings for that path and leaves every other path alone:

```
$matcher = ExampleMatcher::matching()
    ->withIgnoreCase()
    ->withMatcher('email', GenericPropertyMatcher::startsWith()->caseSensitive());
```

```

expect($matcher->isIgnoreCase('status'))->toBeTrue()
->and($matcher->isIgnoreCase('email'))->toBeFalse()
->and($matcher->stringMatcherFor('email'))->toBe(StringMatcher::STARTING)
->and($matcher->stringMatcherFor('status'))->toBe(StringMatcher::EXACT)

```

`StringMatcher` has four cases — `EXACT`, `CONTAINING`, `STARTING`, `ENDING` — and the three `LIKE` ones escape the value before it goes anywhere near a pattern:

```

public function pattern(string $value): string
{
    $escaped = str_replace(['!', '%', '_'], ['!!!', '!', '_'], $value);

    return match ($this) {
        self::EXACT => $escaped,
        self::CONTAINING => '%'.$escaped.'%',
        self::STARTING => $escaped.'%',
        self::ENDING => '%'.$escaped,
    };
}

```

So a probe of `under_` searches for a literal underscore, not for "any character". The fragment `Example` emits carries `ESCAPE '!'` — the one escape spelling that reads the same on `sqlite`, `mysql`, `pgsql` and `sqlsrv`, which a backslash does not (`mysql` consumes it inside the string first).

Spring's `REGEX` matcher is deliberately absent: there is no portable SQL for it.

It composes, because it is a specification

The point of porting `Example` as a `Specification` rather than beside one is this — an example can be one leaf of a larger predicate, and it pages exactly like any other:

```

$page = $repo->findByExamplePaged(Example::of(['status' => 'open']), Pageable::of(1, 2,
Sort::by('amount')->descending()));

expect($page->total)->toBe(3)
  ->and($page->items)->toHaveCount(2)
  ->and($page->items[0]->amount)->toBe(150);

$combined = Specifications::allOf(
  Example::of(['status' => 'closed']),
  Specifications::where(static fn (Builder $q): Builder => $q->where('amount', '>', 100)),
);

expect($repo->findBySpecification($combined))->toHaveCount(1);

```

💡 WHERE QUERY BY EXAMPLE EARNS ITS KEEP

An admin search form with eight optional fields is the canonical case. Build the probe from whatever the user filled in, ignore the rest, and you have written no conditional query-building code at all — the empty fields simply are not in the probe.

Saying more about a method: `#[Modifying]`, `#[Projection]`, `#[Lock]`, `#[EntityGraph]`

Four attributes let a repository method say something the method *name* cannot. Before any of them, one rule that catches every reader exactly once:

⚠ THESE ATTRIBUTES NEED A DECLARED METHOD

PHP cannot attach an attribute to a method that is never declared, and a derived query reached through `__call()` is never declared — it exists only as an `@method` docblock tag. So a method carrying one of these four attributes has a real body, and that body is always the same one line: `return $this->dispatchQuery(__FUNCTION__, func_get_args());` (plus whatever `assert()` PHPStan needs). Declaring the method changes nothing about how it resolves; it only gives the attribute somewhere to live.

The scanner records all four into the compiled manifest at build time, and `EloquentRepository` honours them at dispatch with **no reflection at request time** — a `#[Projection]`'s DTO constructor, for instance, is reflected exactly once, when `firefly:cache` compiles the manifest.

#[Modifying] — this method writes

```
/** A statement that needs a transaction: returns the affected-row count. */
#[Modifying]
#[Query('update records set status = :status where amount < :amount')]
public function closeSmall(string $status, int $amount): int
{
    $affected = $this->dispatchQuery(__FUNCTION__, func_get_args());
    assert(is_int($affected));

    return $affected;
}

/** A statement that may run without a transaction. */
#[Modifying(requiresTransaction: false)]
#[Query('delete from records where status = :status')]
public function purgeStatus(string $status): int
{
    $affected = $this->dispatchQuery(__FUNCTION__, func_get_args());
    assert(is_int($affected));
}
```



```
    return $affected;
}
```

The SQL runs as a statement rather than a query, and the method returns the affected-row count. Two things are refused at *scan* time, so you hear about them from `firefly:cache` or the first boot and never from a user's request: `#[Modifying]` without a `#[Query]` (a derived `deleteBy...` is already a statement and needs no attribute), and a `#[Query]` whose SQL is a `SELECT`, `WITH` or `VALUES`. At run time the statement refuses to execute outside an active transaction on the model's connection — a `TransactionRequiredException` — unless you opt out with `requiresTransaction: false`.

Spring's `clearAutomatically` is accepted for source compatibility and does nothing. Eloquent has no persistence context to clear, and pretending otherwise would be the worst kind of parity.

`#[Projection]` — select less, hydrate a DTO

```
/** A class-based projection over three `records` columns; `email` is nullable in the table
and here. */
final readonly class RecordSummary
{
    public function __construct(
        public int $id,
        public ?string $email,
        public int $amount,
    ) {}
}
```

```
#[Projection(RecordSummary::class)]
public function findByStatusOrderByAmountAsc(string $status): array
{
```

```

    $rows = $this->dispatchQuery(__FUNCTION__, func_get_args());
    assert(is_array($rows));

    /** @var list<RecordSummary> $rows */
    return $rows;
}

```

On a **derived** method the `SELECT` list is inferred from the DTO's constructor parameters (or given explicitly with `columns:`); on a `#[Query]` method the SQL owns its own select list. Each row is hydrated through the constructor: `snake_case` column to `camelCase` parameter, scalars coerced from whatever the driver returned, a backed enum from its backing value, a `DateTimeImmutable` from the string, and an optional parameter whose column is absent takes its default.

Coercion is **lossless or refused**. `'150.75'` into an `int`, a backing value the enum has no case for, a string that is not a date — each is a `ConfigurationException` naming the DTO, the column, the value *and* the parameter, never a bare `TypeError` from somewhere deep in PHP. A missing required column and a `NULL` arriving at a non-nullable parameter are the same kind of error, and the columns are checked against the table before the query runs (sqlite would otherwise read an unknown double-quoted identifier as a string literal and hand back nonsense).

Spring's interface projections (`interface RecordSummary { String getEmail(); }`) are not offered. PHP has no runtime proxy that could implement an interface by column name; declare the DTO class instead.

`#[Lock]` — take the row lock the read needs

```

/** @return list<Record> */
#[Lock(LockMode::PESSIMISTIC_WRITE)]
public function findByEmail(string $email): array
{

```

```

    $rows = $this->dispatchQuery(__FUNCTION__, func_get_args());
    assert(is_array($rows));

    /** @var list<Record> $rows */
    return $rows;
}

```

`PESSIMISTIC_WRITE` compiles to `lockForUpdate()`, `PESSIMISTIC_READ` to `sharedLock()`, spelled by the connection's own grammar. Both refuse to run outside an active transaction — Spring's behaviour, and the right one: a row lock is released when the transaction ends, so outside one it guards nothing. `$repo->findByIdForUpdate($id)` is the programmatic twin of the attribute, and `sqlite`, which has no row locks at all, compiles the clause to nothing and simply succeeds.

The scanner refuses `#[Lock]` on a `#[Query]` method: raw SQL carries its own locking clause, and two of them would fight.

`#[EntityGraph]` — eager-load, declaratively

```

/** @var array<string, list<string>> */
protected array $entityGraphs = ['Record.full' => ['entries']];
// ...
/** @return list<Record> */
#[EntityGraph(attributePaths: ['entries'])]
public function findByStatusOrderByIdDesc(string $status): array
{
    $rows = $this->dispatchQuery(__FUNCTION__, func_get_args());
    assert(is_array($rows));

    /** @var list<Record> $rows */
    return $rows;
}

```

It maps to Eloquent's `with()`, and it is the one attribute that reaches methods you did not write. Every inherited read that **opens a builder** — `findById`, `findAll`, `findAllById`, `findPaged`, `findSorted`, `findSlice`, `findBySpecification`, `findByExample`, `findOneByExample`, `findByIdForUpdate` and the rest — starts from `reading(__FUNCTION__)`, which asks the manifest for the graph registered against *this repository class and that method name*. So annotating an override that does nothing but `return parent::findAll();` is the entire recipe, exactly as it is in Spring. The four reads that answer a question *about* rows rather than returning them — `existsById()`, `count()`, `existsByExample()`, `countByExample()` — go straight to `query()` and deliberately never touch `reading()`: an entity graph means nothing to a `COUNT(*)`, and eager-loading relations to throw them away would be pure cost. A named graph the repository never declared in `$entityGraphs` is a `ConfigurationException` at first use.

⚠ TWO COMBINATIONS THAT DO NOT WORK THE WAY YOU WOULD HOPE

A `#[Projection]` and a trailing `Pageable` **do not combine** — the projection returns the whole, unpaginated result. Page the entities, or put a `LIMIT` in the `#[Query]`. And `#[Query]` binds **positionally**: named `:placeholders` are rewritten to `?` in order of first appearance, so a placeholder used twice needs the argument passed twice.

Pagination: `Page`, `Pageable`, and `Sort`

Three small, immutable value objects carry a page request in and a page result out, end to end. `Pageable` is the request — a 1-based page number, a size, and an optional `Sort`:

```
final readonly class Pageable
{
```

```

    public function __construct(
        public int $page = 1,
        public int $size = 20,
        public ?Sort $sort = null,
    ) {
        if ($page < 1) {
            throw new InvalidArgumentException('Page number is 1-based and must be >= 1.');
```

```
        }

        if ($size < 1) {
            throw new InvalidArgumentException('Page size must be >= 1.');
```

```
        }
    }

    public static function of(int $page, int $size, ?Sort $sort = null): self
    {
        return new self($page, $size, $sort);
    }

    public static function unpaged(?Sort $sort = null): self
    {
        return new self(1, PHP_INT_MAX, $sort);
    }

    public function offset(): int
    {
        return ($this->page - 1) * $this->size;
    }

```

`Sort` and `Order` compose an ordering out of small, immutable pieces — every combinator returns a *new* `Sort`, never mutates the one you started with:

```

final readonly class Sort
{
    /**
     * @param list<Order> $orders
     */

```

```

    public function __construct(public array $orders = []) {}
    // ...
    public static function by(string ...$properties): self
    {
        return new self(array_map(
            static fn (string $property): Order => Order::asc($property),
            array_values($properties),
        ));
    }
    // ...
    public function descending(): self
    {
        return new self(array_map(
            static fn (Order $order): Order => Order::desc($order->property),
            $this->orders,
        ));
    }
}

```

`Order::asc('created_at') / Order::desc('created_at')` pair a property name with a `Direction::Asc / Direction::Desc` enum whose backing *string value* is the Eloquent `orderBy()` direction — `$order->direction->value` needs no translation table at all.

`Page` is what comes back out — the row slice plus everything a client needs to render a pager:

```

/**
 * ...
 * @template T
 */
final readonly class Page
{
    /**
     * @param list<T> $items
     */
}

```

```

    public function __construct(
        public array $items,
        public int $total,
        public int $page = 1,
        public int $size = 20,
    ) {}

    public function totalPages(): int
    {
        return $this->size > 0 ? (int) ceil($this->total / $this->size) : 0;
    }

    public function hasNext(): bool
    {
        return $this->page < $this->totalPages();
    }

    // ...
    /**
     * @template U
     *
     * @param callable(T): U $mapper
     * @return self<U>
     */
    public function map(callable $mapper): self
    {
        return new self(array_map($mapper, $this->items), $this->total, $this->page, $this->size);
    }
}

```

`EloquentRepository::findPaged()` builds a `Page` at the Eloquent edge: a `count()` query for `total`, then `skip($pageable->offset())->take($pageable->size)` for the slice, with `applySort()` folding each `Sort` order into an `orderBy()` call in between. `Page::map()` is what a query handler reaches for to turn a page of entities into a page of DTOs **without losing the pagination metadata** — the same `total / page / size` numbers survive the transformation untouched.

Slice : the page that does not count

`total` is expensive. On a large table, `SELECT COUNT(*)` over the same predicate can cost more than fetching the page itself, and an infinite-scroll list never displays the number anyway — it only needs to know whether to keep the "load more" button. That is what a `Slice` is: everything `Page` has except `total`, plus the one fact the button needs.

```
final readonly class Slice
{
    /**
     * @param list<T> $items
     */
    public function __construct(
        public array $items,
        public bool $hasNext,
        public int $page = 1,
        public int $size = 20,
    ) {}

    public function hasNext(): bool
    {
        return $this->hasNext;
    }

    public function hasPrevious(): bool
    {
        return $this->page > 1;
    }

    public function numberOfElements(): int
    {
        return count($this->items);
    }

    public function nextPageable(): Pageable
    {

```



```

    return Pageable::of($this->page + 1, $this->size);
}

```

`hasNext` is not guessed and it is not counted. The repository fetches **one row more** than the page size, and if that extra row came back it is dropped and remembered as "there is a next page":

```

private function sliceOf(Builder $query, Pageable $pageable): Slice
{
    $take = $pageable->isPaged() ? $pageable->size + 1 : PHP_INT_MAX;

    $rows = $this->applySort($query, $pageable->sort)
        ->skip($pageable->offset())
        ->take($take)
        ->get()
        ->all();

    $hasNext = $pageable->isPaged() && count($rows) > $pageable->size;
}

```

So a slice is one cheap query where a page is one cheap query *plus* one expensive one. `$repo->findSlice(Pageable::of(3, 20, Sort::by('id')))` is the port's own count-free page; `Slice::nextPageable()` hands you the request for the following one, and `Slice::map()` transforms the items exactly as `Page::map()` does.

A derived method can return a slice too — and here the two halves of the rule matter separately. A **trailing `Pageable` argument** is what makes a derived method page at all; the **declared return type** is what makes it a `Slice` rather than a `Page`:

```

#[EntityGraph(attributePaths: ['entries'])]
public function findByStatusOrderByIdAsc(string $status, Pageable $pageable): Slice
{
    $slice = $this->dispatchQuery(__FUNCTION__, func_get_args());
    assert($slice instanceof Slice);
}

```

```

    /** @var Slice<Record> $slice */
    return $slice;
}

```

An undeclared `@method` derived query has no declared return type to read, so a `Pageable` on one always produces a `Page`. That is the same rule the four attributes obey, arriving from a different direction.

Soft delete, auditing, and optimistic locking

Three more building blocks live on `EloquentRepository`, layered on Eloquent's own mechanisms rather than reinvented. None of them is exercised by `Wallet` in this book, but each is one `use` statement away on any model that needs it.

Soft delete reuses Eloquent's native `SoftDeletes` trait as-is:

```

final class SoftRecord extends Model
{
    use SoftDeletes;
    // ...
}

```

Once a model uses it, `delete()` / `deleteById()` through the repository soft-deletes the row, and every ordinary read transparently excludes trashed rows via Eloquent's own global scope. `findAllIncludingDeleted()` and `restore(mixed $id): ?object` round out the lifecycle.

Auditing is an opt-in `Auditable` trait that stamps `created_by` / `updated_by` on write, driven by an `AuditorAware` port:

```
interface AuditorAware
{
    public function currentAuditor(): int|string|null;
}
```

Optimistic locking guards concurrent writes with a `version` integer column via `HasOptimisticLock`, which overrides Eloquent's internal `performUpdate()` to add `WHERE version = <loaded version>` to every `UPDATE` — a write against a stale version matches zero rows and throws `OptimisticLockException` instead of silently overwriting someone else's change. That exception extends the kernel's `OptimisticLockingFailureException`, so a `catch` on either name works and the next section's rule holds for it too.

When the driver fails: the `DataAccessException` family

Everything so far has assumed the database says yes. This section is about what reaches your code when it says no — and the framework's answer is Spring's: **a driver failure never leaves `firefly/data` as a driver failure.**

Laravel's own answer is `QueryException`. It is one class for every possible fault, and two things about it make it a poor thing to hand an application. The first is that you cannot branch on it: a duplicate e-mail address and an unreachable database server arrive as the same type, so telling them apart means matching on driver error codes at the call site, once per driver you support. The second is worse. A `QueryException`'s message is the failing statement **with its bindings interpolated** — which is to say the e-mail address, the token or the tenant id that was being written. Anything that renders that message to a client, or logs it at a level somebody else can read, has leaked user data.

`PersistenceExceptionTranslator` fixes both at once:

```
private static function build(?string $kind, Throwable $cause, ?string $sqlState):
    DataAccessException
{
    $translated = match ($kind) {
        DriverErrorTable::DUPLICATE_KEY => new DuplicateKeyException(previous: $cause),
        DriverErrorTable::INTEGRITY => new DataIntegrityViolationException(previous:
$cause),
        DriverErrorTable::DEADLOCK => new DeadlockLoserDataAccessException(previous:
$cause),
        DriverErrorTable::LOCK => new CannotAcquireLockException(previous: $cause),
        DriverErrorTable::TIMEOUT => new QueryTimeoutException(previous: $cause),
        DriverErrorTable::TRANSIENT => new TransientDataAccessResourceException(previous:
$cause),
        DriverErrorTable::RESOURCE => new DataAccessResourceFailureException(previous:
$cause),
        DriverErrorTable::GRAMMAR => new BadSqlGrammarException(previous: $cause),
        default => new DataAccessException('The database refused the operation.', previous:
$cause),
    };

    return $sqlState === null ? $translated : $translated->withExtensions(['sqlState' =>
$sqlState]);
}
```

Read what is *not* in that method. No `$cause->getMessage()` is copied into the new exception. Each of those constructors carries its own **fixed sentence**, written once, with no statement and no bindings in it. The driver's own text stays exactly where it belongs — on `previous`, for the log and the stack trace — and the one piece of the driver's answer that is both harmless and useful, the SQLSTATE, rides along as the `sqlState` extension member of the problem document.

The type is the diagnosis, and it is what a caller branches on:

Thrown	Extends	HTTP	When
<code>DuplicateKeyException</code>	<code>DataIntegrityViolationException</code>	409	a unique or primary-key violation
<code>DataIntegrityViolationException</code>	<code>DataAccessException</code>	409	any other constraint: not-null, foreign key, check
<code>DeadlockLoserDataAccessException</code>	<code>CannotAcquireLockException</code>	409	this transaction was chosen as the deadlock victim
<code>CannotAcquireLockException</code>	<code>DataAccessException</code>	409	a lock wait timed out
<code>QueryTimeoutException</code>	<code>DataAccessException</code>	504	the statement ran past its timeout
<code>TransientDataAccessResourceException</code>	<code>DataAccessException</code>	503	the connection dropped — retrying may work
<code>DataAccessResourceFailureException</code>	<code>DataAccessException</code>	503	the datasource is not there at all
<code>BadSqlGrammarException</code>	<code>DataAccessException</code>	500	the statement is wrong; retrying cannot help
<code>DataAccessException</code>	<code>InfrastructureException</code>	500	a failure the tables do not recognise

The hierarchy is the point. A handler that cares only about "the write conflicted" catches `DataIntegrityViolationException` and gets duplicate keys too; one that cares about "anything the database refused" catches `DataAccessException` and gets all nine.

Because every one of them is a `FireflyException`, Chapter 4's problem-details renderer already knows the status code and the error code without a single line of mapping — `DUPLICATE_KEY`, `QUERY_TIMEOUT`, `BAD_SQL_GRAMMAR` and the rest travel to the client as `type / code`, and the statement does not travel anywhere.

Two more members of the family come from the repository rather than from a driver:

`getId($id)` throws `EmptyResultDataAccessException` (404) exactly where `findById()` would have returned `null`, and `findOneByExample()` throws `IncorrectResultSizeDataAccessException` when the probe matched more than one row.

Where the translation happens, and where it deliberately does not

```
public function translate(Throwable $e, ?string $driver = null): Throwable
{
    if (! $this->enabled || $e instanceof DataAccessException) {
        return $e;
    }
    // ...
    if (! $e instanceof PDOException) {
        return $e;
    }
    // ...
    $driver ??= $e instanceof QueryException ? self::driverOf($e->getConnectionName()) :
    null;
    $sqlState = self::sqlState($e);
    $kind = DriverErrorTable::kindFor($driver, self::driverCode($e), $sqlState);
```

There are exactly two seams a database error can leave `firefly/data` through, and both `translate`: every `EloquentRepository` method (that is what the `translating()` wrapper you saw around `save()` and `findBySpecification()` is doing), and `TransactionTemplate::execute()` — which means every `#[Transactional]` method

in the next chapter, since the proxy delegates to the template. A raw `DB::` call you make outside both still throws Laravel's `QueryException` exactly as it always did. Nothing is patched globally, and nothing is monkey-patched at all.

Three guards keep that narrow. A throwable **already** in the family is returned untouched, so a repository call translating inside a template that translates again is idempotent rather than double-wrapped. Anything that is not a `PDOException` passes straight through — with an exception for the four failures Laravel itself has already classified (`UniqueConstraintViolationException` , `SQLiteDatabaseDoesNotExistException` , `LostConnectionException` , `DeadlockException`), whose own answer is at least as good as any table's, and which are looked for **on the cause as well** because Laravel wraps whatever the query callback throws. And the whole mechanism is one key away from off: `firefly.data.exception-translation.enabled` , `true` by default.

Which code table applies is decided by the driver name. The tables themselves — driver error codes for sqlite, mysql, mariadb and sqlsrv, exact SQLSTATEs, and SQLSTATE *classes* as the fallback — live in a single file, `DriverErrorTable` , with one test row per table row. sqlite needs one extra step, and it is in the source rather than hidden: sqlite reports every constraint violation as code 19, so the unique/primary-key family is only visible in the message text, and that is the one place the translator reads a driver message at all.

💡 THE RULE WORTH REMEMBERING

Catch the **type**, log the **previous** , render the **fixed sentence**. If you ever find yourself reading `$e->getMessage()` to decide what happened, the type you needed already exists.

The domain-event bridge

`EloquentRepository::save()` does one thing beyond persisting the row that matters enormously for Chapter 6: if the saved entity also implements `RecordsDomainEvents` — which `Wallet` does — **and** a transaction is currently active on that entity's own connection, `save()` registers the entity with `AggregateTracker`, `firefly/data`'s unit-of-work registry:

```
public function save(object $entity): object
{
    return $this->translating(function () use ($entity): object {
        if ($entity instanceof Model) {
            $entity->save();
        }

        if ($entity instanceof RecordsDomainEvents
            && $this->tracker !== null
            && $this->connectionFor($entity)->transactionLevel() > 0) {
            $this->tracker->track($entity);
        }

        return $entity;
    });
}
```

That registration is what lets the framework drain and publish a `Wallet`'s raised domain events after the surrounding `#[Transactional]` method commits — and never if it rolls back. Chapter 6 builds the aggregate that raises those events; this is the seam, on the persistence side, that catches them.

✓ What you learned

Concept	What it does
<code>CrudRepository</code> / <code>PagingAndSortingRepository</code>	The persistence-agnostic ports every repository ultimately implements
<code>EloquentRepository</code>	The Eloquent-backed base: set <code>\$model</code> , inherit full CRUD + paging + sorting
Derived query (<code>findByOwnerId</code>)	A bodiless method name parsed by <code>DerivedQueryParser</code> and driven against the Eloquent <code>Builder</code>
<code>#[Query('sql')]</code>	An explicit-SQL escape hatch, discovered at scan time, resolved with zero per-request reflection
<code>Specification</code> / <code>Specifications</code>	Composable predicates (<code>allOf</code> / <code>anyOf</code> / <code>not</code> / <code>where</code>) applied via <code>findBySpecification(Paged)</code>
<code>Example</code> / <code>ExampleMatcher</code>	Query by example, ported as a <code>Specification</code> , so it composes and pages like any other predicate
<code>#[Modifying]</code> / <code>#[Projection]</code> / <code>#[Lock]</code> / <code>#[EntityGraph]</code>	What a declared method can say that its name cannot: writes, DTO hydration, row locks, eager loading
<code>Page</code> / <code>Pageable</code> / <code>Sort</code> / <code>Order</code>	Immutable value objects carrying a page request in and a page result (with metadata) out
<code>Slice</code>	The same page without the count query: one row over-fetched answers <code>hasNext</code>

Concept	What it does
<code>SoftDeletes</code> / <code>Auditable</code> / <code>HasOptimisticLock</code>	Opt-in traits for soft-delete, <code>created_by</code> / <code>updated_by</code> stamping, and version-guarded writes
<code>DataAccessException</code> family	A driver failure becomes a typed exception with a fixed sentence; the statement never leaves the log
The domain-event bridge	<code>save()</code> registers a <code>RecordsDomainEvents</code> entity with <code>AggregateTracker</code> while a transaction is active

Try it yourself

- Add a derived counter.** In a scratch copy of the project, declare `public function countByCurrency(string $currency): int` with no body on a repository extending `EloquentRepository`, and confirm calling it compiles to `SELECT COUNT(*) ... WHERE currency = ?` with no SQL of your own.
- Compose two specifications.** Build a `Specification` for "balance at least N" and another for "in currency C" with `Specifications::where(...)`, combine them with `Specifications::allOf(...)`, and run the composite through `findBySpecificationPaged` against a small seeded table.
- Turn a page into a slice.** Take the paged read from the previous exercise and ask for `findSlice(Pageable::of(1, 2))` instead. Enable the query log and confirm what the section claims: the slice runs **one** query where the page ran two, and `hasNext` is still right.
- Make the database say no.** Add a unique index to the wallets table, save two wallets with the same owner id, and catch the result. Check three things: the type is

`DuplicateKeyException`, `getMessage()` does *not* contain the owner id you wrote, and `getPrevious()` is Laravel's `QueryException` with the statement still on it.

5. **Read a page of wallets.** Call `EloquentWalletRepository::findAll(...)`'s paging counterpart (add a `findPaged` caller through `WalletRepository` in your scratch project) with `Pageable::of(1, 2, Sort::by('created_at')->descending())` against three seeded wallets, and confirm `total`, `totalPages`, and `hasNext` all match what you expect before and after `Page::map()`.

Domain-Driven Design

By the end of this chapter you will know the difference `firefly/domain` draws between an `Entity` and a `ValueObject`, how `AggregateRoot` (and its trait twin, `HasDomainEvents`) makes an object the sole source of its own domain events, how a `DomainEvent` is built and identified, how the `Wallet` aggregate enforces its own overdraft and currency rules with nothing outside it able to bypass them, and — closing the loop opened in Chapter 5 — exactly how a raised event travels from an aggregate method to a published fact on the other side of a committed transaction.

NEW TERM: INVARIANT

An **invariant** is a rule that must always hold, no matter which code path got there — for `Wallet`, "the balance is never negative" is an invariant, not a convention. An invariant lives *inside* the object it protects; a rule enforced only by a service that callers can bypass is not truly an invariant at all.

Entities and value objects

`firefly/domain` is deliberately the smallest package in the whole framework:

`Entity`, `ValueObject`, `AggregateRoot`, and `DomainEvent` — four plain-PHP building blocks with **zero framework dependency and zero reflection**. (A dedicated test greps the package's own source for `ReflectionClass` / `ReflectionMethod` / `getAttributes` and asserts no hits — your domain model never has to know LaraFly exists.)

`Entity` draws the first of DDD's two foundational distinctions: identity, not value, decides whether two entities are the same thing.

```

abstract class Entity
{
    public function __construct(protected int|string|null $id = null) {}

    public function id(): int|string|null
    {
        return $this->id;
    }

    public function isTransient(): bool
    {
        return $this->id === null;
    }

    public function equals(self $other): bool
    {
        if ($this === $other) {
            return true;
        }

        if ($this::class !== $other::class) {
            return false;
        }

        if ($this->isTransient() || $other->isTransient()) {
            return false;
        }

        return $this->id === $other->id;
    }
}

```

Two entities are `equals()` only when they are the same concrete class **and** have equal, non-null ids. A **transient** entity — one whose id is still `null`, not yet persisted — is equal only to itself, by plain object identity; two distinct transient instances are never

equal even with identical other state. PHP has no runtime generics, so `id()` is typed `int|string|null` — wide enough to cover both an auto-increment integer key and a string id like `Wallet`'s own `wlt-...`.

`ValueObject` is the second distinction: no identity at all, immutable by convention, equal purely by the values it holds.

```
interface ValueObject {}
```

It is a bare marker interface on purpose — a value object's *equality* is supplied separately, by a small trait:

```
trait ValueObjectEquality
{
    public function equals(self $other): bool
    {
        // ...
        return get_class($this) === get_class($other)
            && get_object_vars($this) == get_object_vars($other);
    }
}
```

`get_object_vars($this)`, not `ReflectionProperty` — deliberately, so `firefly/domain` stays reflection-free — compares every scope-visible property. This is exactly right for a **flat readonly** value object; a value object that nests another value object should compose equality by delegating to the nested object's own `equals()` rather than trusting `==` to recurse correctly through the outer property array.

i NEW TERM: AGGREGATE

An **aggregate** is a small cluster of objects that must stay consistent as a group, entered and mutated through exactly one object: its **aggregate root**. The next section builds one.

Money : the textbook value object

Money is `firefly/domain`'s `ValueObject` applied to Lumen's actual currency handling — real, shipped code from `samples/lumen/src/Domain/Money.php` :

```
<?php

declare(strict_types=1);

namespace Lumen\Domain;

use Firefly\Domain\ValueObject;
use Firefly\Domain\ValueObjectEquality;
use Firefly\Kernel\Exception\Business\ConflictException;

final readonly class Money implements ValueObject
{
    use ValueObjectEquality;

    public function __construct(public int $minorUnits, public Currency $currency) {}

    public static function zero(Currency $currency): self
    {
        return new self(0, $currency);
    }

    public function add(Money $other): self
    {
        $this->assertSameCurrency($other);

        return new self($this->minorUnits + $other->minorUnits, $this->currency);
    }
}
```



```

    public function subtract(Money $other): self
    {
        $this->assertSameCurrency($other);

        return new self($this->minorUnits - $other->minorUnits, $this->currency);
    }

    public function isPositive(): bool
    {
        return $this->minorUnits > 0;
    }

    public function isNegative(): bool
    {
        return $this->minorUnits < 0;
    }

    public function __toString(): string
    {
        return number_format($this->minorUnits / 100, 2, '.', '').' '.$this->currency->value;
    }

    private function assertSameCurrency(Money $other): void
    {
        if ($this->currency !== $other->currency) {
            throw new ConflictException(
                "cannot combine {$this->currency->value} with {$other->currency->value}"
            );
        }
    }
}

```

Three design choices here repay close reading. `$minorUnits` is stored as an **integer** — cents, not a `float` — precisely so financial arithmetic never accumulates rounding error; `Money(1050, Currency::EUR)` is exactly €10.50, and `add / subtract` operate

on that integer directly. `add()` and `subtract()` both return a **new** `Money` instance — `final readonly class` makes this the only option, not a discipline you have to remember — so a deposit never mutates the amount it started from; nothing else in the codebase can be silently holding a stale reference to a `Money` that just changed underneath it. And `assertSameCurrency()` is a private helper both public methods route through first, so combining EUR and USD amounts is a `ConflictException` raised at the exact call site of the mistake, never a silently-wrong sum discovered during reconciliation.

`final readonly class Money implements ValueObject { use ValueObjectEquality; }` is the entire equality story — two `Money(1050, Currency::EUR)` instances constructed independently are `equals()` to each other, because `ValueObjectEquality` compares their public state, not their object identity.

LARAVEL PARITY

`ValueObject` + `ValueObjectEquality` map to the `@ValueObject` / `@Embeddable` idea from the JPA world, and to PHP's own `readonly` class feature doing the immutability half of the job. In a plain Laravel/Eloquent application, the nearest equivalent to `Money` is an Eloquent cast object — except `Money` here has no idea Eloquent exists, and is exercised in a unit test with two constructor arguments and nothing else.

The aggregate root: the only source of its own events

`Money` solves representation. Something still has to *own* the decision of whether a deposit or withdrawal is allowed at all — that is the aggregate root's job.

`AggregateRoot` extends `Entity` and adds exactly one thing: a private buffer of pending domain events, and the single protected method that appends to it.

```

abstract class AggregateRoot extends Entity implements RecordsDomainEvents
{
    /** @var list<DomainEvent> */
    private array $pendingEvents = [];

    protected function raiseEvent(DomainEvent $event): void
    {
        $this->pendingEvents[] = $event;
    }

    /** @return list<DomainEvent> */
    public function pendingEvents(): array
    {
        return $this->pendingEvents;
    }

    /** @return list<DomainEvent> */
    public function pullEvents(): array
    {
        $events = $this->pendingEvents;
        $this->pendingEvents = [];

        return $events;
    }

    public function clearEvents(): void
    {
        $this->pendingEvents = [];
    }
}

```

`raiseEvent()` is `protected` — deliberately. Only the aggregate's *own* methods can call it, which is what makes an aggregate the **consistency boundary** the chapter introduction promised: no external caller can reach in and make the aggregate emit an event on its behalf, the same way no external caller can reach in and mutate its state without going through a method that enforces the rules. `pendingEvents()` is a non-

draining snapshot — safe to call from a test assertion or a log line without side effects.

`pullEvents()` drains the buffer and hands it to whoever asked, which is what the framework's after-commit machinery calls. `clearEvents()` drops the buffer without publishing anything — used on rollback.

`RecordsDomainEvents` is the interface that names exactly this drain-side contract, deliberately **without** `raiseEvent()` — because only the aggregate itself may raise its own events, so that method stays `protected` on whichever class implements the interface:

```
interface RecordsDomainEvents
{
    // ...
    public function pendingEvents(): array;
    // ...
    public function pullEvents(): array;
    // ...
    public function clearEvents(): void;
}
```

Why `Wallet` doesn't extend `AggregateRoot`

Here is where LaraFly's aggregate story has to solve a problem PyFly and Java never face: PHP has single inheritance, and `Wallet` needs to be a **persisted Eloquent model** as well as an aggregate root. It cannot `extends AggregateRoot` and `extends Model` at the same time. `HasDomainEvents` is the trait that closes this gap — the exact same buffer and the exact same four methods as `AggregateRoot`, but as a trait any class can `use`, regardless of what it already extends:

```
trait HasDomainEvents
{
    /** @var list<DomainEvent> */
```

```

    private array $pendingEvents = [];

    protected function raiseEvent(DomainEvent $event): void
    {
        $this->pendingEvents[] = $event;
    }

    /** @return list<DomainEvent> */
    public function pendingEvents(): array
    {
        return $this->pendingEvents;
    }

    /** @return list<DomainEvent> */
    public function pullEvents(): array
    {
        $events = $this->pendingEvents;
        $this->pendingEvents = [];

        return $events;
    }

    public function clearEvents(): void
    {
        $this->pendingEvents = [];
    }
}

```

Here is the whole, real `Wallet` aggregate — `samples/lumen/src/Domain/Wallet.php` — combining exactly this trait with `extends Model` and `implements RecordsDomainEvents`:

```

<?php

declare(strict_types=1);

```

```

namespace Lumen\Domain;

use Firefly\Domain\HasDomainEvents;
use Firefly\Domain\RecordsDomainEvents;
use Firefly\Kernel\Exception\Business\ConflictException;
use Illuminate\Database\Eloquent\Model;
use Lumen\Domain\Event\FundsDeposited;
use Lumen\Domain\Event\FundsWithdrawn;
// ...
use Lumen\Domain\Event\WalletOpened;

final class Wallet extends Model implements RecordsDomainEvents
{
    use HasDomainEvents;

    protected $table = 'wallets';

    public $incrementing = false;

    protected $keyType = 'string';

    /** @var list<string> */
    protected $fillable = ['id', 'owner_id', 'currency', 'balance_minor'];

    public $timestamps = false;

    public static function open(string $id, string $ownerId, Currency $currency): self
    {
        if (trim($ownerId) === '') {
            throw new ConflictException('owner_id is required');
        }

        $wallet = new self([
            'id' => $id,
            'owner_id' => $ownerId,
            'currency' => $currency->value,
            'balance_minor' => 0,
        ]);

        $wallet->raiseEvent(new WalletOpened($id, $ownerId, $currency->value));
    }
}

```

```

    return $wallet;
}

public function currency(): Currency
{
    $value = $this->getAttribute('currency');

    return Currency::from(match (true) {
        is_string($value) => $value,
        default => '',
    });
}

public function balanceMoney(): Money
{
    $value = $this->getAttribute('balance_minor');

    $minor = match (true) {
        is_int($value) => $value,
        is_numeric($value) => (int) $value,
        default => 0,
    };

    return new Money($minor, $this->currency());
}

/**
 * `getKey()` is declared `mixed` (a primary key may be any attribute type); Wallet's
 * own key is always the
 * string `id` set in open(), so this narrows for the event payload rather than casting
 * mixed directly.
 */
private function walletId(): string
{
    $key = $this->getKey();

    return match (true) {
        is_string($key) => $key,
    };
}

```

```

        is_int($key) => (string) $key,
        default => '',
    };
}

public function deposit(Money $amount): void
{
    $this->assertCurrency($amount);
    if (! $amount->isPositive()) {
        throw new ConflictException('deposit amount must be > 0');
    }
    $new = $this->balanceMoney()->add($amount);
    $this->setAttribute('balance_minor', $new->minorUnits);
    $this->raiseEvent(new FundsDeposited(
        $this->walletId(), $amount->minorUnits, $amount->currency->value, $new->minorUnits
    ));
}

public function withdraw(Money $amount): void
{
    $this->assertCurrency($amount);
    if (! $amount->isPositive()) {
        throw new ConflictException('withdrawal amount must be > 0');
    }
    $remaining = $this->balanceMoney()->subtract($amount);
    if ($remaining->isNegative()) {
        throw new ConflictException(
            "cannot withdraw {$amount}; balance is {$this->balanceMoney()}"
        );
    }
    $this->setAttribute('balance_minor', $remaining->minorUnits);
    $this->raiseEvent(new FundsWithdrawn(
        $this->walletId(), $amount->minorUnits, $amount->currency->value, $remaining->minorUnits
    ));
}

// ...

private function assertCurrency(Money $amount): void

```



```

    {
        if ($amount->currency !== $this->currency()) {
            throw new ConflictException('currency mismatch');
        }
    }
}
}

```

This is a genuinely different shape from a language with proper multiple inheritance — and it is deliberate, not a workaround. `Wallet` is the Eloquent row (`protected $table = 'wallets'` , `$fillable` , a non-incrementing string key) **and** the aggregate root, in one object. There is no separate `WalletEntity` ORM class and no mapper crossing a boundary between them for this aggregate: `Wallet::open()` both builds the row's attributes *and* raises `WalletOpened` in the same static factory method, and `deposit()` / `withdraw()` both mutate the Eloquent attribute *and* raise their event in the same call. `HasDomainEvents` 's buffer, `$pendingEvents` , is a genuinely **declared** private property, not a database attribute — Eloquent's magic `__get` / `__set` never intercepts it, because a declared property always shadows the magic accessors.

Three invariants live inside these three methods, and nowhere else. `open()` refuses a blank `owner_id` . `deposit()` and `withdraw()` both refuse a non-positive amount and a currency mismatch (via the shared `assertCurrency()` helper). `withdraw()` additionally refuses to let the balance go negative — the overdraft rule. Every one of these checks runs *before* `setAttribute()` is called, so a rejected operation leaves the wallet's persisted attributes completely untouched, and the corresponding event is never raised for an operation that never happened.

⚠ KEEP INVARIANTS IN THE MODEL, NOT THE SERVICE

If the overdraft check lived in a service method instead of inside `withdraw()`, anything that called `$wallet->save()` directly — a background job, an admin script, a future developer in a hurry — could silently bypass it. Because the check is inside the aggregate's own method, there is no code path into an overdrawn `Wallet` at all: the rule and the only door into the state it protects are the same piece of code.

Domain events: `DomainEvent`, and the four Lumen ships

`DomainEvent` is the flat, immutable base every concrete event extends. It auto-populates two fields you never have to set yourself:

```
abstract readonly class DomainEvent
{
    public string $eventId;

    public DateTimeImmutable $occurredAt;

    public function __construct(?string $eventId = null, ?DateTimeImmutable $occurredAt = null)
    {
        $this->eventId = $eventId ?? self::uuid4();
        $this->occurredAt = $occurredAt ?? new DateTimeImmutable;
    }

    public function eventId(): string
    {
        return $this->eventId;
    }

    public function occurredAt(): DateTimeImmutable
    {
        return $this->occurredAt;
    }
}
```

```

    }

    public function eventType(): string
    {
        $class = static::class;
        $pos = strrpos($class, '\\');

        return $pos === false ? $class : substr($class, $pos + 1);
    }
    // ...
}

```

`eventId` is a fresh uuid-v4 built from `random_bytes()` — no `ramsey/uuid` dependency, no reflection — and `occurredAt` defaults to "now" at construction time; both accept an explicit value too, for reconstitution or a test that needs a fixed timestamp. `eventType()` is the concrete class's own short name — `strrpos` / `substr` on `static::class`, again no reflection — and it is this string, not the PHP class, that a downstream listener matches against (you will see exactly that matching rule in this chapter's closing section).

Lumen ships four such events, one per state transition the wallet or a transfer can produce. Here are the three `Wallet` itself raises — real, shipped code from `samples/lumen/src/Domain/Event/`:

```

<?php

declare(strict_types=1);

namespace Lumen\Domain\Event;

use Firefly\CQRS\Attributes\PublishDomainEvent;
use Firefly\Domain\DomainEvent;

#[PublishDomainEvent('wallet.events')]

```

```

final readonly class WalletOpened extends DomainEvent
{
    public function __construct(
        public string $walletId,
        public string $ownerId,
        public string $currency,
    ) {
        parent::__construct();
    }
}

```

```

<?php

declare(strict_types=1);

namespace Lumen\Domain\Event;

use Firefly\Cqrs\Attributes\PublishDomainEvent;
use Firefly\Domain\DomainEvent;

#[PublishDomainEvent('wallet.events')]
final readonly class FundsDeposited extends DomainEvent
{
    public function __construct(
        public string $walletId,
        public int $amountMinor,
        public string $currency,
        public int $balanceMinor,
    ) {
        parent::__construct();
    }
}

```

```

<?php

declare(strict_types=1);

```

```

namespace Lumen\Domain\Event;

use Firefly\CQRS\Attributes\PublishDomainEvent;
use Firefly\Domain\DomainEvent;

#[PublishDomainEvent('wallet.events')]
final readonly class FundsWithdrawn extends DomainEvent
{
    public function __construct(
        public string $walletId,
        public int $amountMinor,
        public string $currency,
        public int $balanceMinor,
    ) {
        parent::__construct();
    }
}

```

A fourth event, `TransferCompleted`, is raised not by `Wallet` itself but by the application layer once both legs of a transfer succeed — `TransferHandler`, later in this chapter, shows exactly where. Every one carries

`#[PublishDomainEvent('wallet.events')]` — an attribute from `firefly/cqrs`, not `firefly/domain`, which names the **destination** an event republishes to once it crosses into the integration-event world; `firefly/domain` itself has no opinion about destinations at all, only about raising and draining.

Notice what each event does and does not carry. `FundsDeposited` and `FundsWithdrawn` both carry the **post-operation** balance (`$balanceMinor`), not just the amount that moved — a subscriber updating a read-model never has to reload the wallet to learn its new balance; everything it needs is already in the fact it received. That design pays off directly in `LedgerProjector`, later in this chapter.

⬆ LARAVEL PARITY

`AggregateRoot / HasDomainEvents` + `DomainEvent` correspond to Spring Data's `AbstractAggregateRoot` with its `registerEvent() / @DomainEvents / @AfterDomainEventPublication` mechanism. `raiseEvent()` is `registerEvent()`; `pullEvents()` is the drain `@AfterDomainEventPublication` performs automatically. In plain Laravel/Eloquent, the closest built-in idiom is a model firing a Laravel event directly from inside a mutator — except that fires **immediately**, win or lose, where `raiseEvent()` only buffers, and nothing is published until a surrounding transaction actually commits.

The after-commit event model

Buffering an event is not the same as publishing it — and the gap between the two is exactly what protects you from a listener ever reacting to a change that got rolled back. The full path from a raised event to a published fact runs through three packages working together:

1. A mutating aggregate method — `Wallet::deposit()`, say — calls `raiseEvent()`, appending to the private buffer. Nothing is published yet.
2. When the entity is persisted through a Firefly repository, `EloquentRepository::save()` (Chapter 5's closing section) registers it with `firefly/data`'s `AggregateTracker` — but **only** while a transaction is actively open on that entity's own connection.
3. When the surrounding `#[Transactional]` method commits, the framework drains every tracked recorder's `pullEvents()` and publishes each one, scheduled via Laravel's own `DB::afterCommit()` on the same connection the transaction ran on.
4. Laravel only fires `afterCommit()` callbacks after the **outermost** real commit, and **discards** them entirely on rollback — so a listener only ever observes an event from a unit of work that genuinely succeeded.

DepositHandler — the real, shipped command handler behind

WalletController::deposit() from Chapter 4 — is this whole cycle in five lines:

```
<?php

declare(strict_types=1);

namespace Lumen\Application\Command;

use Firefly\CQRS\Attributes\CommandHandler;
use Firefly\Data\Transaction\Attributes\Transactional;
use Firefly\Kernel\Exception\Business\ResourceNotFoundException;
use Lumen\Domain\Money;
use Lumen\Infrastructure\WalletRepository;

/**
 * Handles Deposit: loads the aggregate, credits the amount in the wallet's own currency,
 * persists, and returns the
 * new balance in minor units. #[Transactional] for the same load-bearing reason as
 * OpenWalletHandler — it is the only
 * thing that runs save() at transactionLevel() > 0 so the aggregate is tracked and
 * FundsDeposited publishes on commit.
 *
 * Intentionally NOT final: the generated transactional proxy subclasses this handler.
 */
#[CommandHandler]
class DepositHandler
{
    public function __construct(private readonly WalletRepository $wallets) {}

    #[Transactional]
    public function handle(Deposit $command): int
    {
        $wallet = $this->wallets->findById($command->walletId)
        ?? throw new ResourceNotFoundException("Wallet [{ $command->walletId }] not
found");
    }
}
```

```

        $wallet->deposit(new Money($command->amountMinor, $wallet->currency()));
        $this->wallets->save($wallet);
    }

    return $wallet->balanceMoney()->minorUnits;
}
}

```

`#[Transactional]` (the subject of a later chapter in full) is what makes step 2 above possible at all: without it, `save()` would flush the write but never see `transactionLevel() > 0`, so the aggregate would never be tracked and `FundsDeposited` would never publish, no matter how correctly `Wallet::deposit()` raised it. `$wallet->deposit(...)` runs the invariant checks and queues the event; `$this->wallets->save($wallet)` persists the row **and** registers the aggregate for after-commit dispatch, in the same call; the method returns, the transaction commits, and only then does `FundsDeposited` reach any listener.

`TransferHandler` shows the same cycle with two aggregates and a genuine all-or-nothing guarantee:

```

<?php

declare(strict_types=1);

namespace Lumen\Application\Command;

use Firefly\CQRS\Attributes\CommandHandler;
use Firefly\Data\Transaction\Attributes\Transactional;
use Firefly\Data\Transaction\Propagation;
use Firefly\Kernel\Exception\Business\ResourceNotFoundException;
use Lumen\Domain\Money;
use Lumen\Infrastructure\WalletRepository;

/**
 * Handles Transfer: an ATOMIC debit(source) + credit(destination) + save(both) inside one

```



```

#[Transactional] boundary.
*
* The transferred value carries the SOURCE currency (a transfer moves a specific amount of
money, not an abstract
* number of minor units), so the credit leg deposits that same Money into the destination.
When the two wallets share
* a currency the credit succeeds and both saves commit together; when they differ the
destination's deposit() throws
* a currency-mismatch ConflictException AFTER the source was already debited, and – because
#[Transactional] rolls
* back on any Throwable – the source debit is undone too. That is the money-cannot-vanish
invariant: there is no path
* where the source loses funds the destination never receives.
*
* Intentionally NOT final: the generated transactional proxy subclasses this handler (same
reason as the S4 handlers).
*/
#[CommandHandler]
class TransferHandler
{
    public function __construct(private readonly WalletRepository $wallets) {}

    #[Transactional(propagation: Propagation::REQUIRED)]
    public function handle(Transfer $command): void
    {
        $source = $this->wallets->findById($command->sourceWalletId)
        ?? throw new ResourceNotFoundException("Wallet [{$command->sourceWalletId}] not
found");
        $destination = $this->wallets->findById($command->destinationWalletId)
        ?? throw new ResourceNotFoundException("Wallet [{$command-
>destinationWalletId}] not found");

        $amount = new Money($command->amountMinor, $source->currency());
        $source->withdraw($amount); // debit (raises FundsWithdrawn)
        $this->wallets->save($source); // persist + track the debit INSIDE the tx, so it
can genuinely roll back
        $destination->deposit($amount); // credit – throws on currency mismatch -> whole
tx rolls back
        $this->wallets->save($destination);
    }
}

```

```

    $source->recordTransferTo($command->destinationWalletId, $amount); // both legs
succeeded -> raise TransferCompleted

    // commit here -> FundsWithdrawn + FundsDeposited + TransferCompleted drain
atomically after the unit of work commits.

    // ...
}
}

```

`recordTransferTo()` is where the `TransferCompleted` fact promised earlier gets raised — deliberately the *last* line, after both legs have already succeeded, on the source wallet (which `save()` already registered with the tracker, so an event raised after that save still drains at commit). It is a small `Wallet` method that just calls `raiseEvent(new TransferCompleted(...))`; because it runs only on the success path, a rolled-back transfer never reaches it and never publishes it.

If the destination's `deposit()` throws — a currency mismatch — `#[Transactional]`'s default `rollbackFor` (every `Throwable`) rolls the whole method back, undoing the source's debit too. None of `FundsWithdrawn`, `FundsDeposited`, or `TransferCompleted` was published yet at that point — each was only buffered on its aggregate — so a listener never sees the half of a transfer that never actually happened. Money can neither vanish nor double: either both legs commit and their events publish, or neither does.

REHYDRATION AND THE FACTORY

Every `findById()` call above rebuilds a `Wallet` from a stored row through Eloquent's own hydration — never through `Wallet::open()`. That is correct: `open()` is for **new** wallets, and calling it again on an already-persisted row would re-raise `WalletOpened` for a wallet that has existed for months. A `Wallet` loaded from storage is exactly as valid as a freshly opened one; it simply carries no fresh event, because nothing new happened to it yet.

From domain event to read model: **LedgerProjector**

The last leg of the journey — a published **FundsDeposited** actually reaching something useful — closes with a real, shipped listener. **LedgerProjector** turns every committed wallet event into an append-only row in **ledger_entries**, the table **GetLedgerHandler** (Chapter 4's **WalletController::ledger()**) reads back:

```
<?php

declare(strict_types=1);

namespace Lumen\Application\Listener;

use Firefly\Container\Attributes\Component;
use Firefly\Eda\Attributes\EventListener;
use Firefly\Eda\EventEnvelope;
use Lumen\Domain\LedgerEntry;

/**
 * The read-model projector: turns committed wallet integration events into `ledger_entries`
 * rows (an append-only
 * audit ledger the GetLedger query reads back). It closes the full event chain the sample
 * exercises – a
 * #[Transactional] command commits, the DomainEventDispatcher drains the aggregate's domain
 * events after commit, the
 * firefly/cqrs domain->integration bridge republishes each one to the eda EventPublisher
 * (the memory InMemoryEventBus
 * on the default provider), and the SubscriberRegistry delivers the matching envelope here.
 *
 * The #[EventListener] MUST enumerate the event TYPE names, not the
 * #[PublishDomainEvent('wallet.events')] DESTINATION:
 * SubscriberRegistry::deliver() calls fnmatch($pattern, $envelope->eventType), matching the
 * pattern against the
 * eventType (the short class name, e.g. 'FundsDeposited') and NEVER against the
 * destination. A 'wallet.*'-style pattern
 * would therefore never match any wallet event and this projector would silently never
```

```

fire.
*/
#[Component]
final class LedgerProjector
{
    #[EventListener(['WalletOpened', 'FundsDeposited', 'FundsWithdrawn',
    'TransferCompleted'])]
    public function onWalletEvent(EventEnvelope $envelope): void
    {
        // The envelope payload is array<string, mixed> (get_object_vars of the domain
        event, seen through the broker
        // boundary), so each field is narrowed to its projected type – a WalletOpened
        carries no amount/balance, so
        // those default to 0. TransferCompleted names its wallet `sourceWalletId` (not
        `walletId`), so the ledger row
        // for a completed transfer is keyed to the source wallet via the fallback below.
        $walletId = $envelope->payload['walletId'] ?? $envelope->payload['sourceWalletId'] ?
        ? '';

        $amountMinor = $envelope->payload['amountMinor'] ?? 0;
        $balanceMinor = $envelope->payload['balanceMinor'] ?? 0;

        LedgerEntry::query()->create([
            'wallet_id' => is_string($walletId) ? $walletId : '',
            'event_type' => $envelope->eventType,
            'amount_minor' => is_int($amountMinor) ? $amountMinor : 0,
            'balance_minor' => is_int($balanceMinor) ? $balanceMinor : 0,
            'occurred_at' => now(),
        ]);
    }
}

```

Read the docblock closely — it names every hop the event actually takes: the `#[Transactional]` command commits; `firefly/data`'s dispatcher drains the aggregate's buffered events; `firefly/cqrs`'s domain-to-integration bridge republishes each one onto the `firefly/eda` event bus (the in-memory bus by default); and `SubscriberRegistry` delivers the resulting `EventEnvelope` to every

`#[EventListener]` whose pattern matches the event's **type name** — `FundsDeposited`, not the `#[PublishDomainEvent('wallet.events')]` destination string. `$envelope->payload['balanceMinor']` existing at all, with no need to reload the wallet, is the direct payoff of `FundsDeposited` carrying the post-operation balance from the moment `Wallet::deposit()` raised it.

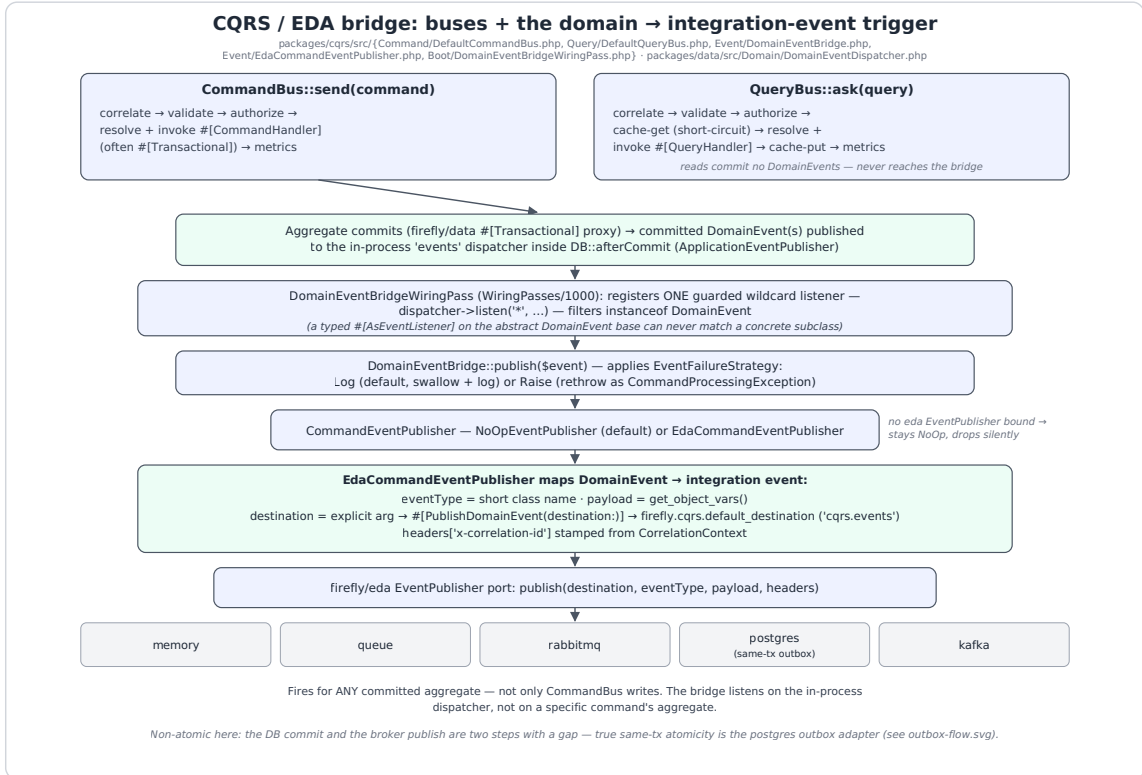


Figure 6.1 — The domain-to-integration bridge: a command commits, buffered domain events drain and republish onto the event bus, and a listener like LedgerProjector reacts on the other side.

A full account of `firefly/cqrs`'s command bus and `firefly/eda`'s event bus is a later chapter's territory. What matters here is the shape: `Wallet` never imports `firefly/eda`, never imports `LedgerProjector`, and has no idea a ledger exists. It only raises a fact. Everything downstream of that fact is somebody else's concern — which is the entire point of a domain event.

✓ What you learned

Concept	What it does
<code>Entity</code>	Identity-based equality: same class, equal non-null ids; a transient entity equals only itself
<code>ValueObject</code> / <code>ValueObjectEquality</code>	A marker for no-identity, immutable state, plus structural equality via <code>get_object_vars()</code>
<code>AggregateRoot</code>	The persistence-free consistency boundary: a private event buffer plus a <code>protected raiseEvent()</code>
<code>HasDomainEvents</code>	The identical buffer as a trait, for a class (like an Eloquent <code>Model</code>) that already extends something else
<code>RecordsDomainEvents</code>	The drain-side contract (<code>pendingEvents</code> / <code>pullEvents</code> / <code>clearEvents</code>) — deliberately without <code>raiseEvent()</code>
<code>DomainEvent</code>	A flat, immutable base auto-populating <code>eventId</code> (uuid-v4) and <code>occurredAt</code> ; <code>eventType()</code> is the class's short name
<code>Wallet::open</code> / <code>deposit</code> / <code>withdraw</code>	The three real invariants — required owner, positive amount, no overdraft, currency match — enforced before any mutation
The after-commit model	<code>save()</code> tracks a <code>RecordsDomainEvents</code> entity while a transaction is open; events drain and publish only after that transaction commits

Concept	What it does
<code>LedgerProjector</code>	A real <code>#[EventListener]</code> reacting to the published facts, matched by event type name

Try it yourself

- Add a `freeze()` behaviour.** In a scratch copy of the project (not the shipped `samples/lumen` package), add a `WalletFrozen` event (`walletId: string`, `reason: string`), a `freeze(string $reason): void` method on `Wallet` that raises it, and guard `deposit()` / `withdraw()` with a check that throws `ConflictException` when the wallet is frozen. Write a small test proving a frozen wallet's balance never changes on a deposit attempt, and that no `FundsDeposited` is ever queued for the rejected attempt.
- Prove the invariant holds.** Open a wallet, deposit a small amount, then attempt to withdraw more than the balance. Assert the thrown `ConflictException`, then assert the wallet's `balanceMoney()` is completely unchanged and `pendingEvents()` shows no `FundsWithdrawn` — the rule fired before any mutation, exactly as this chapter describes.
- Trace the transaction boundary.** Temporarily remove `#[Transactional]` from a copy of `DepositHandler`, deposit into a wallet, and confirm (via a fresh read in a new request) that the row was never actually committed — proving that `EloquentRepository::save()`'s *flush* alone is not durability, and that domain-event publication depends on the same commit.

PART III

Coordinating and Securing the Application

CQRS: Commands and Queries

By the end of this chapter you will know the shape of `firefly/cqrs`'s two buses (`CommandBus::send` / `QueryBus::ask`), how `#[CommandHandler]` / `#[QueryHandler]` register a handler as both a DI bean and a dispatch target — with the handled message type *inferred* from the handler's own `handle()` parameter — exactly what the bounded pipeline each bus runs looks like, how a handler's own exception surfaces to a caller wrapped in a `CommandProcessingException` / `QueryProcessingException`, and how `HandlerScanner` compiles handler discovery into a manifest `firefly:cache` writes right alongside the component and route manifests you already met in Chapters 2 and 4.

NEW TERM: CQRS

Command Query Responsibility Segregation treats writes and reads as two distinct kinds of message with two distinct dispatch paths — a **command** is an intent to change the write model and returns at most an id or a count; a **query** is a pure read that never mutates anything. `firefly/cqrs` gives each its own bus, its own handler attribute, and its own pipeline, so the two are never accidentally mixed at a single call site.

Two buses, one shape

Both buses are one-method ports:

```
interface CommandBus
{
    public function send(object $command): mixed;
}
```

```
interface QueryBus
{
    public function ask(object $query): mixed;
}
```

`send` is the write-side verb; `ask` is the read-side verb — chosen deliberately over the more obvious `query()`, which would collide with Eloquent's own `Model::query()` at every call site inside a repository. Neither interface requires its message to implement anything: `Firefly\CQRS\Command\Command` and `Firefly\CQRS\Query\Query` exist purely as *optional* documentary markers —

```
interface Command {}
```

— so a command is free to be exactly what `Lumen\Application\Command\OpenWallet` already is: a `final readonly` DTO with no base class and no interface at all:

```
<?php

declare(strict_types=1);

namespace Lumen\Application\Command;

use Lumen\Domain\Currency;

/**
 * Command: open a new wallet for an owner in a given currency. A plain readonly DTO — the
 * CQRS bus needs no marker
 * interface (per docs/modules/cqrs.md); the handler is resolved by the command's class-
 * string. The wallet id is
 * generated by OpenWalletHandler and returned to the caller.
 */
final readonly class OpenWallet
{
```

```

    public function __construct(
        public string $ownerId,
        public Currency $currency,
    ) {}
}

```

Dispatching one is a single call, whether it comes from a controller, a test, or another service:

```

/** @var CommandBus $commands */
$walletId = $commands->send(new OpenWallet('owner-A', Currency::EUR));

```

`$commands->send()` returns *whatever the handler returns* — here, the freshly minted wallet id — because `send()` / `ask()` are typed `mixed`. The bus itself has no idea what an `OpenWallet` even means; that is entirely the handler's job, which is exactly what the next section builds.

Handlers: one attribute, two jobs

`#[CommandHandler]` and `#[QueryHandler]` both **specialise** `#[Component]` from Chapter 2 — they *extend* it rather than merely resembling it:

```

#[Attribute(Attribute::TARGET_CLASS)]
final class CommandHandler extends Component
{
    public function __construct(public readonly ?string $command = null)
    {
        parent::__construct();
    }
}

```

```
#[Attribute(Attribute::TARGET_CLASS)]
final class QueryHandler extends Component
{
    public function __construct(public readonly ?string $query = null)
    {
        parent::__construct();
    }
}
```

Because of that `extends Component`, one attribute does **two** independent jobs at once: `ComponentScanner` (Chapter 2) discovers the class through `ReflectionAttribute::IS_INSTANCEOF` matching exactly as it would a `#[Service]`, so the handler becomes a constructor-injected singleton bean with **no separate `#[Service]` needed** — and `firefly/cqrs`'s own `HandlerScanner` reads the very same attribute to build the command/query dispatch table. Here is the whole, real handler behind `OpenWallet` — `samples/lumen/src/Application/Command/OpenWalletHandler.php`:

```
<?php

declare(strict_types=1);

namespace Lumen\Application\Command;

use Firefly\Cqrs\Attributes\CommandHandler;
use Firefly\Data\Transaction\Attributes\Transactional;
use Lumen\Domain\Wallet;
use Lumen\Infrastructure\WalletRepository;

/**
 * Handles OpenWallet: mints a new wallet id, opens the aggregate, and persists it. The
 * handled command type is
 * inferred from the sole handle() parameter (HandlerScanner param inference) — no explicit
 * #[CommandHandler(...)].
```

```

*
* #[Transactional] is LOAD-BEARING, not cosmetic: DefaultCommandBus opens no transaction of
its own, and
* EloquentRepository::save() only tracks the aggregate when transactionLevel() > 0. The
generated transactional proxy
* installs the TransactionTemplate that is the sole caller of
DomainEventDispatcher::dispatchAfterCommit(), so without
* this attribute the WalletOpened domain event would never publish and SS's ledger
projector would never fire.
*
* Intentionally NOT final: the generated transactional proxy subclasses this handler
(mirroring the framework's own
* SignatureBag/TxWidget fixtures and the PlaceOrderService pattern in docs/modules/
domain.md).
*/
#[CommandHandler]
class OpenWalletHandler
{
    public function __construct(private readonly WalletRepository $wallets) {}

    #[Transactional]
    public function handle(OpenWallet $command): string
    {
        $id = 'wlt-'.bin2hex(random_bytes(8));
        $wallet = Wallet::open($id, $command->ownerId, $command->currency);
        $this->wallets->save($wallet);

        return $id;
    }
}

```

Notice `#[CommandHandler]` here carries **no argument** — `OpenWallet::class` is never written anywhere in this file. `HandlerScanner` — the one reflection site in the whole package — resolves the handled message type by reading `handle()`'s own parameter:

```

final class HandlerScanner
{
    // ...

    private function messageType(ReflectionClass $reflection, ?string $explicit, string
$handlerClass): string
    {
        if ($explicit !== null) {
            return $explicit;
        }

        if (! $reflection->hasMethod('handle')) {
            throw new CqrsConfigurationException("Handler [{$handlerClass}] declares no
public handle() method. Add handle(<Message>) or an explicit
#[CommandHandler(Message::class)].");
        }

        $parameters = $reflection->getMethod('handle')->getParameters();
        if (count($parameters) !== 1) {
            throw new CqrsConfigurationException("Handler [{$handlerClass}]::handle() must
take exactly one message parameter, got ".count($parameters).'.');
        }

        $type = $parameters[0]->getType();
        if (! $type instanceof ReflectionNamedType || $type->isBuiltin()) {
            throw new CqrsConfigurationException("Cannot infer the message type for
[{$handlerClass}]::handle() from its parameter. Annotate #[CommandHandler(Message::class)] /
#[QueryHandler(Message::class)] explicitly.");
        }

        return $type->getName();
    }
    // ...
}

```


Same rule Chapter 6 already showed you for `#[EventListener]`'s sibling scanner. An explicit `#[CommandHandler(OpenWallet::class)]` always wins if present — useful when `handle()` takes a shared base type or an `object` parameter the scanner cannot narrow — but for the common case of one handler, one concrete message, one typed parameter, you never write the class name twice. Get the shape wrong — no `handle()` method, more than one parameter, a scalar/union parameter type the scanner can't resolve to a class — and `HandlerScanner` throws `CqrsConfigurationException` at **cache time**, not at some later request.

The query side is the mirror image, and reads a query with no side effects at all — `GetBalanceHandler`, real, shipped code:

```
<?php
declare(strict_types=1);

namespace Lumen\Application\Query;

/**
 * Query: read a wallet's current balance in minor units. GetBalanceHandler returns null for
 * an unknown wallet.
 */
final readonly class GetBalance
{
    public function __construct(public string $walletId) {}
}
```

```
<?php
declare(strict_types=1);

namespace Lumen\Application\Query;
```

```

use Firefly\Cqrs\Attributes\QueryHandler;
use Lumen\Infrastructure\WalletRepository;

/**
 * Handles GetBalance: reads the aggregate through the port and returns its balance in minor
 * units, or null when no
 * such wallet exists. A pure read – no #[Transactional]; the query type is inferred from
 * the handle() parameter.
 */
#[QueryHandler]
final class GetBalanceHandler
{
    public function __construct(private readonly WalletRepository $wallets) {}

    public function handle(GetBalance $query): ?int
    {
        $wallet = $this->wallets->findById($query->walletId);

        return $wallet?->balanceMoney()->minorUnits;
    }
}

```

`GetBalanceHandler` is `final` and carries no `#[Transactional]` at all — a query never mutates state, so there is no unit of work to demarcate and nothing for a proxy to intercept. `GetLedgerHandler` (behind Chapter 4's `WalletController::ledger()`) is the same shape reading `LedgerEntry` rows straight back:

```

<?php

declare(strict_types=1);

namespace Lumen\Application\Query;

use Firefly\Cqrs\Attributes\QueryHandler;
use Lumen\Domain\LedgerEntry;

```

```

/**
 * Handles GetLedger: returns the wallet's ledger rows in insertion order. S5's projector
 writes `ledger_entries`, so
 * this reads the projected rows directly (an empty list for a wallet with no entries yet).
 A pure read –
 * no #[Transactional]; the query type is inferred from the handle() parameter.
 */
#[QueryHandler]
final class GetLedgerHandler
{
    /** @return list<LedgerEntry> */
    public function handle(GetLedger $query): array
    {
        return array_values(
            LedgerEntry::query()
                ->where('wallet_id', $query->walletId)
                ->orderBy('id')
                ->get()
                ->all()
        );
    }
}

```

📌 LARAVEL PARITY

Plain Laravel has no read/write split at the dispatch layer at all — you either `dispatch()` a queued job or call a service method directly, and nothing stops a "query" object from quietly mutating something. `#[CommandHandler]` / `#[QueryHandler]` make the split a first-class, attribute-discovered concept: a query handler that tried to call `$this->wallets->save(...)` would still compile, but it would be an obvious code-review red flag, because nothing about `QueryBus::ask()`'s pipeline (no transaction, no domain-event drain) supports it.

The bounded pipeline

Neither bus is a bare function call to the handler. `DefaultCommandBus::send()` — the shipped implementation — runs a small, fixed sequence of collaborators around the dispatch:

```
final class DefaultCommandBus implements CommandBus
{
    // ...
    public function __construct(
        private readonly HandlerRegistry $registry,
        private readonly MessageValidator $validator,
        private readonly CommandAuthorizer $authorizer,
        private readonly CorrelationContext $correlation,
        private readonly CqrsMetrics $metrics,
        // ...
    ) {
        public function send(object $command): mixed
        {
            $prior = $this->correlation->begin();
            $startedAt = microtime(true);

            try {
                // ...
                $this->validator->validate($command);
                $this->authorizer->authorize($command);

                $handler = $this->registry->findCommandHandler($command::class);
                // ...
                $this->metrics->recordCommandSuccess($command, microtime(true) - $startedAt);

                return $result;
            } catch (Throwable $e) {
                $this->metrics->recordCommandFailure($command, microtime(true) - $startedAt);

                throw $e instanceof CommandProcessingException ? $e : new
                CommandProcessingException($command::class, $e);
            } finally {
```

```

        $this->correlation->end($prior);
    }
}
}

```

Read it as **correlate** → **validate** → **authorize** → **resolve + invoke the handler** → **metrics**, wrapped in a `try / catch / finally` that guarantees the correlation id is restored no matter what happens. `DefaultQueryBus::ask()` is the same shape with one extra step — a cache short-circuit — inserted between authorize and invoke:

```

final class DefaultQueryBus implements QueryBus
{
    // ...
    public function ask(object $query): mixed
    {
        $prior = $this->correlation->begin();
        $startedAt = microtime(true);

        try {
            // ...
            $this->validator->validate($query);
            $this->authorizer->authorize($query);

            $key = $query instanceof Cacheable ? $query->cacheKey() : null;

            if ($key !== null) {
                $cached = $this->cache->get($key);
                if ($cached !== null) {
                    return $cached;
                }
            }

            $handler = $this->registry->findQueryHandler($query::class);
            $result = $handler($query);

            if ($key !== null) {

```

```

        $this->cache->put($key, $result, $this->cacheTtl);
    }
    // ...
    $this->metrics->recordQuerySuccess($query, microtime(true) - $startedAt);
}

    return $result;
} catch (Throwable $e) {
    $this->metrics->recordQueryFailure($query, microtime(true) - $startedAt);
}

    throw $e instanceof QueryProcessingException ? $e : new
QueryProcessingException($query::class, $e);
} finally {
    $this->correlation->end($prior);
}
}
}
}

```

Every collaborator except `HandlerRegistry` itself defaults to a **no-op**, so the pipeline is a chain of stages you can light up one at a time rather than an all-or-nothing framework:

Stage	Default	What lights it up
Validation	Skipped unless the message implements <code>Validatable</code>	The shipped <code>firefly/validation</code> <code>Validator</code>
Authorization	<code>AllowAllAuthorizer</code> — allow everything	<code>firefly/security</code> 's <code>SecurityCommandAuthorizer</code> / <code>SecurityQueryAuthorizer</code> (Chapter 10)
Correlation	Always on — a uuid4 id per dispatch, restored after	Nothing to enable; stamped into <code>x-correlation-id</code> when a domain event reaches the eda bus (Chapter 8)
Metrics	<code>NoOpCqrsMetrics</code>	<code>firefly/observability</code> (a later chapter)
Query cache	<code>NoOpQueryCache</code> — always a miss	A future <code>firefly/cache</code> adapter

Neither bus opens a transaction, and neither bus drains domain events.

`#[Transactional]` on `OpenWalletHandler::handle()` above is doing that work entirely on its own, through the generated proxy Chapter 9 explains in full — the bus is completely oblivious to it, and would behave identically if `handle()` were a plain, unproxied method that happened to call `DB::beginTransaction()` by hand.

When a handler throws: category-preserving wrapping

`WithdrawHandler`'s `handle()` can fail two very different ways — the wallet might not exist, or the domain's own no-overdraw invariant might refuse the amount. Either way, `send()`'s `catch` block wraps whatever was thrown:

```
final class CommandProcessingException extends CqrsException
{
    public function __construct(string $commandClass, Throwable $cause)
    {
        parent::__construct(
            "Processing command [{commandClass}] failed: {cause->getMessage()}",
            // ...
            $cause instanceof FireflyException ? $cause->httpStatus() : 500,
            $cause instanceof FireflyException ? $cause->category() :
            ErrorCategory::Internal,
            $cause instanceof FireflyException ? $cause->severity() : ErrorSeverity::Error,
            $cause,
        );
    }
}
```

Three things to notice. First, the cause is always available on `getPrevious()` — the constructor's fifth argument to the parent — so nothing about the original failure is lost, only wrapped. Second, this is **category-preserving**: when the cause is a `FireflyException` (every kernel exception is), its own `httpStatus()` / `category()` / `severity()` are copied onto the wrapper rather than defaulting to a generic `500` / `Internal`. A `ResourceNotFoundException` (404, business) thrown for an unknown wallet id still renders as a 404 problem-details response even after being wrapped — the bus adds context, it never masks the fault. Third, `send()` checks `$e instanceof CommandProcessingException` before wrapping, so a nested command dispatch (a

handler that itself calls `$commands->send(...)`) never gets double-wrapped in two layers of the same exception. `QueryProcessingException` is the read-side twin, identical in shape.

`HandlerRegistry` throws its own, narrower exceptions for a **wiring** mistake rather than a business fault — an unmapped message sent through the bus:

```
final class HandlerRegistry
{
    // ...
    public function findCommandHandler(string $commandClass): callable
    {
        return $this->commandHandlers[$commandClass] ?? throw new
        CommandHandlerNotFoundException($commandClass);
    }
    // ...
}
```

`CommandHandlerNotFoundException` / `QueryHandlerNotFoundException` are both `ErrorCategory::Framework`, `500` — a genuinely different signal from a `ConflictException`'s `409`: the first means "nothing was ever registered for this message," the second means "something was registered, and it correctly refused the request." The registry also refuses a **second** registration for the same message class outright:

```
final class HandlerRegistry
{
    // ...
    public function registerCommandHandler(string $commandClass, callable $invoker): void
    {
        if (isset($this->commandHandlers[$commandClass])) {
            throw new CqrsConfigurationException("Duplicate command handler registered for
            message [{$commandClass}]. One command maps to exactly one handler.");
        }
    }
}
```

```

    }

    $this->commandHandlers[$commandClass] = $invoker;
}
// ...
}

```

One command, one handler — enforced at wiring time, not left to whichever registration happened to run last.

💡 UNWRAPPING THE CAUSE IN A TEST

A test that wants to assert *why* a command was rejected reaches straight past the wrapper with `getPrevious()`. `samples/lumen/tests/Application/TransferSecurityTest.php` does exactly this for a denied `#[PreAuthorize]` (Chapter 10 explains the guard itself):

```

$denied = null;
try {
    $commands->send(new Withdraw($walletId, 1000));
} catch (CommandProcessingException $e) {
    $denied = $e;
}

expect($denied)->toBeInstanceOf(CommandProcessingException::class);
expect($denied?->getPrevious())->toBeInstanceOf(AuthorizationException::class);

```

The outer type tells you *which bus operation* failed; the inner type — reached via `getPrevious()` — tells you *why*.

Compiled, not reflected: `HandlerScanner` → `HandlerManifest` → `firefly:cache`

`HandlerScanner` is the **sole** reflection site in `packages/cqrs/src` (the same grep-enforced invariant Chapter 6 described for `firefly/domain`). It walks the app's PSR-4 roots once, resolves every `#[CommandHandler]` / `#[QueryHandler]` (running exactly the `messageType()` inference you just read) into a `HandlerDescriptor`, and hands the whole list to `HandlerManifestCompiler`, which `var_export()`s it to a plain PHP array file — the same zero-reflection technique Chapter 2 introduced for the component manifest and Chapter 3 for `#[ConfigProperties]`.

You never run `HandlerScanner` yourself. It is one of the **twelve** scanner/compiler pairs `php artisan firefly:cache` orchestrates in one pass, alongside the component/context, route, config-properties, and (Chapter 9's) transactional scanners:

```
$ php artisan firefly:cache
firefly:cache – wrote 14 manifest(s) + 1 proxy(ies) to /path/to/my-app/bootstrap/cache/
firefly
```

The compiled file lands at `bootstrap/cache/firefly/handlers.php`;

`FireflyCacheServiceProvider` binds it straight onto the `HandlerManifest` container entry — unconditionally overriding the **empty** default. `CqrsWiringProvider` binds when no cache exists yet (the same `#[ConditionalOnMissingBean]`-style backing-off pattern Chapter 2 showed you for beans, applied here to a whole compiled manifest).

`CqrsHandlerWiringPass` then walks that manifest at boot and populates the `HandlerRegistry` you met above with one invoker per descriptor — each invoker resolving its handler bean **fresh from the container on every dispatch**, never cached

at wiring time, so a `#[Transactional]` handler like `OpenWalletHandler` is always observed through its generated proxy (Chapter 9), never through the bare, unproxied class.

Get this wrong in an application — add a handler, forget to re-run `firefly:cache` — and the message it should have handled resolves nothing:

`HandlerRegistry::findCommandHandler()` never sees the new descriptor, and dispatching that command throws `CommandHandlerNotFoundException` until the cache is regenerated. Fail loud at cache time (a malformed handler) or fail loud at dispatch time (a stale cache) — never a silent no-op.

✓ What you learned

Concept	What it does
<code>CommandBus::send(object): mixed</code> / <code>QueryBus::ask(object): mixed</code>	The two dispatch ports; a command may mutate and returns at most an id, a query never mutates
<code>#[CommandHandler]</code> / <code>#[QueryHandler]</code>	Specialise <code>#[Component]</code> — one attribute registers the handler as a DI bean and a dispatch target
Message-type inference	<code>HandlerScanner</code> reads the sole <code>handle()</code> parameter unless an explicit class-string is given
The bounded pipeline	correlate → validate → authorize → (cache-get, query only) → resolve + invoke → (cache-put, query only) → metrics
<code>CommandProcessingException</code> / <code>QueryProcessingException</code>	Category-preserving wrappers; the original fault is always reachable via <code>getPrevious()</code>
<code>HandlerRegistry</code>	One command/query class maps to exactly one handler — a duplicate registration fails loud at wiring time
<code>HandlerScanner</code> → <code>HandlerManifest</code>	The sole reflection site; compiled by <code>firefly:cache</code> (one of twelve scanner/compiler pairs) into a zero-reflection manifest

Try it yourself

1. **Trigger `CommandHandlerNotFoundException`**. In a scratch copy of the project (not the shipped `samples/lumen` package), define a new command class with no `#[CommandHandler]` anywhere in the app, dispatch it through `CommandBus::send()`, and confirm you get a `CommandHandlerNotFoundException` wrapped in nothing (it is thrown directly by the registry, before the bus's own `catch` block would wrap a handler's fault).
2. **Force an ambiguous `handle()`**. Give a `#[QueryHandler]` class a `handle()` method with two parameters (or a parameter typed `object`) and run `php artisan firefly:cache`. Confirm it fails loud with `CqrsConfigurationException` naming the exact class and method, rather than silently skipping the handler.
3. **Read the compiled manifest**. After running `firefly:cache` in a project with `OpenWalletHandler` and `GetBalanceHandler` present, open `bootstrap/cache/firefly/handlers.php` and find both entries. Match each array's `kind` field (`'command'` or `'query'`) back to the `HandlerKind` enum this chapter described.

Event-Driven Architecture and the Transactional Outbox

By the end of this chapter you will know the two, deliberately distinct event surfaces LaraFly ships and why confusing them is the single most common mistake; the `EventPublisher` port and the `EventEnvelope` it carries; how `#[EventListener]` matches an event's **type name** — not its destination — a real gotcha the `samples/lumen` build hit while wiring `LedgerProjector` in Chapter 6; the in-memory and queue adapters; the retry/dead-letter model every listener is wrapped in; and — closing the durability gap the domain → integration bridge leaves open — the genuine **same-transaction outbox** `firefly/eda-postgres` writes atomically with the aggregate that raised the event.

NEW TERM: INTEGRATION EVENT

Chapter 6 built **domain events** — `WalletOpened`, `FundsDeposited` — facts an aggregate raises for consumers inside the *same* process and the *same* deployable. An **integration event** is the same fact re-emitted across a process or service boundary: a JSON-ish envelope with a type name, a payload, and routing headers, carried by a broker (or, in-process, by the very same bus). This chapter is about the machinery on the far side of that boundary, and the bridge that gets a domain event onto it.

Two event surfaces — not one

LaraFly ships two unrelated event mechanisms that happen to share a similar-looking attribute name:

Surface	Attribute	Subscribes to	Delivery	Package
In-process application events	<code>#[AsEventListener]</code>	a PHP event class	synchronous, same process, no broker	<code>firefly/context</code>
EDA broker bus	<code>#[EventListener]</code>	an event-type pattern (fnmatch glob, e.g. <code>'user.*'</code>)	in-memory (sync) or queue (async)	<code>firefly/eda</code>

`#[AsEventListener]` fires when application code calls

`ApplicationEventPublisher::publish(object $event)` with a typed PHP object — the seam Chapter 6's after-commit dispatch itself rides on (`DomainEventDispatcher` calls exactly this to publish a drained `DomainEvent`). `#[EventListener]`, this chapter's subject, fires when an `EventEnvelope` whose `eventType` string matches your pattern arrives on the **eda** bus — in-memory, via a queue worker, or (this chapter's second half) via a Postgres outbox row. Neither package depends on the other; nothing in `firefly/eda` even imports Illuminate's `Dispatcher`.

The `EventPublisher` port and the `EventEnvelope`

Everything in this chapter routes through one small port:

```
interface EventPublisher
{
    public function subscribe(string $eventTypePattern, callable $handler): void;

    /**
     * @param array<string, mixed> $payload
     */
}
```



```

    * @param array<string, string> $headers
    */
    public function publish(string $destination, string $eventType, array $payload, array
$headers = []): void;

    public function start(): void;

    public function stop(): void;
}

```

`subscribe()` takes an fnmatch-style pattern (`'user.*'`, `'order.created'`, a bare `'*'`); `publish()` takes a destination string, an event-type string, a payload, and optional headers, and builds the envelope every matching subscriber receives:

```

final readonly class EventEnvelope
{
    // ...
    public function __construct(
        public string $eventType,
        public string $destination,
        public array $payload = [],
        public array $headers = [],
        ?string $eventId = null,
        ?DateTimeImmutable $timestamp = null,
        // ...
        public DateTimeImmutable $timestamp;
    // ...
}

```

`eventType` and `destination` are two genuinely different things, and the rest of this chapter hinges on that distinction: `destination` is *where* the envelope is routed on the wire (a queue name, a Postgres outbox row's `destination` column); `eventType` is *what fact it carries*, and it is the **only** field a `#[EventListener]` pattern is ever matched against.

`withHeaders(array $extra): self` returns a copy with merged headers — used internally to stamp `x-correlation-id`, and (you'll meet this again below) `x-original-topic` / `x-exception` on a dead-lettered envelope. `start()` / `stop()` are no-ops on the in-memory and queue adapters; the outbox's own consumer (later in this chapter) is where they start doing real work.

`#[EventListener]` and the `eventType-match` gotcha

```
#[Attribute(Attribute::TARGET_METHOD | Attribute::IS_REPEATABLE)]
final class EventListener
{
    /** @var list<string> */
    public readonly array $patterns;
    // ...
    /**
     * // ...
     */
    // ...
    $this->patterns = is_string($patterns) ? [$patterns] : array_values($patterns);
    // ...
}
}
```

`#[EventListener]` is **inert metadata only**, same rule as every Firefly attribute — a single string pattern normalises to a one-element list, and `IS_REPEATABLE` lets one method carry several separately-ordered subscriptions. The subscription itself happens inside the shared registry both shipped adapters compose:

```
final class SubscriberRegistry
{
    /** @var list<array{pattern: string, handler: callable}> */
```

```

    private array $subscribers = [];

    public function subscribe(string $pattern, callable $handler): void
    {
        $this->subscribers[] = ['pattern' => $pattern, 'handler' => $handler];
    }

    public function deliver(EventEnvelope $envelope): void
    {
        foreach ($this->subscribers as $subscriber) {
            if (fnmatch($subscriber['pattern'], $envelope->eventType)) {
                ($subscriber['handler'])($envelope);
            }
        }
    }
}

```

Read that `fnmatch()` call closely: it matches `$subscriber['pattern']` against `$envelope->eventType` — **never** against `$envelope->destination`. This is exactly the gotcha the `samples/lumen` build hit wiring `LedgerProjector` in Chapter 6, and it is worth restating in full here because it is the single easiest way to wire a listener that silently never fires. Every wallet domain event carries `#[PublishDomainEvent('wallet.events')]` — that string is the **destination** the event routes to once it crosses into the integration-event world. `LedgerProjector`'s own docblock states the rule as a warning for exactly this reason:

```

/**
 * // ...
 * The #[EventListener] MUST enumerate the event TYPE names, not the
 * #[PublishDomainEvent('wallet.events')] DESTINATION:
 * SubscriberRegistry::deliver() calls fnmatch($pattern, $envelope->eventType), matching the
 * pattern against the
 * eventType (the short class name, e.g. 'FundsDeposited') and NEVER against the
 * destination. A 'wallet.*'-style pattern

```

```

* would therefore never match any wallet event and this projector would silently never
fire.
*/
// ...
#[EventListener(['WalletOpened', 'FundsDeposited', 'FundsWithdrawn',
'TransferCompleted'])]
public function onWalletEvent(EventEnvelope $envelope): void
{
    // ...
    $walletId = $envelope->payload['walletId'] ?? $envelope-
>payload['sourceWalletId'] ?? '';
    $amountMinor = $envelope->payload['amountMinor'] ?? 0;
    $balanceMinor = $envelope->payload['balanceMinor'] ?? 0;

    LedgerEntry::query()->create([
        'wallet_id' => is_string($walletId) ? $walletId : '',
        'event_type' => $envelope->eventType,
        'amount_minor' => is_int($amountMinor) ? $amountMinor : 0,
        'balance_minor' => is_int($balanceMinor) ? $balanceMinor : 0,
        'occurred_at' => now(),
    ]);
}
}

```

A tempting-looking `#[EventListener(['wallet.*'])]` — reasoning by analogy from the destination string — compiles, deploys, and then never once invokes `onWalletEvent()`, because no wallet event's `eventType` (`'FundsDeposited'`, `'WalletOpened'`, ...) ever begins with `'wallet.'`; that word only ever appears in the *destination*. Chapter 6 showed you `LedgerProjector`'s full source and its consequences for the read model; this chapter is where the rule it depends on actually lives.

⚠ **EVENTTYPE** , **NEVER** **DESTINATION**

Whenever you write a `#[EventListener]` pattern, ask "does this glob match the short class name of the event I want, not the queue/topic string it happens to route through?" The two are unrelated strings that are easy to conflate precisely because they often *look* related (`'wallet.events'` destination, `'WalletOpened'` event type) — `fnmatch()` only ever sees the second one.

A second, minimal illustration from `firefly/eda`'s own documentation makes the call site concrete, pattern and imperative publish together:

```
use Firefly\Container\Attributes\Component;
use Firefly\Eda\Attributes\EventListener;
use Firefly\Eda\EventEnvelope;
use Firefly\Eda\EventPublisher;

#[Component]
final class OrderNotifier
{
    #[EventListener('order.*')]
    public function onOrderEvent(EventEnvelope $envelope): void
    {
        // $envelope->eventType is e.g. "order.placed"; $envelope->payload carries the data.
    }
}

final class OrderService
{
    public function __construct(private readonly EventPublisher $events) {}

    public function place(int $orderId): void
    {
        $this->events->publish('orders', 'order.placed', ['id' => $orderId]);
    }
}
```

Here the destination (`'orders'`) and the event type (`'order.placed'`) are deliberately different strings, and the `'order.*'` pattern matches the **second** one — precisely so this example cannot be mistaken for evidence that patterns ever look at the destination.

`EventListenerScanner` (`packages/eda/src/Scanner/EventListenerScanner.php`) is the sole reflection site in `firefly/eda`, walking the app's PSR-4 roots once and compiling every `#[EventListener]` method into an `EventListenerDescriptor`; `EventListenerManifest` loads the result with zero reflection, exactly the `HandlerManifest` idiom Chapter 7 just showed you. `firefly:cache` runs this scanner as one of its twelve pairs, writing `bootstrap/cache/firefly/event-listeners.php`; `FireflyCacheServiceProvider` binds it, overriding the empty default an uncached app would otherwise boot with. `EventListenerWiringPass` then walks that manifest at boot — in every process, web request and queue worker alike — wraps each target invocation in the retry/DLQ decorator below, and calls `$bus->subscribe($pattern, $wrapped)` per pattern, resolving the target bean **fresh from the container on every dispatch**.

Retry and dead-letter: every listener is wrapped, always

`RetryingEventHandler::wrap()` decorates every subscribed handler — there is deliberately no unwrapped fast path:

```
final class RetryingEventHandler
{
    // ...
    public static function wrap(callable $handler, int $retries, float $retryDelay, ?
    DeadLetterStore $dlq): Closure
    {
        return static function (EventEnvelope $envelope) use ($handler, $retries,
```

```

$retryDelay, $dlq): void {
    $attempt = 0;

    while (true) {
        try {
            $handler($envelope);
        }

        return;
    } catch (Throwable $e) {
        $attempt++;

        if ($attempt > $retries) {
            if ($dlq === null) {
                throw $e;
            }

            $dlq->store(
                $envelope->withHeaders([
                    'x-original-topic' => $envelope->destination,
                    'x-exception' => $e->getMessage(),
                ]),
                $e,
            );

            return;
        }

        if ($retryDelay > 0.0) {
            usleep((int) ($retryDelay * $attempt * 1_000_000));
        }
    }
};
}
}

```

A throw retries up to `firefly.eda.retries` more times with **linear** backoff (attempt N sleeps `retryDelay * N` seconds); on final failure, a bound `DeadLetterStore` receives the envelope — enriched with `x-original-topic / x-exception` — and the failure is swallowed; with no store bound, the original exception re-throws unchanged. With `retries === 0` and no DLQ, the handler still runs through the wrapper, just once, the same observable outcome as an unwrapped call. `InMemoryDeadLetterStore` — an in-process, append-only list of `DeadLetterEntry` (envelope + exception class + message) — is the `#[ConditionalOnMissingBean]` default: nothing is silently lost, but nothing survives the process either.

TWO INDEPENDENT RETRY LAYERS

On the `QueueEventBus` adapter, Laravel's own job-level retry/ `failed_jobs` machinery governs the **envelope delivery** job itself. `RetryingEventHandler` governs whether an individual `#[EventListener]` **handler's own failure** is retried and dead-lettered — completely independently, on both adapters. A job that delivers successfully but whose listener throws is a `RetryingEventHandler` concern, not a queue-retry concern.

In-memory and queue: the two M9 adapters

`InMemoryEventBus` is the skeleton default (`firefly.eda.provider` unset, or `memory`) — synchronous, zero external services:

```
final class InMemoryEventBus implements EventPublisher
{
    // ...
    public function subscribe(string $eventPattern, callable $handler): void
    {
        $this->registry->subscribe($eventPattern, $handler);
    }
}
```



```

    // ...
    public function publish(string $destination, string $eventType, array $payload, array
$headers = []): void
    {
        // Delivery is synchronous, so the consume side is NESTED in the publish side: the
PRODUCER span
        // wraps the CONSUMER span, and the envelope carries the traceparent between them
exactly as it
        // would across a broker.
        $this->tracing->tracePublish($destination, $eventType, $headers, function (array
$headers) use ($destination, $eventType, $payload): void {
            $envelope = new EventEnvelope($eventType, $destination, $payload, $headers);

            $this->tracing->traceConsume($envelope, fn (EventEnvelope $received) => $this-
>registry->deliver($received));
        });
    }

    public function start(): void {}

    public function stop(): void {}
}

```

`publish()` builds the envelope and hands it to `SubscriberRegistry::deliver()` **synchronously** — every matching handler has already run by the time `publish()` returns, and a handler that throws throws into the publishing call. The two `EdaTracing` calls wrapped around it are a no-op until Chapter 11 installs a real tracer; what they buy is that the in-memory bus emits the same producer-then-consumer span pair a broker would, so a trace does not change shape when you change provider.

`QueueEventBus` (`firefly.eda.provider=queue`) keeps the identical `EventPublisher` contract but makes `publish()` fire-and-forget:

```

final class QueueEventBus implements EventPublisher
{

```

```

    // ...

    public function publish(string $destination, string $eventType, array $payload, array
$headers = []): void
    {
        $this->tracing->tracePublish($destination, $eventType, $headers, function (array
$headers) use ($destination, $eventType, $payload): void {
            $job = (new DispatchEventJob(new EventEnvelope($eventType, $destination,
$payload, $headers)))
                ->onConnection($this->connection)
                ->onQueue($this->queue);
        });

        // Resolved EVERY call so Bus::fake() intercepts – never hoist into the
constructor.
        $this->container->make(Dispatcher::class)->dispatch($job);
    }
}

    public function deliver(EventEnvelope $envelope): void
    {
        $this->tracing->traceConsume($envelope, fn (EventEnvelope $received) => $this-
>registry->deliver($received));
    }
}
// ...
}

```

Same two seams, split across two processes: `publish()` is the producer side in the web request, `deliver()` the consumer side on the worker, and the traceparent riding in the envelope's headers is the only thing joining them.

`publish()` returns before any listener has run — delivery happens on whichever queue worker picks up the `DispatchEventJob`. That worker's own boot repopulates its `SubscriberRegistry` from the same compiled manifest `EventListenerWiringPass` reads everywhere, so a share-nothing worker reconstructs the identical subscriber set

every time it starts. Under the `sync` queue driver, delivery collapses to the same synchronous behaviour as `InMemoryEventBus` — which is exactly how a test exercises the async path with no worker actually running.

The domain → integration-event bridge

Chapter 6 showed you the *effect* of the bridge — a committed `WalletOpened` reaching `LedgerProjector` as an `EventEnvelope`. Here is the mechanism. Every `DomainEvent` committed through `firefly/data`'s after-commit dispatch (Chapter 6's closing section) is also published, in-process, to `ApplicationEventPublisher` — the very `#[AsEventListener]` surface this chapter opened by contrasting with `#[EventListener]`. Because Illuminate's own dispatcher matches an object event only by its **concrete class** (plus interfaces, never parent classes), a single `#[AsEventListener]` typed on the abstract `DomainEvent` base could never fire for any concrete subclass. `DomainEventBridgeWiringPass` works around exactly this limitation with one guarded wildcard listener instead:

```
final class DomainEventBridgeWiringPass implements BootPass
{
    // ...
    public function run(BootContext $context): void
    {
        $container = $context->container;
        // ...
        $dispatcher = $container->make('events');

        $dispatcher->listen('*', DispatcherEventPublisher::guardListener(
            // ...
            static function (string $eventName, array $payload) use ($container): void {
                $event = $payload[0] ?? null;
                if ($event instanceof DomainEvent) {
```

```

        $container->make(DomainEventBridge::class)->publish($event);
    }
    },
    ));
}
}
}

```

`DomainEventBridge::publish()` maps the `DomainEvent` onto the `eda EventPublisher` port through `EdaCommandEventPublisher: eventType = $event->eventType()` (the short class name — the very string `#[EventListener]` patterns match against), `payload` = the event's public fields via `get_object_vars()`, and `destination` = an explicit override, else the event's `#[PublishDomainEvent(destination:)]` (`'wallet.events'` for every wallet event), else `firefly.cqrs.default_destination` (`'cqrs.events'`). The active correlation id (Chapter 7's `CorrelationContext`) is stamped into the envelope's `x-correlation-id` header.

```

final class DomainEventBridge
{
    // ...
    public function publish(DomainEvent $event): void
    {
        $eventClass = $event::class;

        try {
            $this->publisher->publish($event);
        } catch (Throwable $e) {
            if ($this->failureStrategy === EventFailureStrategy::Raise) {
                throw $e instanceof CommandProcessingException ? $e : new
CommandProcessingException($eventClass, $e);
            }

            $this->logger?->error(
                "CQRS domain-event bridge failed to publish [{ $eventClass }]: { $e-

```

```

>getMessage()}",
    ['exception' => $e],
    );
    }
    }
}

```

This runs **after** the aggregate's own transaction has already committed — the write already succeeded by the time this code executes.

`firefly.cqrs.event_failure_strategy` (default `log`) decides what happens if the broker publish itself then fails: `log` swallows and logs it — the command's result stands; the integration publish was best-effort. `raise` re-throws it, wrapped in `CommandProcessingException`, surfacing the post-commit failure to the original caller even though the underlying write is already durable. Either way, the bridge fires **only** for a genuinely committed unit of work: a rolled-back transaction raises no `DomainEvent` at all (Laravel discards `afterCommit` callbacks on rollback), so nothing is ever published for it — the same guarantee Chapter 6's after-commit model already gave you for in-process listeners, now extended across the process boundary.

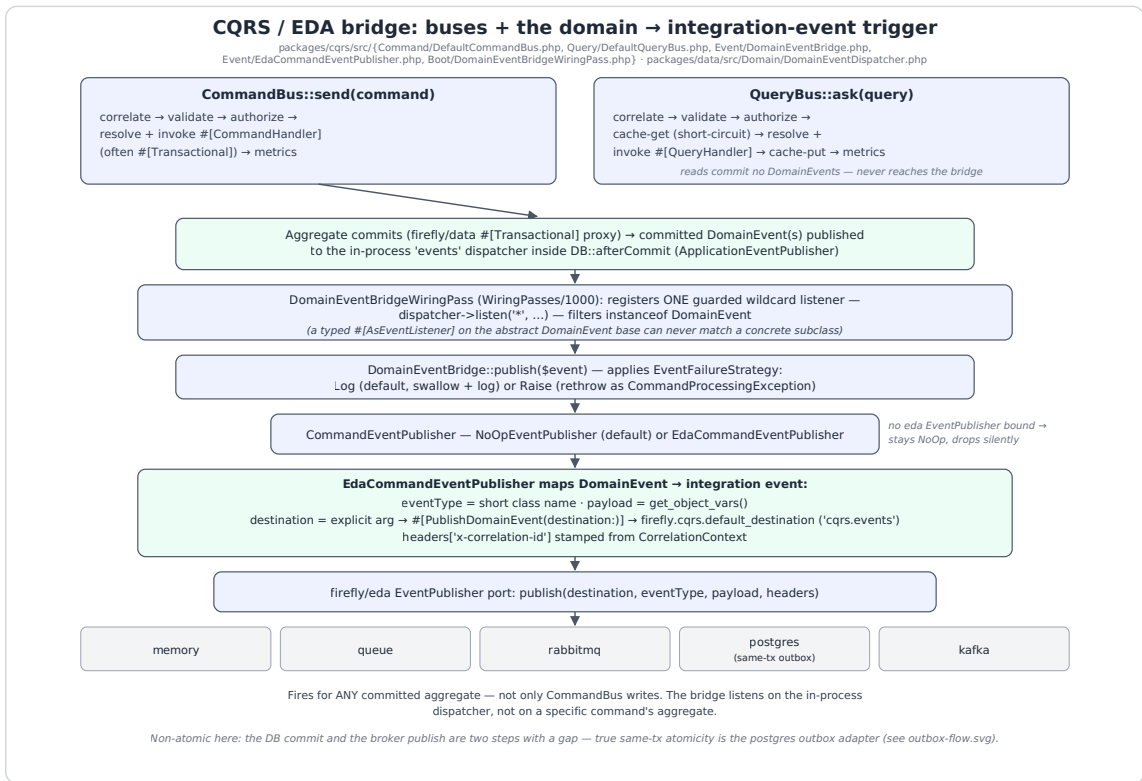


Figure 8.1 — The domain-to-integration bridge: a guarded wildcard listener on the in-process dispatcher catches every committed DomainEvent and republishes it through the resolved CommandEventPublisher.

When no `firefly/eda EventPublisher` is bound at all, `CqrsAutoConfiguration` falls back to `NoOpEventPublisher`, which drops the event (optionally logging at debug) — installing `firefly/cqrs` without `firefly/eda` still boots cleanly and simply emits no integration events.

The gap this leaves open. The write commits; *then*, in a separate step, the broker publish is attempted. Between those two steps there is a real window — a crashed process, a broker outage — in which the aggregate's own database state has moved forward but the integration event describing that move was never sent. This is an

honest, at-least-once-*ish*, non-atomic guarantee, adequate for a great many applications and explicitly not adequate for money movements that other services must react to reliably. Closing that gap is exactly what the rest of this chapter builds.

The genuine same-transaction outbox (`firefly/eda-postgres`)

`firefly/eda-postgres` closes the gap by writing the outbox row **inside** the aggregate's own open transaction, on the aggregate's own connection — so it commits or rolls back atomically with the aggregate, with no dual-write and nothing to "hope" about after the fact.

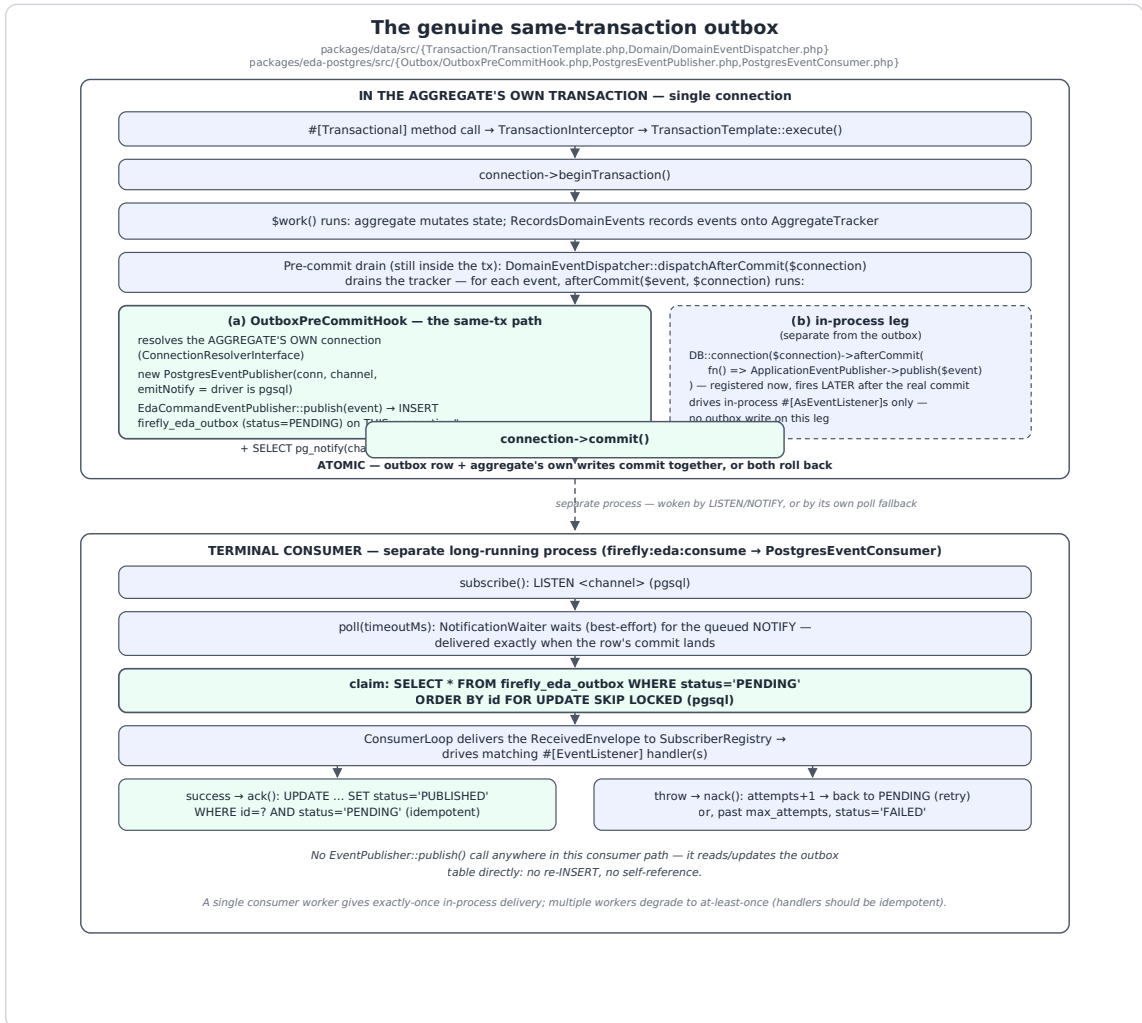


Figure 8.2 — The same-transaction outbox: the aggregate's own commit and the outbox INSERT are one atomic unit; a consumer claims PENDING rows independently.

The seam: **PreCommitEventHook**

Chapter 6 showed `DomainEventDispatcher::dispatchAfterCommit()` draining the `AggregateTracker` and scheduling each event via `DB::afterCommit()`. It carries one more, optional collaborator — a hook that runs **before** that scheduling, while the transaction is still open:


```
interface PreCommitEventHook
{
    public function handle(object $event, ?string $connection = null): void;
}
```

With no hook bound — no `firefly/eda-postgres`, or any provider other than `postgres` — this is a complete no-op and `DomainEventDispatcher` behaves exactly as Chapter 6 described. `firefly/eda-postgres`'s `OutboxPreCommitHook` is the implementation that turns it on:

```
final class OutboxPreCommitHook implements PreCommitEventHook
{
    // ...
    public function handle(object $event, ?string $connection = null): void
    {
        if (! $event instanceof DomainEvent) {
            return;
        }
        // ...
        $emitNotify = $conn instanceof Connection && $conn->getDriverName() === 'pgsql';
        // ...
        (new EdaCommandEventPublisher($publisher, $this->defaultDestination, $this->destinations, $this->correlation))
            ->publish($event);
    }
}
```

`$connection` is threaded through unchanged from the dispatcher, so a `#[Transactional(connection: 'x')]` aggregate's outbox row lands on connection `x` — never a hard-coded default — and the hook reuses the *exact same*

`EdaCommandEventPublisher` mapping (event type, payload, destination, correlation id) the after-commit bridge uses, so the row this writes is byte-for-byte identical in shape to what the ordinary bridge would have produced.

The INSERT, and `pg_notify` in the same transaction

```
final class PostgresEventPublisher implements EventPublisher
{
    // ...
    public function publish(string $destination, string $eventType, array $payload, array
$headers = []): void
    {
        $this->tracing->tracePublish($destination, $eventType, $headers, function (array
$headers) use ($destination, $eventType, $payload): void {
            $id = $this->connection->table(OutboxSchema::TABLE)->insertGetId([
                'destination' => $destination,
                'channel' => $this->channel,
                'event_type' => $eventType,
                'payload' => json_encode($payload, JSON_THROW_ON_ERROR),
                'headers' => json_encode($headers === [] ? new stdClass : $headers,
JSON_THROW_ON_ERROR),
                'transaction_id' => $headers['x-correlation-id'] ?? null,
                'status' => OutboxSchema::STATUS_PENDING,
                'attempts' => 0,
                'created_at' => $this->connection->raw('CURRENT_TIMESTAMP'),
            ]);

            if ($this->emitNotify) {
                // In-tx pg_notify: Postgres queues it and delivers on COMMIT, so the
consumer's LISTEN wakes exactly
                // ...
                $this->connection->statement('SELECT pg_notify(?, ?)', [$this->channel,
(string) $id]);
            }
        });
    }
}
```

```
// ...
}
```

Because `publish()` runs on **whatever connection it was constructed with**, and that connection is mid-transaction when `OutboxPreCommitHook` calls it, the `INSERT` enlists in the aggregate's own transaction: it commits when the aggregate commits, and it rolls back when the aggregate rolls back. When the driver is `pgsql`, the same closure also fires `SELECT pg_notify(?, ?)` — **inside the same transaction**. Postgres queues a `NOTIFY` raised inside a transaction and only actually delivers it once that transaction commits, so a listening consumer wakes with low latency at exactly the moment the row becomes visible to other connections — and never wakes for a row that gets rolled back.

The whole of that body sits inside `EdaTracing::tracePublish()` — the single port all three broker publishers now route through, `eda-rabbitmq` and `eda-kafka` alongside this one. Because propagation means *writing* something on the envelope, the port hands the send closure the headers to carry and the adapter builds the row out of those, so the `traceparent` the producer span stamps is on the row that commits with the aggregate — this is the one adapter where the producer span and the durable record are genuinely atomic, rather than a span that ends the moment a socket accepted the message. With tracing off the bound `EdaTracing` is `NoOpEdaTracing`, which calls the closure with the caller's headers unchanged, so the path is byte-for-byte what it was; and `transaction_id` is still read from `x-correlation-id`, never from a `traceparent`, so the relay's idempotency key does not change just because a request happened to be traced.

The outbox table (`firefly_eda_outbox` , one schema shared by the migration, the publisher, the consumer, and the relay) carries: `id`, `destination`, `channel`, `event_type`, `payload`, `headers`, `transaction_id`, `status`, `attempts`, `error_message`, `created_at`, `processed_at`, `failed_at` , with `status` one of `PENDING` / `PUBLISHED` / `FAILED` .

A genuine, shipped test proves both halves of the guarantee directly against a real transaction:

```
it('writes the outbox row INSIDE the caller transaction (present after commit)', function () {
  {
    // ...
    DB::transaction(function () use ($publisher): void {
      $publisher->publish('users', 'user.created', ['id' => 1], ['x-a' => 'b']);
    });
    // ...
    expect(DB::table(OutboxSchema::TABLE)
      ->where('event_type', 'user.created')
      ->where('status', OutboxSchema::STATUS_PENDING)
    // ...
  });
}

it('rolls the outbox row back WITH the aggregate (absent after rollback)', function () {
  // ...
  try {
    DB::transaction(function () use ($publisher): void {
      $publisher->publish('users', 'user.created', ['id' => 2]);
      throw new RuntimeException('business failure after the outbox write');
    });
  } catch (RuntimeException) {
    // expected
  }

  // The INSERT rolled back atomically with the aggregate's tx: no row survives.
```

```
expect(DB::table(OutboxSchema::TABLE)->count())->toBe(0);
});
```

No amount of prose substitutes for that second test: the outbox row from the failed unit of work simply does not exist afterward, because it was never a separate write to begin with — it was one more statement inside the same transaction that rolled back.

Double-publish avoidance

With `provider=postgres` bound, both `OutboxPreCommitHook` (in-tx) *and* the ordinary after-commit `DomainEventBridge` could, in principle, both try to write the same event to the outbox. `PostgresOutboxAutoConfiguration` prevents that at `#[Order(900)]` — ahead of `firefly/data`'s and `firefly/cqrs`'s `#[Order(1000)]` defaults — by binding, together, as one package:

- its **own** `DomainEventDispatcher`, carrying the `OutboxPreCommitHook`, which wins `DataAutoConfiguration`'s `#[ConditionalOnMissingBean(DomainEventDispatcher::class)]`, and
- a `NoOpEventPublisher` as the `CommandEventPublisher`, which wins `CqrsAutoConfiguration`'s `#[ConditionalOnMissingBean(CommandEventPublisher::class)]` — silencing **only** the after-commit eda leg.

The result: a domain event is written to the outbox **exactly once**, in-tx, atomically with the aggregate. The generic (non-eda) `#[AsEventListener]` after-commit dispatch this chapter opened with is completely untouched — those in-process listeners keep firing exactly as before. All four beans are additionally gated `#[ConditionalOnProperty('firefly.eda.provider', 'postgres')]`, so installing the package without selecting it as the active provider is fully inert.

Claiming rows: `PostgresEventConsumer`

Once a row is `PENDING`, `PostgresEventConsumer` is the always-available, terminal in-process delivery path — `php artisan firefly:eda:consume` drives it as a long-running worker:

```
final class PostgresEventConsumer implements EventConsumer
{
    // ...
    public function subscribe(array $destinations): void
    {
        if ($this->connection->getDriverName() === 'pgsql') {
            $this->connection->getPdo()->exec('LISTEN '.$this->channel);
        }
    }
    // ...
    public function poll(int $timeoutMs): ?ReceivedEnvelope
    {
        try {
            ($this->awaitNotification)($timeoutMs);
        } catch (\Throwable) {
            // ...
        }

        $pgsql = $this->connection->getDriverName() === 'pgsql';
        $query = $this->connection->table(OutboxSchema::TABLE)
            ->where('status', OutboxSchema::STATUS_PENDING)
            ->orderBy('id');
        if ($pgsql) {
            // ...
        }
        $row = $query->first();

        if ($row === null) {
            return null;
        }

        return new ReceivedEnvelope(new EventEnvelope(
```

```

        OutboxRow::asString($row->event_type),
        OutboxRow::asString($row->destination),
        OutboxRow::asMap($row->payload),
        OutboxRow::asStringMap($row->headers),
    ), OutboxRow::asInt($row->id));
    }

    public function ack(ReceivedEnvelope $received): void
    {
        $this->connection->table(OutboxSchema::TABLE)
            ->where('id', $received->deliveryTag)
            ->where('status', OutboxSchema::STATUS_PENDING)
            // ...
    }
    // ...
}

```

`poll()` first waits briefly (best-effort, guarded against any failure) on a `NOTIFY`, then — regardless of whether one arrived — claims the oldest `PENDING` row with `FOR UPDATE SKIP LOCKED` on postgres (a plain `ORDER BY id` under sqlite, for tests). Note there is **no `EventPublisher::publish()` call anywhere in this class** — it reads and updates the outbox table directly, so there is no re-`INSERT` and no risk of a consumer feeding its own output back into the table it just read from. `ConsumerLoop` delivers the claimed envelope to the same `SubscriberRegistry` your `#[EventListener]` handlers already subscribed to, exactly as the in-memory and queue adapters do; `ack()`'s `WHERE status = 'PENDING'` guard makes acknowledgement idempotent, and a handler throw instead calls `nack()`, which increments `attempts` and either resets the row to `PENDING` (retry) or, past `firefly.eda.postgres.max_attempts` (default 3), marks it `FAILED` in place — there is no separate dead-letter table for Postgres; a `status='FAILED'` row is the dead letter, queryable on the same `firefly_eda_outbox` table.

This is a **durable** `PENDING` → `PUBLISHED` status window, not an in-memory high-water mark: a restarted consumer resumes at the oldest still-`PENDING` row and never replays a `PUBLISHED` one. A single consumer worker gives exactly-once in-process delivery; running more than one concurrently degrades to at-least-once (write your listeners idempotently if you do).

LARAVEL PARITY

Plain Laravel has no first-party transactional-outbox pattern at all — the idiom you'd hand-roll is an `outbox` table plus a scheduled command that polls it, with no framework help keeping the `INSERT` inside the same transaction as your domain write. `firefly/eda-postgres` is exactly that idiom, but with the atomicity guaranteed by construction (`PreCommitEventHook` runs while the transaction is still open) rather than left to developer discipline.

✓ What you learned

Concept	What it does
Two event surfaces	<code>#[AsEventListener]</code> (in-process, <code>firefly/context</code>) vs. <code>#[EventListener]</code> (broker bus, <code>firefly/eda</code>) — unrelated mechanisms
<code>EventPublisher</code> / <code>EventEnvelope</code>	The broker-bus port; <code>destination</code> is <i>where</i> , <code>eventType</code> is <i>what fact</i>
<code>SubscriberRegistry::deliver()</code>	Matches <code>fnmatch(\$pattern, \$envelope->eventType)</code> — never the destination; the exact gotcha <code>LedgerProjector</code> hit
<code>RetryingEventHandler</code>	Always wraps every listener: linear-backoff retry, then dead-letter (or re-throw with no store bound)
<code>InMemoryEventBus</code> / <code>QueueEventBus</code>	Synchronous vs. queue-backed (fire-and-forget) <code>EventPublisher</code> adapters, identical port
<code>DomainEventBridge</code>	The after-commit trigger republishing a committed <code>DomainEvent</code> as an integration event — best-effort, non-atomic
<code>OutboxPreCommitHook</code> / <code>PostgresEventPublisher</code>	Writes the outbox row inside the aggregate's own open transaction — genuinely atomic
<code>PostgresEventConsumer</code>	Claims <code>PENDING</code> rows (<code>FOR UPDATE SKIP LOCKED + LISTEN / NOTIFY</code>), drives <code>#[EventListener]</code> handlers directly, no re- <code>INSERT</code>

Try it yourself

1. **Reproduce the gotcha, deliberately.** In a scratch copy of the project, change `LedgerProjector`'s `#[EventListener]` patterns to `['wallet.*']` and re-run its test suite. Confirm the projector silently never fires and no `ledger_entries` rows are ever written — then restore the original type-name patterns and confirm they pass again.
2. **Force a dead letter.** Write a small `#[EventListener]` handler that always throws, configure `firefly.eda.retries=2` and bind an `InMemoryDeadLetterStore`, publish a matching event, and inspect `DeadLetterStore::all()` afterward. Confirm the entry's `exceptionMessage` matches what your handler threw.
3. **Prove the outbox rollback yourself.** Using a Postgres connection (`@group('integration')`, per the framework's own test conventions), reproduce the two `OutboxSameTransactionTest` cases this chapter quoted — publish-then-commit and publish-then-throw — and confirm the row's presence or absence matches this chapter's description.

Transactions and the `#[Transactional]` Proxy

By the end of this chapter you will know exactly what `#[Transactional]` does — its seven propagation modes, its isolation/read-only/rollback settings, and the generated proxy that gives it teeth — how self-invocation bypasses that proxy (and what to do instead), how a `timeout:` is actually enforced — a monotonic wall clock that judges the method after it returns, plus a best-effort driver statement timeout that can interrupt a running query — how `#[TransactionalEventListener]` defers *the listener* rather than the event into one of four transaction phases, of which `BEFORE_COMMIT` can veto the commit outright, how `firefly:cache` and the in-process scan together give every application a working proxy with no wiring of its own, and the single most consequential fact this whole book has been building toward: **a domain event publishes only because `TransactionTemplate` — the machinery behind `#[Transactional]` — is the sole caller of the after-commit dispatch.** No `#[Transactional]`, no publish, no matter how correctly an aggregate raised its event.

NEW TERM: DECLARATIVE TRANSACTION DEMARCATION

Instead of writing `DB::beginTransaction()` / `DB::commit()` / `DB::rollback()` by hand inside a method body, you *declare* the boundary with an attribute and let a generated proxy enforce it. This is Spring's `@Transactional` model, and it is why Chapters 6 and 7 could already show you `#[Transactional]` on `OpenWalletHandler`, `DepositHandler`, `WithdrawHandler`, and `TransferHandler` without a single explicit `DB::` call inside any of their `handle()` bodies.

The #[Transactional] attribute

```
#[Attribute(Attribute::TARGET_CLASS | Attribute::TARGET_METHOD)]
final class Transactional
{
    /**
     * @param list<class-string<Throwable>> $rollbackFor
     * @param list<class-string<Throwable>> $noRollbackFor
     */
    public function __construct(
        public Propagation $propagation = Propagation::REQUIRED,
        public Isolation $isolation = Isolation::DEFAULT,
        public bool $readOnly = false,
        public array $rollbackFor = [Throwable::class],
        public array $noRollbackFor = [],
        public ?string $connection = null,
        public ?int $timeout = null,
    ) {}
}
```

On a **class**, it sets the default for every public method. On a **method**, it *replaces* — never merges with — the class-level attribute for that one method (the same Spring semantics). `packages/data/tests/Fixtures/Capstone/AccountService.php` is real, shipped test fixture code that puts both forms to work at once:

```
<?php

declare(strict_types=1);

namespace Firefly\Data\Tests\Fixtures\Capstone;

use Firefly\Container\Attributes\Service;
use Firefly\Data\Transaction\Attributes\Transactional;
use Firefly\Data\Transaction\Propagation;
use Firefly\Data\Transaction\TransactionalDescriptor;
```

```

use Firefly\Data\Transaction\TransactionTemplate;
use Illuminate\Support\Facades\DB;
use RuntimeException;

// ...
#[Service]
#[Transactional]
class AccountService
{
    public function __construct(private readonly TransactionTemplate $template) {}

    public function transferAndCommit(): void
    {
        DB::table('accounts')->insert(['name' => 'a']);
        DB::table('accounts')->insert(['name' => 'b']);
    }

    public function transferAndFail(): void
    {
        DB::table('accounts')->insert(['name' => 'a']);
        DB::table('accounts')->insert(['name' => 'b']);

        throw new RuntimeException('boom');
    }

    // ...
    #[Transactional(noRollbackFor: [IgnorableException::class])]
    public function logButKeep(): void
    {
        DB::table('accounts')->insert(['name' => 'kept']);

        throw new IgnorableException('ignored');
    }

    public function outerWithNested(): void
    {
        DB::table('accounts')->insert(['name' => 'outer']);

        try {
            $this->template->execute(function (): void {
                DB::table('accounts')->insert(['name' => 'inner']);
            });
        }
    }
}

```

```

        throw new RuntimeException('inner fail');
    }, new TransactionalDescriptor(propagation: Propagation::NESTED));
    } catch (RuntimeException) {
    }
}
}
}

```

Every public method here inherits the class-level `#[Transactional]` default (`REQUIRED`, rollback on any `Throwable`) **except** `logButKeep()`, whose own method-level attribute *completely replaces* that default for this one method — it does not layer `noRollbackFor` on top of the class default, it *is* the effective configuration. Default `rollbackFor = [Throwable::class]`: PHP has no checked/unchecked exception split, so by default *any* throwable rolls the transaction back, unless it also matches `noRollbackFor`, which always wins.

The seven propagation modes

`Propagation` is an unbacked enum with every Spring mode, including `NESTED` — made possible over a plain relational connection by Laravel's own automatic savepoints:

```

enum Propagation
{
    case REQUIRED;
    case REQUIRES_NEW;
    case NESTED;
    case SUPPORTS;
    case NOT_SUPPORTED;
    case MANDATORY;
    case NEVER;
}

```

```
// ...
}
```

`TransactionTemplate::execute()` is the single source of truth both the generated proxy and any direct, programmatic caller run through — there is no second code path to keep in sync:

```
public function execute(Closure $work, ?TransactionalDescriptor $descriptor = null): mixed
{
    $d = $descriptor ?? new TransactionalDescriptor;
    $connection = DB::connection($d->connection);
    $active = $connection->transactionLevel() > 0;

    try {
        return match ($d->propagation) {
            Propagation::MANDATORY => $active ? $work() : throw new
TransactionRequiredException,
            Propagation::NEVER => $active ? throw new TransactionNotAllowedException :
$work(),
            Propagation::SUPPORTS, Propagation::NOT_SUPPORTED => $work(),
            Propagation::REQUIRED => $active ? $work() : $this-
>runInTransaction($connection, $work, $d, true),
            Propagation::REQUIRES_NEW, Propagation::NESTED => $this-
>runInTransaction($connection, $work, $d, !$active),
        };
    } catch (Throwable $e) {
        throw $this->translator->translate($e, $connection->getDriverName());
    }
}
```

The whole method is one `match` over the propagation mode and a `catch` that hands whatever came out to the exception translator from Chapter 5. Everything else — the savepoints, the isolation statements, the timeout, the after-commit drain — is inside `runInTransaction()`, which only three of the seven modes ever reach.

Mode	Behaviour
<code>REQUIRED</code> (default)	Joins the caller's transaction if one is active; otherwise starts a new outermost one.
<code>REQUIRES_NEW</code>	Always runs in a transaction. With none active it becomes the new outermost one; with one already active on the same connection , Laravel has no suspend primitive, so it degrades to a nested <code>beginTransaction()</code> — a savepoint, not a truly independent transaction.
<code>NESTED</code>	The same mechanics as <code>REQUIRES_NEW</code> in this implementation: a fresh outermost transaction if none is active, otherwise a savepoint — so a <code>NESTED</code> failure unwinds only to its own savepoint, never the whole unit of work.
<code>SUPPORTS</code>	Runs in the caller's transaction if one is active; otherwise with no transaction at all. Never starts one.
<code>NOT_SUPPORTED</code>	Always runs with no transaction — but on the <i>same</i> connection there is no suspend primitive, so an already-active transaction is simply not paused; the work still runs inside it.
<code>MANDATORY</code>	Requires an active transaction; runs in it, or throws <code>TransactionRequiredException</code> .
<code>NEVER</code>	Forbids an active transaction; throws <code>TransactionNotAllowedException</code> if one is active, otherwise runs with none.

`outerWithNested()` above is `NESTED` in action: the outer method inserts `'outer'` under the class-level `REQUIRED` default, then calls `$this->template->execute(...)` directly with `propagation: Propagation::NESTED`, inserts `'inner'`, and throws. A real, shipped capstone test proves exactly what unwinds and what survives:

```
it('unwinds a NESTED inner rollback to a savepoint, leaving the outer row intact', function () {
    // ...
    accountService($this->app())->outerWithNested();

    expect(DB::table('accounts')->pluck('name')->all())->toBe(['outer']);
});
```

The `'inner'` row vanishes with the savepoint; `'outer'` — inserted before the nested unit of work even began — survives and later commits normally when the outer method returns.

Isolation, read-only, and the rollback decision

`Isolation` is a string-backed enum whose value is the SQL clause:

```
enum Isolation: string
{
    case DEFAULT = 'DEFAULT';
    case READ_UNCOMMITTED = 'READ UNCOMMITTED';
    case READ_COMMITTED = 'READ COMMITTED';
    case REPEATABLE_READ = 'REPEATABLE READ';
    case SERIALIZABLE = 'SERIALIZABLE';
}
```

On the outermost transaction of a unit of work, a non-`DEFAULT` isolation issues `SET TRANSACTION ISOLATION LEVEL {value}`, and `readOnly: true` issues `SET TRANSACTION READ ONLY` — both **best-effort**: either statement failing (a driver that ignores it, like SQLite) is caught and silently ignored rather than failing the whole unit of work.

```
private function shouldRollBack(Throwable $translated, Throwable $original,
TransactionalDescriptor $d): bool
{
    foreach ($d->noRollbackFor as $type) {
        if ($translated instanceof $type || $original instanceof $type) {
            return false; // noRollbackFor wins: commit-and-rethrow
        }
    }

    foreach ($d->rollbackFor as $type) {
        if ($translated instanceof $type || $original instanceof $type) {
            return true;
        }
    }

    return false; // not listed in rollbackFor: commit-and-rethrow
}
```

Read in order: an exception matching `noRollbackFor` always **commits** (checked first, so it wins even over a matching `rollbackFor`); otherwise, a match in `rollbackFor` (the default is everything) **rolls back**; otherwise — only reachable with a deliberately narrowed `rollbackFor` — it **commits**. `AccountService::logButKeep()` exercises exactly the first branch, and a real capstone test proves the row survives the exception it's thrown alongside:

```
it('commits despite a method-level noRollbackFor exception (override beats class-level)',
function () {
```

```
// ...
try {
    accountService($this->app())->logButKeep();
} catch (IgnorableException) {
}

// Class-level default would roll back any Throwable; the method-level override keeps
the row.
expect(DB::table('accounts')->where('name', 'kept')->count())->toBe(1);
});
```

Notice `TransactionTemplate` uses **manual** `beginTransaction() / commit() / rollBack()` throughout, never Laravel's own `DB::transaction($closure)` — only manual control lets a caught exception be *committed*-and-rethrown when `noRollbackFor` says so, instead of Laravel's own helper, which always rolls back on any exception with no such override point.

The proxy model

A `#[Transactional]` bean is never called directly. `TransactionalBeanPostProcessor` — a `BeanPostProcessor` discovered exactly like any other bean, installed at phase 700 — swaps it, on its second pass (after `#[PostConstruct]` has already run on the real bean), for an instance of a generated `final class {Target}` `__FireflyTransactionalProxy extends {Target}`. `ProxyClassGenerator` emits that class, overriding every transactional method with:

```
final class TransferService__FireflyTransactionalProxy extends TransferService
{
    private \Firefly\Data\Proxy\MethodInterceptor $__fireflyTxInterceptor;

    public function transfer(int $amount): int
```

```

    {
      return (new \Firefly\Data\Proxy\MethodInvocation(
        $this,
        TransferService::class,
        'transfer',
        [$amount],
        [$this->__fireflyTxInterceptor],
        [\Firefly\Data\Transaction\TransactionalDescriptor::class =>
self::__fireflyTxDescriptor('transfer')],
        fn (array $__fireflyArgs) => parent::transfer(...$__fireflyArgs),
      ))->proceed();
    }
  }
}

```

— handing the call to `MethodInvocation::proceed()`, which walks the interceptors the compiled plan named for this method and finishes in the terminal closure that calls `parent::`. For a bean whose only advice is `#[Transactional]` that list holds one link: `TransactionInterceptor::invoke()` reads the baked `TransactionalDescriptor` off the invocation and passes `fn () => $invocation->proceed()` to its unchanged `run()`, which delegates straight to `TransactionTemplate::execute()`. A real capstone test confirms the swap actually happened — the resolved bean's class is **not** the plain service class at all:

```

it('proxies the #[Service] and rolls back BOTH inserts when the method throws', function ()
{
  // ...
  $service = accountService($this->app());

  expect($service::class)->not->toBe(AccountService::class); // it is the generated proxy
subclass
  expect($service)->toBeInstanceOf(AccountService::class);

  try {
    $service->transferAndFail();
  }
}

```

```

    } catch (RuntimeException) {
    }

    expect(DB::table('accounts')->count())->toBe(0);
});

```

`$service::class` is the generated `AccountService_FireflyTransactionalProxy`, yet `$service instanceof AccountService` is still true — the proxy *is-a* `{Target}`, so every container call and `#[PreDestroy]` resolve against it exactly as they would the original bean. `ProxyFactory` instantiates it **state-preservingly**:

`newInstanceWithoutConstructor()` (so `#[PostConstruct]` is not re-run), then the real bean's initialised state is copied onto the proxy slot by slot, each slot written by a closure bound to the class that *declares* it — never

`ReflectionProperty::setValue()`. Writing from the declaring class is not a detail: a closure bound to the declared class alone sees only that class's own privates, so writing from where each slot was declared is what lets the copy reach a `private` on a parent (`EloquentRepository`'s translator, under every `#[Repository]`) and initialise a parent's `protected readonly` (`EloquentRepository`'s manifest and tracker) on the PHP 8.3 floor, where a `readonly` property is initialisable from its declaring class's scope alone. The `(array)` cast is what makes the slots exact: it mangles a private as `"\00wner\0name"`, so a private a parent and a child both declare under one name stays two slots, and a typed property the bean never initialised is simply absent and stays uninitialised on the proxy.

⚠ SELF-INVOCATION BYPASSES THE PROXY

A method calling `$this->otherMethod()` from *inside* the proxied class calls straight through `parent::`, skipping `__fireflyTxInterceptor` entirely — the same well-known Spring limitation. This is exactly why `AccountService::outerWithNested()` above doesn't just call some hypothetical `$this->innerNested()` method — it goes through the **injected `TransactionTemplate`** instead, which is the correct escape hatch for getting transactional semantics on an inner unit of work from within another method on the same instance.

One proxy, many advices

That proxy is no longer only about transactions. Any package can contribute a kind of advice by shipping one `#[Component]` implementing `AdviceSource` — `advice()` names its interceptor bean, its descriptor class and an **order**; `scan()` returns the rows it claims; `render()` turns a row back into the PHP literal the generated class bakes in. `ProxyPlanner` merges every source's rows into one `ProxyPlan`, applying the sources in `Advice::order`, so each method's advice list is outermost-first by construction. The generated class keeps the name it has always had, `{Target}`

`__FireflyTransactionalProxy`, but it now declares one private interceptor property and one baked static descriptor factory **per advice kind** the class uses, and

`MethodInvocation::proceed()` walks that list before reaching the terminal `fn (array $__fireflyArgs) => parent::m(...$__fireflyArgs)`. Lower `Advice` order runs *outer*: `MethodSecurityAdviceSource`'s advice is `100` and the transactional one is `1000`, so a refusal is thrown before a transaction is ever opened. An advice whose interceptor bean is absent fails the boot with a `ConfigurationException` unless it declared `inertWhenUnbound` — security's does, because "annotations are inert until `firefly.security.enabled` is on" is its documented state — and `InterceptorRegistry` then hands the proxy a `PassThroughInterceptor` in its place.

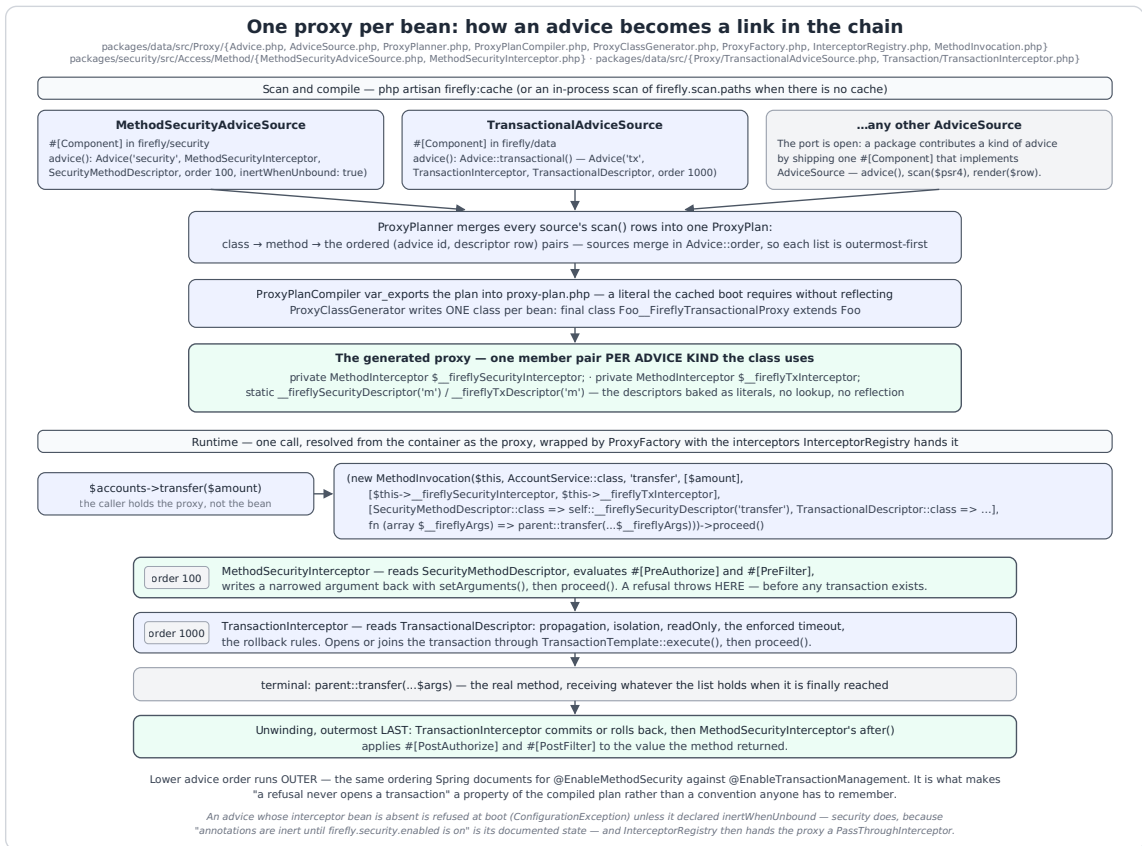


Figure 9.1 — Every AdviceSource contributes rows to one compiled ProxyPlan; a call then runs method security at advice order 100 before the transaction at 1000.

The timeout is enforced, not carried

`#[Transactional(timeout: 5)]` used to be metadata a reader could set and nothing would read. It is now enforced, and the way it is enforced is worth understanding, because one half of it can interrupt a running statement and the other half cannot.

```
$timeout = $outermost ? $this->effectiveTimeout($d) : 0;
$restore = $timeout > 0 && $this->settings->statementTimeout ? $this->timeouts-
```



```
>apply($connection, $timeout) : null;
// (int): hrtime(true) is an int on every 64-bit build; the cast keeps PHPStan's int|float|
false union out.
$deadline = $timeout > 0 ? (int) hrtime(true) + $timeout * 1_000_000_000 : null;
```

The wall clock is the half that always applies. A monotonic deadline is taken right after `beginTransaction()`, and when the work *returns* the template compares the clock against it. Overrun means roll back and throw `TransactionTimedOutException` — 504, error code `TRANSACTION_TIMED_OUT`:

```
if ($deadline !== null && (int) hrtime(true) > $deadline) {
    // Drain the tracker either way (it must not leak into the next unit of work); the
    rollback discards
    // the after-commit callbacks it queued.
    $this->dispatcher?->dispatchAfterCommit($d->connection);
    $connection->rollBack();
}

throw new TransactionTimedOutException(sprintf(
    'The transaction ran for longer than its %d second timeout and was rolled back.',
    $timeout,
));
}
```

Note what that cannot do: PHP is not going to interrupt a `SELECT` that is still inside the driver. The wall-clock check judges the method *after* it returns, which catches an overrun but does not stop one.

The driver statement timeout is the half that can. Immediately after `beginTransaction()` on the outermost transaction, `StatementTimeoutApplier` tells the database itself to give up, in whatever dialect that database speaks — `SET LOCAL statement_timeout` on postgres (transaction-scoped, which is precisely why it must be issued after the `BEGIN`), `max_execution_time` plus `innodb_lock_wait_timeout` on

mysql, mariadb's own `max_statement_time` on mariadb, and the busy timeout on sqlite, which is the only knob sqlite has. `apply()` returns the step that undoes it, and the template runs that in its `finally` — because the mysql variables are *session* variables, and on a persistent connection the next request would otherwise inherit this request's budget.

Every one of those statements is best-effort, exactly like the isolation `SET`: a driver that refuses one does not fail the transaction, and the wall-clock check still applies. You can turn the driver half off entirely with `firefly.data.transaction.statement-timeout=false` and keep the wall clock.

Three rules decide the number itself, and a test pins each one:

```
it('applies firefly.data.transaction.default-timeout when the attribute names none, and the
attribute wins when it does', function () {
  $template = new TransactionTemplate(null, null, new DataSettings(defaultTimeout: 1));

  expect(fn () => $template->execute(overrunOneSecond(...)))-
    >toThrow(TransactionTimedOutException::class)
    ->and(DB::table('widgets')->count()->toBe(0);

  $template->execute(overrunOneSecond(...), new TransactionalDescriptor(timeout: 10));

  expect(DB::table('widgets')->count()->toBe(1);
});
```

`#[Transactional(timeout:)]` wins over `firefly.data.transaction.default-timeout`; `0` on both means no deadline at all; and a **joined** transaction never times out on its own — only the outermost transaction takes a deadline, which is the Spring rule and the only one that makes sense when an inner method's budget would otherwise cut short work the outer method is still responsible for.

#[TransactionalEventListener] : hearing an event in a phase

Chapter 8's `#[AsEventListener]` hears an event when it is published. Inside a `#[Transactional]` method, "when it is published" is often the wrong moment: the row is written but not committed, so a listener that sends an e-mail or enqueues a job may be acting on a transaction that is about to roll back.

There are two different answers to that, and it is worth being precise about which one you want:

- `DomainEventDispatcher::publishAfterCommit()` — covered earlier in this chapter — defers **the event**. Nobody hears it at all before the commit.
- `#[TransactionalEventListener]` defers **the listener**. The event is published immediately (a plain `#[AsEventListener]` on the same class still sees it inside the transaction); *this* method is queued on the current transaction and invoked in the phase it asked for.

```
enum TransactionPhase: string
{
    case BEFORE_COMMIT = 'BEFORE_COMMIT';
    case AFTER_COMMIT = 'AFTER_COMMIT';
    case AFTER_ROLLBACK = 'AFTER_ROLLBACK';
    case AFTER_COMPLETION = 'AFTER_COMPLETION';
}
```

Phase	When it runs	What it sees
<code>BEFORE_COMMIT</code>	inside the transaction, just before the commit	the uncommitted row, at transaction level 1 — a throw aborts the commit
<code>AFTER_COMMIT</code> (the default)	after a successful commit	the committed row, at level 0
<code>AFTER_ROLLBACK</code>	after a rollback	the row gone
<code>AFTER_COMPLETION</code>	after either	whichever of the two happened

The framework's own fixture declares one listener per phase, which is the clearest way to read them:

```
#[TransactionalEventListener(phase: TransactionPhase::BEFORE_COMMIT, order: -10)]
public function beforeCommit(NoteSaved $event): void
{
    self::record('before-commit', $event);

    if ($event->title === 'veto') {
        throw new RuntimeException('vetoed before commit');
    }

    if ($event->title === 'chain') {
        $this->events->publish(new NoteIndexed($event->title));
    }
}
```

Four details, each of which you will eventually need:

- **The event class is inferred** from the method's first parameter type, exactly as `#[AsEventListener]` does it. `event:` is there for the case where you want to be explicit.

- **order:** follows the `#[Order]` convention — lower first — among transactional listeners *of the same phase*.
- **No transaction, no listener.** With nothing active the method is skipped, silently and on purpose: it asked to run in a phase that does not exist. `fallbackExecution: true` says "run me at once instead", which is what you want for a listener that must fire either way.
- **The attribute is inert metadata.** `TransactionalScanner` compiles it into the manifest's `listeners` map, `TransactionalEventListenerWiringPass` registers it, and `TransactionSynchronizationRegistry` queues it against the running transaction — no reflection at publish time. The whole mechanism is behind `firefly.data.transactional-event-listeners.enabled`, `true` by default.



`BEFORE_COMMIT` IS A VETO, AND THAT IS THE USEFUL PART

A last-moment invariant check belongs here: it runs inside the transaction, so throwing from it aborts the commit and takes the whole unit of work with it. Publishing from it also works — a second event queued while the `BEFORE_COMMIT` queue is draining is handled inside the same commit, not deferred to the next one.

How a real app actually gets a working proxy

Chapter 2 taught you the shape of a `#[Configuration]` class and its `#[Bean]` factory methods. For one release of this framework, every LaraFly application had to write one of them by hand — a `#[Configuration]` whose single `#[Bean]` loaded the cached `TransactionalManifest` — or `#[Transactional]` did nothing at all. **You no longer need that class**, and it is worth a page: the bug it worked around was one of the sharpest this framework has had, and the fix is a small lesson in how every compiled artifact is resolved.

`php artisan firefly:cache` emits **three** separate artifacts the proxy machinery reads. It runs `TransactionalScanner / TransactionalManifestCompiler` — one of the twelve scanner/compiler pairs — to write the manifest data to `bootstrap/cache/firefly/transactional.php`. It runs `ProxyPlanCompiler` over the merged `ProxyPlan` to write `bootstrap/cache/firefly/proxy-plan.php` — the artifact that decides which beans get a proxy at all and which advice each of their methods runs. And it runs `ProxyClassGenerator` to emit one `{Target}` `FireflyTransactionalProxy` source file **per class the plan names** — every class any `AdviceSource` claims, so a bean carrying only method-security rules gets one too — into a `proxies/` directory, plus a `proxies.php` classmap.

For a long time, nothing loaded the first of those three. `DataAutoConfiguration` bound an unconditional *empty* `TransactionalManifest`, the compiled `transactional.php` was referenced only by its own declaration, and so `hasProxyFor()` was always false and `TransactionalBeanPostProcessor` handed back every bean unwrapped. `#[Transactional]` was a **silent no-op** in any application that did not hand-write its own manifest configuration — which is exactly what that hand-written `#[Configuration]` existed to do, and exactly the failure mode the previous section warned about, arriving from the framework's own side.

It is fixed at the source. Both beans now resolve their artifact the way every Category-B artifact in this framework is resolved — **compiled file first, in-process scan second, empty last**:

```
#[Bean]
#[ConditionalOnMissingBean(TransactionalManifest::class)]
public function transactionalManifest(Container $container): TransactionalManifest
{
    if (($file = AppScan::cachedFile($container, AppScan::TRANSACTIONAL)) !== null) {
        ProxyMaterializer::classmap($container);
    }
}
```

```

    return TransactionalManifest::load($file);
}

$paths = AppScan::paths($container);
if ($paths === []) {
    return new TransactionalManifest([], []);
}

return (new TransactionalScanner)->scan($paths);
}

```

Three things in eighteen lines are worth reading slowly. The compiled file wins when it exists, which is the cached-production path. When it does not, the scanner runs in process over `firefly.scan.paths`, which is the development path — no `firefly:cache` required to get a working proxy while you are writing code. And `ProxyMaterializer::classmap()` is called **before** the manifest is returned, so the generated proxy classes are loadable before anything can ask for one; `TransactionalBeanPostProcessor` throws a `ConfigurationException` on a manifest that promises a proxy class it cannot find, so a half-written cache fails loudly at boot instead of quietly running unproxied.

`proxyPlan()` sits beside it with the same shape and one extra rung: `proxy-plan.php` first, then — for a cache written by a `firefly:cache` from before proxy plans existed — a transactional-only plan bridged from the manifest, then the in-process scan through every `AdviceSource` bean, then, with no scan paths at all, a transactional-only plan from whatever `TransactionalManifest` is bound. That last rung is what lets the framework's own capstone fixtures compile a manifest by hand and still get their `#[Transactional]` beans wrapped.

So the skeleton ships no `CachedTransactionalConfiguration`, and neither should your application. A LaraFly app gets a working `#[Transactional]` proxy in development from the scan and in production from the cache, with no wiring of its own.

The framework's test suites still supply an equivalent `#[Configuration]` inline — `packages/data/tests/Fixtures/Capstone/CapstoneTransactionalConfiguration` for the `AccountService` fixture above, and `samples/lumen/tests/Support/LumenTransactionalConfiguration` for the handlers this book takes from `samples/lumen` — because those suites do not boot through an application at all and have no `firefly.scan.paths` to scan.

💡 IF YOU WROTE ONE OF THESE CLASSES, DELETE IT

A hand-bound `TransactionalManifest` still works — `#[ConditionalOnMissingBean]` makes the framework back off from any competing bean definition — but it now pins your application to whatever that class does, and it was only ever a workaround.

The key lesson: domain events publish only through a `#[Transactional]` boundary

Everything in this section has been building to one fact, and it is the single most important thing this chapter teaches. Look again at where `DomainEventDispatcher::dispatchAfterCommit()` is actually called from inside `TransactionTemplate` — the catch arm, the timeout arm and the success arm of one method:

```
if ($outermost) {
    // Queue after-commit events BEFORE resolving the tx, on THIS descriptor's connection:
    Laravel fires
    // then on that connection's commit, discards on rollBack.
    $this->dispatcher?->dispatchAfterCommit($d->connection);
}

if ($this->shouldRollBack($translated, $e, $d)) {
```



```

    $connection->rollBack();
} else {
    // ...
    if ($deadline !== null && (int) hrtime(true) > $deadline) {
        // Drain the tracker either way (it must not leak into the next unit of work); the
        // rollback discards
        // the after-commit callbacks it queued.
        $this->dispatcher?->dispatchAfterCommit($d->connection);
        $connection->rollBack();
    }
    // ...
    if ($outermost) {
        $this->dispatcher?->dispatchAfterCommit($d->connection);
    }

    $this->commit($connection);

    return $result;
}

```

`dispatchAfterCommit()` is called from exactly **three** places, and all three are inside that one `TransactionTemplate` method — there is no fourth call site anywhere in the framework. Two of the three are there so the tracker is *drained* on the failure paths: draining on a rollback empties it without scheduling any publish, which is what stops one unit of work's events leaking into the next. And `dispatchAfterCommit()` itself only has anything to dispatch because of a second, equally load-bearing fact:

`EloquentRepository::save()` (Chapter 5) only registers an entity with `AggregateTracker` when `$connection->transactionLevel() > 0` — and the *only* code path that ever makes that true is `TransactionTemplate::runInTransaction()`'s own `beginTransaction()` call, a few lines above.

Chain those two facts together and the conclusion is unavoidable: **a method with no #[Transactional] never gets its transaction level raised, so save() never tracks the aggregate, so there is nothing for dispatchAfterCommit() to drain even if it were somehow called.** OpenWalletHandler's own docblock states this as the reason the attribute is there at all, not decoration:

```
/**
 // ...
 * #[Transactional] is LOAD-BEARING, not cosmetic: DefaultCommandBus opens no transaction of
 its own, and
 * EloquentRepository::save() only tracks the aggregate when transactionLevel() > 0. The
 generated transactional proxy
 * installs the TransactionTemplate that is the sole caller of
 DomainEventDispatcher::dispatchAfterCommit(), so without
 * this attribute the WalletOpened domain event would never publish and S5's ledger
 projector would never fire.
 // ...
 */
#[CommandHandler]
class OpenWalletHandler
{
    public function __construct(private readonly WalletRepository $wallets) {}

    #[Transactional]
    public function handle(OpenWallet $command): string
```

Recall from Chapter 7 that `DefaultCommandBus::send()` opens **no** transaction of its own — it correlates, validates, authorizes, and invokes the handler, full stop. Every bit of transactional behaviour you have seen in `OpenWalletHandler`, `DepositHandler`, `WithdrawHandler`, and `TransferHandler` comes *entirely* from the `#[Transactional]` attribute on their `handle()` methods, through the exact proxy mechanism this chapter just walked through. Strip the attribute from any one of them and the command still "succeeds" — the row still gets written by a plain, unproxied `save()` call — but

`WalletOpened` / `FundsDeposited` / `FundsWithdrawn` are raised into the aggregate's private event buffer and then **silently discarded**, because nothing ever drains that buffer. `LedgerProjector` (Chapter 6) would simply never fire, with no error, no warning, and a perfectly successful-looking HTTP response.

⚠ NO #[TRANSACTIONAL] , NO DOMAIN-EVENT PUBLISH — SILENTLY

This is not a slow failure or a misleading error message; it is a **complete, silent no-op** for every domain event the handler's aggregate raised. The write commits. The event vanishes. Nothing in the response, the logs, or the database schema tells you it happened — the only way to notice is that a downstream projector or listener you expected to fire never does. Treat `#[Transactional]` on a command handler that touches an aggregate as load-bearing, not stylistic.

The atomic `Transfer` : money can't vanish

Chapter 6 already showed you `TransferHandler`'s full source — one `#[Transactional(propagation: Propagation::REQUIRED)]` boundary wrapping a debit, a save, a credit, and a second save. What Chapter 6 didn't show you is the rigorous proof that the atomicity claim actually holds under a real failure. `samples/lumen/tests/Application/TransferSecurityTest.php` dispatches through the real `CommandBus` / `QueryBus` — the exact ports Chapter 7 introduced — and proves both directions:

```
it('transfers atomically: money is conserved across debit + credit', function () {
    // ...
    $commands = $this->fireflyContext()->get(CommandBus::class);
    // ...
    $queries = $this->fireflyContext()->get(QueryBus::class);
    // ...
});
```

```

    // ...
    $src = $commands->send(new OpenWallet('owner-A', Currency::EUR));
    // ...
    $dst = $commands->send(new OpenWallet('owner-B', Currency::EUR));
    $commands->send(new Deposit($src, 10000));

    $commands->send(new Transfer($src, $dst, 4000));

    // Debit + credit committed as one unit of work: the 10000 that left nowhere reappears
    split 6000/4000.
    expect($queries->ask(new GetBalance($src)))->toBe(6000);
    expect($queries->ask(new GetBalance($dst)))->toBe(4000);
  }->group('lumen');

  it('rolls the whole transfer back when the credit leg fails (money cannot vanish)', function
  () {
    // ...
    $commands = $this->fireflyContext()->get(CommandBus::class);
    // ...
    $queries = $this->fireflyContext()->get(QueryBus::class);

    // Destination in a DIFFERENT currency: the debited EUR amount cannot be credited into a
    USD wallet, so the
    // credit leg throws currency-mismatch AFTER the debit already ran -> the whole
    #[Transactional] tx rolls back.
    // ...
    $src = $commands->send(new OpenWallet('owner-A', Currency::EUR));
    // ...
    $dst = $commands->send(new OpenWallet('owner-B', Currency::USD));
    $commands->send(new Deposit($src, 10000));

    expect(fn () => $commands->send(new Transfer($src, $dst, 4000)))
      ->toThrow(CommandProcessingException::class);

    // Load-bearing, non-tautological proof that money cannot vanish: the source debit was
    ROLLED BACK (still 10000,
    // not 6000) and the destination never received anything (still 0). No value was created
    or destroyed.
    expect($queries->ask(new GetBalance($src)))->toBe(10000);
  }

```

```
    expect($queries->ask(new GetBalance($dst))>->toBe(0);
  }->group('lumen');
```

The second test is the one that matters. `Wallet::withdraw()` on the source ran successfully and raised `FundsWithdrawn` into its own buffer; then `Wallet::deposit()` on the destination threw a currency-mismatch `ConflictException` — *after* the debit's `save()` call had already executed inside the same, still-open transaction. Because `#[Transactional]`'s default `rollbackFor` catches every `Throwable`, the entire method rolls back: the source's debit is undone at the database level, and — because rollback happens *before* `dispatchAfterCommit()` would ever be reached on the success path, and `TransactionTemplate` also calls it on the `catch` arm precisely so a rolled-back drain still empties the tracker without ever scheduling a publish — neither `FundsWithdrawn` nor a `FundsDeposited` that never even got raised reaches any listener. Chapter 7's `CommandProcessingException` wraps the underlying `ConflictException`, and the two balance assertions are the whole point: not "the transfer failed" in the abstract, but the source's own `10000` came back **exactly**, and the destination's `0` never moved. No value was created; none was destroyed.

✓ What you learned

Concept	What it does
<code>#[Transactional]</code>	Class-level default, method-level <i>replacement</i> (not merge); <code>rollbackFor = [Throwable::class]</code> by default
<code>Propagation</code> (7 modes)	<code>TransactionTemplate::execute()</code> is the one source of truth for all seven, proxy and programmatic caller alike
<code>Isolation</code> / <code>readOnly</code>	Best-effort <code>SET TRANSACTION ...</code> statements on the outermost transaction only
<code>shouldRollBack()</code>	<code>noRollbackFor</code> wins over <code>rollbackFor</code> ; neither list matching also commits
<code>#[Transactional(timeout:)]</code>	A monotonic deadline taken on the outermost transaction only and judged when the work returns — overrun rolls back and throws <code>TransactionTimedOutException</code> (504, <code>TRANSACTION_TIMED_OUT</code>); the attribute beats <code>firefly.data.transaction.default-timeout</code> , <code>0</code> means none
<code>StatementTimeoutApplier</code>	The half that can actually interrupt a running statement: a best-effort per-dialect <code>SET</code> , issued after <code>BEGIN</code> and undone in <code>finally</code> (the mysql variables are session-wide); switched off with <code>firefly.data.transaction.statement-timeout</code>

Concept	What it does
<code>#[TransactionalEventListener]</code>	Defers the listener , not the event: four phases, <code>BEFORE_COMMIT</code> runs inside the transaction so a throw from it aborts the commit; no active transaction means the method is skipped unless <code>fallbackExecution: true</code>
The generated proxy	<code>{Target}__FireflyTransactionalProxy</code> extends <code>{Target}</code> ; routes each call through <code>TransactionInterceptor::run()</code> then <code>parent::</code>
Self-invocation bypass	<code>\$this->other()</code> inside the proxied class skips the interceptor entirely — use the injected <code>TransactionTemplate</code> instead
<code>DataAutoConfiguration::transactionalManifest() / proxyPlan()</code>	Compiled artifact first, in-process scan second, empty last — the reason no application needs to bind a manifest by hand
The key lesson	<code>TransactionTemplate</code> is the sole caller of <code>dispatchAfterCommit()</code> — no <code>#[Transactional]</code> , no domain-event publish, silently
<code>Transfer</code>	Debit + credit + both <code>save()</code> s in one boundary — a failed credit leg rolls back the already-run debit too

Try it yourself

1. **Reproduce the silent event loss.** In a scratch copy of the project (not the shipped `samples/lumen` package), remove `#[Transactional]` from a copy of `DepositHandler::handle()`, deposit into a wallet through the HTTP API, and confirm the balance *does* update (the row still gets written) while the ledger (`LedgerProjector`'s `ledger_entries` table) gets **no new row at all** — with no error anywhere.
2. **Prove `NOT_SUPPORTED` cannot suspend.** Give a method `#[Transactional(propagation: Propagation::NOT_SUPPORTED)]`, call it from inside another `#[Transactional(propagation: Propagation::REQUIRED)]` method on the *same* connection (through `TransactionTemplate`, not self-invocation), and confirm — per this chapter's "Known-latent" description — that the inner work still runs inside the outer transaction rather than truly outside one.
3. **Read the generated proxy source.** After running `php artisan firefly:cache` in a project with a `#[Transactional]` class, open the emitted file under `bootstrap/cache/firefly/proxies/` and find the four kinds of member `ProxyClassGenerator` renders: the `(new \Firefly\Data\Proxy\MethodInvocation(...))->proceed()` override it writes for every advised method, the private `$_fireflyTxInterceptor` property, the private static `__fireflyTxDescriptor('m')` factory the descriptor literal is baked into, and the public static `__fireflyAdvice()` table `ProxyFactory` reads to know which interceptor bean belongs in which property. Then give a *second* bean a `#[PreAuthorize]` rule and **no** `#[Transactional]` at all, re-run `firefly:cache`, and confirm the plan named it too: it gets a proxy of its own, carrying `$_fireflySecurityInterceptor` and `__fireflySecurityDescriptor('m')` instead. Finally put both advices on one method and read the chain order straight off the generated source — the override's

interceptor array is `[$this->__fireflySecurityInterceptor, $this->__fireflyTxInterceptor]`, outermost first, exactly the order Figure 9.1 draws.

4. **Make a timeout fire, then take the driver half away.** Set

`firefly.data.transaction.default-timeout` to `1`, give a `#[Transactional]` method a `sleep(2)` between a `save()` and its return, and confirm you get a `504` with error code `TRANSACTION_TIMED_OUT` and **no** row. Then set `firefly.data.transaction.statement-timeout` to `false` and run it again: the same failure, from the wall clock alone. Finally raise the budget with `#[Transactional(timeout: 10)]` on the method and confirm the attribute beats the configured default — and that calling the method from *inside* another `#[Transactional]` method gives it no deadline of its own at all.

5. **Veto a commit from BEFORE_COMMIT.** Declare a

`#[TransactionalEventListener(phase: TransactionPhase::BEFORE_COMMIT)]` that throws whenever an event carries some sentinel value, publish that event from inside a `#[Transactional]` method, and confirm the whole unit of work rolls back. Then change the phase to `AFTER_COMMIT` and confirm the identical throw now leaves the row committed. Last, call the same publishing method with no transaction anywhere and confirm the listener is skipped in silence — then add `fallbackExecution: true` and watch it run immediately.

Security: Authentication and Authorization

By the end of this chapter you will know `firefly/security`'s immutable principal model (`Authentication` , `SecurityContext` , `SecurityContextHolder`), how `DaoAuthenticationProvider` authenticates a username/password pair while defeating enumeration attacks, how `JwtService` refuses to boot with a weak secret and refuses to accept a token with no expiry, the deny-by-default `HttpSecurity` URL DSL, how a person actually signs in through `SecurityContextPersistenceFilter` and `FormLoginFilter` — including the login page the framework ships and the one it refuses to replace — how logging out and staying logged in work, and why a remember-me request is *marked* rather than freshly authenticated, how `firefly.security.http.entry_point` decides between a login redirect and a bare `401` , how the principal is injected into a controller signature, the security event family a sign-in, a logout and a denial each publish, and — the centrepiece of this chapter — exactly how `#[PreAuthorize]` is evaluated by a hand-rolled, **closed-whitelist** expression grammar that never calls `eval()` , normalises `hasRole('X')` to a check against the granted authority `'ROLE_X'` , is enforced at the CQRS bus you met in Chapter 7, and now holds on *any* stereotyped bean through Chapter 9's proxy rather than only at the bus.

i NEW TERM: AUTHENTICATION VS. AUTHORIZATION

Authentication answers "who is making this request?" — it produces a principal.

Authorization answers "is that principal allowed to do *this specific thing*?" `firefly/security` keeps the two strictly separate: `Authentication / SecurityContextHolder` carry the answer to the first question; `HttpSecurity`, `#[PreAuthorize]`, and `AuthorizationChecker` all answer the second, by evaluating an expression **against** whatever the first question already settled.

The principal model

`Authentication` is an immutable token built exclusively through two named factories, so the two states it can be in — an unauthenticated request still carrying raw credentials, and an authenticated principal carrying granted authorities — can never be confused:

```
final class Authentication
{
    // ...
    private function __construct(
        public readonly string $name,
        public readonly mixed $principal,
        public readonly mixed $credentials,
        public readonly array $authorities,
        public readonly bool $authenticated,
        public readonly array $attributes,
    ) {}
    // ...
    public static function authenticated(string $name, mixed $principal, array $authorities,
        array $attributes = []): self
    {
        return new self($name, $principal, null, $authorities, true, $attributes);
    }
}
```

```

    public static function unauthenticated(string $name, mixed $principal, mixed
$credentials): self
    {
        return new self($name, $principal, $credentials, [], false, []);
    }
    // ...
    public function eraseCredentials(): self
    {
        $principal = $this->principal instanceof CredentialsContainer ? $this->principal-
>eraseCredentials() : $this->principal;

        return new self($this->name, $principal, null, $this->authorities, $this-
>authenticated, $this->attributes);
    }
    // ...
}

```

The private constructor means the only way to build one is `authenticated()` (credentials always `null`, `authenticated` always `true`) or `unauthenticated()` (authorities always `[]`, `authenticated` always `false`) — there is no path that lets you construct an "authenticated" token with leftover raw credentials still attached.

`eraseCredentials()` is the same argument made once more, one level down, and its two lines are worth reading closely. It returns a **new** instance rather than clearing a field, so a reference someone else is already holding can never observe a token that was cleared and then repopulated. And it does not stop at its own `credentials`: a principal that is itself a `CredentialsContainer` — the shipped `User`, whose `getPassword()` is the encoded hash — is asked for its own credential-free copy, because a password hash reachable through `$authentication->getPrincipal()` is a password hash on the wire the moment something serialises the context. This is the form `SessionSecurityContextRepository` writes to the session. Notice what does *not* call it: the authentication manager, on success. The token a filter receives still carries

the principal's encoded password, because the remember-me cookie's signature is computed from it — so only a **store** ever needs the erased copy. `GrantedAuthority` / `SimpleGrantedAuthority` wrap a bare authority string (`'ROLE_ADMIN'` , `'orders:read'`):

```
interface GrantedAuthority
{
    public function getAuthority(): string;
}
```

```
final readonly class SimpleGrantedAuthority implements GrantedAuthority
{
    public function __construct(private string $authority) {}

    public function getAuthority(): string
    {
        return $this->authority;
    }
}
```

`SecurityContext` is an immutable snapshot of the current `Authentication` (`anonymous()` is the unauthenticated zero-value), and `SecurityContextHolder` is where a request finds it:

```
final class SecurityContextHolder
{
    private const KEY = 'firefly.security.context';

    public static function getContext(): SecurityContext
    {
        // ...
        $context = Context::get(self::KEY);
    }
}
```

```

    return $context instanceof SecurityContext ? $context :
SecurityContext::anonymous();
}

    public static function setContext(SecurityContext $context): void
    {
        Context::add(self::KEY, $context);
    }

    public static function clearContext(): void
    {
        Context::forget(self::KEY);
    }

    public static function getAuthentication(): ?Authentication
    {
        return self::getContext()->getAuthentication();
    }
}

```

It is `static` on purpose — Spring's own `SecurityContextHolder` shape — because it is a request-local accessor, not an injected collaborator, backed by Laravel's own `Context` facade (per-request, Octane-safe). `getContext()` never requires a null-check at the call site: an unset context resolves to `SecurityContext::anonymous()` automatically. The load-bearing guarantee against one request's principal leaking into the next is that every authentication filter calls `clearContext()` in a `finally` block on exit — PHP always runs a `finally`, so no exception path can skip it.

Authenticating a request: `DaoAuthenticationProvider`

`DaoAuthenticationProvider` checks a username/password pair against a `UserDetailsService`, and it is written specifically to defeat username-enumeration timing attacks:

```
final class DaoAuthenticationProvider implements AuthenticationProvider
{
    // ...
    private const DUMMY_PASSWORD = 'firefly-dummy-password-for-timing-mitigation';

    private readonly string $dummyHash;

    public function __construct(
        private readonly UserDetailsService $users,
        private readonly PasswordEncoder $encoder,
    ) {
        // Precompute with the REAL encoder (same algorithm/cost) so the user-not-found
        verify below is
        // ...
        $this->dummyHash = $encoder->encode(self::DUMMY_PASSWORD);
    }
    // ...
    public function authenticate(Authentication $authentication): Authentication
    {
        $raw = $authentication->getCredentials();
        $presented = is_string($raw) ? $raw : '';

        try {
            $user = $this->users->loadUserByUsername($authentication->getName());
        } catch (UsernameNotFoundException) {
            // Enumeration mitigation: a real verify against the dummy hash (equal timing) +
            the SAME generic 401
            // (identical message/body) as a bad password. Unknown user === wrong password
            to any observer.
            $this->encoder->matches($presented, $this->dummyHash);
        }
    }
}
```



```

        throw new BadCredentialsException('Bad credentials.');
```

```

    }

    if (! $this->encoder->matches($presented, $user->getPassword())) {
        throw new BadCredentialsException('Bad credentials.');
```

```

    }

    if (! $user->isAccountNonLocked()) {
        throw new LockedException('Account is locked.');
```

```

    }

    if (! $user->isEnabled()) {
        throw new DisabledException('Account is disabled.');
```

```

    }

    return Authentication::authenticated($user->getUsername(), $user, $user-
>getAuthorities());
}
}

```

An unknown username is deliberately made **indistinguishable** from a wrong password: on `UsernameNotFoundException`, the provider still runs a real password verification — against a fixed dummy hash, precomputed at construction time with the *same* encoder and cost factor the real users use — before throwing the exact same generic `BadCredentialsException`, with the same message, that a genuine bad password produces. No timing difference, no content difference. Account-status failures (`locked` / `disabled`) are checked only **after** a correct password — the OWASP-recommended order — so an attacker without the right password learns nothing about whether the account even exists, let alone its state. On success, the returned token is freshly constructed via `Authentication::authenticated()`, which never carries the raw password forward at all.

JWT: mandatory expiry, weak-secret boot refusal

`JwtService` enforces two fail-closed invariants that never require a developer to remember them:

```
final class JwtService
{
    private const MIN_SECRET_BYTES = 32;

    private const PLACEHOLDERS = ['changeme', 'change-me', 'secret', 'password', 'insecure',
    'your-secret-key', 'null', ''];

    public function __construct(
        private readonly string $secret,
        private readonly string $algorithm = 'HS256',
        private readonly int $leewaySeconds = 0,
    ) {
        // ...
        if (in_array(strtolower($secret), self::PLACEHOLDERS, true) || strlen($secret) <
self::MIN_SECRET_BYTES) {
            throw new WeakSigningSecretException(
                // ...
            );
        }
    }
    // ...
    public function decode(string $token): array
    {
        JWT::$leeway = $this->leewaySeconds;

        try {
            $decoded = JWT::decode($token, new Key($this->secret, $this->algorithm));
        } catch (ExpiredException $e) {
            throw new TokenExpiredException('JWT has expired.', 'TOKEN_EXPIRED', $e);
        } catch (SignatureInvalidException $e) {
            throw new InvalidTokenException('JWT signature is invalid.', 'INVALID_TOKEN',
            $e);
        }
    }
}
```

```

    // ...
    }
    // ...
    $claims = (array) $decoded;
    if (! array_key_exists('exp', $claims)) {
        throw new InvalidTokenException('JWT is missing the mandatory exp claim.');
```

The first invariant fires at **construction**, not at first use: a secret shorter than 32 bytes, or one of a fixed list of common placeholders ('changeme' , 'secret' , 'password' , an empty string...), throws `WeakSigningSecretException` — the application refuses to boot at all rather than sign tokens with a guessable key. The second fires on every `decode()`: even a cryptographically valid signature is rejected outright if the token carries no `exp` claim, so a forgotten expiry can never produce an eternally valid token. `firefly.security.jwt.enabled` and `firefly.security.oauth2.resource_server.enabled` are additionally **mutually exclusive** — enabling both is refused at boot, matching Spring's one-bearer-mechanism model.

Deny-by-default URL rules: `HttpSecurity`

`HttpSecurity` is a fluent DSL building an ordered list of URL rules, evaluated first-match-wins by `HttpSecurityFilter`, with **no match at all denying the request**:

```

public function requestMatcher(string $pattern): self
{
    $this->pending = self::normalisePattern($pattern);
```

```

    return $this;
}

public function anyRequest(): self
{
    return $this->requestMatcher('*');
}

public function permitAll(): self
{
    return $this->finalise('permitAll()');
}

public function denyAll(): self
{
    return $this->finalise('denyAll()');
}

public function authenticated(): self
{
    return $this->finalise('isAuthenticated()');
}

public function hasRole(string $role): self
{
    return $this->finalise("hasRole('".self::assertSafeValue($role)."'");
}

public function hasAuthority(string $authority): self
{
    return $this->finalise("hasAuthority('".self::assertSafeValue($authority)."'");
}

public function hasScope(string $scope): self
{
    return $this->finalise("hasScope('".self::assertSafeValue($scope)."'");
}

// ...

```

```
private static function normalisePattern(string $pattern): string
{
    $normalised = ltrim($pattern, '/');

    return $normalised === '' ? '/' : $normalised;
}
```

Every rule compiles to the **exact same expression grammar** `#[PreAuthorize]` uses below — `HttpSecurity` is a builder that emits `permitAll()` / `hasRole('ADMIN')` - shaped strings, not a second authorization engine. `HttpSecurityFilter` evaluates the compiled rules against the request path and, on the first pattern match, checks the rule's expression; a request matching **no** rule at all is denied — fail-closed, not fail-open. A denial renders as a `401` when the context is anonymous (authenticate first) and a `403` when authenticated but under-privileged.

`requestMatcher()` is the one door every rule comes through — `anyRequest()` and `fromConfig()` both call it — and it normalises the pattern before storing it. The filter matches `Str::is($rule->pattern, $request->path())`, and Laravel's `path()` never carries a leading slash, so `normalisePattern()` strips one: `/api/*` and `api/*` are the **same** rule. A `'/actuator/health'` written the way half the world writes it — the way every route in a `RouteManifest` spells it — is a live rule, not the silently dead one it used to be, where the request matched nothing, deny-by-default refused it, and the operator read a `401` on the one path they had explicitly opened. The root path is the single exception that keeps its slash, because `path()` answers `'/'` for it and never `''`. What normalisation does **not** do is rewrite a route template: a pattern is matched against `api/orders/7`, never against `api/orders/{id}`, so a pattern carrying a placeholder is still a dead rule no normalisation can rescue — `api/orders/*` is the spelling that covers it.

`assertSafeValue()` rejects any role/authority value containing a single quote, for a reason that matters a great deal once you've read the next section: a legitimate role or authority string never contains one, but a value that did could otherwise splice extra grammar into the fixed expression literal it gets interpolated into.

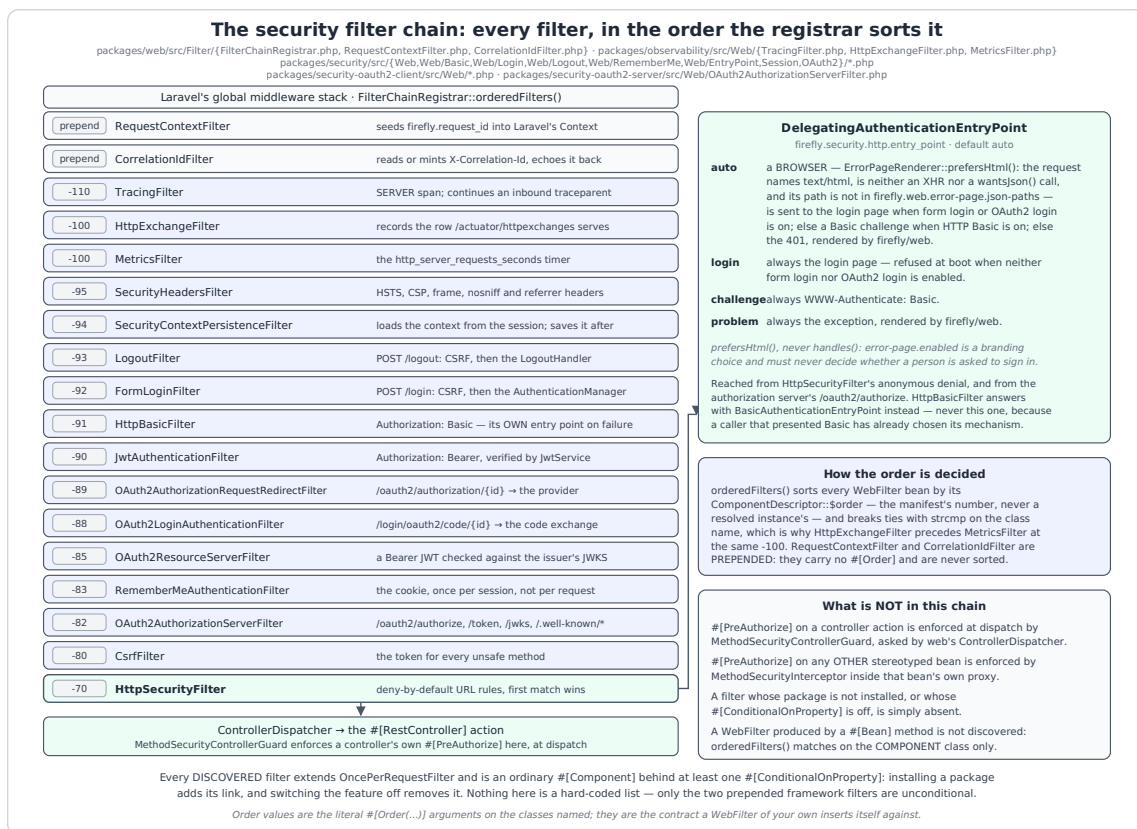


Figure 10.1 — `HttpSecurityFilter` is the last link of an ordered chain: every filter's real `#[Order]` value, the two framework filters prepended ahead of all of them, and the `DelegatingAuthenticationEntryPoint` an anonymous denial reaches — a login redirect, a Basic challenge or a 401.

Three of those numbers do one job between them.

`OAuth2AuthorizationRequestRedirectFilter` (-89) starts a sign-in with a provider, `OAuth2LoginAuthenticationFilter` (-88) finishes it, and `OAuth2AuthorizationServerFilter` (-82) is what a request hits when you are the

provider. The first two are `firefly/security-oauth2-client`, the third is `firefly/security-oauth2-server`, and neither package needs the other: either half works against any conformant counterpart on the far side.

Figure 10.2 follows one sign-in through both halves at once, because the thing worth understanding about the authorization-code grant is not any single step but which secret each party holds at which moment. Start at the top left with a rule you have already written: `GET /orders` matches no `permitAll()`, `HttpSecurityFilter (-70)` denies it, and the entry point does exactly what it does for form login — saves the GET and redirects to `/login`. What is different is that the login page now carries a button per registration, and that button goes to `/oauth2/authorization/{id}`. From there the browser never carries a secret again: the code it brings back is worthless without the PKCE verifier, which never left the relying party's session, and is spent in a back-channel `POST /oauth2/token` no browser ever sees.

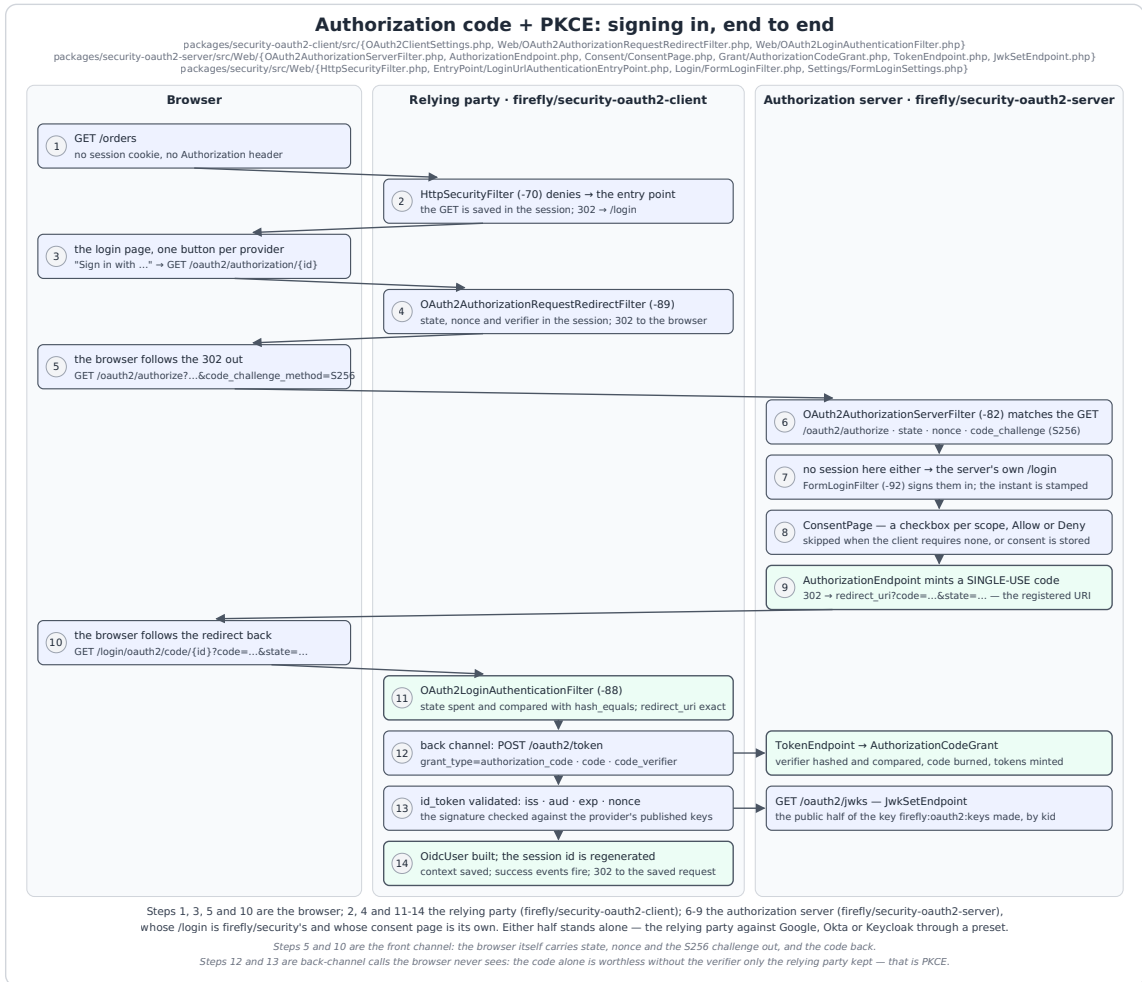


Figure 10.2 — One sign-in, end to end: HttpSecurityFilter denies the request and the entry point saves it and redirects to /login, the login page's provider button starts the round trip at /oauth2/authorization/{id} with a single-use state, a nonce and an S256 code challenge, the authorization server runs its own login and consent pages before minting a single-use code, and the relying party spends that code and the verifier in a back-channel token exchange whose id token is checked against /oauth2/jwks.

Signing a person in: the session, the form and the page

Everything so far in this chapter authenticates a request that arrives carrying its own credential — a JWT, a Basic header. That is the right shape for an API and the wrong shape for a person, who signs in once and then browses. Three filters and one page make that work, and all three are on by one key.

The context that survives a request

```
protected function doFilter(Request $request, Closure $next): mixed
{
    $entry = SecurityContextHolder::getContext();
    $loaded = null;

    if (! $entry->isAuthenticated()) {
        $loaded = $this->repository->load($request);
        if ($loaded !== null) {
            SecurityContextHolder::setContext($loaded);
        }
    }

    try {
        return $next($request);
    } finally {
        $current = SecurityContextHolder::getContext();
        if ($current->isAuthenticated() && $current !== $entry && $current !== $loaded) {
            $this->repository->save($current, $request);
        }
        SecurityContextHolder::clearContext();
    }
}
```

`SecurityContextPersistenceFilter` runs at `#[Order(-94)]` — right after the headers filter and **ahead of every authentication filter**, so a session-held principal is what `HttpSecurityFilter` and the method-security guards see. Twenty lines, and every one of them is deliberate:

- It does **not** load when the holder already carries an authenticated context. A test's acting principal, or an outer middleware, established one on purpose.
- On the way out it saves only a context that **changed** during the request and is **still on the holder**. It never deletes the stored one: an inner authentication filter that clears the holder in its own `finally` does so before this exit runs, and an empty holder at that point means "the request is over", not "sign out".
- The `finally` clear is the load-bearing Octane guarantee. Whatever happened, nothing bleeds into the next request on the same worker.

What is stored never carries a credential: a `CredentialsContainer` principal — the shipped `User` is one — is written without its encoded password.

⚠ FIREFLY FILTERS ARE GLOBAL; LARAVEL'S SESSION IS NOT

Firefly filters are global kernel middleware, and Laravel starts the session in the `web` route group, *later*. When session security is on, a boot pass called `SessionSecurityBootstrap` pushes `EncryptCookies`, `AddQueuedCookiesToResponse` and `StartSession` onto the **global** stack ahead of the filter chain, removes them from the `web` group, and excludes them on the matched route at dispatch time. Without that, a route carrying the group would decrypt already-decrypted cookies, null them, and mint a fresh session — losing the login. A session driver is required; boot refuses without one.

The form

`firefly.security.form_login.enabled` turns on two things at once.

`FormLoginFilter` (`-92`) handles `POST /login` **before routing**, exactly as Spring's `UsernamePasswordAuthenticationFilter` does — so it needs no route, no CSRF

group and no controller. Its order is: the session CSRF token first (a login CSRF is an attack, so this check is independent of `CsrfFilter`), then the `AuthenticationManager`, then, on success, a regenerated session id, the context stored, the events published, the remember-me cookie when it was asked for, and a redirect to whatever the entry point saved.

The capstone test walks that whole path, and it is the clearest description of the mechanism there is:

 \$THIS->EVENTS IN THIS CHAPTER'S LISTINGS

Every listing below that reads `$this->events->interactive()`, `->successes()`, `->logouts()` or `->denials()` is holding `Firefly\Testing\Double\RecordingAuthenticationEvents` — the security-aware recording double Chapter 12 catalogues, which answers the five event kinds by name instead of making a test filter a mixed list by class. It replaces the `ApplicationEventPublisher` port and must be bound **before boot**, from `defineFireflyEnvironment()`; Chapter 12 also explains the `$forwardTo` argument that keeps real listeners hearing the event.

```
it('signs in with the right password: a NEW session id, the context in the session, the
saved request honoured, the page then accessible', function () {
    // ...
    // 1. Refused at /home → redirected to /login with /home saved.
    $refused = $this->get('/home');
    $refused->assertRedirect('/login');

    // 2. The login page, in that same session.
    $this->forgetSession();
    $page = $this->followSession($refused)->get('/login');
    $page->assertOk();
    $before = (string) $page->getCookie($this->sessionCookieName())?->getValue();

    // 3. The POST: right password → 302 to the SAVED request, with a regenerated session
```

```

id.
    $this->forgetSession();
    $login = $this->followSession($page)->post('/login', ['username' => 'ada', 'password' =>
'secret', '_token' => $this->csrfTokenFrom($page)]);
    $login->assertRedirect('/home');
    $after = (string) $login->getCookie($this->sessionCookieName())?->getValue();

    expect($after)->not->toBe($before)
        ->and($this->events->interactive())->toHaveCount(1)
        ->and($this->events->interactive()[0]->mechanism)-
toBe(InteractiveAuthenticationSuccessEvent::FORM)
        ->and($this->events->interactive()[0]->authentication->getName())->toBe('ada')
        ->and($this->events->successes())->toHaveCount(1)
        ->and(SecurityContextHolder::getContext()->isAuthenticated())->toBeFalse();

    // 4. The protected page, from the session alone.
    $this->forgetSession();
    $this->followSession($login)->get('/home')->assertOk()->assertSee('Signed in as ada');
    $this->followSession($login)->getJson('/whoami')->assertJson(['name' => 'ada',
'authorities' => ['ROLE_USER']]);

```

The session id changes on every interactive sign-in — session-fixation protection, and it is not optional. A failure publishes an `AuthenticationFailure*Event` carrying the username and the source IP and **never the password**, then redirects to `/login?error`.

The page the framework ships, and the one it refuses to replace

The framework renders its own login page, in the error page's design. It is a real class, `LoginPage`, mounted by a boot pass — and that boot pass carries the single most considerate rule in the package:

```

/**
// ...
* A LOGIN PAGE THE APPLICATION ALREADY OWNS IS LEFT ALONE. Laravel's RouteCollection keeps

```

```

one route per
* method and URI and the LAST registration wins, and this pass runs after the application's
routes – so
* mounting unconditionally would replace a Breeze/Fortify-style controller, or any
#[GetMapping('/login')],
* the moment `form_login.enabled` went on, with no log, no refusal and no failing test: the
framework page
* would simply appear where the application's used to be. Spring's rule is the one applied
instead: a custom
* login page belongs to the application (formLogin().loginPage() switches the default page
generator off),
* and the framework generates a page only when nobody else answers that address. So when a
GET route at
// ...
*/
final class LoginRouteRegistrar implements BootPass

```

So: if a **GET** route at the configured login path already exists when the pass runs — an attribute route, a routes file, Breeze, Fortify, anything — **nothing is mounted**.

Everything else still works: **HttpSecurityFilter** permits the page by path (or the redirect would loop), the entry point redirects to it, and **FormLoginFilter** still answers the POST before routing. Your own form needs only to post the session token to the login processing URL.

Between those two extremes there is a middle option: point **view** at a Blade template of your own and the framework will render that instead, handing it a **LoginPageModel** with the form's fields and every provider link.

Where the users come from

AuthenticationManager asks a **UserDetailsService**, and the one most applications want reads their own table:

```

/**
// ...
* The expected schema (every name configurable under firefly.security.users):
*   email       the username (unique)
*   password    the ENCODED password, `{id}`-prefixed for the delegating encoder (`{bcrypt}
$2y$...`)
*   enabled     boolean, optional (enabled_column; empty means every account is enabled)
*   locked      boolean, optional (locked_column; empty means no account is locked)
*   authorities a JSON list of strings – `["ROLE_USER", "orders:read"]` – or a relation
(`roles.name`);
*             optional (authorities; empty means the model carries none and every account
gets [])
// ...
*/
final class EloquentUserDetailsService implements UserDetailsService
{
    public function __construct(private readonly UserStoreSettings $settings) {}

    public function loadUserByUsername(string $username): UserDetails

```

One query per lookup, and the row is mapped to the immutable `User` value object rather than handed out as the model — the principal ends up in the session and inside event objects, and a model there would drag its connection, its relations and its attributes along with it.

One detail is worth more than it looks. A configured column name the row does not carry is **refused**, not read as null. Laravel's `Model::getAttribute()` answers `null` for an attribute a row does not have, and null is the wrong answer in every position here — for the locked column it is the dangerous one, because `locked_column: 'is_locked'` against a table whose column is `locked` would make every row unlocked and switch account lockout off with no error anywhere. Opting out of an optional column is the **empty string**, which a typo never is.

Laravel's stock `users` table (`id`, `name`, `email`, `password`) works with no schema change at all: set `authorities` to `''`.

Logging out, and staying logged in

Logging out

```
it('ends the session on POST /logout, expires the named cookies, publishes the event, and
lands on ?logout', function () {
    // ...
    $logout = $this->followSession($login)->withCookie('theme', 'dark')->post('/logout',
['_token' => $this->csrfTokenFrom($this->followSession($login)->get('/login'))]);
    $logout->assertRedirect('/login?logout');

    $theme = $logout->getCookie('theme');
    expect($theme)->not->toBeNull()
        ->and($theme->getExpiresTime())->toBeLessThan(time())
        ->and($this->events->logouts())->toHaveCount(1)
        ->and($this->events->logouts()[0]->authentication?->getName())->toBe('ada');

    // The OLD session is gone: its cookie no longer authenticates anyone.
    $this->forgetSession();
    $this->followSession($login)->getJson('/whoami')->assertStatus(401);
    $this->followSession($login)->get('/home')->assertRedirect('/login');
});
```

`LogoutFilter` (-93) handles `POST /logout` — **POST only, CSRF-checked**; a `GET` falls through to whatever route is there, which is what you want, because a link that logs people out is a CSRF vector. It expires the remember-me cookie and every name listed in `delete_cookies`, invalidates the session, publishes `LogoutSuccessEvent`, and redirects.

That sequence lives in one bean, `LogoutHandler`, and **every path that ends a session calls it** — the filter above, and OpenID Connect RP-initiated logout when the authorization-server package is on. So the three settings that govern signing out govern both by construction; a second path cannot quietly obey a subset of them.

Staying logged in

`TokenBasedRememberMeServices` is Spring's, including the signature:

```
username : expiry : HMAC-SHA256(username : expiry : password-hash, key)
```

That the **password hash** is inside the signature is the whole design. Change the password and every outstanding cookie stops verifying, everywhere, with no server-side token table to clean up. The key is held to the same rule as the JWT secret and the boot refuses a weak one.

```
// The session is gone; the cookie ALONE signs ada back in – through the remember-me
mechanism.
$this->destroySessionOf($remembered);
$this->forgetCookies();
$this->events->reset();
$back = $this->withCookie('remember-me', $token)->get('/home');
$back->assertOk()->assertSee('Signed in as ada');

expect($this->events->interactive()->toHaveCount(1)
    ->and($this->events->interactive()[0]->mechanism)-
    >toBe(InteractiveAuthenticationSuccessEvent::REMEMBER_ME);

// And the NEW session it opened carries ada from now on, without the cookie.
$this->forgetSession();
$this->followSession($back)->getJson('/whoami')->assertJson(['name' => 'ada']);

// A tampered cookie is anonymous, not an error.
$this->forgetSession();
```



```
$this->forgetCookies();
$this->withCookie('remember-me', stripslashes($token))->get('/home')->assertRedirect('/login');
```

`RememberMeAuthenticationFilter` (-83) only runs when the session holds no principal, and when it succeeds it performs a **full interactive sign-in**: a new session id, the context stored, an `InteractiveAuthenticationSuccessEvent`. So the cookie is consulted once per *session*, not once per request. A tampered cookie is anonymous rather than an error — there is nothing to report, because anyone can send bytes.

A disabled or locked account is refused even on a genuine cookie, and logout expires it.

⚠ REMEMBERED IS NOT THE SAME AS AUTHENTICATED

A cookie sign-in is **marked remembered**. It is evidence that this browser signed in at some point in the past, not evidence that the person is at the keyboard now. Spring draws exactly this line, and it is why a "change password" or "delete account" page should require a fresh authentication rather than accepting a remembered one.

Redirect or 401: the entry point decides

One deny-by-default rule has to produce two completely different responses. A person browsing to `/orders` should end up at a login page. A JavaScript client calling `/api/orders` should get a 401 it can act on — and a login page rendered into a `fetch()` is the single most confusing failure mode a web framework can produce.

`HttpSecurityFilter` hands every **unauthenticated** denial to the `AuthenticationEntryPoint` named by `firefly.security.http.entry_point`:

Mode	What an anonymous denial gets
<code>auto</code> (the default)	a browser gets the login page with the request saved; a Basic-enabled app answers <code>401</code> + <code>WWW-Authenticate: Basic</code> ; otherwise a 401 for firefly/web to render
<code>login</code>	always the login-page redirect — refused at boot when no login is enabled, because there would be nowhere to send anyone
<code>challenge</code>	always <code>401</code> + <code>WWW-Authenticate: Basic</code>
<code>problem</code>	always the 401, as <code>application/problem+json</code> or the HTML 401 page

"A browser" is not a guess about the `User-Agent`. It is the error page's own content negotiation, `ErrorPageRenderer::prefersHtml()`: the request names `text/html` (or `application/xhtml+xml`), is neither an `XMLHttpRequest` nor a `wantsJson()` call, and is not under `firefly.web.error-page.json-paths`. The same test the error page uses, reused rather than re-invented — so the two can never disagree:

```
it('redirects a browser to the login page and answers a JSON client with a 401 problem',
function () {
    // ...
    $this->get('/home')->assertRedirect('/login');

    $this->getJson('/api/orders')
        ->assertStatus(401)
        ->assertHeader('Content-Type', 'application/problem+json')
        ->assertJson(['code' => 'AUTHENTICATION_FAILED']);

    // An api/* path is a machine surface whatever the Accept header says
    (firefly.web.error-page.json-paths).
```

```

    $this->get('/api/orders', ['Accept' => 'text/html'])->assertStatus(401);

    expect($this->events->denials()->toBe([]);
});

```

Read the last line. An **anonymous** denial publishes nothing — it is not a security event, it is a visitor who has not signed in yet, and a log full of them is noise that hides the real thing. An **authenticated** denial is the real thing: it stays a 403 and it publishes `AuthorizationDeniedEvent` naming the principal, the subject and the expression that refused it:

```

    $this->followSession($signedIn)->getJson('/admin/panel')->assertStatus(403)-
    >assertJson(['code' => 'ACCESS_DENIED']);

    expect($this->events->denials()->toHaveCount(1)
        ->and($this->events->denials()[0]->authentication->getName()->toBe('ada')
        ->and($this->events->denials()[0]->subject)->toBe('GET /admin/panel')
        ->and($this->events->denials()[0]->expression)->toBe("hasRole('ADMIN')"));

```

Two callers reach the entry point: `HttpSecurityFilter`'s anonymous denial, and the authorization server's `/oauth2/authorize` when it needs the person signed in first. `HttpBasicFilter` deliberately does **not** — it answers with its own `BasicAuthenticationEntryPoint` whatever `entry_point` says, exactly as Spring's `BasicAuthenticationFilter` keeps its own. A caller that presented Basic credentials has already chosen its mechanism, and redirecting it to an HTML page is the wrong answer to an API client.

Method security: the closed-whitelist expression grammar

This is the section the rest of the chapter has been building toward. `#[PreAuthorize]`, `#[Secured]`, and `#[RolesAllowed]` all guard a method with a boolean expression:

```
#[Attribute(Attribute::TARGET_METHOD | Attribute::TARGET_CLASS)]
final readonly class PreAuthorize
{
    /**
     * @param ?string $code the error code a refusal carries instead of ACCESS_DENIED
     * @param ?string $message the sentence a refusal carries instead of the framework's
     */
    public function __construct(
        public string $expression,
        public ?string $code = null,
        public ?string $message = null,
    ) {}
}
```

```
/** Requires ANY of the listed authorities (JSR-250 / Spring @Secured). Compiled to
hasAnyAuthority(...). */
#[Attribute(Attribute::TARGET_METHOD | Attribute::TARGET_CLASS)]
final readonly class Secured
{
    /** @var list<string> */
    public array $authorities;

    public function __construct(string ...$authorities)
    {
        $this->authorities = array_values($authorities);
    }
}
```

```

/** Requires ANY of the listed roles (JSR-250 @RolesAllowed). Compiled to hasAnyRole(...).
*/
#[Attribute(Attribute::TARGET_METHOD | Attribute::TARGET_CLASS)]
final readonly class RolesAllowed
{
    /** @var list<string> */
    public array $roles;

    public function __construct(string ...$roles)
    {
        $this->roles = array_values($roles);
    }
}

```

`#[PreAuthorize]` 's `code:` and `message:` are what the **client** reads when the expression refuses. Without them the guard answers `403 ACCESS_DENIED` with the framework's own sentence and the authorities the expression named; with them it answers the application's product code and wording. `#[Secured]` and `#[RolesAllowed]` have no slot for words and always use the framework's.

`#[Secured]` / `#[RolesAllowed]` are convenience sugar `MethodSecurityScanner` compiles down to a `hasAnyAuthority(...)` / `hasAnyRole(...)` expression — everything ultimately reduces to the one grammar `#[PreAuthorize]` writes directly. And that grammar is deliberately, provably closed: `SecurityExpressionEvaluator` is a hand-written tokenizer plus a recursive-descent parser, and it **never** calls `eval()`, `create_function()`, or `call_user_func()` on anything derived from the expression string. Function dispatch is a hard-coded `match`:

```

final class SecurityExpressionEvaluator
{
    // ...
    private function dispatch(string $name, array $args): bool
    {

```

```

    // parse()-only mode has no root: validate the whitelist but return a placeholder
    bool.
    $root = $this->root;
    if ($root === null) {
        return match ($name) {
            'hasRole', 'hasAnyRole', 'hasAuthority', 'hasAnyAuthority', 'hasScope',
            'hasAnyScope', 'hasPermission',
            'isAuthenticated', 'permitAll', 'denyAll' => true,
            default => throw new ExpressionParseException("Unknown function
            '{$name}'."),
        };
    }

    return match ($name) {
        'hasRole' => $root->hasRole($this->str($args, 0, $name)),
        'hasAnyRole' => $root->hasAnyRole(...$this->strings($args, $name)),
        'hasAuthority' => $root->hasAuthority($this->str($args, 0, $name)),
        'hasAnyAuthority' => $root->hasAnyAuthority(...$this->strings($args, $name)),
        'hasScope' => $root->hasScope($this->str($args, 0, $name)),
        'hasAnyScope' => $root->hasAnyScope(...$this->strings($args, $name)),
        'hasPermission' => $root->hasPermission($args[0] ?? null, $this->str($args, 1,
        $name)),
        'isAuthenticated' => $root->isAuthenticated(),
        'permitAll' => $root->permitAll(),
        'denyAll' => $root->denyAll(),
        default => throw new ExpressionParseException("Unknown function '{$name}'."),
    };
}
// ...
}

```

Ten function names, total — `hasRole`, `hasAnyRole`, `hasAuthority`, `hasAnyAuthority`, `hasScope`, `hasAnyScope`, `hasPermission`, `isAuthenticated`, `permitAll`, `denyAll`, plus `#param` references — and nothing else is reachable. The same ten are spelled twice on purpose: the first `match` is the whitelist tokenizer's parse-only mode, which `HttpSecurity` and `MethodSecurityScanner` use to reject a

bad expression at BOOT rather than on the request that happens to hit it, and the second is the evaluation proper. The grammar the tokenizer accepts is boolean `and / or / not` (spelled `&& / || / !` at the character level), parentheses, string literals in single quotes, and `#identifier` parameter references — and nothing else:

```
final class SecurityExpressionEvaluator
{
    // ...
    public function evaluate(string $expression, SecurityExpressionRoot $root): bool
    {
        // ...
        try {
            $this->tokens = $this->tokenize($expression);
            $this->pos = 0;
            $this->root = $root;
            $result = $this->parseExpression();
            $this->expect('eof');
        }
        return $result;
    } catch (\Throwable) {
        // ...
        return false;
    }
    // ...
}
// ...
}
```

⚠ WHAT IS NOT IN THE GRAMMAR

There is no `==`, no `!=`, no comparison operator of any kind, and no `.property` navigation. You cannot write `#command.ownerId == authentication.name` — the tokenizer has no notion of a dot-access or an equality operator at all, so that text either fails to tokenize or fails to parse, and either way `evaluate()` catches the `ExpressionParseException` and returns `false`. This is not an oversight; it is what makes the grammar provably closed. If a comparison like that seems like exactly what you need, the intended escape hatch is `hasPermission(#target, 'action')` backed by a custom `PermissionEvaluator` bean that does the actual field comparison in real PHP code — the grammar routes to your logic, it never becomes a general-purpose expression language itself.

Every failure mode — an unknown function name, a stray character, a malformed string literal, an exception thrown by a custom `PermissionEvaluator` mid-evaluation — is caught by the same `catch (\Throwable)` and turned into `false`. A hostile or broken expression can only ever deny; it can never accidentally grant.

`SecurityExpressionRoot` is the **only** object an expression's function calls can ever reach — there is no way to call anything else, because there is no property-access syntax and no way to obtain a reference to any other object:

```
final class SecurityExpressionRoot
{
    // ...
    public function hasRole(string $role): bool
    {
        return $this->hasAuthority(str_starts_with($role, 'ROLE_') ? $role : 'ROLE_'. $role);
    }
    // ...
    public function hasAuthority(string $authority): bool
    {
        return in_array($authority, $this->reachable, true);
    }
}
```



```
// ...
public function arg(string $name): mixed
{
    return $this->args[$name] ?? null;
}
}
```

`hasRole('ADMIN')` and `hasRole('ROLE_ADMIN')` are equivalent — a bare role name is normalised by prepending `ROLE_` if it isn't already there, exactly matching Spring's own convention. `#param` — tokenized as a bare `#` followed by an identifier — resolves through `arg()` against the guarded method's own parameter names, bound positionally by `MethodSecurityScanner` at scan time.

Here is the whole point made concrete, in real, shipped code — `WithdrawHandler`, real, from `samples/lumen/src/Application/Command/WithdrawHandler.php`:

```
<?php

declare(strict_types=1);

namespace Lumen\Application\Command;

use Firefly\Cqrs\Attributes\CommandHandler;
use Firefly\Data\Transaction\Attributes\Transactional;
use Firefly\Kernel\Exception\Business\ResourceNotFoundException;
use Firefly\Security\Access\Attributes\PreAuthorize;
use Lumen\Domain\Money;
use Lumen\Infrastructure\WalletRepository;

/**
 * Handles Withdraw: loads the aggregate, debits the amount (the domain enforces no-overdraw
 * – an over-balance debit
 *
 * raises a ConflictException that rolls the transaction back and surfaces as a
 * CommandProcessingException), persists,
 *
 * and returns the new balance in minor units. #[Transactional] for the same reason as the
```

```

sibling handlers.
*
* Guarded by #[PreAuthorize]: only an ADMIN or the WALLET_OWNER may debit.
SecurityExpressionEvaluator is a CLOSED
* whitelist (booleans and/or/not, string literals, #param, and eight fixed functions – no
`==`, no `.property`
* navigation), so ownership is modelled as a granted WALLET_OWNER authority rather than an
`#command.ownerId ==
* authentication.name` comparison, which the parser cannot express.
SecurityCommandAuthorizer enforces this at the bus
// ...
* (The faithful owner-only alternative – hasPermission(#command, 'withdraw') backed by a
custom PermissionEvaluator
* bean that loads the wallet and compares its owner_id to the current principal – is left
to a later step.)
*
* Intentionally NOT final: the generated transactional proxy subclasses this handler.
*/
#[CommandHandler]
class WithdrawHandler
{
    public function __construct(private readonly WalletRepository $wallets) {}

    #[PreAuthorize("hasRole('ADMIN') or hasRole('WALLET_OWNER')")]
    #[Transactional]
    public function handle(Withdraw $command): int
    {
        $wallet = $this->wallets->findById($command->walletId)
        ?? throw new ResourceNotFoundException("Wallet [{ $command->walletId}] not
found");

        $wallet->withdraw(new Money($command->amountMinor, $wallet->currency()));
        $this->wallets->save($wallet);

        return $wallet->balanceMoney()->minorUnits;
    }
}

```

Read that docblock's second paragraph carefully — it is the whole lesson in miniature. The *naïve* way to express "only the wallet's own owner may withdraw" would be something like `#command.ownerId == authentication.name`, and the grammar simply cannot say that: there is no `.` operator and no `==`. The real, shipped alternative is coarser but genuinely expressible: grant a `WALLET_OWNER` authority to whoever should be allowed, and check `hasRole('WALLET_OWNER')`. A true per-resource ownership check, when you need one, routes through `hasPermission(#command, 'withdraw')` and a custom `PermissionEvaluator` bean that does the actual comparison in ordinary PHP — the grammar dispatches to your code, it does not try to become expressive enough to write the comparison itself.

Enforcement at the bus: closing Chapter 7's `AllowAllAuthorizer` loop

Chapter 7 showed you `CommandBus / QueryBus`'s pipeline running an `authorize()` stage that defaults to `AllowAllAuthorizer` — allow everything. `firefly/security` is what actually replaces it, and does so only when you opt in:

`SecurityAutoConfiguration` is `#[Order(500)]`, strictly below

`CqrsAutoConfiguration`'s `#[Order(1000)]`, so — the same win-the-race pattern

Chapter 8 showed you for the Postgres outbox at `#[Order(900)]` — its beans register first and satisfy `CqrsAutoConfiguration`'s own

`#[ConditionalOnMissingBean(CommandAuthorizer::class)]` before the allow-all default ever gets a chance to bind, but **only** when `firefly.security.enabled=true` (every security bean here is additionally gated on that master flag):

```
final class SecurityCommandAuthorizer implements CommandAuthorizer
{
    public function __construct(private readonly MethodSecurityMessageEnforcer $enforcer) {}
}
```

```

    public function authorize(object $command): void
    {
        $this->enforcer->enforce($command, HandlerKind::Command);
    }
}

```

`MethodSecurityMessageEnforcer` is the shared join both `SecurityCommandAuthorizer` and its query-side twin `SecurityQueryAuthorizer` delegate to. It resolves the message's handler class and method from the **same** `HandlerManifest` Chapter 7 introduced, looks up any compiled `#[PreAuthorize]` / `#[Secured]` / `#[RolesAllowed]` rule for that `Class::method`, and — only if one exists — evaluates it:

```

    public function enforce(object $message, HandlerKind $kind): void
    {
        if (! $this->config->bool('firefly.security.enabled', false)) {
            return;
        }

        $handler = $this->handlerByMessage[$kind->value.':'.$message::class] ?? null;
        if ($handler === null) {
            return;
        }

        $rule = $this->methods->ruleFor($handler['class'], $handler['method']);
        if ($rule === null) {
            return;
        }

        $this->evaluator->before($rule, isset($rule->params[0]) ? [$rule->params[0] =>
$message] : []);
    }

```

Read it as four early returns and one line of work. Security off, no registered handler for this message, or no compiled rule for that handler — each passes through unchecked, because method security here is **additive**, never a second deny-by-default gate (that job belongs entirely to `HttpSecurityFilter`). When a rule is found, the last line hands it to `MethodSecurityEvaluator::before()`, the single evaluator every enforcement seam shares, with the message itself bound to the rule's one `#param`.

So when `WithdrawHandler::handle()` carries a rule, this is what runs it, **before** `handle()` itself ever executes. A refusal becomes an `AuthorizationException`, which `DefaultCommandBus::send()` (Chapter 7) wraps in `CommandProcessingException`, exactly like any other handler-side fault.

Method security everywhere, not just at the bus

The bus was the *first* place these rules were enforced. It is no longer the only one, and the difference matters: a `#[PreAuthorize]` on a plain `#[Service]` used to be a comment.

There are now three enforcement seams, and all three share one `MethodSecurityEvaluator`, so they cannot disagree about what an expression means:

Seam	Guards	Runs
The CQRS bus	a command or query handler	before <code>handle()</code> , via <code>SecurityCommandAuthorizer</code> / <code>SecurityQueryAuthorizer</code>
The controller dispatcher	a controller action	<code>ControllerSecurityGuard::check()</code> before, <code>afterInvocation()</code> after
The proxy chain	any stereotyped bean — <code>#[Service]</code> , <code>#[Component]</code> , <code>#[Repository]</code>	as the outermost interceptor on the generated proxy

The third is the new one, and it is not a new mechanism: it is Chapter 9's proxy, with a second kind of advice on it. `MethodSecurityAdviceSource` contributes its rows to the same `ProxyPlan`, and declares where in the chain its interceptor runs:

```
public function advice(): Advice
{
    // inertWhenUnbound: the interceptor bean exists only under the master flag, and its
    // absence is the
    // documented "annotations are inert until security is enabled" state, not a
    // misconfiguration – so the
    // InterceptorRegistry may hand the proxy a PassThroughInterceptor instead of refusing
    // to boot.
    return new Advice(self::ID, MethodSecurityInterceptor::class,
        SecurityMethodDescriptor::class, 100, inertWhenUnbound: true);
}
```

Order **100**, against the transactional advice's **1000**. Lower runs out, so security refuses *before* a transaction is ever opened — which is the only sensible order, because opening and rolling back a transaction to discover the caller was never allowed is work done for nothing.

`inertWhenUnbound` is the other half of the same thought.

`MethodSecurityInterceptor` only exists as a bean under `firefly.security.enabled`, and its absence is the framework's documented "annotations are inert until security is on" state rather than a misconfiguration — so the registry hands the proxy a `PassThroughInterceptor` rather than failing the boot, which is what it does for any *other* advice whose interceptor is missing.

The rules that need the returned value

Once the seam can run *after* the call, two more attributes become possible — and this is exactly what the bus could never do:

```
#[Service]
class ReportService
{
    /** @return array<string, int> */
    #[PreAuthorize("hasRole('ADMIN')")]
    public function totals(): array
    {
        return ['total' => 42];
    }

    #[PostAuthorize("hasPermission(#returnObject, 'READ')", code: 'REPORT_NOT_YOURS',
message: 'That report belongs to someone else.')]
    public function find(int $id): Report
    {
        return new Report($id, $id % 2 === 0 ? 'ada' : 'bob');
    }

    /** @return list<Report> */
```

```

    #[PostFilter("hasPermission(#filterObject, 'READ')")]
    public function all(): array
    {
        return [new Report(1, 'bob'), new Report(2, 'ada'), new Report(3, 'bob'), new
Report(4, 'ada')];
    }

```

- `#[PostAuthorize]` runs after `proceed()` with `#returnObject` bound to what the method returned. "You may call this, but only if what came back is yours" is a rule you cannot express before the call, because the answer is what decides it.
- `#[PostFilter]` runs over an iterable result with `#filterObject` bound to each element, keeping the ones the expression accepts. Arrays keep their keys; an `Enumerable` is filtered in kind; a non-iterable return is refused rather than quietly passed through.
- `#[PreFilter]` is the mirror image on the way in, naming which parameter it filters with `filterTarget:`.

Notice `code:` and `message:` on the `#[PostAuthorize]`. A refusal's product code and the sentence a person reads live on the same line as the rule that produced them, beside the method they guard — rather than in some service the caller has to remember to invoke first. Without them the framework's own wording applies: a 403 reading "You do not have permission to do this.", with the authorities the rule asked for carried as an RFC 9457 extension member, and the guarded class and method sent to the **log** rather than to the wire, because a PHP class name is not something a stranger should read on a panel.

⚠ TWO BOOT REFUSALS YOU WANT, AND ONE YOU MUST OPT INTO

A `final` bean carrying rules only a proxy could enforce is refused at scan time — the proxy cannot extend it, so compiling the row would produce a rule nothing runs. And `SecurityWiringPass` refuses to boot over a `security-methods.php` written without a `proxy-plan.php` beside it: the rules would compile and the bean-level ones would silently not be enforced. The one you must ask for is `firefly.security.method.strict`, which turns "no compiled artifact found" from a fallback into a startup failure. Set it in any image that runs `firefly:cache`; it is the only defence against a build that ships without the manifest.

Method security remains **additive**. "No rule recorded for this method" means allow, at every one of the three seams — deny-by-default is `HttpSecurityFilter`'s job and nothing else's. That is the right semantics and it has a sharp edge, which is what `strict` exists for: an empty manifest is indistinguishable from an application that declares no rules.

The principal, injected

A controller that needs to know who is calling should say so in its signature. It should not reach into a holder, and it should certainly not receive a `principal` that a query string could set.

```
#[GetMapping('/profile')]
public function profile(Authentication $auth, ?UserDetails $user, #[AuthenticationPrincipal]
mixed $principal, #[CurrentSecurityContext] SecurityContext $context): array
{
    return [
        'name' => $auth->getName(),
        'authorities' => $auth->authorityStrings(),
        'user' => $user?->getUsername(),
    ];
}
```

```

    'principal' => is_string($principal) ? $principal : ($principal instanceof
UserDetails ? 'details:'.$principal->getUsername() : get_debug_type($principal)),
    'authenticated' => $context->isAuthenticated(),
];
}

```

Four parameters, four ways of asking, and `firefly/security` binds all of them before the container is ever consulted — because of a port `firefly/web` exposes for exactly this:

```

interface HandlerMethodArgumentResolver
{
    /**
     * @param array<string, mixed> $binding
     */
    public function supports(array $binding): bool;
}

/**
 * @param array<string, mixed> $binding
 */
public function resolve(array $binding, Request $request): mixed;
}

```

A resolver sees the compiled binding plan — the parameter's name, kind, type, attributes and nullability — and claims it with `supports()`. `ArgumentResolver` asks the registered resolvers **before** its own kinds, so a class-typed parameter a resolver understands never reaches `$container->make()`. `SecurityArgumentResolver` claims `Authentication`, `UserDetails`, `#[AuthenticationPrincipal]` and `#[CurrentSecurityContext]`.

The nullability rules are worth stating precisely, because they are what let one action serve two mechanisms:

Declaration	Signed in	Anonymous
<code>Authentication \$auth</code>	the authentication	401
<code>?Authentication \$auth</code>	the authentication	<code>null</code>
<code>?UserDetails \$user</code>	the principal when it is one	<code>null</code>
<code>#[AuthenticationPrincipal] mixed \$principal</code>	whatever the principal is	the anonymous principal
<code>#[CurrentSecurityContext] SecurityContext \$context</code>	the context	the anonymous context

An **attributed** principal is handed over only when it *is* what the parameter declares, and `null` otherwise. That is what stops a `TypeError` — a JWT's principal is its `sub` **string**, not a `UserDetails` — and it is what lets the same action serve a form login and a bearer token:

```
// A JWT's principal is its `sub` string, not a UserDetails:
`#[AuthenticationPrincipal] ?UserDetails` is the
// null it allowed for, not the string a `?UserDetails` parameter would refuse with a
TypeError (a 500).
$this->withHeader('Authorization', $this->bearerFor('svc-42'))->getJson('/open/
principal-user')->assertOk()->assertJson(['user' => null]);
```

Two more facts, both defensive. The resolver is registered **whether or not the master flag is on**, so with security off the annotations are inert — a `null`, an anonymous context, an honest 401 — rather than being misread as query parameters by the ordinary binding plan. And nothing here ever reads a principal from the request:

```
it('never reads a principal from the query string, whoever asks', function () {  
  // ...  
  $this->getJson('/open/whoami?principal=admin')->assertOk()->assertJson(['principal' =>  
    null]);  
});
```

The events

Every mechanism in this chapter reports what it did, through

`AuthenticationEventPublisher` over the context's ordinary

`ApplicationEventPublisher`. So an `#[AsEventListener]` method receives them

exactly like any other application event from Chapter 8, and `Event::fake()` sees them in a test.

The family is small and complete — this is every class in `packages/security/src/Event/`, and there are no others:

Event	Published when
<code>AuthenticationSuccessEvent</code>	any successful authentication, interactive or not
<code>InteractiveAuthenticationSuccessEvent</code>	a person signed in: <code>form</code> , <code>basic</code> or <code>remember-me</code>
<code>AuthenticationFailureBadCredentialsEvent</code>	a wrong password — and an unknown username
<code>AuthenticationFailureLockedEvent</code>	the credentials were right, the account is locked
<code>AuthenticationFailureDisabledEvent</code>	the credentials were right, the account is disabled
<code>LogoutSuccessEvent</code>	a session ended through <code>LogoutHandler</code>
<code>AuthorizationDeniedEvent</code>	a URL rule or a method rule refused an authenticated principal

Three of those rows carry a decision rather than a fact.

An unknown user is reported as bad credentials, on purpose. A distinct "no such user" event would be a username oracle, and the whole point of

`DaoAuthenticationProvider` equalising the two paths — the same timing, the same answer — would be undone by the very listener written to monitor it.

A locked or disabled account is only reported after the right password. Otherwise the events would tell an attacker which accounts exist, which is the same oracle by a slower route.

AuthorizationDeniedEvent fires only for someone who is signed in. An anonymous denial is a visitor who has not logged in yet, published nowhere; an authenticated denial is a person who tried something they are not allowed to do, and it arrives with the principal, the subject (`GET /admin/users` , or `App\Reports::totals`) and the expression that refused them.

What none of them carry is a credential. A failure event names the username and the source IP; the password is not in it, and the abstract base class the three failures share has no slot for one.

💡 THE THREE LISTENERS WORTH WRITING ON DAY ONE

Count `AuthenticationFailureBadCredentialsEvent` per source IP and you have brute-force detection. Log `InteractiveAuthenticationSuccessEvent` with its `mechanism` and you have a sign-in audit that distinguishes a password from a cookie. Alert on `AuthorizationDeniedEvent` and you are watching the only denial that means somebody who *is* known tried something they should not have.

Proof end-to-end

Two real, shipped test files exercise the entire chain — HTTP request in, `SecurityCommandAuthorizer` enforcing `WithdrawHandler`'s `#[PreAuthorize]` at the bus, out to an RFC-9457 response. And the first thing they prove is that **a denial is not one answer but two**.

`samples/lumen/tests/Web/WalletRestTest.php`, with no principal set at all:

```
$this->postJson("/api/v1/wallets/{id}/withdraw", ['amount_minor' => 1000])
    ->assertStatus(401)
    ->assertHeader('Content-Type', 'application/problem+json')
```

```

->assertJsonPath('status', 401)
->assertJsonPath('code', 'AUTHENTICATION_FAILED')
->assertJsonPath('category', 'security');

// Proof the refusal happened BEFORE the handler touched the balance.
$this->getJson("/api/v1/wallets/{id}/balance")->assertJson(['balance_minor' => 5000]);

```

401, not 403 — and the distinction is the whole point. `MethodSecurityEvaluator` answers an **anonymous** caller with an `AuthenticationException`: *authenticate first*. It reserves 403 for a caller who *is* signed in and still lacks the authority, which is the next test in the same file:

```

$this->postJson("/api/v1/wallets/{id}/withdraw", ['amount_minor' => 1000])
->assertStatus(403)
->assertHeader('Content-Type', 'application/problem+json')
->assertJsonPath('status', 403)
->assertJsonPath('code', 'ACCESS_DENIED')
->assertJsonPath('category', 'security');

```

The three enforcement seams give the same pair of answers, because they share the same evaluator: 401 means *signing in would help*, 403 means *it would not*. A client can tell which, and act accordingly.

The bus wraps either one in `CommandProcessingException`, which copies the cause's error code, HTTP status and category — so the wire carries `AUTHENTICATION_FAILED` or `ACCESS_DENIED` rather than the bus's own generic code. And in both tests the balance assertion afterwards is doing real work: it proves the debit genuinely never happened, not merely that the HTTP response looked like a rejection.

Grant the right authority and the same command succeeds:

```

it('allows an authorized withdraw and renders an overdraw as 409 problem+json', function ()
{
    // ...
    SecurityContextHolder::setContext(new SecurityContext(
        Authentication::authenticated('owner-4', 'owner-4', [new
SimpleGrantedAuthority('ROLE_WALLET_OWNER')])
    ));
    // ...
    $id = $this->postJson('/api/v1/wallets', ['owner_id' => 'owner-4', 'currency' =>
'EUR'])->json('wallet_id');
    $this->postJson("/api/v1/wallets/{$id}/deposit", ['amount_minor' => 3000]);

    // Authorized + within balance: succeeds and reflects the new balance.
    $this->postJson("/api/v1/wallets/{$id}/withdraw", ['amount_minor' => 1000])
        ->assertStatus(200)
        ->assertJson(['wallet_id' => $id, 'balance_minor' => 2000]);
    // ...
    $this->postJson("/api/v1/wallets/{$id}/withdraw", ['amount_minor' => 999999])
        ->assertStatus(409)
    // ...
    ->assertJsonPath('category', 'business');
}

```

The granted authority is `'ROLE_WALLET_OWNER'` — already carrying the `ROLE_` prefix — and `WithdrawHandler`'s expression calls `hasRole('WALLET_OWNER')`, which `SecurityExpressionRoot::hasRole()` normalises to the identical `'ROLE_WALLET_OWNER'` string before checking membership; the two spellings meet in the middle. The `409` in the second call is the important contrast: authorization and domain validation are two genuinely independent layers — being *allowed* to withdraw says nothing about whether the withdrawal itself is *valid*, and `Wallet::withdraw()`'s own no-overdraw invariant (Chapter 6) still applies exactly as before.

`samples/lumen/tests/Application/TransferSecurityTest.php` proves the same guard directly against the `CommandBus`, with the `getPrevious()` unwrap pattern Chapter 7 introduced:


```

it('enforces #[PreAuthorize] on withdraw: denies without the owner role, allows with it',
function () {
    // ...
    $commands = $this->fireflyContext()->get(CommandBus::class);
    // ...
    $queries = $this->fireflyContext()->get(QueryBus::class);
    // ...
    $walletId = $commands->send(new OpenWallet('owner-C', Currency::EUR));
    $commands->send(new Deposit($walletId, 5000));
    // ...
    SecurityContextHolder::setContext(new SecurityContext(
        Authentication::authenticated('mallory', 'mallory', [new
SimpleGrantedAuthority('ROLE_USER')])
    ));

    $denied = null;
    try {
        $commands->send(new Withdraw($walletId, 1000));
    } catch (CommandProcessingException $e) {
        $denied = $e;
    }
    // ...
    expect($denied)->toBeInstanceOf(CommandProcessingException::class);
    expect($denied?->getPrevious())->toBeInstanceOf(AuthorizationException::class);
    expect($queries->ask(new GetBalance($walletId)))->toBe(5000);

    // Same withdraw, now with ROLE_WALLET_OWNER granted: the guard passes and the debit
    goes through.
    SecurityContextHolder::setContext(new SecurityContext(
        Authentication::authenticated('owner-C', 'owner-C', [new
SimpleGrantedAuthority('ROLE_WALLET_OWNER')])
    ));

    $commands->send(new Withdraw($walletId, 1000));

    expect($queries->ask(new GetBalance($walletId)))->toBe(4000);

```

`$denied?->getPrevious()` reaches straight past `CommandProcessingException` to the real `AuthorizationException` `MethodSecurityMessageEnforcer` threw — the exact unwrap pattern Chapter 7 showed you, now with its concrete security use.

📌 LARAVEL PARITY

Plain Laravel authorizes with Gates/Policies and a `$this->authorize()` call inside a controller — imperative, and only ever reachable from code that remembers to call it. `#[PreAuthorize]` is attribute-discovered and enforced automatically at the bus for every dispatch of a guarded command or query, with no call site able to forget it; `AuthorizationChecker::check()` (backed by the identical evaluator) is the closest LaraFly equivalent to a Gate check for the rarer case where you need to test an expression imperatively at an arbitrary call site.

✓ What you learned

Concept	What it does
<code>Authentication</code>	Immutable; <code>authenticated()</code> / <code>unauthenticated()</code> factories make the two states impossible to confuse
<code>SecurityContextHolder</code>	Request-scoped static holder; every auth filter clears it in a <code>finally</code>
<code>DaoAuthenticationProvider</code>	Constant-time-equivalent, enumeration-mitigated username/password check
<code>JwtService</code>	Refuses to boot with a weak/placeholder secret; refuses a token with no <code>exp</code> claim
<code>HttpSecurity</code>	Deny-by-default URL rules, first-match-wins, compiling to the same expression grammar as method security
<code>SecurityExpressionEvaluator</code>	A hand-rolled, closed-whitelist tokenizer/parser — no <code>eval</code> , no <code>==</code> , no <code>.property</code> navigation; ten functions total (<code>hasRole</code> , <code>hasAnyRole</code> , <code>hasAuthority</code> , <code>hasAnyAuthority</code> , <code>hasScope</code> , <code>hasAnyScope</code> , <code>hasPermission</code> , <code>isAuthenticated</code> , <code>permitAll</code> , <code>denyAll</code>) plus <code>#param</code> references
<code>hasRole('X')</code>	Normalises to <code>hasAuthority('ROLE_X')</code> — bare and <code>ROLE_</code> -prefixed spellings are equivalent
<code>SecurityCommandAuthorizer</code> / <code>SecurityQueryAuthorizer</code>	The real <code>#[Order(500)]</code> implementations that replace Chapter 7's <code>AllowAllAuthorizer</code> , only when <code>firefly.security.enabled=true</code>

Concept	What it does
<code>MethodSecurityMessageEnforcer</code>	Joins the CQRS <code>HandlerManifest</code> to the compiled <code>SecurityMethodManifest</code> ; enforces before <code>handle()</code> ever runs
<code>SecurityContextPersistenceFilter</code>	Loads the context from the session at <code>-94</code> , saves what changed, and clears the holder in a <code>finally</code>
<code>FormLoginFilter</code> / <code>LoginRouteRegistrar</code>	<code>POST /login</code> handled before routing; the framework's page mounted only when nothing else claims that address
<code>EloquentUserDetailsService</code>	Your own users table as the user store — and a configured column the row lacks is refused, never read as <code>null</code>
<code>LogoutFilter</code> / <code>TokenBasedRememberMeServices</code>	POST-only, CSRF-checked logout; a cookie signed over the password hash , so a password change revokes every one
<code>AuthenticationEntryPoint</code>	The four <code>entry_point</code> modes; <code>ErrorPageRenderer::prefersHtml()</code> — the request names <code>text/html</code> or <code>application/xhtml+xml</code> , is no XHR and is not under <code>json-paths</code> — decides login page versus 401
<code>MethodSecurityAdviceSource</code>	Method security on any stereotyped bean, at advice order 100 — outside the transaction at 1000
<code>#[PostAuthorize]</code> / <code>#[PostFilter]</code>	Rules that need the returned value: <code>#returnObject</code> after the call, <code>#filterObject</code> per element

Concept	What it does
<code>SecurityArgumentResolver</code>	<code>Authentication</code> , <code>UserDetails</code> , <code>#[AuthenticationPrincipal]</code> , <code>#[CurrentSecurityContext]</code> bound before the container is asked
The event family	Seven events; an unknown user reports as bad credentials, and only an authenticated denial is published

Try it yourself

- Write a rejected expression.** Try compiling `#[PreAuthorize("#command.ownerId == authentication.name")]` on a scratch handler and run `php artisan firefly:cache`. Confirm `MethodSecurityScanner` fails loud with a `ConfigurationException` naming the exact class and method, rather than silently accepting an expression that would always deny (or fail to parse) at runtime.
- Add a real ownership check.** In a scratch copy of the project, implement a `PermissionEvaluator` bean that loads a wallet by id and compares its `owner_id` to `Authentication::getName()`, bind it to replace `DenyAllPermissionEvaluator`, and change `WithdrawHandler`'s expression to `hasPermission(#command, 'withdraw')`. Confirm a `WALLET_OWNER`-less principal who genuinely owns the wallet can now withdraw, while a different owner cannot.
- Own the login page.** Add a `#[GetMapping('/login')]` action of your own to a scratch project with `form_login.enabled` on, boot it, and confirm the framework's page does *not* replace yours — then confirm your form still signs people in by posting the session token to the processing URL, because `FormLoginFilter` answers before routing either way.

4. **Watch the entry point switch.** Against a protected path, send `Accept: text/html` and then `Accept: application/json`, and confirm you get a redirect to `/login` and a `401 application/problem+json` respectively. Then set `firefly.security.http.entry_point` to `problem` and confirm the browser request gets the 401 too.
5. **Put a rule on a plain service.** Add `#[PreAuthorize("hasRole('ADMIN')")]` to a method on a non-final `#[Service]` that is neither a controller nor a handler, run `php artisan firefly:cache`, and confirm the call is refused. Then mark the class `final`, re-run the cache, and confirm the scanner refuses at compile time rather than shipping a rule nothing could enforce.
6. **Reproduce the enumeration-mitigation timing.** Time `DaoAuthenticationProvider::authenticate()` for a known username with a wrong password against an unknown username, over enough iterations to smooth out noise, and confirm the two are statistically indistinguishable — proving the dummy-hash verify this chapter described is genuinely doing its job, not just documented as if it were.

OAuth2 and OpenID Connect: Signing In, and Being the Provider

By the end of this chapter you will know which of the two OAuth2 packages you are holding and why they are two — `firefly/security-oauth2-client` makes your application a **relying party** that signs people in through Google, Keycloak, Okta, Entra or any conformant provider, and `firefly/security-oauth2-server` makes it an **authorization server** that issues the tokens somebody else consumes. You will know the two filters the client half adds at `-89` and `-88`, the two base URIs they answer on, and what the framework puts in an authorization request that Chapter 10's form login never had to think about: a single-use `state`, a `nonce`, and an S256 PKCE challenge. You will know what an `OidcUser` is once it reaches your controller, how a `groups` claim becomes a `ROLE`, how logging out reaches the provider's session as well as yours, and how a queued job calls an API as the application with no person involved at all. Then the picture turns around: the same round trip, from the side that mints the code — the consent page, the token endpoint, refresh-token rotation with reuse detection, introspection, revocation, userinfo, the JWKS, and the one command that generates the key it all hangs from.

NEW TERM: RELYING PARTY

In OpenID Connect the application that sends a person to a provider to be identified is the **relying party** (the RP) — it *relies* on somebody else's assertion about who the person is, instead of holding the password itself. The provider is the **OpenID Provider** or, in plain OAuth2 terms, the **authorization server**. The two packages in this chapter are those two roles, and the words matter because a single deployment is regularly both: an internal portal that signs its staff in through corporate SSO *and* issues tokens to the mobile app it owns.

Two packages, two directions

The root library includes both OAuth2 components. Each builds on `firefly/security`; their internal dependency descriptors do not depend on one another. Configuration enables each role independently. These optional component requirements are satisfied by the already-installed library through `replace`:

```
composer require firefly/security-oauth2-client # be a relying party: sign people in
through a provider
composer require firefly/security-oauth2-server # be the provider: issue the tokens
yourself
```

They are not the same weight, though, and the two `require` blocks say so plainly. `firefly/security-oauth2-client` names seven framework packages — `autoconfigure`, `config`, `container`, `context`, `kernel`, `security` and `web` — and `firefly/security-oauth2-server` names eleven framework packages: those same seven plus `actuator`, `data`, `resilience` and `scheduling`, one for each thing the provider role owns that a relying party never does. `firefly/data`, because the registered clients, the authorizations and the consents are Eloquent stores built on `EloquentRepository`; `firefly/actuator`, because the server contributes an endpoint that lists the clients it has registered; `firefly/resilience`, because the token endpoint is rate limited; and `firefly/scheduling`, because expired authorizations are purged on a cron (`*/15 * * * *` by default) the deployment can change. What is true in both blocks is the part that matters here: `firefly/security` is the only `security` package either one names — the third OAuth2 role, the resource server, lives inside `firefly/security` itself — and neither block names the other package.

`firefly/security-oauth2-client` contributes `OAuth2AuthorizationRequestRedirectFilter` (-89) and `OAuth2LoginAuthenticationFilter` (-88), which sit between the JWT filter and the resource-server filter in the chain you already know.

`firefly/security-oauth2-server` contributes exactly one, `OAuth2AuthorizationServerFilter` (-82), which answers **every** endpoint the server owns. Both halves default to off, key by key, and both are configured under `firefly.security.oauth2.*` — the same block Chapter 10 used for `resource_server`, which is the third OAuth2 role and the one that only *verifies* a token somebody else minted.

One pairing is refused outright at boot, and it is worth reading the refusal rather than the summary of it, because the reason is a filter order and you now know how to read one:

```
/**
 * Refusals (1)–(4), as one static so the rules are testable against a bare Config without a
 * boot; a no-op
 * while the server is off.
 */
public static function assertRunnable(Config $config): void
{
    if (! $config->bool('firefly.security.oauth2.server.enabled', false)) {
        return;
    }

    // ...
    if ($config->bool('firefly.security.http_basic.enabled', false)) {
        throw new ConfigurationException(
            'firefly.security.oauth2.server.enabled and firefly.security.http_basic.enabled
            are both on: HttpBasicFilter (-91) '
            . 'answers every Authorization: Basic header as a user login – a 401 and a
            failure event – before the server\'s '
            . 'filter (-82) could read a client_secret_basic credential at the token,
            introspection or revocation endpoint. '
        );
    }
}
```

```

        .Turn http_basic.enabled off; the server authenticates its clients itself.'
    );
}
}

```

`HttpBasicFilter` (-91) runs before `OAuth2AuthorizationServerFilter` (-82), and an OAuth2 client authenticating with `client_secret_basic` sends exactly the header the Basic filter was built to consume. With both switches on, every confidential client in your deployment would be answered 401 as a failed *user* login — with a failure event published for it — before the authorization server's filter ever saw the credential. Nothing about that failure would mention OAuth2. So the boot refuses the pair and names both keys, which is the framework's standing rule for a combination that could only ever produce a confusing runtime error. The same static refuses three more: the server without `firefly.security.enabled`, the server with nothing to carry a principal between requests, and the server beside `firefly.security.jwt.enabled` (whose HMAC filter at -90 would reject every RS256 token this server issues).

Signing in with a provider

Everything in this section is the relying-party half. The shape of it is Spring Boot's, key for key, and the shipped reference is where the defaults live:

```

'client' => [
  'enabled' => env('FIREFLY_OAUTH2_CLIENT_ENABLED', false),
  // 'authorization_endpoint_base_uri' => '/oauth2/authorization',
  // 'redirection_endpoint_base_uri' => '/login/oauth2/code',
  // 'default_success_url' => '/',
  'login' => [
    'enabled' => env('FIREFLY_OAUTH2_LOGIN_ENABLED', false),
    // 'authorization_endpoint_base_uri' => '/oauth2/authorization',
    // 'redirection_endpoint_base_uri' => '/login/oauth2/code',
    // 'default_success_url' => '/',
  ]
]

```

```

    // 'always_use_default_success_url' => false,
    // 'failure_url' => '/login?error',
  ],

  'logout' => [
    'oidc_initiated' => env('FIREFLY_OAUTH2_OIDC_LOGOUT', false),
    // 'post_logout_redirect_uri' => '{baseUrl}/login?logout',
  ],

  // Seconds of leeway on exp/iat of an id token, and the margin before an access token's
  expiry
  // at which the manager refreshes it. Default: 60.
  'clock_skew' => 60,

  // Outbound calls (discovery, token endpoint, JWKS, userinfo): the same bounded timeouts
  the
  // resource server's JWKS fetch uses, and whether Http::oauth2Client('{id}') is
  registered.
  'http' => [
    'connect_timeout' => 5,
    'timeout' => 5,
    'macro' => true,
  ],

  'discovery' => [
    'cache_ttl' => 3600,
    'eager' => env('FIREFLY_OAUTH2_DISCOVERY_EAGER', false),
  ],

  'jwk_set' => [
    'cache_ttl' => 3600,
  ],

  // How long a user-bound client that holds a refresh token stays in the cache service.
  'authorized_client' => [
    'cache_ttl' => 86400,
  ],

  'registration' => [

```

```

    // 'google' => [
    //   'client_id' => env('GOOGLE_CLIENT_ID'),
    //   'client_secret' => env('GOOGLE_CLIENT_SECRET'),
    // ],
    // 'corp' => [
    //   'provider' => 'keycloak',
    //   'client_id' => 'portal',
    //   'client_secret' => env('KEYCLOAK_CLIENT_SECRET'),
    //   'client_authentication_method' => 'client_secret_basic', //
client_secret_basic | client_secret_post | none
    //   'authorization_grant_type' => 'authorization_code', //
authorization_code | client_credentials
    //   'redirect_uri' => '{baseUrl}/login/oauth2/code/{registrationId}',
    //   'scope' => ['openid', 'profile', 'email'],
    //   'client_name' => 'Corporate SSO',
    //   'pkce' => true,
    // ],
  ],
  'provider' => [
    // 'keycloak' => [
    //   'issuer_uri' => 'https://sso.example.com/realms/corp',
    //   'authorization_uri' => null,
    //   'token_uri' => null,
    //   'jwk_set_uri' => null,
    //   'user_info_uri' => null,
    //   'user_name_attribute' => 'sub',
    //   'end_session_uri' => null,
    // ],
  ],
],
],

```

Two commented-out blocks carry the whole model. A **registration** is one relationship with one provider: a client id, a secret, the scopes to ask for, the grant to use. A **provider** is one issuer's endpoints. A registration names its provider (or, when it does not, its own id is taken as the provider's name), and five names are **presets** —

`google`, `github`, `okta`, `keycloak` and `microsoft` (alias `entra`) — whose endpoints, default scopes and `client_name` are built in, and — for `google` and `github` alone — the client-authentication method, because neither provider issues a web client without a secret. On the three per-tenant presets the method is deduced instead: `client_authentication_method` when the registration sets one, `client_secret_basic` when it carries a secret, and `none` — a public client, PKCE mandatory — when it does not. Google and GitHub are complete as they stand; the other three are per-tenant and need `provider.{id}.issuer_uri`, from which **OIDC discovery** reads the rest out of `{issuer}/.well-known/openid-configuration`. Discovery is lazy — a registration resolves on first use and is memoised for the process, so `firefly:cache` and console boots never need the provider to be reachable — unless `discovery.eager` is on, which resolves every registration at boot and fails the boot on a dead issuer, exactly as Spring Boot does.

Everything that can be checked without a request **is** checked without one: a missing `client_id`, a secret-less registration on a method that needs a secret, an unknown grant type, a provider that is neither a preset nor configured, a per-tenant preset with no issuer, a provider with no `issuer_uri` that does not spell out its endpoints, a misspelt key (`client-id` is refused, not ignored) and `login.enabled` with no registration at all are each a `ConfigurationException` naming the key, at boot.

With `login.enabled` on, two URLs exist. `GET /oauth2/authorization/{id}` starts a sign-in and `GET /login/oauth2/code/{id}` finishes it — the defaults of `login.authorization_endpoint_base_uri` and `login.redirection_endpoint_base_uri`, and the second is also the path the default `redirect_uri` template points at. Both are answered by their own filter, *before* `HttpSecurityFilter` (-70), so neither needs a URL rule of its own. The framework's

login page from Chapter 10 grows one "Sign in with {client_name}" button per `authorization_code` registration — after the password form when form login is also on, and alone when it is not.

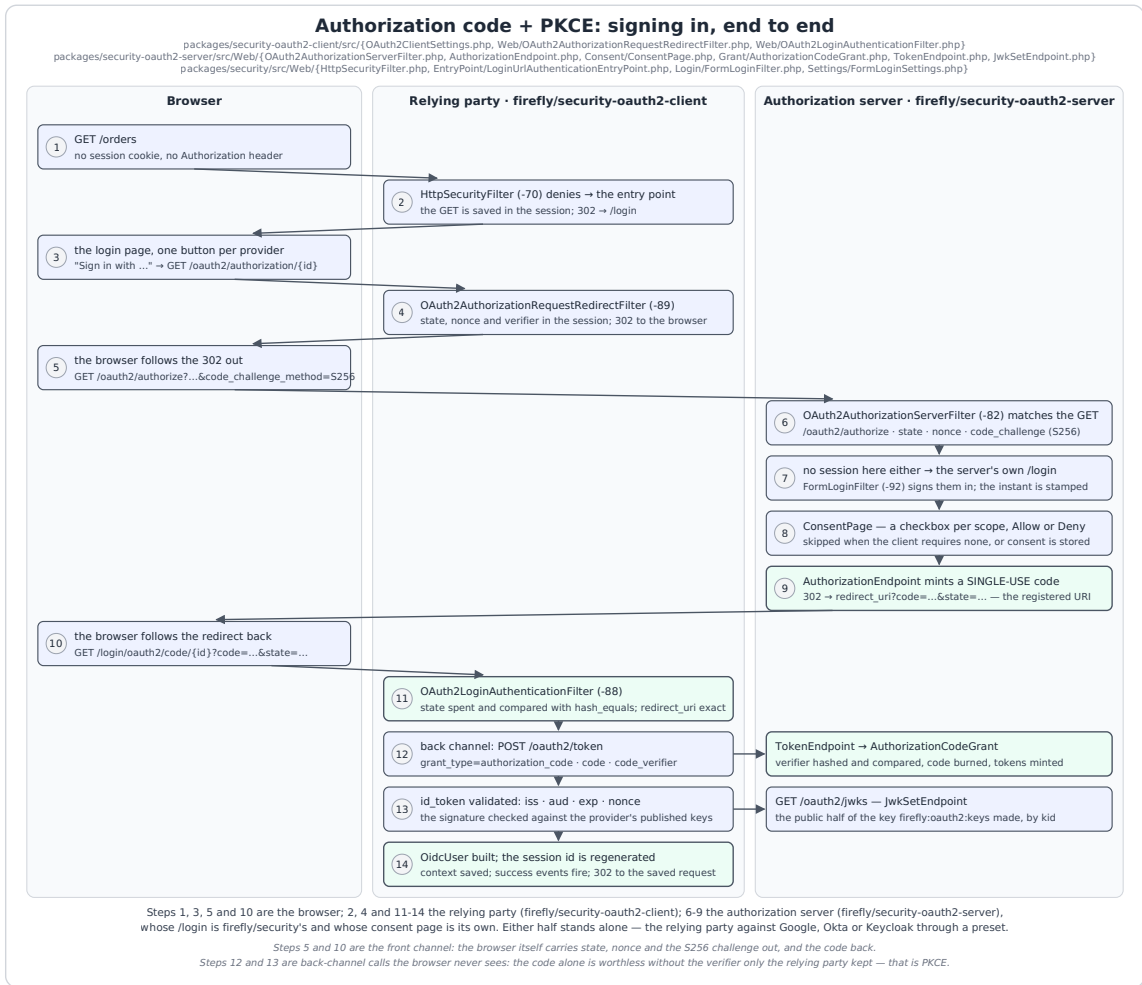


Figure 10A.1 — The authorization-code round trip with PKCE: the browser, the relying party and the authorization server, with every real endpoint path.

Read the figure left to right and notice what the browser never carries. The start URL builds an authorization request and keeps it in the session; only its public halves — the `state`, the `nonce`, the `code_challenge` — travel through the browser. The verifier that the challenge was derived from stays behind, and is spent in a back-channel `POST` the browser never makes. Three random values, one generator:

```
/**
 * Builds the authorization request for a registration (Spring's
 * DefaultOAuth2AuthorizationRequestResolver):
 * a fresh `state` every time (32 random bytes, base64url – 43 characters), a `nonce`
 * whenever `openid` is
 * requested (OIDC Core §3.1.2.1 makes it optional; this package makes it mandatory, because
 * the nonce is
 * what ties the id token to THIS browser's request and there is no reason to ever leave it
 * out), a PKCE
 * verifier whenever the registration uses PKCE (by default; always for a public client),
 * and the redirect
 * URI expanded from the application's base URL. The three random values share one
 * generator: 256 bits from
 * random_bytes(), and the base64url alphabet is a subset of the RFC 7636 verifier alphabet.
 */
final class OAuth2AuthorizationRequestResolver
{
    public function resolve(ClientRegistration $registration, string $baseUrl):
    OAuth2AuthorizationRequest
    {
        return new OAuth2AuthorizationRequest(
            registrationId: $registration->registrationId,
            authorizationUri: $registration->providerDetails->authorizationUri,
            clientId: $registration->clientId,
            redirectUri: RedirectUriTemplate::expand($registration->redirectUri, $baseUrl,
$registration->registrationId),
            scopes: $registration->scopes,
            state: self::random(),
            nonce: $registration->usesOpenId() ? self::random() : null,
            codeVerifier: $registration->usesPkce() ? self::random() : null,
```



```

    );
}

/** 32 random bytes as base64url: 43 characters of `[A-Za-z0-9_-]`. */
public static function random(): string
{
    return rtrim(strtr(base64_encode(random_bytes(32)), '+/', '-_'), '=');
}
}

```

The return leg is where the refusals live, and the order they run in is itself a decision. The saved authorization request is **pulled** from the session first — single use, whatever happens next — so a replayed callback finds nothing to compare against. Then, in order: a provider that answered `?error=` hands back its own RFC 6749 code; no saved request (or one belonging to another registration) is `authorization_request_not_found`; a `state` that does not match, compared with `hash_equals`, is `invalid_state_parameter`; a callback URL that is not exactly the request's `redirect_uri` is `invalid_redirect_uri`; a missing code is `invalid_request`. Only then is the code exchanged.

What comes back is examined before anybody is signed in. For an `openid` registration the id token's signature is verified against the provider's JWKS (cached for `jwk_set.cache_ttl`), and then `iss`, `aud`, `azp`, `iat`, `sub` and the `nonce` are checked per OIDC Core §3.1.3.7 with `clock_skew` seconds of leeway; userinfo is fetched when the provider has that endpoint and a `profile`, `email`, `address` or `phone` scope was granted, and its `sub` must match the id token's. A plain OAuth2 registration — no `openid` — has no id token to check and is identified from `user_info_uri` alone. Then the session id is regenerated, the `SecurityContext` is saved the way Chapter 10's form login saves it, and the browser is sent to the request that was saved when it was first refused, or to `login.default_success_url`.

⚠ A 503 FROM THE PROVIDER IS NOT A REFUSED LOGIN

Discovery, the JWKS fetch and userinfo all go through Laravel's Http client with both timeouts bounded at five seconds — Laravel's own default is thirty, which is PHP's execution limit, so a slow issuer would be a fatal error rather than an error page. When one of them is genuinely unreachable the result is a **503**, not a 401 and not a redirect to `/login?error`: nothing about the person's credentials was examined, and telling them "signing in did not work" would send them to try a password they do not have. The login page omits a provider whose discovery is down at that moment, with a warning logged, rather than taking every other way in down with it.

The person in your controller

After a successful login the principal in the session is an `OidcUser` (or an `OAuth2User` for a registration without `openid`), and Chapter 10's `SecurityArgumentResolver` injects it with no extra wiring. The package's own fixture controller is every way an application reads one:

```
/**
 * Every way an application reads an OAuth2 principal: `[AuthenticationPrincipal] OidcUser`
 (a 401 when the
 * principal is not one), the nullable `?OAuth2User` that serves both kinds, `hasScope` as a
 URL rule (the
 * capstone's `api/scoped`) and as a method rule, and a role a GrantedAuthoritiesMapper must
 have added.
 */
#[RestController]
final class AccountController
{
    /** @return array{name: string, subject: string, email: ?string, fullName: ?string,
    groups: mixed, idTokenValue: string, hasUserInfo: bool, issuer: mixed} */
    #[GetMapping('/account')]
}
```

```

    public function account([AuthenticationPrincipal] $oidcUser $user): array
    {
        return [
            'name' => $user->getName(),
            'subject' => $user->getSubject(),
            'email' => $user->getEmail(),
            'fullName' => $user->getFullName(),
            'groups' => $user->getClaim('groups'),
            'idTokenValue' => $user->getIdToken()->getTokenValue(),
            'hasUserInfo' => $user->getUserInfo() != null,
            'issuer' => $user->getAttribute('iss'),
        ];
    }
    // ...
    /** @return array{email: bool} */
    #[PreAuthorize("hasScope('email')")]
    #[GetMapping('/api/email')]
    public function email(): array
    {
        return ['email' => true];
    }

    /** @return array{engineers: bool} */
    #[PreAuthorize("hasRole('ENGINEERING')")]
    #[GetMapping('/api/engineers')]
    public function engineers(): array
    {
        return ['engineers' => true];
    }
}

```

`OIDCUser` extends `OAuth2User` and `ClaimAccessor`: on top of `getName()`, `getAttributes()` and `getAuthorities()` it has `getIdToken()`, `getUserInfo()`, `getSubject()`, `getEmail()`, `getFullName()`, `getPreferredUsername()` and the

`getClaim()` / `hasClaim()` pair. Its claims are the id token's with the userinfo's written over them — Spring's precedence — and `getName()` reads whichever claim `provider.{id}.user_name_attribute` names (`sub` by default; `id` for GitHub).

Three of those return values are worth pausing on, and the test that pins them says so out loud:

```
/**
 * The principal after an OIDC login, as a controller sees it on the NEXT request – read
 * back from the session:
 * an OidcUser with the merged claims, no raw id token, the userinfo, and `SCOPE_x` honoured
 * by a URL rule and a
 * method rule.
 */
uses(OAuth2ClientCapstoneTestCase::class, SecurityFlows::class);

it('injects the OidcUser into a controller with its claims, and honours hasScope in a URL
rule and a method rule', function () {
    /** @var OAuth2ClientCapstoneTestCase $this */
    $callback = $this->signInThroughProvider();
    $this->forgetSession();
    $client = $this->followSession($callback);

    $account = $client->getJson('/account');
    $account->assertOk()->assertJson([
        'name' => 'ada',
        'subject' => 'ada',
        'email' => 'ada@example.com',
        'fullName' => 'Ada Lovelace',
        'groups' => ['engineering'],
        'idTokenValue' => '',
        'hasUserInfo' => true,
        'issuer' => OAuth2ClientCapstoneTestCase::ISSUER,
    ]);

    $client->getJson('/account/any')->assertOk()->assertJson(['name' => 'ada', 'kind' =>
'oidc']);
```

```

    $client->getJson('/api/scoped')->assertOk()->assertJson(['scoped' => true]);
    $client->getJson('/api/email')->assertOk()->assertJson(['email' => true]);
    $client->getJson('/api/engineers')->assertStatus(403);

    expect($this->events->denials()->toHaveCount(1);
});

```

'idTokenValue' => '' is the first. `DefaultOidcUser` is a `CredentialsContainer`, exactly as the shipped `User` of Chapter 10 is, so the copy the session stores has the id token's **raw value blanked** while every claim decoded out of it survives. The raw id token, the access token and the refresh token live in one place only — the authorized-client entry — and that entry is **encrypted** with the application key. A principal in a session is a set of facts about a person, not a bearer credential somebody could lift out of a session file.

The second is `/api/engineers` answering `403`. The provider granted `ada` a `groups` claim of `['engineering']`, and nothing in OAuth2 says what a group means to your application — so by default it means nothing, and `hasRole('ENGINEERING')` is false. The authorities an OAuth2 login carries are `OIDC_USER` (an `OidcUserAuthority` holding the claims) or `OAUTH2_USER`, plus one `SCOPE_x` per granted scope — the same spelling Chapter 10's resource-server filter produces, which is why `hasScope('email')` reads both without knowing which filter authenticated the request. Turning a claim into a role is the application's decision, and it has a port:

```

/** How an application binds its mapper: a #[Bean] of the port type; the login provider's
optional dependency picks it up. */
#[Configuration]
final class GroupsAuthoritiesConfiguration
{
    #[Bean]
    public function grantedAuthoritiesMapper(): GrantedAuthoritiesMapper
    {

```

```

    return new GroupsAuthoritiesMapper;
}
}

```

```

/** The mapper an application writes: `groups: [engineering]` from the provider becomes
`ROLE_ENGINEERING`, on top of what was granted. */
final class GroupsAuthoritiesMapper implements GrantedAuthoritiesMapper
{
    public function mapAuthorities(array $authorities): array
    {
        $mapped = $authorities;
        foreach ($authorities as $authority) {
            if (! $authority instanceof OAuth2UserAuthority) {
                continue;
            }
            $groups = $authority->getAttributes()['groups'] ?? [];
            foreach (is_array($groups) ? $groups : [] as $group) {
                if (is_string($group) && $group !== '') {
                    $mapped[] = new SimpleGrantedAuthority('ROLE_'.strtoupper($group));
                }
            }
        }
    }

    /** @var list<GrantedAuthority> $mapped */
    return $mapped;
}
}

```

Bind that `#[Bean]` and the same test's `/api/engineers` answers `200`, because `GrantedAuthoritiesMapper` receives the granted list — the authority carrying the claims first — and returns the list the `Authentication` will carry. The principal keeps the granted one, so a mapper can add without erasing what the provider actually said.

The third is that the `Authentication` also remembers **which** registration signed the person in, on its attributes, read back with

`OAuth2AuthenticationToken::registrationId($authentication)`. That is not bookkeeping: logging out and calling an API both need to know which provider to talk to, and the next two sections are those two needs.

Signing out again

`logout.oidc_initiated` binds one bean on the `LogoutSuccessHandler` port Chapter 10 introduced. `LogoutFilter` (-93) asks its success handler **before** it invalidates the session — which is the only reason the handler can still read the encrypted authorized client and find the raw id token to send:

```
/**
 * OpenID Connect RP-Initiated Logout 1.0 (Spring's
 * OidcClientInitiatedLogoutSuccessHandler): after the
 * application's own logout, the browser is sent to the provider's `end_session_endpoint` so
 * the provider's
 * session ends too, with `id_token_hint` (the raw id token of the login – read from the
 * encrypted authorized
 * client the session still holds, because the LogoutFilter asks its success handler BEFORE
 * invalidating the
 * session), `client_id`, and `post_logout_redirect_uri` – `logout.post_logout_redirect_uri`
 * with `{baseUrl}`
 * (and `{registrationId}`) expanded from the application's root, the way the redirect_uri
 * is.
 *
 * WHICH PROVIDER: the registration id the login recorded on the token's attributes. A
 * principal that did not
 * sign in through OAuth2 (a form login in the same application), a registration that no
 * longer exists, a
 * provider with no end-session endpoint (explicitly configured, or a discovery document
 * that names none), or
```

```

* a provider whose discovery is down at that moment, all hand back to the default redirect
– the person IS
* signed out of this application either way, and the last three are logged at warning so an
operator sees
* that the provider's session outlives the application's. Nothing here reads the token's
raw value into a
* log or an exception.
*/
final class OidcClientInitiatedLogoutSuccessHandler implements LogoutSuccessHandler
{
    public function __construct(
        private readonly ClientRegistrationRepository $registrations,
        private readonly OAuth2AuthorizedClientRepository $authorizedClients,
        private readonly OAuth2ClientSettings $settings,
        private readonly Container $container,
        private readonly ?LoggerInterface $logger = null,
    ) {}

    // ...

    public function onLogoutSuccess(Request $request, ?Authentication $authentication): ?
Response
    {
        $registrationId = OAuth2AuthenticationToken::registrationId($authentication);
        if ($registrationId === null) {
            return null;
        }
        // ...
        $endSession = $registration->providerDetails->endSessionUri;
        if ($endSession === null || $endSession === '') {
            $this->logger?->warning("RP-initiated logout skipped for [{registrationId}]:
its provider has no end_session_endpoint (set provider.{registrationId}.end_session_uri, or
accept that the provider's session outlives this one).", ['registration' =>
registrationId]);

            return null;
        }

        /** @var UrlGenerator $urls */
        $urls = $this->container->make(UrlGenerator::class);

```



```

        $idToken = $this->authorizedClients->loadAuthorizedClient($registrationId,
$request)?->idToken;

        $query = [];
        if ($idToken !== null && $idToken !== '') {
            $query['id_token_hint'] = $idToken;
        }
        $query['client_id'] = $registration->clientId;
        $query['post_logout_redirect_uri'] = RedirectUriTemplate::expand($this->settings-
>postLogoutRedirectUri, $urls->to('/'), $registrationId);

        return new RedirectResponse($endSession.(str_contains($endSession, '?') ? '&' :
'').http_build_query($query, '', '&', PHP_QUERY_RFC3986));
    }
}

```

Every branch that returns `null` is a branch where the person is still signed out of *this* application — the handler only decides whether the browser also visits the provider. That is the right shape for a handler at this position: a provider that has no `end_session_endpoint`, a registration somebody deleted from the configuration this morning, or a provider whose discovery is down cannot be allowed to turn "log out" into an error page. Three of the four are logged at warning, so an operator can see that the provider's session is outliving the application's rather than guessing.

`post_logout_redirect_uri` defaults to `{baseUrl}/login?logout` — the framework login page with its signed-out notice, the same one form login lands on — and it has to be **registered at the provider**, which is the one step of this that is not in your own configuration file. The whole round trip is driven end to end in Chromium:

```

// 6. Sign out: the logout POST went to the provider's end-session endpoint with the id
token, which sent the
// browser back to the post-logout URI – the framework login page, signed-out notice
showing.
$page->navigate('/browser-fixture/account')

```

```

->assertSee('Signed in as ada')
->press('Sign out')
->assertPathIs('/login')
->assertQueryStringHas('logout')
->assertSee('You have signed out.')
->assertSee('Sign in with Fake IdP')
->assertNoJavaScriptErrors()
->screenshot(filename: 'oauth2-signed-out');

    expect($this->idp->endSessionRequests)->toHaveCount(1)
    ->and($this->idp->endSessionRequests[0]['id_token_hint'] ?? null)->toBe($this->idp-
>issuedIdTokens[0])
    ->and($this->idp->endSessionRequests[0]['client_id'] ?? null)-
>toBe(FakeAuthorizationServer::CLIENT_ID)
    ->and($this->idp->endSessionRequests[0]['post_logout_redirect_uri'] ?? null)-
>toBe($origin.'/login?logout');

    // 7. The session is gone: the protected page is refused again.
    $page->navigate('/browser-fixture/account')->assertPathIs('/login')-
>assertDontSee('Signed in as');

```

Calling an API as the application

Signing a person in is one of the two things an OAuth2 client does. The other is holding a token so that *your* code can call somebody else's API — a nightly job that posts invoices, a controller that reads a partner's catalogue. That has no browser in it at all, which is why `firefly.security.oauth2.client.enabled` deliberately does **not** require `firefly.security.enabled`: a queue worker with an outbound token has no inbound security to speak of.

`OAuth2AuthorizedClientManager::authorize()` is the one entry point, and it is Spring's two managers in one class because the only thing that differs between them is where the client is looked up:

```

public function authorize(string $clientRegistrationId, ?string $principalName = null):
OAuth2AuthorizedClient
{
    $registration = $this->registrations->findByRegistrationId($clientRegistrationId)
        ?? throw new ConfigurationException("There is no client registration
[{$clientRegistrationId}] under firefly.security.oauth2.client.registration to authorize.");
    $now = Date::now()->getTimestamp();

    if ($registration->authorizationGrantType === AuthorizationGrantType::ClientCredentials)
    {
        return $this->clientCredentials($registration, $principalName ?? $registration-
>clientId, $now);
    }

    return $this->userBound($registration, $principalName, $now);
}
// ...
private function clientCredentials(ClientRegistration $registration, string $principalName,
int $now): OAuth2AuthorizedClient
{
    $id = $registration->registrationId;
    $client = $this->service->loadAuthorizedClient($id, $principalName);
    if ($client !== null && ! $client->accessToken->isExpired($now, $this->settings-
>clockSkewSeconds)) {
        return $client;
    }

    $response = $this->tokens->clientCredentials($registration, $registration->scopes);
    $client = new OAuth2AuthorizedClient($id, $principalName, $response->accessToken,
$response->refreshToken);
    $this->service->saveAuthorizedClient($client);

    return $client;
}

```

A `client_credentials` registration is the application's own token: looked up in the `OAuth2AuthorizedClientService` (the Laravel cache, encrypted), fetched when it is absent or within `clock_skew` seconds of expiring, and cached — one token per process pool, not one per request. An `authorization_code` registration is a *person's* token: read from the session first when the current request has one (the login put it there, and the request is what binds it to that browser), else from the cache service under the principal name — the path a queued job takes — and refreshed with the refresh token when it is about to expire, keeping the rotated refresh token and the id token and writing the result back to both stores. With nothing to return and nothing to refresh with, the answer is a `ClientAuthorizationRequiredException`: a **401** `CLIENT_AUTHORIZATION_REQUIRED` that says, in effect, *sign in through that registration first*.

Notice what an `OAuth2AuthorizedClient` holds: the registration `id`, the principal name and the tokens — never the `ClientRegistration` itself. A client secret therefore never enters a session or a cache entry, whatever an application later does with the object.

You will rarely call the manager yourself, because there is a macro over it:

```
/**
 * `Http::oauth2Client('{registrationId}')->` – Spring's
 ServletOAuth2AuthorizedClientExchangeFilterFunction as a
 * Laravel Http macro: a PendingRequest from the application's Http factory carrying the
 bearer the
 * OAuth2AuthorizedClientManager hands out for the registration (and, optionally, for a
 named principal), so
 * `Http::oauth2Client('billing')->get($url)` fetches or refreshes the token, attaches it,
 and is otherwise
 * exactly `Http::get($url)`: faked by Http::fake(), traced by the observability wave's
 client middleware,
 * retried with ->retry(), and so on.
 *
 * REGISTERED AT PROVIDER register() TIME, not from a boot pass, so Larastan – which boots
```

```

the discovered providers
* when it analyses – types the call; behind `firefly.security.oauth2.client.http.macro`
(default true). The
* closure resolves the factory AND the manager from the container on every call: the
factory is
* the instance Http::fake() stubs (and the one the facade swaps), and the manager is a bean
of the package
* master, so calling the macro with the package off is a ConfigurationException naming the
key – not a
* container error deep inside Laravel. The closure does not read `$this` (the factory it is
bound to) on
* purpose: resolving the factory from the container answers the same instance and keeps
PHPStan out of the
* closure-binding question.
*/
final class OAuth2ClientHttpMacros
{
    public const string OAUTH2_CLIENT = 'oauth2Client';

    public static function register(Container $app): void
    {
        // NOT a static closure: Macroable::__call() binds the closure to the factory
instance, and PHP warns
        // ("Cannot bind an instance to a static closure") on every call when asked to bind
a static one.
        HttpFactory::macro(self::OAUTH2_CLIENT, function (string $registrationId, ?string
$principalName = null) use ($app): PendingRequest {
            if (! $app->bound(OAuth2AuthorizedClientManager::class)) {
                throw new ConfigurationException("Http::oauth2Client('{ $registrationId }')
needs firefly.security.oauth2.client.enabled: the OAuth2AuthorizedClientManager is a bean of
the package master.");
            }

            /** @var OAuth2AuthorizedClientManager $manager */
            $manager = $app->make(OAuth2AuthorizedClientManager::class);

            /** @var HttpFactory $http */
            $http = $app->make(HttpFactory::class);

            return $http->withToken($manager->authorize($registrationId, $principalName)-

```

```

>accessToken->tokenValue);
    });
}
}

```

`Http::oauth2Client('billing')->get($url)` is `Http::get($url)` with a bearer on it, and it stays an ordinary Laravel Http call in every way that matters to a test:

`Http::fake()` intercepts it, the observability wave's client middleware traces it, and `->retry()` still works. `Http::oauth2Client('corp', 'ada')` names the principal for a user-bound registration. The one thing to remember about the cache-backed store is that `authorized_client.cache_ttl` (a day, by default) is how long a renewable client survives there; a registration whose tokens must outlive that — a worker acting as a person for weeks — wants an `OAuth2AuthorizedClientService` binding of your own over a table, which is a single `#[Bean]`.

Running your own authorization server

Turn the picture around. `firefly/security-oauth2-server` is Spring Authorization Server's model inside your application: registered clients, the authorization-code grant with PKCE and a consent page, client credentials, refresh tokens, JWT or opaque access tokens, and the discovery document that tells everybody else where all of it lives. Its configuration block is longer than the client's because it is a protocol surface rather than a relationship, but every value below is a default you can ignore until you need it:

```

'server' => [
    'enabled' => env('FIREFLY_OAUTH2_SERVER_ENABLED', false),
    // ...
    // 'authorization_endpoint' => '/oauth2/authorize',

```

```

    // 'token_endpoint' => '/oauth2/token',
    // 'jwk_set_endpoint' => '/oauth2/jwks',
    // 'token_introspection_endpoint' => '/oauth2/introspect',
    // 'token_revocation_endpoint' => '/oauth2/revoke',
    // 'oidc_user_info_endpoint' => '/userinfo',
    // 'oidc_logout_endpoint' => '/connect/logout',
    // 'oidc_client_registration_endpoint' => '',
    // ...
    'jwt' => [
        'signing_key' => env('FIREFLY_OAUTH2_SERVER_SIGNING_KEY', ''),
        // 'key_id' => '',
        // 'algorithm' => 'RS256',
        // 'previous_keys' => [],
    ],
    // ...
    'access_token' => [
        // 'format' => 'self_contained',
        // 'ttl' => 300,
    ],
    // ...
    'refresh_token' => [
        // 'ttl' => 3600,
        // 'reuse' => false,
    ],
    // ...
    'authorization_code' => [
        // 'ttl' => 300,
    ],
    // ...
    'id_token' => [
        // 'ttl' => 1800,
    ],
    // ...
    // 'require_pkce' => true,
    // 'require_proof_key_for_public_clients' => true,
    // ...
    'consent' => [
        // 'required' => true,
        // 'view' => '',

```

```

    ],
    // ...
    'clients' => [
        'driver' => env('FIREFLY_OAUTH2_SERVER_CLIENTS_DRIVER', 'memory'),
        // 'web-app' => [
            // 'client_id' => 'web-app',
            // 'client_secret' => '{bcrypt}$2y$10$...',
            // 'client_name' => 'The web application',
            // 'client_authentication_methods' => ['client_secret_basic'],
            // 'authorization_grant_types' => ['authorization_code', 'refresh_token'],
            // 'redirect_uris' => ['https://app.example.com/login/oauth2/code/web-app'],
            // 'post_logout_redirect_uris' => ['https://app.example.com/'],
            // 'scopes' => ['openid', 'profile', 'email'],
            // 'client_settings' => ['require_pkce' => true, 'require_authorization_consent'
=> true],
            // 'token_settings' => ['access_token_ttl' => 300, 'refresh_token_ttl' => 3600,
'reuse_refresh_tokens' => false, 'access_token_format' => 'self_contained',
'authorization_code_ttl' => 300, 'id_token_ttl' => 1800],
            // ],
        ],
    // ...
    'authorizations' => [
        'driver' => env('FIREFLY_OAUTH2_SERVER_AUTHORIZATIONS_DRIVER', 'memory'),
        'purge' => [
            // 'enabled' => false,
            // 'cron' => '*/15 * * * *',
        ],
    ],
    // ...
    'rate_limit' => [
        // 'enabled' => false,
        // 'max_tokens' => 60,
        // 'refill_rate' => 1.0,
        // 'idle_ttl' => 2592000,
    ],
],
],

```


Those paths — plus the two `/.well-known/` documents that advertise them — are what `OAuth2AuthorizationServerFilter` (`-82`) answers, and its position in the chain answers two questions people ask immediately. It runs *after* the session filter at `-94` and the bearer filters — because the authorization endpoint needs whatever principal there is — and *before* `CsrfFilter` (`-80`) and `HttpSecurityFilter` (`-70`), which is why a machine's `POST /oauth2/token` is never asked for a session token and why no deny-by-default URL rule can accidentally close the server. Your `http.rules` may end with `*` → `authenticated` and the metadata, the JWKS and the token endpoint still answer; the authorization and userinfo endpoints check the principal themselves. Nothing needs adding to `http.rules` or to `csrf.except`, and the one browser form the server owns — the consent POST — is checked against the session token by the endpoint itself, exactly as the login and logout filters check theirs. A request whose path is none of the server's costs one path comparison and falls through.

Path (default)	Method	Answers
<code>/.well-known/openid-configuration</code> , <code>/.well-known/oauth-authorization-server</code>	GET	One discovery document (OIDC's is a superset of RFC 8414's): every endpoint URL, the grants, the client-authentication methods, <code>S256</code> , and the signing algorithm
<code>/oauth2/jwks</code>	GET	The public keys — current first, previous after — with <code>Cache-Control: public, max-age=3600</code>
<code>/oauth2/authorize</code>	GET, POST	The authorization endpoint and the consent form
<code>/oauth2/token</code>	POST	<code>authorization_code</code> (with <code>code_verifier</code>), <code>client_credentials</code> , <code>refresh_token</code> ; credentials are read from the body only, never the query string
<code>/oauth2/introspect</code>	POST	RFC 7662: <code>{active: false}</code> for anything unknown, revoked, expired, or a code
<code>/oauth2/revoke</code>	POST	RFC 7009: a refresh token takes its access token with it; unknown is <code>200</code>
<code>/userinfo</code>	GET, POST	The claims of the <code>OidcUserInfoMapper</code> , for a bearer carrying <code>openid</code>

Path (default)	Method	Answers
<code>/connect/logout</code>	GET, POST	RP-initiated logout: the other side of the section above
<code>/connect/register</code> (off)	POST	RFC 7591 dynamic registration, behind a single-use bearer carrying <code>client.create</code>

The authorization endpoint is the one with a person in it, and it runs in a fixed order: the `client_id` and an exactly-matching registered `redirect_uri` first — refused **on the page**, never redirected, because RFC 6749 §4.1.2.1 says an unverified redirect target is where an attacker would like the error to go — then `response_type=code`, the client's grants, the requested scopes and PKCE, each of which *is* a redirect carrying `error=`. Only then does it look for a resource owner, and it looks in one place: the principal the `SecurityContextRepository` holds **between** requests. A bearer authenticated by the resource-server filter at -85 is not a resource owner and cannot redeem itself for a code. With nobody signed in, the `AuthenticationEntryPoint` of Chapter 10 takes over — the login page, with the authorization request saved — and the return visit lands back on `/oauth2/authorize` to meet the consent page.

Consent is the framework's own page, built from the login page's design and carrying no JavaScript, unless `consent.view` names a Blade view of yours; either way the model is a `ConsentPageModel` with the client name, the scopes as `{scope, description, approved}` triples, the CSRF token and the form action. Approving records an `OAuth2AuthorizationConsent` (merged with whatever was approved before), so the next authorization request for the same scopes skips the page; denying is `access_denied` back at the client.

What the token endpoint then hands out is a short-lived access token — a JWT with RFC 9068's claims by default, or 32 opaque random bytes when

`access_token.format` is `reference` — and, for a confidential client with the grant, a refresh token. The refresh token is where the interesting rule lives, and it is easier to read as a test than as a paragraph:

```
it('rotates the refresh token, invalidates the previous access token, narrows the scope on
request, and detects reuse by revoking the family', function () {
    /** @var RefreshCapstoneTestCase $this */
    $this->signIn();
    $oauth2 = $this->oauth2();
    $first = $oauth2->tokens('web-app', OAuth2ServerCapstoneTestCase::REDIRECT_URI, 'openid
profile orders:read', OAuth2ServerCapstoneTestCase::WEB_APP_SECRET);
    /** @var string $firstRefresh */
    $firstRefresh = $first['refresh_token'];
    /** @var string $firstAccess */
    $firstAccess = $first['access_token'];

    $refreshed = $oauth2->refresh('web-app', $firstRefresh,
OAuth2ServerCapstoneTestCase::WEB_APP_SECRET, 'openid orders:read');
    $refreshed->assertOk()->assertJson(['token_type' => 'Bearer', 'scope' => 'openid
orders:read']);
    /** @var array<string,mixed> $second */
    $second = $refreshed->json();
    /** @var string $secondRefresh */
    $secondRefresh = $second['refresh_token'];
    expect($secondRefresh)->not->toBe($firstRefresh)
        ->and($second['access_token'])->not->toBe($firstAccess)
        ->and($second)->toHaveKey('id_token');

    /** @var OAuth2AuthorizationService $service */
    $service = $this->app()->make(OAuth2AuthorizationService::class);
    $authorization = $service->findByToken($secondRefresh, OAuth2TokenType::RefreshToken);
    expect($authorization?->authorizedScopes)->toBe(['openid', 'profile', 'orders:read'])
        ->and($authorization?->refreshTokenFamily()->toHaveCount(1)
        // The previous access token was replaced on the record: it no longer resolves, so
```

```

introspection says inactive.

    ->and($service->findByToken($firstAccess, OAuth2TokenType::AccessToken))-
>toBeNull();

    // The new token widens back to the original grant when asked for nothing narrower.
    $oauth2->refresh('web-app', $secondRefresh,
OAuth2ServerCapstoneTestCase::WEB_APP_SECRET)->assertOk()->assertJson(['scope' => 'openid
profile orders:read']);

    // Presenting the FIRST refresh token again is a replay: the whole authorization is
gone.
    $oauth2->refresh('web-app', $firstRefresh,
OAuth2ServerCapstoneTestCase::WEB_APP_SECRET)->assertStatus(400)->assertJson(['error' =>
'invalid_grant']);
    expect($service->findById($authorization->id ?? ''))->toBeNull();
    $oauth2->refresh('web-app', $secondRefresh,
OAuth2ServerCapstoneTestCase::WEB_APP_SECRET)->assertStatus(400)->assertJson(['error' =>
'invalid_grant']);
});

```

Four rules in one test. Every refresh **rotates** the token and invalidates the access token it replaces. A refresh may ask for a *narrower* scope, and the authorization still remembers the original grant, so the next refresh can widen back to it. Presenting a superseded refresh token is treated as theft rather than as a mistake: the whole authorization is revoked — which is why the last line, using the *good* token, also fails. And a client that genuinely cannot rotate can set

`token_settings.reuse_refresh_tokens`, in which case it gets its own token back, because a server that stores only hashes has no value to echo.

Two more things about storage, because they decide whether the defaults survive contact with a real deployment. Tokens are stored **by SHA-256 hash**, never by value, so a dump of `oauth2_authorizations` is not a set of credentials. And `clients.driver` and `authorizations.driver` both default to `memory`, which means *this process* — fine for a test and for a single dev server, and wrong the moment there

are two workers, because a code issued by one must be redeemable by the other.

`eloquent` is the answer there, and `php artisan vendor:publish --tag=firefly-oauth2-server-migrations` publishes the tables.

Keys, and rotating them

Everything above hangs from one private key. The framework generates it:

```
php artisan firefly:oauth2:keys           # RSA 2048 → storage/oauth2/
private.pem, chmod 0600
php artisan firefly:oauth2:keys --algorithm=ES256      # an EC P-256 key instead
php artisan firefly:oauth2:keys --out=/run/secrets/oauth2.pem --force
php artisan firefly:oauth2:keys --print               # to stdout, for a secret
manager
```

The command prints the two lines you need next — the environment variable to set, and the `kid` the JWKS will publish — and its options are exactly these:

```
/**
 * Generates the private key the OAuth2 authorization server signs with: RSA 2048 for RS256
 (the default) or
 * P-256 for ES256, written to storage/oauth2/private.pem with owner-only permissions (or
 printed with --print),
 * and the two lines a developer needs next – the env variable to set and the kid the JWKS
 will publish. A file
 * that already exists is never overwritten without --force: a key replaced by accident
 invalidates every token
 * in flight. Rotation is documented in docs/modules/security-oauth2-server.md (move the old
 key to
 * jwt.previous_keys, generate a new one here).
 */
final class OAuth2KeysCommand extends Command
```

```

{
  /** @var string */
  protected $signature = 'firefly:oauth2:keys
    {--algorithm=RS256 : RS256 (an RSA key) or ES256 (an EC P-256 key)}
    {--bits=2048 : the RSA modulus size (ignored for ES256)}
    {--out= : where to write the PEM (default: storage/oauth2/private.pem)}
    {--force : overwrite an existing file}
    {--print : print the PEM to the console instead of writing a file}';

  /** @var string */
  protected $description = 'Generate a private signing key for the OAuth2 authorization
server (firefly.security.oauth2.server.jwt.signing_key).';

```

`--force` exists so that overwriting is deliberate: a key replaced by accident invalidates every token in flight, and no error message anywhere would say why. `jwt.key_id` defaults to the RFC 7638 thumbprint of the public key, which means every node running the same key publishes the same `kid` with no coordination at all.

Rotating is two moves and no downtime. Generate a new key; move the old one into `jwt.previous_keys` as `[[key, key_id]]` — its public half is enough, since it now only needs to *verify* — and point `jwt.signing_key` at the new one. Tokens signed with the old key keep verifying until they expire, the JWKS publishes both, and a resource server reading the local key source sees both. Publishing two keys under one `kid` is refused at boot, because a verifier keeps one key per id and the pair would fail every fresh token at first use — so an explicit `key_id` has to change with the key.

💡 BE YOUR OWN RESOURCE SERVER

The commonest deployment of this package is an application that issues tokens *and* accepts them. Turn on `firefly.security.oauth2.resource_server` from Chapter 10 with `jwt_source: local`, and the filter reads the key set through the `JwksDocumentSource` bean the server already binds — no HTTP self-fetch, which on a single-process dev server would be a deadlock. Say `local` explicitly rather than leaving `auto: auto` recognises only `/.well-known/jwks.json` as "this application's own", and the server publishes at `/oauth2/jwks`.

Seeing it work

Both halves ship with a way to exercise them that involves no external provider and no network.

For the client half it is `Firefly\Testing\Security\OAuth2\FakeAuthorizationServer` — an entire OpenID Connect provider in one class. `install()` mounts its **front channel** as real routes on the application under test (`/authorize`, optionally rendering a consent page; `/end-session`) and fakes its **back channel** with `Http::fake()` for exactly four URLs: the discovery document, `/token`, `/jwks` and `/userinfo`. Anything else stays unfaked, so a stray outbound request is still a stray outbound request. Tokens are real RS256 JWTs signed with a key pair generated once per process; codes are single-use, bound to their `redirect_uri` and PKCE-checked; the clock is `Date::now()`, so `$this->travel()` expires tokens. Every hop is recorded and every failure is one method away — `overrideIdTokenClaims()`, `signWithUnknownKey()`, `refuseToken()`, `refuseAuthorization()`, `withoutUserInfo()`, `takeDiscoveryDown()`. Because the browser plugin serves the application in-process, the same fake drives a real Chromium:


```

it('signs in through the provider from the login page – every hop, the consent page, the
page the person was refused at – and signs out at the provider too', function (): void {
    /** @var OAuth2LoginBrowserTestCase $this */
    $this->idp->requireConsent();
    $origin = OAuth2LoginBrowserTestCase::origin();

    // 1. The entry point sent the browser to the framework's login page: the provider is
    listed, there is no password form.
    $page = visit('/browser-fixture/account');
    $page->assertPathIs('/login')
        ->assertSee('Sign in with Fake IdP')
        ->assertDontSee('Username')
        ->assertNoJavaScriptErrors()
        ->screenshot(filename: 'oauth2-login-page');

    // 2. The start URL redirected to the provider with state, nonce and a PKCE challenge;
    the fake stopped at its consent page.
    $page->click('Sign in with Fake IdP')
        ->assertPathIs('/fake-idp/authorize')
        ->assertQueryStringHas('response_type', 'code')
        ->assertQueryStringHas('client_id', FakeAuthorizationServer::CLIENT_ID)
        ->assertQueryStringHas('redirect_uri', $origin.'/login/oauth2/code/fake')
        ->assertQueryStringHas('scope', 'openid profile email')
        ->assertQueryStringHas('state')
        ->assertQueryStringHas('nonce')
        ->assertQueryStringHas('code_challenge')
        ->assertQueryStringHas('code_challenge_method', 'S256')
        ->assertSee('asks to sign you in as ada')
        ->assertNoJavaScriptErrors()
        ->screenshot(filename: 'oauth2-consent');

    // 3. Approving sent the code back; the callback exchanged it and landed on the page the
    person was refused at.
    $page->press('Allow')
        ->assertPathIs('/browser-fixture/account')
        ->assertQueryStringMissing('code')
        ->assertQueryStringMissing('state')
        ->assertSee('Signed in as ada')

```

```

->assertSee('ada@example.com')
->assertNoJavaScriptErrors()
->screenshot(filename: 'oauth2-signed-in');

$token = $this->idp->lastTokenRequest();
expect($this->idp->lastAuthorizationRequest()['redirect_uri'] ?? null)->toBe($origin.'/login/oauth2/code/fake')
->and($token['grant_type'])->toBe('authorization_code')
->and($token['authorization'])->toStartWith('Basic ')
->and($token['form']['redirect_uri'] ?? null)->toBe($origin.'/login/oauth2/code/fake')
->and($token['form']['code_verifier'] ?? '')->toMatch('/^[A-Za-z0-9_-]{43}$/')
->and($this->idp->userInfoRequests)->toBe([$this->idp->issuedAccessTokens[0]])
->and($this->idp->jwksRequests)->toBe(1)
->and($this->idp->discoveryRequests)->toBe(1);

```

For the server half it is `Firefly\Testing\Security\OAuth2\OAuth2ServerTestClient`, which drives your *own* server through Laravel's test client: `authorize()` with a fresh PKCE pair and state, `approveConsent()`, `obtainCode()`, `exchangeCode()`, `clientCredentials()`, `refresh()`, `introspect()`, `revoke()` and `userInfo()`. It reads the endpoint paths off the `AuthorizationServerSettings` bean, so renaming one needs no change in a test. The one thing it insists on is that you **sign the person in through the login page and carry the session cookie**: `actingAsPrincipal()` is deliberately not enough at the authorization endpoint, because the endpoint demands the principal the `SecurityContextRepository` holds between requests, so no test double can be a resource owner without a session behind it. The machine helpers need no sign-in at all.

And the whole server round trip is driven in Chromium too, against a relying-party fixture served by the same application:

```

it('drives the whole authorization-code flow in Chromium: login page, consent, the redirect back with the code, then the exchange and a protected API call in-process', function () {

```

```

void {
    /** @var OAuth2ServerBrowserTestCase $this */
    // ...
    // Hop 1: the authorization request from an anonymous browser lands on the framework's
login page.
    $login = visit(authorizeUrl($redirectUri, $proof, $state, $nonce));
    $login->assertPathIs('/login')
        ->assertSee('Sign in')
        ->assertNoJavaScriptErrors()
        ->screenshot(filename: 'oauth2-login');
    // ...
    // Hop 2: signing in returns to the authorization request, which now shows the consent
page.
    $consent = $login->fill('username', 'ada')->fill('password', 'secret')-
>press(SIGN_IN_BUTTON);
    $consent->assertPathIs('/oauth2/authorize')
        ->assertQueryStringHas('client_id', OAuth2ServerBrowserTestCase::CLIENT_ID)
        ->assertQueryStringHas('state', $state)
        ->assertQueryStringHas('nonce', $nonce)
        ->assertSee('Allow access?')
        ->assertSee('The browser relying party')
        ->assertSee('openid')
        ->assertSee('profile')
        ->assertNoJavaScriptErrors()
        ->screenshot(filename: 'oauth2-consent');
    // ...
    // In-process, as the relying party's back channel: the exchange with the verifier and
the client secret.
    $tokens = $this->withBasicAuth(OAuth2ServerBrowserTestCase::CLIENT_ID,
OAuth2ServerBrowserTestCase::CLIENT_SECRET)
        ->post('/oauth2/token', ['grant_type' => 'authorization_code', 'code' => $code,
'redirect_uri' => $redirectUri, 'code_verifier' => $proof->verifier], ['Accept' =>
'application/json']);
    $this->flushHeaders();
    $tokens->assertOk()->assertJson(['token_type' => 'Bearer', 'scope' => 'openid
profile']);
    /** @var array<string,mixed> $body */
    $body = $tokens->json();
    expect($body)->toHaveKeys(['access_token', 'refresh_token', 'id_token']);
}

```

```

    /** @var string $accessToken */
    $accessToken = $body['access_token'];
    /** @var string $idToken */
    $idToken = $body['id_token'];

    $id = $this->app()->make(JwtGenerator::class)->decode($idToken);
    expect($id)->toMatchArray(['sub' => 'ada', 'aud' =>
[OAuth2ServerBrowserTestCase::CLIENT_ID], 'nonce' => $nonce]);
    // ...
    $this->forgetSession();
    $this->withHeader('Authorization', 'Bearer '.$accessToken)->getJson('/api/browser-
fixture/profile')
        ->assertOk()
        ->assertJson(['sub' => 'ada', 'authorities' => ['SCOPE_openid', 'SCOPE_profile']]);
    $this->flushHeaders();
    $this->getJson('/api/browser-fixture/profile')->assertStatus(401);

```

The last four lines are the part worth keeping: a token this application minted, sent by a process holding no session, authenticates on a route this application protects — through `OAuth2ResourceServerFilter` (-85) and nothing else — and the same route without the token is a `401`. That is the whole loop closed inside one test.

In production the same question is answered by the actuator. `/actuator/oauth2clients` lists every registered client with its methods, grants, scopes, redirect URIs and settings, plus how many authorizations are alive for it — and never a secret, because the payload is assembled field by field and `clientSecret` is named nowhere in it. It is unexposed by default like everything past `health` and `info`, and the admin dashboard's **OAuth2 clients** page (`/firefly/oauth2`, in the Wiring group) renders it in-process. Two details on that page repay a moment: PKCE is reported twice — as the client registered it and as the endpoints actually enforce it, which differ whenever the server-wide `require_pkce` is on — and the authorization counts carry a `processLocal` flag, which is `true` on the `memory` driver and is the difference between "this client has no live authorizations" and "this worker has issued none".

 LARAVEL PARITY

The client half is the ground Socialite covers, and covers differently: Socialite gives you a driver per provider and expects you to write the controller, call `Auth::login()` and decide what to do with the payload, and it reads userinfo rather than validating an id token. Here the flow is configuration in Spring Boot's shape, the id token is checked signature-and-claim by claim, PKCE and a single-use constant-time `state` are not optional, and the result is a principal the rest of the framework already understands. The server half is Passport's ground, and the differences are the protocol surface: Passport has no discovery document, no introspection endpoint and no OIDC, while this package ships RFC 8414, RFC 7662, RFC 7009, OIDC Core's userinfo, RP-Initiated Logout and RFC 7591 — over the same session, login page, entry point and `PasswordEncoder` Chapter 10 already gave you, rather than over a parallel stack of its own.

✓ What you learned

Concept	What it does
<code>firefly/security-oauth2-client</code>	Your application as a relying party : registrations, discovery, authorization-code login, an <code>OidcUser</code> principal, RP-initiated logout, outbound tokens
<code>firefly/security-oauth2-server</code>	Your application as an authorization server : clients, consent, <code>/oauth2/token</code> , introspection, revocation, userinfo, JWKS, discovery
<code>OAuth2AuthorizationRequestRedirectFilter</code> (-89)	Answers <code>GET /oauth2/authorization/{id}</code> : builds the request, keeps it in the session, redirects to the provider
<code>OAuth2LoginAuthenticationFilter</code> (-88)	Answers <code>GET /login/oauth2/code/{id}</code> : pulls the saved request (single use), checks <code>state</code> , exchanges the code, signs the person in
<code>OAuth2AuthorizationServerFilter</code> (-82)	The server's one filter — after the session and bearer filters, before CSRF and the URL rules, so no rule and no <code>csrf.except</code> entry is needed
Registrations and providers	<code>registration.{id}</code> is one relationship, <code>provider.{id}</code> is one issuer; five presets, and <code>issuer_uri</code> discovers the rest
<code>OAuth2AuthorizationRequestResolver</code>	A fresh 43-character <code>state</code> every time, a <code>nonce</code> for every <code>openid</code> request, and an S256 PKCE verifier — one generator, 256 bits each

Concept	What it does
<code>OidcUser</code>	Merged claims (userinfo over id token), <code>SCOPE_*</code> per granted scope, and a blanked raw id token in the session copy
<code>GrantedAuthoritiesMapper</code>	The <code>#[Bean]</code> that turns a <code>groups / roles</code> claim into <code>ROLE_*</code> ; it adds to the granted list rather than replacing it
<code>OidcClientInitiatedLogoutSuccessHandler</code>	<code>logout.oidc_initiated</code> : the provider's <code>end_session_endpoint</code> with <code>id_token_hint</code> , read before the session is invalidated
<code>OAuth2AuthorizedClientManager</code>	<code>client_credentials</code> from the cache, <code>authorization_code</code> from the session or the cache, refreshed within <code>clock_skew</code> of expiry
<code>Http::oauth2Client('{id}')</code>	An ordinary <code>PendingRequest</code> with that registration's bearer on it: fakeable, traced, retryable
PKCE, <code>state</code> , <code>nonce</code>	S256 by default and mandatory for a public client; <code>state</code> single-use, session-bound and compared in constant time; <code>nonce</code> always sent and always checked
Refresh-token rotation	Every refresh rotates and invalidates the access token it replaces; presenting a superseded token revokes the whole authorization
<code>firefly:oauth2:keys</code>	RSA 2048 or P-256, <code>0600</code> , never overwritten without <code>--force</code> ; <code>kid</code> defaults to the RFC 7638 thumbprint

Concept	What it does
<code>jwt.previous_keys</code>	Rotation with no downtime: the old key still verifies and still appears in the JWKS; two keys under one <code>kid</code> is a boot refusal
<code>memory</code> vs <code>eloquent</code> drivers	<code>memory</code> is one process — a test or a single dev server; more than one worker needs <code>eloquent</code> , or a code one process issued is unredeemable by another
<code>FakeAuthorizationServer</code>	A whole OIDC provider in one class: real routes for the front channel, <code>Http::fake()</code> for exactly four back-channel URLs
<code>OAuth2ServerTestClient</code>	Drives your own server end to end, over a real sign-in — because the authorization endpoint demands a session-held principal
<code>/actuator/oauth2clients</code>	Every client, its grants and its live authorization count, never a secret; <code>processLocal</code> says whether that count describes the deployment

Try it yourself

1. **Sign in through a real provider, twice.** Add a `google` registration with nothing but `client_id` and `client_secret` and confirm the login page grows a button; then add a second registration `corp` on `keycloak` with only `provider.keycloak.issuer_uri` set, and watch discovery fill in the five endpoints you never typed — authorization, token, JWKS, userinfo and end-session. Take the issuer offline and reload `/login`: one button disappears with a warning in the log, the other still works.

2. **Watch the state get spent.** Start a login, copy the whole callback URL out of the address bar before the page finishes, and replay it. The second attempt is `authorization_request_not_found` — not a state mismatch — because the saved request is pulled from the session before anything is compared. Then start again, tamper with one character of `state`, and see `invalid_state_parameter` instead.
3. **Turn a claim into a role.** Bind a `GrantedAuthoritiesMapper` that maps a `groups` claim to `ROLE_*`, put `#[PreAuthorize("hasRole('ENGINEERING')")]` on an action, and confirm the same person who got a `403` before the bean existed gets a `200` after it. Then confirm `hasScope('email')` was already true without any mapper at all, and say why.
4. **Be both halves at once.** In one application, turn on `oauth2.server` with a `web-app` client, and `oauth2.resource_server` with `jwt_source: local`. Mint a token with `OAuth2ServerTestClient`, call a protected API with it, then rotate the signing key into `jwt.previous_keys` and call the API again with the *old* token: it still verifies. Now remove `previous_keys` and watch the same token stop.
5. **Break the pair on purpose.** Turn `firefly.security.http_basic.enabled` on beside the authorization server and read the boot refusal. Then reason it out from Chapter 10's filter table alone, without rerunning the boot: which filter answers first, what it answers, and why no OAuth2 error would ever have been logged.

PART IV

Observability, Testing & Delivery

Observability: Health, Metrics, and the Actuator

By the end of this chapter you will know `firefly/actuator`'s `HealthIndicator` SPI and the built-in `Ping / DiskSpace / Db` indicators, how `HealthEndpoint` aggregates them into a single `/actuator/health` response — and how a probe **group** (the mechanism behind "liveness" and "readiness") is nothing more than a named, configured subset of indicators, how the whole management surface is **unexposed by default** so a forgotten endpoint fails closed as a 404 rather than an information leak, and `firefly/observability`'s pure-PHP `MeterRegistry`, its locale-safe Prometheus exporter, the per-meter histogram buckets that turn a Prometheus summary into something `histogram_quantile()` can answer, the structured log formats (`json` , `ecs` , `logstash`) that `firefly.logging.structured.format` switches a channel to and the four ids every line then carries, and the exact `#[Order(500)]` precedence trick — the same one Chapter 10 showed you for security — that lets `MeterRegistryCqrsMetrics` replace the CQRS bus's `NoOpCqrsMetrics` with no code change to `firefly/cqrs` at all. The chapter closes on `firefly/admin`, the server-rendered browser dashboard over those same endpoints — a drawn **bean graph** that resolves every constructor dependency through the interface it is wired by and reports the cycles a boot would otherwise die on with no message. It reads those endpoints **in-process**, so it renders pages the JSON surface deliberately keeps unexposed, which makes its own URL the entire security boundary and its default (`app.debug`) the most important line in the package.

i NEW TERM: ACTUATOR

An **actuator** is a management endpoint that reports on the *running process itself* — is it healthy, what did it boot with, how fast are its requests — rather than on the business domain the process serves. The term and the shape both come from Spring Boot Actuator; `firefly/actuator` is a first-party, dependency-light PHP analogue: framework endpoints mounted directly on the same Illuminate `Router` your own controllers use, not a separate admin process.

The `HealthIndicator` SPI

A health check in LaraFly is any `#[Component]` implementing one method:

```
interface HealthIndicator
{
    public function health(): Health;
}
```

`Health` is an immutable status-plus-details reading, built exclusively through four named factories:

```
final readonly class Health
{
    // ...
    public function __construct(
        public Status $status,
        public array $details = [],
    ) {}
    // ...
    public static function up(array $details = []): self
    {
        return new self(Status::Up, $details);
    }
}
```

```

    }
    // ...
    public static function down(array $details = []): self
    {
        return new self(Status::Down, $details);
    }
    // ...
    public static function outOfService(array $details = []): self
    {
        return new self(Status::OutOfService, $details);
    }
    // ...
    public static function unknown(array $details = []): self
    {
        return new self(Status::Unknown, $details);
    }
}

```

`Status` is a backed enum carrying its own severity ordering and the HTTP status it maps to — `DOWN` and `OUT_OF_SERVICE` both render as `503`, so a load balancer needs no special-casing to treat either as "take this instance out of rotation":

```

enum Status: string
{
    case Up = 'UP';
    case Down = 'DOWN';
    case OutOfService = 'OUT_OF_SERVICE';
    case Unknown = 'UNKNOWN';

    public function severity(): int
    {
        return match ($this) {
            self::Down => 3,
            self::OutOfService => 2,
            self::Up => 1,
            self::Unknown => 0,
        };
    }
}

```

```

    };
  }

  public function httpStatus(): int
  {
    return match ($this) {
      self::Down, self::OutOfService => 503,
      default => 200,
    };
  }
}

```

The built-in indicators: Ping, DiskSpace, Db

Three `HealthIndicator`s ship with `firefly/actuator`, and all three are worth reading end to end — they are short, and each teaches a different design decision.

`PingHealthIndicator` is the trivial always-up probe, and its own docblock names exactly what it is for:

```

/** The trivial liveness probe – always UP. Discovered by HealthContributorRegistrar under
the name 'ping'. */
#[Component]
final class PingHealthIndicator implements HealthIndicator
{
    public function health(): Health
    {
        return Health::up();
    }
}

```

`DiskSpaceHealthIndicator` reads a path and a byte threshold from config, and degrades gracefully — an unreadable path is reported DOWN with an explanatory detail rather than throwing:

```
#[Component]
final class DiskSpaceHealthIndicator implements HealthIndicator
{
    public function __construct(private readonly Config $config) {}

    public function health(): Health
    {
        $path = $this->config->string('firefly.management.endpoint.health.diskspace.path',
getcwd() ?: '.');
        $threshold = $this->config-
>int('firefly.management.endpoint.health.diskspace.threshold', 10_485_760);

        $free = @disk_free_space($path);
        $total = @disk_total_space($path);
        if ($free === false || $total === false) {
            return Health::down(['path' => $path, 'error' => 'unable to determine disk
space']);
        }

        $details = ['total' => (int) $total, 'free' => (int) $free, 'threshold' =>
$threshold, 'path' => $path];

        return $free >= $threshold ? Health::up($details) : Health::down($details);
    }
}
```


`DbHealthIndicator` is **on by default** — `matchIfMissing: true` on its `#[ConditionalOnProperty]`, exactly as Spring Boot's `DataSourceHealthIndicator` auto-configuration is. It used to be opt-in, and the change is worth understanding, because "on by default" is not what stops a database-less project from seeing a surprise `DOWN`. A second interface is:

```
#[Component]
#[ConditionalOnProperty(name: 'firefly.management.endpoint.health.db.enabled', havingValue:
'true', matchIfMissing: true)]
final class DbHealthIndicator implements ConditionalHealthIndicator
{
    // ...
    public function available(): bool
    {
        $default = $this->config->get('database.default');
        if (! is_string($default) || $default === '') {
            return false;
        }

        $driver = $this->config->get("database.connections.{$default}.driver");

        return is_string($driver) && $driver !== '';
    }

    public function health(): Health
    {
        try {
            $connection = $this->connections->connection();
            $connection->selectOne('select 1 as ok');

            $driverName = method_exists($connection, 'getDriverName') ? $connection-
>getDriverName() : null;
            $driver = is_string($driverName) ? $driverName : 'unknown';

            return Health::up(['database' => $driver]);
        } catch (Throwable $e) {
```

```

        return Health::down(['error' => $e::class.': '.$e->getMessage()]);
    }
}
}

```

`ConditionalHealthIndicator` adds exactly one method to the SPI, and `available()` is this framework's answer to Spring's `@ConditionalOnBean(DataSource)`.

`HealthEndpoint` asks it before running the check, and an indicator that answers `false` contributes **no component at all** — not an `UNKNOWN`, not a `DOWN`, simply no `db` key in the response. Here the answer is read out of Laravel's own configuration: `database.default` has to name a connection, and that connection's entry has to declare a `driver`. So an application that genuinely has no database gets a health document that does not mention one, which is the honest answer; an application that *has* a database and still wants the check gone sets

`firefly.management.endpoint.health.db.enabled=false` and the bean is never registered.

That is the distinction worth carrying forward: the **property** decides whether the indicator exists, and `available()` decides whether an existing indicator has anything to say. Any `HealthIndicator` you write may implement the same interface and get the same treatment.

Notice `DbHealthIndicator` depends directly on Illuminate's own `ConnectionResolverInterface` rather than on anything from `firefly/data` — there is no `Actuator → Data` edge in `deptrac.yaml` at all, so a DB health check costs `firefly/actuator` no new dependency. And every indicator here follows the same fail-safe shape: a query, a comparison, or a filesystem call that could throw is always caught and turned into `Health::down()` with a detail explaining why — never an unhandled exception, never a `500` where a `503` belongs.

Aggregating health: `HealthEndpoint` and probe groups

`HealthEndpoint` is itself a `#[Component]` (not a plain framework internal — it has to be discoverable like any other bean) that reads every registered indicator, runs each one fail-safe, and folds the results down to the single most-severe status — and, since [26.09.4](#), takes one collaborator more than that, a `HealthDetailsAuthorizer` it asks before publishing any per-component detail:

```
#[Component]
final class HealthEndpoint implements ActuatorEndpoint
{
    public function __construct(
        private readonly HealthContributorRegistry $registry,
        private readonly StatusAggregator $aggregator,
        private readonly Config $config,
        private readonly HealthDetailsAuthorizer $authorizer,
    ) {}

    public function endpointId(): string
    {
        return 'health';
    }

    public function enabled(): bool
    {
        return true;
    }

    public function handle(EndpointRequest $request): ?EndpointResponse
    {
        $indicators = $this->registry->all();

        if ($request->subPath !== []) {
            $members = $this->group($request->subPath[0]);
            if ($members === null) {
                return null; // unknown group → 404
            }
        }
    }
}
```

```

    }

    $indicators = array_intersect_key($indicators, array_flip($members));
}

$components = [];
$statuses = [];
foreach ($indicators as $name => $indicator) {
    $health = $this->readFailSafe($indicator);
    $statuses[] = $health->status;
    $components[$name] = ['status' => $health->status->value, 'details' => $health-
>details];
}

$status = $this->aggregator->aggregate($statuses);
$body = ['status' => $status->value];

if ($this->showDetails()) {
    $body['components'] = $components;
}

return EndpointResponse::json($body, $status->httpStatus());
}

private function readFailSafe(HealthIndicator $indicator): Health
{
    try {
        return $indicator->health();
    } catch (Throwable $e) {
        return Health::down(['error' => $e::class.' '. $e->getMessage()]);
    }
}

// ...

private function group(string $name): ?array
{
    $key = "firefly.management.endpoint.health.group.{ $name }.include";
    if (! $this->config->has($key)) {
        return null;
    }
}

```

```

    return array_values(array_filter(
        array_map('trim', explode(',', $this->config->string($key))),
        static fn (string $segment): bool => $segment !== '',
    ));
}

/**
 * `always` publishes details to every caller; `never` to none; `when-authorized` asks
the
 * HealthDetailsAuthorizer, which is the deny-by-default port firefly/security fills
when its master flag
 * is on (see that interface for why the question lives here and the answer lives
there). Anything else –
 * including a typo – is `never`: the fail-closed reading, because the cost of
withholding a detail is a
 * support question and the cost of publishing one is a disclosure.
 */
private function showDetails(): bool
{
    return match ($this->config->string('firefly.management.endpoint.health.show-
details', 'never')) {
        'always' => true,
        'when-authorized' => $this->authorizer->mayReadDetails(),
        default => false,
    };
}
}

```

Two things are worth pausing on. First, with no request path at all, every registered indicator counts and `StatusAggregator` folds an empty set to `UP` (empty means nothing to be unhealthy about) and otherwise picks the single most-severe `Status` by `severity()` — one `DOWN` indicator is enough to bring the whole aggregate down, whatever the others report. Second, **there is no separate "liveness" or "readiness"**

endpoint class anywhere in the source — `/actuator/health/{group}` is a single generic mechanism, and "liveness"/"readiness" are simply *conventional names* you configure, not a hard-coded Kubernetes-shaped feature:

```
<?php
declare(strict_types=1);

return [
    'management' => [
        'endpoint' => [
            'health' => [
                'group' => [
                    'liveness' => ['include' => 'ping'],
                    'readiness' => ['include' => 'ping,db'],
                ],
            ],
        ],
    ],
];
```

Configured this way, `GET /actuator/health/liveness` aggregates only `ping` (so a healthy-but-momentarily-database-less process still reports alive) and `GET /actuator/health/readiness` folds in `db` too — but the mechanism underneath is the exact same `group()` lookup and the exact same `Health / Status` types the rest of this section already showed you. A group name nobody configured returns `null` from `group()`, which `handle()` turns into a plain `404`, not a `200` with an empty body.

`show-details` (`never / when-authorized / always`, default `never`) governs whether the response includes the per-component `details` map at all — with `never`, an unauthenticated caller sees only the aggregate `status`, never *which* indicator failed or

why. Anything else, *including a typo*, reads as `never`: the fail-closed direction, because the cost of withholding a detail is a support question and the cost of publishing one is a disclosure.

And `when-authorized` now means what its name says. For two releases it degraded to `never`, because `firefly/actuator` has no code edge to `firefly/security` and therefore could not say *who was asking* — safe, and also a documented lie about what the value did. 26.09.4 closed it with a shape this book has already shown you — Chapter 7's `CqrsMetrics` port and its no-op default are the same move: the package that owns the **question** declares a port and ships the conservative answer, and the package that owns the **answer** fills it. The port is

`Firefly\Actuator\Health\HealthDetailsAuthorizer`, a deliberately argument-less `mayReadDetails(): bool`; actuator's own implementation is `DenyHealthDetailsAuthorizer`, which refuses everybody and is the old behavior by another name. Enable the included `firefly/security` component with `firefly.security.enabled` and `PrincipalHealthDetailsAuthorizer` takes over: it reads the session-held principal from the same `SecurityContextHolder` Chapter 10's rules read, and matches it against `firefly.management.endpoint.health.roles` through the same `RoleHierarchy`, so `ROLE_ADMIN > ROLE_ACTUATOR` grants what it says. An empty `roles` is Spring's "any authenticated principal"; a non-empty one (a list, or Spring's CSV string — a bare `ADMIN` is read as `ROLE_ADMIN`) narrows it further, and a value that cannot be read as a role list at all refuses everybody rather than admitting them. The new dependency edge is `Security → Actuator` and only that direction: `firefly/actuator` still names no principal type and still works in an application with no security at all. Both beans are `#[ConditionalOnMissingBean]`, so your own `HealthDetailsAuthorizer` displaces either one.

⚠ BREAKING CHANGE IN 26.09.4 — SEE [CHANGELOG.MD](#) UNDER [26.09.4]

Two things moved, and both are recorded there under **BREAKING**. First, **an application already running `show-details: when-authorized` with security on starts disclosing what it used to withhold**: at the default `roles ( [], "any authenticated principal"`), every authenticated caller now reads the component details — which name database drivers, disk paths, broker hosts and indicator error messages. Decide rather than inherit: set `show-details: never`, or list the roles that may read them in `firefly.management.endpoint.health.roles`, or bind your own `HealthDetailsAuthorizer`. Second, **HealthEndpoint's constructor gained a required fourth parameter**, `HealthDetailsAuthorizer $authorizer`, so code that constructs the endpoint directly instead of resolving the bean must now pass one — `new DenyHealthDetailsAuthorizer` reproduces the old behavior exactly. An application with `firefly/security` absent, or its master flag off, is unaffected: the deny default stands and the response body is byte for byte what it was.

Secure by default: exposure and the 404

The single most important operational fact about the actuator is this: **most endpoints are unreachable until you say otherwise**. `ExposureModel` is the whole mechanism, and it is short enough to read in full:

```
final readonly class ExposureModel
{
    // ...
    public function __construct(
        private array $include,
        private array $exclude,
        public string $basePath,
    ) {}

    public static function fromConfig(Config $config): self
```



```

    {
        $include = self::csv($config-
>string('firefly.management.endpoints.web.exposure.include', 'health,info'));
        $exclude = self::csv($config-
>string('firefly.management.endpoints.web.exposure.exclude', ''));
        $base = trim($config->string('firefly.management.endpoints.web.base-path', '/
actuator'), '/');

        return new self($include, $exclude, $base === '' ? 'actuator' : $base);
    }
    // ...
    public function isExposed(string $id): bool
    {
        // ...
        return false;
    }
    // ...
}
// ...
}
}
}

```

The default `include` is `"health,info"` — every other endpoint id (`env`, `beans`, `conditions`, `mappings`, `loggers`, `scheduledtasks`, and observability's `metrics` / `prometheus` once that package is installed) is **not exposed** until you explicitly add it, and `ActuatorDispatchAction` renders an unexposed or unknown id as a plain `404` through the same `ProblemDetailsRenderer` Chapter 4 introduced — never a raw framework error, and never a silent `200` with a body an unauthenticated caller shouldn't see. `.exclude` always wins over `.include`, so `include: '*'` plus a short `exclude` list is a legitimate "expose everything except..." policy.

`/actuator/env` layers a second, independent safety net on top of exposure: even once exposed, a key whose name looks like a credential is masked before the response is ever built. The endpoint itself is four lines, because the rule lives in one shared class:

```
public function handle(EndpointRequest $request): EndpointResponse
{
    /** @var array<string, mixed> $firefly */
    $firefly = (array) $this->config->get('firefly', []);

    return EndpointResponse::json(['firefly' => SensitiveValueMasker::mask($firefly)]);
}
```

```
final class SensitiveValueMasker
{
    public const string MASK = '*****';

    private const string SENSITIVE = '/password|secret|token|key|credential|passwd|
authorization|headers/i';

    // ...

    public static function mask(array $values): array
    {
        $masked = [];

        foreach ($values as $key => $value) {
            if (self::isSensitive($key)) {
                $masked[$key] = self::MASK;

                continue;
            }

            $masked[$key] = is_array($value) ? self::mask($value) : $value;
        }
    }
}
```

```
    return $masked;
  }
```

That the rule lives in a class of its own rather than inside the endpoint is not tidiness. `/configprops` needs the *same* rule, and a second copy is how a masking rule rots: the two lists drift on the very next word somebody thinks to add, and the endpoint that missed the addition leaks.

The ordering inside `mask()` is the part worth studying, because it is a bug fix you can read. The rule as it was first written tested the key only on the branch where the value was a **scalar** — an array-valued key was recursed into, and each leaf was then judged on its own key. So `firefly.security.jwt.keys => ['active' => 'PRIVATE...', 'previous' => '...']` rendered both private keys in full: `keys` matched the pattern but was an array, and `active` and `previous` matched nothing. Every real-world shape of a secret — a keyring, a credentials pair, a per-tenant token map — is exactly that shape, so the bypass covered the cases that mattered most. Here the **key decides first**, and a sensitive key masks its whole subtree whatever type the value has.

A sensitive array masks to the scalar `*****` rather than to a same-shaped array of masks, and that is deliberate too: the shape of a secret is itself information — how many keys are in the keyring, which tenants have tokens — and a caller who may not see the values has no business counting them.

`authorization` and `headers` are in the list because the framework introduced a key that carries a credential: `firefly.observability.tracing.otlp.headers` documents `authorization=Bearer ...` as its intended contents, and none of the six original words appear in `headers`. A bag of headers is where an outbound client's credential travels, whatever the individual header is called — which is exactly why the **bag's** key must decide and not the leaves.

Beyond exposure and masking, `firefly/security`'s `HttpSecurity` (Chapter 10) is what actually locks the surface down for real traffic, and it needs **zero** code changes to do it — `HttpSecurityFilter` is a global middleware, so it runs for the actuator's own directly-registered routes exactly as it runs for your controllers:

```
<?php
declare(strict_types=1);

return [
    'security' => [
        'enabled' => true,
        'http' => [
            'enabled' => true,
            'rules' => [
                ['pattern' => 'actuator/health', 'access' => 'permitAll'],
                ['pattern' => 'actuator/info', 'access' => 'permitAll'],
                ['pattern' => 'actuator/*', 'access' => 'hasRole:ACTUATOR'],
            ],
        ],
    ],
];
```

`firefly/actuator`'s own `composer.json` has no dependency on `firefly/security` at all — securing the actuator this way is **pure configuration**, drawing on the same deny-by-default URL DSL Chapter 10 already taught you, with no new mechanism to learn.

⚠ **EVERY OTHER MANAGEMENT ENDPOINT IS A STANDALONE ENDPOINT, NOT AN** **/** **INFO** **SUB-KEY**

`/actuator/info` genuinely has exactly three fragments — `runtime` (from `RuntimeInfoContributor`, registered by default, which is why a freshly generated application already answers something: PHP version/SAPI/OPcache, Laravel version, LaraFly version, current and peak memory; turn it off with `firefly.management.info.runtime.enabled = false`), `app` (from `AppInfoContributor`, read from `firefly.management.info.app.*`) and `build` (from `BuildInfoContributor`, reading a JSON file at `firefly.management.info.build.path`). `env`, `beans`, `conditions`, `mappings`, `loggers`, `scheduledtasks`, `configprops`, and `caches` are each their own `ActuatorEndpoint`, mounted at their own `/actuator/{id}` — not nested under `/info`. `firefly:about` (Chapter 13) renders several of these together at the terminal, which is a convenience of that one command, not evidence they share a route.

The HAL index and the endpoint contract

Every framework endpoint — health, info, metrics, or your own — implements the same three-method contract:

```
interface ActuatorEndpoint
{
    /** The stable id under the base path, e.g. 'health' → /actuator/health. */
    public function endpointId(): string;

    /** Per-endpoint kill switch; ActuatorDispatchAction also honours
    firefly.management.endpoint.{id}.enabled. */
    public function enabled(): bool;

    public function handle(EndpointRequest $request): ?EndpointResponse;
}
```

`ActuatorRouteRegistrar`, a `BootPass` at `WiringPasses` phase, discovers every `#[Component]` implementing this interface, resolves each one exactly once, and mounts exactly **two** native Illuminate routes — a `GET {base}` index and a catch-all `GET|POST {base}/{path}` dispatcher — under the configured base path, so nothing about the mechanism collides with your own application routes. The index itself, `ActuatorIndexAction`, renders a HAL-style `_links` map filtered to whatever is both enabled **and** exposed:

```
final class ActuatorIndexAction
{
    public function __construct(
        private readonly ActuatorRegistry $registry,
        private readonly ExposureModel $exposure,
        private readonly Config $config,
        // ...
    ) {}

    public function __invoke(Request $request): Response
    {
        // ...
        $links = ['self' => ['href' => $base]];
        foreach ($this->registry->all() as $id => $endpoint) {
            if (! $endpoint->enabled() || ! $this->config->bool("firefly.management.endpoint.{ $id }.enabled", true) || ! $this->exposure->isExposed($id)) {
                continue;
            }
            $links[$id] = ['href' => $base.'/' . $id];
        }

        return new Response(
            (string) json_encode(['_links' => $links], JSON_THROW_ON_ERROR | JSON_UNESCAPED_SLASHES),
            200,
            ['Content-Type' => 'application/json'],
        );
    }
}
```

```
    }
}
```

`GET /actuator` on a default install therefore lists only `self`, `health`, and `info` — exactly the two ids `include` exposes by default — and grows only as far as your own `.include` configuration lets it.

Metrics: `MeterRegistry`, `MetricsRecorder`, and idempotent registration

`firefly/observability` is a second, independent opt-in package layered on top of the actuator — it ships its own `MeterRegistry` (the read-facing factory a consumer queries) and a narrower `MetricsRecorder` (the write-facing port instrumentation actually depends on, so a filter or a metrics recorder never needs the full registry API):

```
interface MeterRegistry
{
    /** @param array<string, string> $tags */
    public function counter(string $name, array $tags = []): Counter;

    /** @param array<string, string> $tags */
    public function timer(string $name, array $tags = []): Timer;

    /**
     * @param array<string, string> $tags
     * @param callable(): float $supplier
     */
    public function gauge(string $name, array $tags, callable $supplier): Gauge;

    /** @return list<Meter> */
```

```

    public function meters(): array;
}

```

```

interface MetricsRecorder
{
    /** @param array<string, string> $tags */
    public function increment(string $name, array $tags = [], float $amount = 1.0): void;

    /** @param array<string, string> $tags */
    public function record(string $name, array $tags = [], float $seconds = 0.0): void;

    /** @param array<string, string> $tags */
    public function setGauge(string $name, array $tags, float $value): void;
}

```

`SimpleMeterRegistry` implements both ports as a first-party, pure-PHP, in-memory store — no `ext-prometheus`, no OpenTelemetry SDK. Registration is idempotent, keyed by `type|name|sorted-tags`, so calling `counter('cQRS_commands_seconds', [...])` twice with the same name and tag set returns the very same `Counter` instance both times. It also fails loud, on purpose, if you try to register one metric name under two different types:

```

final class SimpleMeterRegistry implements MeterRegistry, MetricsRecorder
{
    // ...
    private function guardType(string $name, MeterType $type): void
    {
        $existing = $this->namesToTypes[$name] ?? null;
        if ($existing !== null && $existing !== $type) {
            throw new InvalidArgumentException(
                "Metric '{$name}' already registered as {$existing->value}; cannot re-
register as {$type->value}."
            );
        }
    }
}

```



```

    $this->namesToTypes[$name] = $type;
}
// ...
}

```

This is not pedantry: Prometheus's own text format scopes exactly one `# TYPE` declaration per metric name, so a name registered as both a counter and a gauge somewhere in your codebase would make `PrometheusTextFormat` emit two conflicting `# TYPE` lines for the same name — invalid exposition that a scraper would reject. `SimpleMeterRegistry` turns that mistake into an immediate `InvalidArgumentException` at the call site responsible, rather than a scrape failure discovered later in production.

Exposition: the locale-safe Prometheus exporter

`PrometheusTextFormat` renders every registered meter as format-0.0.4 text at `/actuator/prometheus`. The interesting part is a six-line private method most teams get wrong the first time they write one:

```

final class PrometheusTextFormat
{
    // ...
    private function value(float $value): string
    {
        if (is_nan($value)) {
            return 'NaN';
        }
        if (is_infinite($value)) {
            return $value > 0 ? '+Inf' : '-Inf';
        }
        if ($value == floor($value) && abs($value) < 1.0e15) {

```

```

    return (string) (int) $value;
}

// number_format() (unlike sprintf('%f')) is locale-INDEPENDENT here: the decimal
point and thousands
// separator are passed explicitly as arguments, so LC_NUMERIC (e.g. a comma-decimal
locale such as
// de_DE) cannot leak a ',' into the exposition and produce unscreapeable Prometheus
output.
return rtrim(rtrim(number_format($value, 10, '.', ''), '0'), '.');
}
// ...
}

```

`sprintf('%f', $value)` is the instinctive way to format a float in PHP, and it is *locale-dependent*: on a host whose `LC_NUMERIC` is set to a comma-decimal locale like `de_DE`, `sprintf('%f', 1.5)` renders `"1,500000"` — a comma the Prometheus text format's own grammar does not permit inside a sample value, silently producing exposition a scraper cannot parse. `number_format($value, 10, '.', '')` takes the decimal point and thousands separator as **explicit arguments**, so the process's ambient locale can never leak into the output — the same class of "which layer controls formatting" question the rest of this book keeps circling back to, here answered once, in one function, rather than at every call site that happens to print a float.

`/actuator/metrics` (`MetricsEndpoint`) exposes the same registry as Micrometer-flavored JSON instead — no sub-path lists every registered name; a specific name resolves to its measurements and available tags, or a plain `404` if the registry has never seen it.

Histogram buckets: turning a summary into something you can take a quantile of

A `Timer` exposes as a Prometheus **summary** by default — `_count` and `_sum`, and nothing else. That is enough for an average and useless for a p99, because an average latency is the one number that hides every bad experience the service delivered.

`histogram_quantile()` needs buckets, and buckets are a per-meter decision:

```
public const string BUCKETS_KEY = 'firefly.observability.metrics.distribution.buckets';

public const string PER_METER_KEY = 'firefly.observability.metrics.distribution.per-meter';
// ...
/** @return list<float> ascending upper bounds in seconds; [] when the meter is a summary */
public function bucketsFor(string $meterName): array
{
    return $this->perMeter[$meterName] ?? $this->buckets;
}
```

`firefly.observability.metrics.distribution.buckets` is the global list of upper bounds in seconds; `distribution.per-meter.<name>` overrides it for one meter, and an **empty** per-meter list turns that one meter back into a summary. A timer that has buckets exposes as a Prometheus **histogram** — `_bucket{le=...}`, `_count`, `_sum` — and one that does not stays a summary.

Two design decisions are visible in that six-line method. Bounds are keyed by meter **name**, never by tag set, because a Prometheus family has exactly one type and one bucket layout; a per-tag layout would produce a family that no scraper could make sense of. And the registries ask for the buckets when a timer is **created**, not on every `record()` — the layout is fixed for the meter's life.

The default is **no buckets**, and not because buckets are dangerous. Turning a summary into a histogram changes a scrape's `# TYPE` line and adds a series per bound, and an upgrade has no business doing that to a running dashboard without being asked. The list to start from is the one Prometheus clients ship:

```
[0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1, 2.5, 5, 10]
```

Every bound is validated at configuration time: a non-numeric entry, a zero, a negative, an infinity or a NaN is a `ConfigurationException` naming the key. `+Inf` is implied and must not be listed.

Auto-instrumentation: `MetricsFilter` and bounded cardinality

Once `firefly/observability` is installed and enabled, every HTTP request is timed automatically by `MetricsFilter`, a `#[Component]` `WebFilter` discovered by `firefly/web`'s filter chain with no wiring of your own:

```
#[Component]
#[Order(-100)]
#[ConditionalOnProperty(name: 'firefly.observability.metrics.enabled', havingValue: 'true',
matchIfMissing: true)]
#[Lazy]
final class MetricsFilter extends OncePerRequestFilter
{
    // ...

    private const string UNMATCHED_ROUTE_URI = 'UNKNOWN';

    public function __construct(private readonly MetricsRecorder $recorder) {}
}
```

```

    protected function doFilter(Request $request, Closure $next): mixed
    {
        $start = microtime(true);

        try {
            $response = $next($request);
            $status = $response instanceof Response ? $response->getStatusCode() : 200;
            $this->record($request, $start, $status, $this->outcome($status), 'none');
        }

        return $response;
    } catch (Throwable $e) {
        $this->record($request, $start, 500, 'SERVER_ERROR', $e::class);
    }

    throw $e;
}

private function record(Request $request, float $start, int $status, string $outcome,
string $exception): void
{
    $this->recorder->record('http_server_requests_seconds', [
        'method' => $request->getMethod(),
        'uri' => $request->route() !== null ? '/' . ltrim((string) $request->route()-
>uri(), '/') : self::UNMATCHED_ROUTE_URI,
        'status' => (string) $status,
        'outcome' => $outcome,
        // ...
    ], microtime(true) - $start);
}
// ...
}

```

Two details make this production-safe rather than merely convenient. `#[Order(-100)]` puts it outermost among discovered filters, so it times the *entire* inner chain — security, validation, your controller — not just a slice of it. And the `uri` tag is always the matched route's **template** (`/orders/{id}`), never the raw request path — an

unmatched request (any `404`) is tagged with the fixed sentinel `'UNKNOWN'` rather than the attacker- or crawler-controlled path itself. Tagging by raw path would let anyone generate an unbounded number of distinct metric label combinations just by hitting made-up URLs; under Octane, where the registry is a process-lifetime singleton rather than a fresh one per request, that is not a cosmetic wart — it is an unbounded memory-growth vector. On a thrown request, the filter records the outcome and **rethrows** rather than swallowing, so `ProblemDetailsRenderer` still gets to render the error exactly as it would without the filter installed.

`MeterBindingsPass`, a companion `BootPass`, registers a handful of pull-based gauges the same way: `process_resident_memory_bytes / php_memory_peak_bytes` for the running process, and one `resilience_circuit_breaker_state{name}` gauge per configured circuit breaker (`closed=0`, `open=1`, `half_open=2`) — each sampled *live* at scrape time via a closure, so a fresh `/actuator/prometheus` request always reflects the breaker's current state rather than a snapshot from boot.

The CQRS metrics seam: `#[Order(500)]`, one more time

Chapter 7 left you with `NoOpCqrsMetrics` bound behind `#[ConditionalOnMissingBean(CqrsMetrics::class)]`, and a docblock promising a real recorder would drop in later. `firefly/observability` is that drop-in, and the winning mechanism is *exactly* the precedence trick Chapter 10 used for `SecurityCommandAuthorizer`:

```
#[Configuration]
#[Order(500)]
final class ObservabilityAutoConfiguration
{
    // ...
```

```

    #[Bean]
    #[ConditionalOnProperty(name: 'firefly.observability.metrics.enabled', havingValue:
'true', matchIfMissing: true)]
    #[ConditionalOnMissingBean(MeterRegistry::class)]
    public function meterRegistry(Container $container, Config $config): MeterRegistry
    {
        $distribution = DistributionStatisticConfig::fromConfig($config);

        $store = $config->string('firefly.observability.metrics.store', '');
        if ($store === '' || ! $container->bound('cache')) {
            return new SimpleMeterRegistry($distribution);
        }

        /** @var Factory $factory */
        $factory = $container->make('cache');
        $ttl = $config->int('firefly.observability.metrics.ttl', 0);

        return new CacheMeterRegistry($factory->store($store), 'firefly:metrics:', $ttl >
0 ? $ttl : null, $distribution);
    }
    // ...
    #[Bean]
    #[ConditionalOnMissingBean(CqrsMetrics::class)]
    #[ConditionalOnProperty(name: 'firefly.observability.metrics.enabled', havingValue:
'true', matchIfMissing: true)]
    public function cqrsMetrics(MetricsRecorder $recorder): CqrsMetrics
    {
        return new MeterRegistryCqrsMetrics($recorder);
    }
    // ...
}

```

`meterRegistry()` is shown beside it because the two beans answer different questions with the same machinery. It picks the registry: `SimpleMeterRegistry` by default, in process memory; `CacheMeterRegistry` over a named cache store when `firefly.observability.metrics.store` names one, so counters accumulate across

PHP-FPM workers rather than each scrape seeing only its own request. Both are handed the same `DistributionStatisticConfig`, so a timer's histogram buckets do not change when the backing store does.

`cqrsMetrics()` is the one this section is about, and

`ObservabilityAutoConfiguration` is `#[Order(500)]`, strictly below

`CqrsAutoConfiguration`'s `#[Order(1000)]`. The incremental condition pass evaluates low-`#[Order]`-first and registers survivors immediately, so this class's `cqrsMetrics()` bean registers **first**; by the time `CqrsAutoConfiguration` evaluates its own `#[ConditionalOnMissingBean(CqrsMetrics::class)]`, a bean is already bound, and the `NoOp` default backs off. No line inside `firefly/cqrs` changes — the win is pure auto-configuration ordering, the same shape you have now seen twice.

`MeterRegistryCqrsMetrics` itself is the concrete recorder that ordering installs — a small, direct implementation of the `CqrsMetrics` port, recording each command or query as a timer tagged by message type and outcome:

```
final class MeterRegistryCqrsMetrics implements CqrsMetrics
{
    public function __construct(private readonly MetricsRecorder $recorder) {}

    public function recordCommandSuccess(object $command, float $seconds): void
    {
        $this->recorder->record('cqrs_commands_seconds', ['type' => $this->type($command),
        'outcome' => 'success'], $seconds);
    }

    public function recordCommandFailure(object $command, float $seconds): void
    {
        $this->recorder->record('cqrs_commands_seconds', ['type' => $this->type($command),
        'outcome' => 'failure'], $seconds);
    }

    public function recordQuerySuccess(object $query, float $seconds): void
```



```

    {
        $this->recorder->record('cQRS_queries_seconds', ['type' => $this->type($query),
'outcome' => 'success'], $seconds);
    }

    public function recordQueryFailure(object $query, float $seconds): void
    {
        $this->recorder->record('cQRS_queries_seconds', ['type' => $this->type($query),
'outcome' => 'failure'], $seconds);
    }

    private function type(object $message): string
    {
        $class = $message::class;

        return ($pos = strpos($class, '\\')) !== false ? substr($class, $pos + 1) : $class;
    }
}

```

Every observability bean gates on the **same** property,

`firefly.observability.metrics.enabled` — not on `MeterRegistry`'s presence.

That is a deliberate choice, not an oversight: a

`#[ConditionalOnBean(MeterRegistry::class)]` on `cQRSMetrics()` or `MetricsFilter` would be evaluated *before* `ObservabilityAutoConfiguration`'s own `#[Order(500)]` has registered `MeterRegistry` at all, wrongly dropping every downstream consumer even with metrics genuinely enabled — an ordering trap, not a logic bug. Gating everything on one flag instead makes survival order-independent: flip `firefly.observability.metrics.enabled` and the registry, the CQRS recorder, the filter, and both endpoints all survive — or all back off — together.

Finally, a `Tracer` port rounds out the package — and it is no longer the placeholder it started as. The interface is three methods: `startSpan(string $name, SpanKind $kind = SpanKind::Internal, array $attributes = [], ?SpanContext $parent = null):`

`Span`, `currentSpan(): ?Span`, and the `trace(string $name, callable $callback, SpanKind $kind = SpanKind::Internal, array $attributes = []): mixed` convenience that starts a span, runs the callback, records a throwable as an `ERROR` status plus an `exception` event, rethrows, and ends. `NoOpTracer` is still the shipped default — non-recording spans, an invalid `SpanContext`, `currentSpan()` of `null` — and an OpenTelemetry adapter now binds itself over it at `#[Order(400)]`, ahead of `ObservabilityAutoConfiguration`'s `#[Order(500)]`, whenever the SDK is installed and `firefly.observability.tracing.enabled` is on. The identical "port now, adapter later" shape you have just seen for `CqrsMetrics`; what changed is that the adapter arrived, and the instrumentation on both sides of it with it.

Tracing: one `traceparent`, five boundaries

What the port buys is one id that survives every boundary a request crosses.

`W3CTraceContextPropagator` speaks `W3C Trace Context` over a plain header map, with no SDK involved — `extract(array $carrier): ?SpanContext` on the way in, `inject(SpanContext): array<string,string>` on the way out — so propagation works identically under an SDK-backed tracer and under a test one.

Five boundaries are instrumented, and the figure is all five at once. `TracingFilter` (`#[Order(-110)]`, outermost of the discovered filters) continues an inbound `traceparent` as a remote parent and opens a `SERVER` span, then publishes its ids to Laravel `Context` and to `Request::$attributes` as `firefly.trace_id` and `firefly.span_id`. `TracerCqrsTracing` opens an `INTERNAL` span around every command and every query — nothing to carry, since the bus is in-process, so the span simply nests under whatever is current. `TracerEdaTracing` opens a `PRODUCER` span named `publish <destination>` on the way out, writing the `traceparent` into the envelope headers, and a `CONSUMER` span named `process <destination>` on delivery,

reading it back — which is what joins a queue worker's span to the request that published. Six callers reach that seam today: `InMemoryEventBus`, `QueueEventBus`, the shared `SubscriberRegistrySink` every broker consumer delivers through, and — since 26.09.4 — the three broker publishers themselves, `RabbitMqEventPublisher`, `KafkaEventPublisher` and `PostgresEventPublisher`, each of which routes `publish()` through `tracePublish()` rather than building its envelope alone. A broker *publish* therefore opens a `PRODUCER` span and stamps a `traceparent` of the framework's own making. `firefly.eda.tracing.brokers.enabled` downgrades those three to the no-op without touching the in-memory and queue buses, which is the one thing that can still leave a broker record unstamped. And `HttpClientTracingMiddleware`, which `HttpClientTracingPass` installs on Laravel's `Http` factory at boot, opens a `CLIENT` span on every outbound call and injects the header onto the PSR-7 request.

Both seams are the `CqrsMetrics` shape you have just read: `CqrsTracing` lives in `firefly/cqrs` and `EdaTracing` in `firefly/eda`, each with a no-op default behind `#[ConditionalOnMissingBean]`, and observability's `#[Order(500)]` auto-configuration registers the real implementation first. Neither `firefly/cqrs` nor `firefly/eda` depends on `firefly/observability` — check their `composer.json` files if you want to see it.

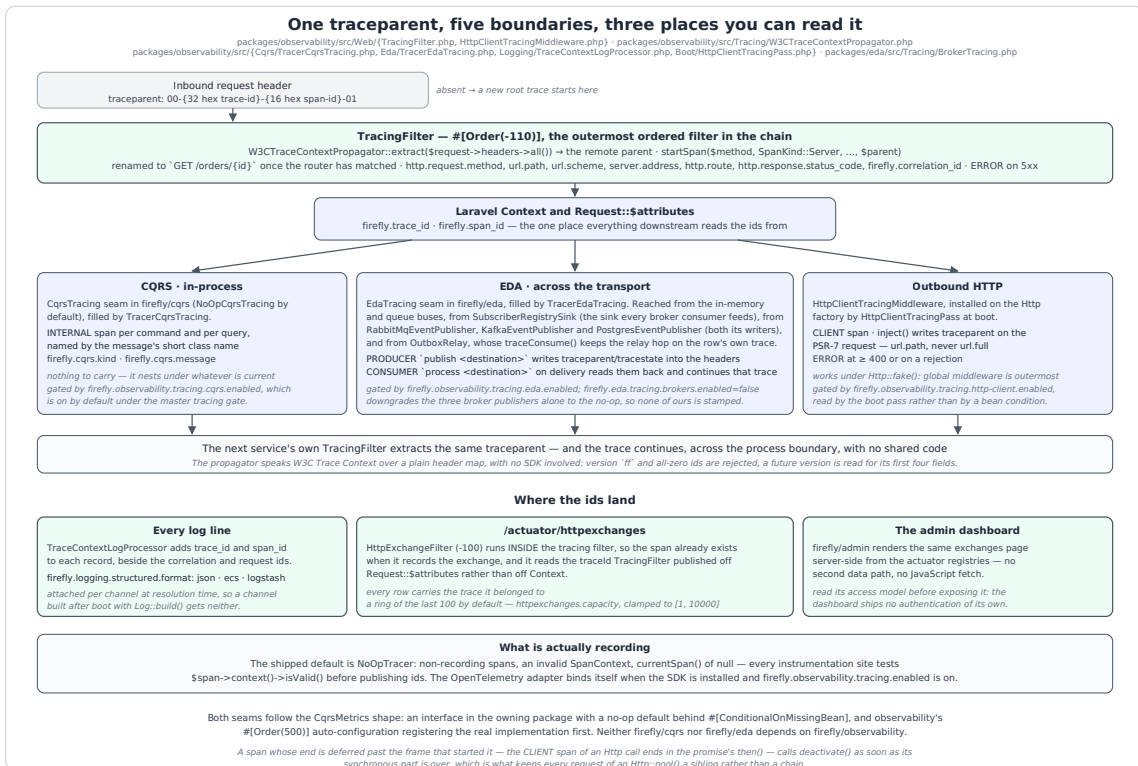


Figure 11.1 — One traceparent enters at TracingFilter, is published to Laravel Context as `firefly.trace_id` and `firefly.span_id`, and rides out again through the CQRS bus, the EDA envelope and the outbound Http client — landing on every log line, on `/actuator/httpexchanges` and on the dashboard.

The three places it lands are the point of the whole exercise.

`TraceContextLogProcessor` stamps `trace_id` and `span_id` onto every log record, beside the correlation and request ids. `HttpExchangeFilter` — `#[Order(-100)]`, so it runs *inside* the tracing filter and the span already exists when it records — reads the trace id back off `Request::$attributes`, which is why every row of `/actuator/httpexchanges` carries the trace it belonged to. And the dashboard renders that same page server-side from the same registries, with no second data path.

Logs a machine can read

The three ids the last section produced are only useful if something can find them. A `LineFormatter` log line with a trace id somewhere inside its message text is not searchable; it is a haystack with a needle in it.

So `firefly.logging.structured.format` — Spring Boot 3.4's `logging.structured.format`, on Laravel's channels — names a formatter, and there are three:

```
public const string FORMAT_KEY = 'firefly.logging.structured.format';

public const string CHANNELS_KEY = 'firefly.logging.structured.channels';
```

Value	What each line becomes
<code>''</code> (the default)	off — Laravel's own formatting, untouched
<code>json</code>	Monolog's <code>JsonFormatter</code>
<code>ecs</code>	Elastic Common Schema 8, through the framework's own <code>EcsFormatter</code>
<code>logstash</code>	Monolog's <code>LogstashFormatter</code> , with the framework's fields under <code>fields</code>

Anything else is a `ConfigurationException` **at boot** — `LogChannelWiringPass` asks for the format from `Application::boot()`, before it touches the log service, which is what makes a typo a startup failure rather than a silent fallback to plain text.

The ECS shape is worth seeing, because it is the one most likely to land straight in an existing stack:

```

/**
 * Elastic Common Schema 8 – the JSON shape Elastic's own ecs-logging libraries emit and
 * Spring Boot's
 * `logging.structured.format=ecs` produces, first-party so the framework adds no logging
 * dependency:
 *
 * { "@timestamp": "2026-09-20T10:11:12.345678+00:00", "log.level": "info", "message": "Order 42
 * shipped",
 *   "ecs.version": "8.11.0", "log": { "logger": "stack", "service":
 * {"name": "ledger", "environment": "production"},
 *   "trace": { "id": "4bf9...", "span": { "id": "00f0...", "labels":
 * {"correlation_id": "...", "request_id": "..."},
 *   "error": { "type": "RuntimeException", "message": "boom", "stack_trace": "#0 ..."},
 *   "context": { "order": 42, "extra": { "memory": 12 } }
 * }
 * // ...
 */
final class EcsFormatter extends NormalizerFormatter
{
    public const string ECS_VERSION = '8.11.0';

```

The four framework ids and the two service fields are **lifted** into their ECS homes; everything a processor or a caller added stays nested under `extra / context` rather than merged at the top level, so an application's own `message` or `error` context key can never collide with an ECS field.

Where the ids come from

```

/**
 // ...
 * trace_id / span_id the CURRENT span's ids when a tracer has one (a log line
 * written inside a
 * command handler or an event listener names THAT span), else
 * the request's ids
 * TracingFilter published in Context; absent when tracing is
 * off.

```

```
// ...
* request_id Context firefly.request_id (RequestContextFilter).
// ...
*/
final class TraceContextLogProcessor
{
    public const string TRACE_ID = 'trace_id';
    // ...
    public function __invoke(LogRecord $record): LogRecord
```

`correlation_id` is the third: the id `problem+json` and the `X-Correlation-Id` header already carry, read from the Laravel `Context` key `CorrelationIdFilter` seeds. So one log line names the span that wrote it, the request it belonged to, and the correlation id a client can quote back at you.

Read the first row twice. A log line written inside a command handler carries **that handler's span id**, not the request's — which is exactly what makes a log search useful, because it tells you which of a request's several spans wrote the line. The processor falls back to the request's ids only when no span is current.

There is one implementation detail with a reason behind it: the tracer is resolved through a closure on **every** record rather than captured at construction. The processor is pushed when the `log` service is first resolved, and that can happen before the container has bound a `Tracer` at all — capturing the null would mean no trace ids for the rest of the process.

Which channels, and what the formatter does not touch

`firefly.logging.structured.channels` lists channel names; empty — the default — means the default channel. A `stack` channel's Monolog logger holds its *members'* handler instances, so formatting the stack formats the members too, and listing both is harmless because applying is idempotent.

What never changes is the handlers themselves. A `daily` file stays a daily file, a `stack` keeps its members, a `slack` channel keeps posting to Slack; **only what a line looks like changes**. Nothing is replaced, added or removed.

A channel name that does not exist under `logging.channels` is refused, and the reason is a good illustration of why this framework checks things:

`LogManager::channel()` never throws for an unknown name — it catches its own *"Log [x] is not defined."* and hands back a throw-away emergency logger. Without the check, a typo would put the processors and the formatter on an object nobody ever writes to, while the real channel quietly kept writing plain text with no ids in it at all.

⚠ A CHANNEL BUILT AFTER BOOT GETS NEITHER

The processors and the formatter are attached **per channel, at resolution time**. A channel created later through `Log::build()` has neither the ids nor the structured formatter. If you build channels dynamically, apply the formatter yourself — or declare the channel in `logging.channels` and name it in `firefly.logging.structured.channels`, which is the path everything else takes.

`gelf` and `logfmt` are not offered. Monolog has a `GelfMessageFormatter`, and a channel can still choose it through Laravel's own `formatter` key.

The admin dashboard: `firefly/admin`

Everything so far in this chapter is JSON, and JSON is the right shape for a load balancer, a Kubernetes probe and a Prometheus scraper. It is not the right shape for a person at 3am who wants to know whether this process compiled its manifests, which auto-configuration backed off, and what `firefly.data.*` actually resolved to.

`firefly/admin` is the package for that person: a server-rendered browser dashboard over the very same actuator endpoints, in the spirit of Spring Boot Admin. It is included

in `fireflyframework/laraflly`, so a skeleton project already has it — and, as with `firefly/actuator`, having it installed is not the same as having it switched on. No separate package is needed.

Then open `/firefly`. There is no npm step at install time and no CDN at request time — the views are plain Blade with inline CSS and system fonts, because a Composer package cannot assume npm has run, and a dashboard that needs the network is useless in exactly the isolated environments where you most want to look at one.

Most pages are a view over one endpoint's payload; four read the container instead. The menu groups them the way an operator thinks rather than the way the packages are laid out — what is it doing right now, what did it wire at boot, what is its data, and how is it configured — because a flat list of seventeen links is a worse menu than four short ones:

Group	Page	Reads	Answers
Runtime	Overview	several	Is it healthy, what is it doing, and what did it wire?
Runtime	Health	health	Every indicator this process registered, with its own status and details
Runtime	Metrics	metrics	Counters, timers and gauges, with their current measurements
Runtime	HTTP traffic	httpexchanges	The most recent requests this application served
Wiring	Beans	beans	Every bean the container registered, with the stereotype that declared it
Wiring	Bean explorer	beans	How your beans depend on one another, resolved through the interfaces they are wired by

Group	Page	Reads	Answers
Wiring	Conditions	<code>conditions</code>	Which auto-configurations applied, and which backed off because you supplied your own
Wiring	Routes	<code>mappings</code>	The compiled route table the dispatcher serves from
Wiring	Scheduled	<code>scheduledtasks</code>	Methods registered by <code>#[Scheduled]</code> , with the cron or interval that drives them
Configuration	Environment	<code>env</code>	Resolved <code>firefly.*</code> configuration, with secrets masked
Configuration	Config properties	<code>configprops</code>	Every <code>#[ConfigProperties]</code> DTO the application bound, with the values it resolved
Configuration	Caches	<code>caches</code>	The cache stores this application has configured

Group	Page	Reads	Answers
Configuration	Loggers	loggers	Log channels and their levels, with a control to change one

A page whose endpoint is not registered in *this* process — or is switched off — is **hidden from the menu** rather than offered as a link that lands on an apology. That matters because the actuator's endpoints are conditional: `metrics` disappears when `firefly.observability.metrics.enabled` is false, and several others exist only if the package that contributes them is installed. The menu has to be built from what this process actually registered, so it is.

It reads endpoints in-process, not over HTTP

The dashboard holds the `ActuatorRegistry` and invokes each `ActuatorEndpoint` bean directly:

```
final readonly class AdminEndpointReader
{
    public function __construct(
        private ActuatorRegistry $registry,
        private Config $config,
        private ?Container $container = null,
    ) {}
    // ...
    public function available(): array
    {
        $ids = [];
        foreach ($this->registry->all() as $id => $endpoint) {
            if ($endpoint->enabled() && $this->config->bool("firefly.management.endpoint.
{$id}.enabled", true)) {
```

```

        $ids[] = $id;
    }
}

return $ids;
}
// ...
public function read(string $id, array $subPath = [], array $query = []): ?array
{
    $endpoint = $this->registry->get($id);
    if ($endpoint === null || !$this->has($id)) {
        return null;
    }

    try {
        $response = $endpoint->handle(new EndpointRequest('GET', $subPath, $query));
    } catch (Throwable) {
        return null;
    }

    return $response === null || is_string($response->body) ? null : $response->body;
}
// ...
}

```

Look at what is **not** in that method: any mention of `ExposureModel`. That is the single most important thing to understand about this package, and it is deliberate.

`firefly.management.endpoints.web.exposure.include` defaults to `health,info`, so fetching `/actuator/beans` or `/actuator/env` over HTTP would 404 — as the whole first half of this chapter insisted it should. The dashboard needs none of that. It renders what the process already knows, in-process, so **it shows you pages the HTTP surface deliberately does not expose**, and the JSON surface stays secure-by-default. Exposing `beans`, `conditions` and `env` to every anonymous caller just so a browser could read them would be exactly the wrong trade.

The per-endpoint kill switch *is* honoured, and the asymmetry is the point:

`firefly.management.endpoint.{id}.enabled` means "this endpoint is off", which is a statement about the endpoint itself; `exposure.include` means "this endpoint is unpublished", which is a statement about the HTTP surface. The dashboard is not the HTTP surface.

A throwing endpoint is caught and reported as `null` rather than allowed to take the page down with it — the same fail-safe discipline `HealthEndpoint::readFailSafe()` applies to indicators, for the same reason: one broken contributor should degrade its own panel, not the dashboard.



HEALTH DETAILS ARE READ FROM THE CONTRIBUTOR REGISTRY, NOT THROUGH THE ENDPOINT

`show-details` defaults to `never`, and that default is right — it stops an anonymous HTTP caller learning your database host from a failed connection. Applying that HTTP disclosure policy to the dashboard, though, produced a Health panel whose entire content was an apology telling the operator to go and change a config key. The dashboard reads `HealthContributorRegistry` directly instead, calling each indicator in isolation so one that throws is reported DOWN with its reason and nothing else is affected.

The bean explorer

`/firefly/graph` now opens with search, entry points, heavily depended-on beans, module coupling and wiring cycles. Select `?bean=<qualified identity>` to explore a bounded neighbourhood; `?module=<namespace>` opens that module's beans and boundary relations; `?q=<term>` searches the complete catalogue. `/firefly/beans` includes components, factory products and bound configuration DTOs and defaults to dependent count descending.

The selected bean sits between its dependents on the left and dependencies on the right. Positions are computed in PHP: 176×38-pixel boxes at 204×46-pixel pitch, with at most 16 rows per column and 72 nodes by default. Two hops occupy 992 pixels horizontally and at most 728 vertically. Headings scroll with the content. Native links, GET forms, pagination and visible keyboard focus work without JavaScript; the graph has no camera or fit operation. Module text and sigils complement decorative colours.

Frontier candidates share each column round robin. Overflow links open the source bean's complete direct-neighbour tables, with interface indirection and injection/production labels. Shortest entry-point chains explain reachability. The details preserve all positive and negative conditions, including those on the producing configuration; a missing Conditions endpoint is tolerated.

Module coupling reports weight (resolved relations), distinct target beans, distinct interfaces and concrete relations. Production edges are excluded unless explicitly included. Module lists remain paginated above the map budget. The module page marks exclusive reachability and beans with no dependents. These are catalogue observations, not proof that removing a module is safe.

Iterative Tarjan SCC analysis reports all cycle members, including self-injection, without recursive walks. Since the wiring graph includes factory production, an SCC is not necessarily a runtime constructor cycle. Competing factory methods keep separate identities, even across configuration classes. The projection lacks primary/qualifier metadata, so ambiguous targets remain unresolved rather than guessing the container winner; factory scope is reported as unknown. Explicitly unbound configuration DTOs are excluded. No bean is instantiated to inspect it.

Migration: `firefly.admin.graph.max-nodes` still parses into `AdminSettings::$graphMaxNodes` (default 220), but no longer controls rendering. In particular, **0 no longer forces a list**. Use the complete catalogue or relation tables for a tabular view.

The new settings are `firefly.admin.graph.focus.depth` (2, bounded 1–4), `focus.max-rows` (16, 4–60), `focus.max-nodes` (72, 8–300), `focus.max-paths` (3, 0–10), `focus.page-size` (50, 10–500), `firefly.admin.graph.starters` (12, 1–50), `firefly.admin.graph.modules.max-nodes` (40, 0–200; 0 disables the map), and `firefly.admin.beans.page-size` (50, 10–500). Relation/catalogue page sizes also obey shared table limits. All keys are documented in `skeleton/config/firefly.php`.

The access model is the whole security boundary

Because the dashboard bypasses exposure, its own URL is the only thing standing in front of `beans`, `env` and `conditions`. That is why it must not be on by default in production, and why the enable flag is written the way it is:

```
final readonly class AdminSettings
{
    // ...
    public function __construct(
        public bool $enabled,
        public string $basePath,
        public string $title,
        // ...
    ) {}

    public static function fromConfig(Config $config): self
    {
        $base = trim($config->string('firefly.admin.base-path', '/firefly'), '/');

        return new self(
            enabled: $config->bool('firefly.admin.enabled', $config->bool('app.debug',
false)),
            basePath: $base === '' ? 'firefly' : $base,
            title: $config->string('firefly.admin.title', $config->string('app.name',
'LaraFly')),
            // ...
        );
    }
}
```



```

    );
}
// ...
}

```

`firefly.admin.enabled` defaults to the value of `app.debug`. The reasoning is that an application already running with debug on is already serving stack traces to whoever asks and is a development environment by definition, so a dashboard there discloses nothing that was not already disclosed. An application with debug off has made the opposite statement about itself, and must opt in explicitly. Setting the key always wins over the debug default, in both directions — you can turn the dashboard off in a debug environment, and on in a production one.

⚠️ TURNING IT ON OUTSIDE DEBUG IS ONLY HALF THE JOB

`firefly.admin.enabled = true` with `app.debug = false` mounts a dashboard that renders your bean graph, your resolved configuration and your route table at a known URL, to anyone who can reach it. The dashboard ships **no authentication of its own** — it has no code dependency on `firefly/security` at all, exactly as `firefly/actuator` does not. An application that turns it on outside debug **must put the route behind its own auth middleware**.

Chapter 10's `HttpSecurity` rules do that as pure configuration, the same way this chapter already secured the actuator — here is a deployment that opts in and locks the route down in one file:

```

<?php
declare(strict_types=1);

return [
    'admin' => [

```

```

    'enabled' => true, // explicit: this deployment wants the dashboard with
app.debug off
    'base-path' => '/firefly',
  ],
  'security' => [
    'enabled' => true,
    'http' => [
      'enabled' => true,
      'rules' => [
        ['pattern' => 'firefly', 'access' => 'hasRole:ADMIN'],
        ['pattern' => 'firefly/*', 'access' => 'hasRole:ADMIN'],
      ],
    ],
  ],
];

```

`AdminRouteRegistrar` mounts the two routes — an index and a `GET|POST {base}/{page}` catch-all — as a `BootPass` at `WiringPasses`, order **60**, one step after `ActuatorRouteRegistrar`'s 50, because it reads the registry that pass populates. They are registered natively on the Illuminate `Router` for the same reason the actuator's and the OpenAPI package's routes are: `firefly.admin.base-path` has to be settable per application, and an attribute route bakes its literal path into a compiled `RouteDescriptor`. When the dashboard is disabled the pass registers *nothing at all* — there is no route to guess at and no handler to reach.

It also backs off silently in one more case that is easy to miss. Blade is required to render the dashboard and is not a dependency of the package, so a JSON-only deployment with no view factory bound gets no routes rather than routes that would fatal on first request; the JSON actuator remains the management surface there.

⚠️ THREE THINGS THE DASHBOARD CAN ONLY SHOW YOU ABOUT THIS PROCESS

Under PHP-FPM every request is a different process, and three pages inherit that.

Changing a log level calls the same endpoint `POST /actuator/loggers/{name}` does, which mutates the current process's Monolog handlers — the next request is a different process, so change `logging.channels` for anything that must persist. **Metrics** are only as durable as the registry: the default `SimpleMeterRegistry` keeps meters in process memory, so the dashboard sees only its own request unless

`firefly.observability.metrics.store` points at a cache store. And **health details** stay hidden on the JSON `/actuator/health` response until `firefly.management.endpoint.health.show-details` is `always` — or is `when-authorized` and the `HealthDetailsAuthorizer` admits the caller — even though the dashboard's own Health page reads the indicators directly.

The data browser, and why it does not inherit that default

`firefly/admin` ships one more surface, and it is the only one in this chapter whose gate is written differently from every other gate you have seen. It is a Django-admin-style **database browser** over the data layer of Chapter 5 — listing, detail, search, sorting and paging over your own repositories — reached through a single `DataBrowser` that `DataBrowser::forContainer($container)` assembles from the application container.

It discovers what to browse the same way the rest of the dashboard discovers everything — from the compiled catalogue. **Every bean whose scan-time interface list contains `CrudRepository` is a browsable resource.** Nothing is registered, nothing is declared: a repository you write is browsable the moment the container has it, and one you delete stops being browsable without anyone editing a list. Each row of `BeansCatalog` already carries the full interface closure `ComponentScanner` recorded with `class_implements()`, so "is this bean a repository, and does it also page?" is two `in_array()` calls over data the process already holds — no reflection, and, decisively, no chance of offering a resource the container never registered.

Now the gate:

```
final readonly class DataBrowserSettings
{
    // ...
    public static function fromConfig(Config $config): self
    {
        $max = min(self::PAGE_SIZE_CEILING, max(1, $config->int('firefly.admin.data.max-
page-size', 200)));

        return new self(
            enabled: $config->bool('firefly.admin.data.enabled', false),
            writable: $config->bool('firefly.admin.data.writable', false),
            pageSize: min($max, max(1, $config->int('firefly.admin.data.page-size', 25))),
            maxPageSize: $max,
            excluded: self::csv($config->string('firefly.admin.data.exclude', '')),
            // ...
        );
    }
    // ...
    public function canWrite(): bool
    {
        return $this->enabled && $this->writable;
    }
    // ...
}
```

Look at the two defaults, and compare them with `AdminSettings`'s `$config->bool('app.debug', false)` a few pages up. The dashboard follows `app.debug`, and the argument for that was sound **for what the dashboard shows**: beans, conditions, mappings and resolved configuration are facts about the *application*, and an application already serving stack traces has already published facts of that kind.

This page shows facts about the application's **users**. That is a categorically bigger disclosure, and the mistakes that expose it are the ordinary ones that cost nothing today: a debug flag left on in a staging environment that shares a database with production, a `.env` copied to a box that was supposed to be internal, a laptop tunnelled for a demo. Each becomes a customer-record disclosure the moment a data browser is wired to `app.debug`. So the gate is separate, explicit and off — `app.debug` **cannot switch it on, and neither can** `firefly.admin.enabled`. All three must be true.

Writes then need a *second* key, and it is ineffective on its own. Reading the wrong row is a disclosure; deleting it is data loss with no undo, from a form, over a session that may be nothing more than "debug was on". Turning on the browser is a decision about **visibility**; turning on writes is a decision about **custody**. Collapse them into one key and the operator who wanted to look at a table has also armed the delete button.

⚠️ CREATE EXISTS FOR ELOQUENT, AND IS REFUSED FOR EVERYTHING ELSE

The browser had no `create()` at all for a while, and the argument was half right. A generic create form over an arbitrary entity is a promise it cannot keep, and Chapter 6 is why: **an aggregate's constructor is where its invariants live**. An `Order` that must have at least one line, a `Wallet` whose balance starts at zero in the currency it was opened in, a value object that rejects a malformed IBAN — a form built from a column list knows none of them. There are only two ways to build such a row: call the constructor, which needs arguments the form cannot supply in the right types or the right order; or write the columns straight to the table, which produces a row the domain model considers impossible. The second is what a "just insert the columns" implementation does, and it is *worse than having no button*, because it looks like it worked. **That case is still refused, by name.**

It was never true for an **Eloquent model**, which is constructed empty and filled by attribute — precisely what `update()` has always done to a row that exists. Create was refusing on a risk update was already taking, and the inconsistency cost every application a CRUD surface that stopped at RUD. So it is offered for an Eloquent-backed resource under the same two switches, with the identifier and any masked column *omitted from the form* rather than disabled in it: a field the browser would refuse to write should not appear to accept.

Two more decisions are worth carrying out of this section, because both are the sort of thing that reads as a detail and is not.

The identifier and any masked secret are refused as update targets — and refused twice, once so the view can render the field read-only and again in the write path, so a hand-crafted POST cannot reach what the form would not offer. Re-keying a row from a generic form is not an edit, it is a different row, and the foreign keys pointing at the old value do not follow. A secret's *displayed* value is `*****`, so round-tripping a rendered form would write the mask over the real credential — a data-loss bug the masking itself

created. Secrets are excluded from **search** for a related reason: a box that answers "yes, some row's `api_token` starts with `sk_live_9`" is an oracle an operator can walk one character at a time.

No error text the page renders is ever an exception message. Laravel's

`QueryException` stringifies the failing SQL *and its bindings* into `getMessage()`, so echoing it would publish the schema and the bound values — which on a search over a users table is the operator's own query, and on a detail lookup is a primary key. Every reason is a fixed sentence composed in the browser layer, plus at most the exception's class name; the message stays in the exception, where the log can have it. This is also why nothing in the layer throws at its caller: reads answer with a listing carrying a reason, writes with one of four outcomes (`Done`, `Refused`, `NotFound`, `Failed`), and a view rendering an admin page never has to be exception-safe to stay on its feet.

ⓘ THE LISTING PATH YOU GET DEPENDS ON THE INTERFACE YOU IMPLEMENTED

A `PagingAndSortingRepository` is paged **in the database**: the repository does the offset, the limit, the `ORDER BY` and the `COUNT`, and the cost is independent of table size. A plain `CrudRepository` cannot express any of that, so the browser calls `findAll()`, sorts and slices **in PHP**, and throws away all but 25 rows — which on ten thousand rows is a slow page and on ten million is an out-of-memory that kills the worker, on the *first* click. The interface has no limit, no offset and no count-with-predicate, so the honest options were "refuse to browse repositories that cannot page" or "browse them and say what it costs". LaraFly does the second, and this is the saying. Implement `PagingAndSortingRepository` on anything you intend to browse against a real table.

Walking the model: relations, filters, and a map

A listing you can only scroll is a table dump. Three things turn it into something you explore.

Relations are discovered by CALLING the methods that declare one, because that is the only way to learn which columns they join on — a method's name says nothing and its return type says only the kind. Which makes "what is safe to call" the load-bearing question, and the answer is the **declared return type**: only a public, non-static, no-argument method returning an Eloquent `Relation` subclass is ever called. A method announcing `: HasMany` is a relation definition by construction — the same signal Laravel's own `with()` validation and IDE tooling rely on — and an accessor cannot claim it without lying about its signature. The call executes no query; Eloquent defers until `get()`.

A `belongsTo` opens the one parent record; a `hasMany` opens the child listing **filtered to this row's key**, which is what the record page needs a filter for. A relation whose other end is not a browsable resource is still shown — it tells you the shape of the model — but is not linked, and that distinction lives in the model rather than the template so a view cannot mint a URL that 404s.

Filtering is eight comparisons over the columns the resource already publishes — `is`, `is not`, `contains`, `starts with`, `greater than`, `less than`, `is empty`, `is not empty` — expressed as a GET URL you can bookmark or paste into a ticket. Every comparison binds its value, including the `LIKE` ones, where the wildcards go around an *escaped* value rather than the value going into a pattern. A column the schema does not publish and an operator outside that set are **dropped** rather than passed to the driver: both arrive in a URL an operator can hand-edit, and a query reaching the driver with an arbitrary identifier in it is a column-name oracle at best. Conditions AND with each other and with the search box, so narrowing a relation's listing cannot escape it.

The same eight are implemented for the in-PHP fallback path, because a repository that cannot page must be filtered by the same rules as one that can. Two implementations of one predicate drift, and the drift shows up as a filter meaning different things on different resources.

`/firefly/data-map` draws it. Every browsable entity as a box with its columns, every foreign key as a labelled edge, boxes linking into their own records. A `hasMany` and the `belongsTo` facing it are *one* key seen from two ends, so each is drawn once — pointing from the table that holds the key to the table it references, which is also what the arrow means.

Two more pages, and the one that changes things

`/firefly/datasource` answers what a config dump cannot. Which database am I talking to (driver, host, database, with `password` masked by the same masker the `env` endpoint uses)? Is it *up* — one connection probed per page load, because opening a socket can hang against a firewalled host and a page that opened every configured connection would take the slowest one's timeout to render, on the page you opened *because* something is wrong. What does pooling mean here — PHP has no connection pool, and rather than invent a gauge the page reports PDO's `ATTR_PERSISTENT` for what it is, and says that under php-fpm the effective pool size is your worker count. And what did `#[Transactional]` compile to, which until now existed only as an artifact under `bootstrap/cache`.

`/firefly/settings` is the only page in the dashboard that *changes* the application rather than describing it, and it is gated accordingly:

`firefly.admin.settings.enabled` (off by default, unlike everything else here), `.writable` on top of it, and a third gate that is **not a configuration key** — in

production every write is refused whatever the other two say. That last one is deliberate. It is the difference between "we made it safe" and "we made it configurable to be safe", and only the first survives someone copying a `.env`.

It is a *feature switch*, not a remote configuration endpoint: the list is fixed and framework-owned, so a crafted POST naming `app.key` or a database host finds nothing to write. A change goes to one JSON file under `bootstrap/cache`, filtered on the way in as well as out, and merged over configuration in the provider's `register()` — not in a boot pass, and that distinction cost a debugging session. Every settings object in this framework is built once from config and held, so applying the overrides from the dashboard's own pass wrote the file and showed the new state on the page while `/openapi.json` kept answering 200. `register()` runs before any bean resolves, which is the only point at which the merge is true.

LARAVEL PARITY

Plain Laravel ships no health-check or metrics endpoint at all — most teams either hand-roll a `/health` route or reach for a third-party package, usually paired with the `ext-prometheus` extension. `firefly/actuator` and `firefly/observability` are first-party, dependency-light analogues of Spring Boot Actuator and Micrometer respectively: framework endpoints mounted on the same `Router` your app already uses, health checks that reuse Laravel's own `DB / Log / config` underneath, and a pure-PHP Prometheus exporter with no extension requirement. Both packages are opt-in Composer dependencies and both are secure-by-default — an app that adds `firefly/actuator` gets `health / info` and nothing else until it configures more. `firefly/admin` completes the set as the analogue of Spring Boot Admin, with the difference that it is not a separate monitoring application you deploy and register instances with: it is Blade views inside the application it reports on, which is why it can read the registry directly and why its access model matters as much as it does.

✓ What you learned

Concept	What it does
<code>HealthIndicator</code>	One-method SPI; a <code>#[Component]</code> bean discovered and aggregated automatically
<code>Health</code> / <code>Status</code>	Immutable reading + a severity-ordered enum; <code>DOWN/OUT_OF_SERVICE</code> both map to HTTP 503
<code>PingHealthIndicator</code> / <code>DiskSpaceHealthIndicator</code> / <code>DbHealthIndicator</code>	Always-up liveness probe; threshold-based disk check; on-by-default <code>SELECT 1</code> DB check that declines registration via <code>ConditionalHealthIndicator::available()</code> when no default connection is configured
<code>HealthEndpoint</code>	Aggregates to the most-severe status; a probe group is just a configured, named indicator subset — there is no separate liveness/readiness endpoint class
<code>ExposureModel</code>	<code>include</code> / <code>exclude</code> CSV gate; default <code>"health,info"</code> ; everything else is a plain 404 until exposed
<code>EnvEndpoint</code> / <code>SensitiveValueMasker</code>	One shared rule masks <code>password\ secret\ token\ key\ credential\ passwd\ authorization\ headers</code> keys with <code>*****</code> , independent of exposure — the key decides first, so a sensitive key masks its whole subtree
<code>MeterRegistry</code> / <code>MetricsRecorder</code>	Read/write metrics ports; <code>SimpleMeterRegistry</code> is idempotent per <code>type\ name\ tags</code> and fails loud on a type conflict

Concept	What it does
<code>PrometheusTextFormat</code>	Locale-safe exposition — <code>number_format()</code> , never <code>sprintf('%f')</code>
<code>DistributionStatisticConfig</code>	Histogram buckets per meter name , fixed when the timer is created; no buckets by default, because adding them changes a scrape's <code># TYPE</code> line
<code>firefly.logging.structured.format</code>	<code>json</code> , <code>ecs</code> or <code>logstash</code> on the channels you name — the handlers are never replaced, only what a line looks like changes
<code>TraceContextLogProcessor</code>	<code>trace_id</code> , <code>span_id</code> , <code>correlation_id</code> and <code>request_id</code> on every record; the current span's ids, not just the request's
<code>MetricsFilter</code>	<code>#[Order(-100)]</code> outermost timing filter; tags by route template , never raw path — bounded cardinality
<code>ObservabilityAutoConfiguration</code> <code>#[Order(500)]</code>	The same precedence trick as Chapter 10's security seam: registers <code>cqrsMetrics()</code> before <code>CqrsAutoConfiguration</code> evaluates its <code>#[ConditionalOnMissingBean]</code>
<code>firefly/admin</code>	A server-rendered Blade dashboard at <code>/firefly</code> ; a page whose endpoint is unregistered or switched off is hidden from the menu rather than linked

Concept	What it does
<code>AdminEndpointReader</code>	Invokes each <code>ActuatorEndpoint</code> in-process from <code>ActuatorRegistry</code> , bypassing <code>ExposureModel</code> — so the dashboard shows what the HTTP surface does not expose, and a throwing endpoint degrades one panel
<code>BeanGraph</code>	Turns the beans catalogue into a drawn dependency graph over three kinds of node — components, <code>#[Bean]</code> products and <code>#[ConfigProperties]</code> DTOs — with <code>injects</code> / <code>produces</code> edges resolved through an interface index (marked <code>via</code>), longest-path layering, cycles reported rather than hung on, and bounded hop exploration with iterative SCCs
<code>#[Bean]</code> products as nodes	A framework's wiring lives in factory methods, not constructors; with only declaring classes as nodes a stock skeleton drew one edge out of 42 beans
<code>ComponentDescriptor::\$dependencies</code>	The graph's edges, recorded by <code>ComponentScanner</code> at scan time — class and interface types only, because a scalar parameter is configuration, not wiring
<code>firefly.admin.enabled</code>	Defaults to <code>app.debug</code> ; an explicit value wins in both directions, and turning it on with debug off obliges you to put your own auth middleware in front of the route

Concept	What it does
<code>firefly.admin.data.enabled</code>	The data browser's own gate, defaulting to <code>false</code> — it does <i>not</i> follow <code>app.debug</code> or <code>firefly.admin.enabled</code> , because this page shows facts about the application's users rather than about the application
<code>RelationIntrospector</code>	Finds relations by CALLING only the methods whose declared return type is an Eloquent <code>Relation</code> , so a record links to what it references and the entity map has edges to draw
<code>DataFilter</code>	Eight comparisons over the columns the resource publishes, always bound, with an unknown column or operator dropped rather than passed to the driver
<code>/firefly/datasource</code>	Connections with secrets masked, one probed per load, PDO persistence reported for what it is, and the compiled <code>#[Transactional]</code> contract
<code>/firefly/settings</code>	The one page that changes the application: off by default, writable by a second key, and refused in production by a gate no key lifts
<code>firefly.admin.data.writable</code>	A second gate, also <code>false</code> and ineffective alone: visibility and custody are different decisions, and one key would arm the delete button for whoever wanted to look at a table

Concept	What it does
No <code>create()</code>	Permanent, not pending: an aggregate's invariants live in its constructor, and a form built from a column list cannot satisfy them — writing the columns anyway produces a row the domain considers impossible

Try it yourself

1. **Add a custom `HealthIndicator`**. Write a `#[Component]` implementing `HealthIndicator` that checks something specific to your own app (a feature flag, a queue depth, a cache connection) and confirm it appears in `GET /actuator/health`'s `components` map once `show-details` is set to `always`.
2. **Configure a real liveness/readiness split**. Add `firefly.management.endpoint.health.group.liveness.include = 'ping'` and `...readiness.include = 'ping,db'` (with the DB indicator enabled) to a scratch project's config, and confirm `GET /actuator/health/liveness` and `GET /actuator/health/readiness` diverge the moment you make the database unreachable.
3. **Watch the CQRS metrics seam win the race**. Use the included `firefly/observability` component in a scratch LaraFly project already using `firefly/cqrs`, send a handful of commands, and inspect `GET /actuator/prometheus` for `cqrs_commands_seconds` samples — then temporarily comment out `ObservabilityAutoConfiguration`'s `#[Order(500)]` attribute (reverting to the class default) and confirm whether the metric still appears, to see the ordering trick actually matter rather than just reading about it.

4. **Prove the dashboard's exposure bypass to yourself.** Enable the included `firefly/admin` component with `firefly.admin.enabled` in the sample, leave `firefly.management.endpoints.web.exposure.include` at its default, and confirm that `GET /actuator/beans` returns a `404` while `/firefly/beans` renders the full bean list in the same process. Then set `firefly.management.endpoint.beans.enabled` to `false` and confirm the Beans entry vanishes from the dashboard's menu — the kill switch is honoured where exposure is not, and the difference between the two keys is the whole design.
5. **Draw your own wiring, then break it.** Open `/firefly/graph` in the sample and find the arrow from `WalletService` to `EloquentWalletRepository` — note that the *Wired by* column says `WalletRepository`, the port, not `class`. Then introduce a deliberate cycle (have a `#[Service]` take a constructor parameter typed as another `#[Service]` that already depends on it), reload the page, and confirm the **Cycles** stat turns red and names both classes. Now boot the app fresh without opening the dashboard, and compare what PHP tells you about the same cycle.
6. **Read the access default as a security decision.** Set `app.debug` to `false` in a scratch project with `firefly/admin` installed and confirm `/firefly` is genuinely unrouted rather than merely unlinked (`php artisan route:list` should not list it). Then set `firefly.admin.enabled` to `true` without adding any `HttpSecurity` rule, and look at what an unauthenticated `GET /firefly/env` now discloses — that is precisely the gap this chapter told you to close with your own auth middleware.
7. **Turn a summary into something you can take a p99 of.** Scrape `GET /actuator/prometheus` in a project with `firefly/observability` installed and find the `http_server_requests_seconds` family: note the `# TYPE ... summary` line and that all you have is `_count` and `_sum`. Then set `firefly.observability.metrics.distribution.buckets` to the client default list this chapter prints, re-scrape, and confirm the `# TYPE` line now reads `histogram` and a `_bucket{le=...}` series appeared per bound. Now silence just that one meter

again with an **empty** `distribution.per-meter.http_server_requests_seconds` list, and confirm every other timer keeps its buckets. Finally put a `0` — or a `-1`, or a string — into the global list and confirm the boot refuses it by name rather than quietly dropping the bound.

8. **Make your logs machine-readable, then read one line.** Set

`firefly.logging.structured.format` to `ecs`, make a request that writes a log line from inside a command handler, and read the resulting line: find `trace.id`, `span.id` and `labels.correlation_id`, and confirm the span id is the **handler's**, not the request's, by comparing it against a line written from a plain controller in the same request. Then check that your own context keys landed under `context / extra` rather than at the top level — try logging a key literally called `message` and confirm it did not overwrite ECS's. Last, set the key to `jsonn` and confirm the application refuses to boot, naming the key, instead of silently falling back to plain text.

Testing LaraFly Applications

By the end of this chapter you will know `firefly/testing`'s two boot-harness families — `FireflyTestCase` / `FireflyDatabaseTestCase` for Testbench-style class tests, and `bootFireflyApp()` / `fireflyApplication()` for bare boots with no test framework glue at all — the eleven recording doubles that let a test assert on what a Firefly port actually saw (something Laravel's own `Event::fake()` / `Bus::fake()` cannot do, because they have never heard of Firefly's ports), the five Firefly-flavored Pest expectations, the `WebSliceTestCase` / `DataSliceTestCase` pair that boots only the beans a test needs, the Pest 4 browser suite that drives the **shipped skeleton** in the same process — same container, same SQLite database, no second server — and why `assertNoJavaScriptErrors()` is the assertion that earns it, the helpers that sign a test in (`actingAsAuthentication()` and the `actingAsPrincipal()` / `actingAsOidcUser()` pair built on it, the declarative `#[WithMockUser]`, and `withoutSecurity()` for the other direction), and — because this book only teaches what is actually shipped — a real, honest limitation `samples/lumen`'s own tests ran into and how they worked around it.

NEW TERM: SLICE TEST

A **slice test** boots only the narrow vertical of the framework a test actually exercises — the web pipeline over one controller, or the data pipeline over one repository — rather than the whole application. It is Spring Boot's `@WebMvcTest` / `@DataJpaTest` idea: faster boots, and a mis-wired slice fails immediately instead of quietly working by accident because some unrelated bean happened to be present too.

The boot harness: `FireflyTestCase`

Every hand-rolled Firefly test base across the monorepo used to re-derive the same handful of boot-order rules by hand. `FireflyTestCase` absorbs them once, as a `Testbench\TestCase` subclass with exactly three hooks to override:

```
/**
 * The Firefly capability providers this test needs, in registration order.
 * FireflyAutoConfigureServiceProvider is ALWAYS prepended by the harness – do NOT list it
 here.
 *
 * @return list<class-string<ServiceProvider>>
 */
protected function fireflyProviders(): array
{
    $attribute = $this->fireflyTestAttribute();

    return $attribute instanceof FireflyTest ? $attribute->providers : [];
}
// ...
protected function configOverrides(): array
{
    $attribute = $this->fireflyTestAttribute();

    return $attribute instanceof FireflyTest ? $attribute->config : [];
}
// ...
/**
 * Lazily-resolved bindings / manifest instances, bound AFTER config. Override to
 * $app->instance(SomeManifest::class, ...) or bind a port fake for the whole test class.
 */
protected function defineFireflyEnvironment(Application $app): void
{
    // no-op by default
}
```

`configOverrides()` returns eager, dot-keyed configuration seeded **before** boot, so that a `#[ConditionalOnProperty]` or `#[ConditionalOnMissingBean]` pass — which scans at *register* time — actually observes it. That is the seam for `firefly.scan.paths` and for any `firefly.<feature>.*` flag a test needs on. Each of the three hooks also has a class-level form: `#[FireflyTest(providers: [...], config: [...])]` on the test class does the same job without an override.

The base class handles the parts every test used to get subtly wrong:

`FireflyAutoConfigureServiceProvider` is *always* registered first (every capability provider's own `register()` assumes the kernel already exists), `configOverrides()` is applied before boot so a `#[ConditionalOnProperty]` scanning at register time actually observes it, and the log channel is switched to `errorlog` so a read-only sandbox never trips over a filesystem-backed logger. A minimal subclass needs nothing more than the one hook it actually uses:

```
<?php

declare(strict_types=1);

use Firefly\Testing\FireflyTestCase;

final class WidgetProbeTest extends FireflyTestCase
{
    protected function configOverrides(): array
    {
        return ['firefly.scan.paths' => [
            'App\\Widgets\\' => app_path('Widgets'),
        ]];
    }
}
```

Two small guardrails round the base out: `app(): Application` throws a clear `LogicException` if you call it before boot, rather than handing back an untyped `null`, and `fireflyContext(): ApplicationContext` gives you straight back the same boot-engine facade Chapter 2 introduced, already resolved.

Database tests: `FireflyDatabaseTestCase`

`FireflyDatabaseTestCase` extends `FireflyTestCase` and mixes in `UsesSqliteMemory`, which binds a shared sqlite `:memory:` connection as the `testing` default — no Docker, no fixture database file, no cross-test state:

```
<?php

declare(strict_types=1);

use Firefly\Testing\FireflyDatabaseTestCase;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Foundation\Application;

final class WidgetRepositoryTest extends FireflyDatabaseTestCase
{
    protected function defineFireflyEnvironment(Application $app): void
    {
        $this->createSchema('widgets', function (Blueprint $table): void {
            $table->id();
            $table->string('name');
        });
    }
}
```

`createSchema(string $table, Closure(Blueprint): void $blueprint): void` is a thin wrapper over `Schema::create()` — nothing new to learn if you already know Laravel migrations, just a shorter path from "I need a table for this one test" to a running schema.

Bare boots: `bootFireflyApp()` and `fireflyApplication()`

Not every test wants a full Testbench `TestCase`. `Firefly\Testing\functions.php` — Composer `files`-autoloaded, so these are always available with no `use` statement — ships a second family for a bare boot:

```
function fireflyApplication(array $config = [], array $providers = [], array $bindings = [],
array $needs = []): Application
// ...
function bootFireflyApp(array $config = [], array $providers = [], array $bindings = [],
array $needs = []): ApplicationContext
```

`$config` is the full config array; `$providers` register **after** `FireflyAutoConfigureServiceProvider`, which is always registered first regardless; `$bindings` are `$app->instance()` overrides bound *before* provider registration — the seam for handing in a port double; and `$needs` is a small "missing-bindings menu" (`'cache'`, `'validation'`, `'http'`) that binds a fresh, isolated fallback for a capability a bare boot would otherwise lack entirely:

```
<?php
declare(strict_types=1);
```

```

use Firefly\Cqrs\CqrsServiceProvider;
use Firefly\Cqrs\CqrsWiringProvider;
use Firefly\Cqrs\Handler\HandlerManifest;

it('boots a bare cqrs app with AutoConfigure first and an explicit binding', function () {
    $context = bootFireflyApp(
        config: ['firefly' => ['cqrs' => []]],
        providers: [CqrsServiceProvider::class, CqrsWiringProvider::class],
        bindings: [HandlerManifest::class => new HandlerManifest([], [])],
    );

    expect($context->has(HandlerManifest::class))->toBeTrue();
});

```

`bootFireflyApp()` is the identical call, returning the booted `ApplicationContext` instead of the raw `Application` — useful whenever a test wants to reach straight for a bean via the context facade rather than Laravel's own container API.

Recording doubles: testing without `Event::fake()`

Laravel's `Event::fake()` / `Bus::fake()` intercept *Laravel's own* event and job dispatch — they have no visibility into `firefly/eda`'s `EventPublisher`, `firefly/cqrs`'s `CommandBus`, or any other Firefly port. `firefly/testing` ships a recording or stub double for exactly that: a plain implementation of the real port that records what it saw, so a test can assert on behavior instead of re-implementing the framework's own internals.

Port	Double (Firefly\Testing\Double*)	Records
Firefly\Eda\EventPublisher	RecordingEventPublisher	<code>\$published</code> — <code>{destination, eventType, payload, headers};</code> <code>publishedOf(string \$eventType)</code> filters
Firefly\Context\Event\ApplicationEventPublisher	RecordingApplicationEventPublisher	<code>\$events</code> — every published event object, in order; <code>ofType(string \$class)</code> filters
Firefly\Cqrs\Command\CommandBus	RecordingCommandBus	<code>\$sent;</code> <code>willReturn(string \$commandClass, mixed \$result)</code> programs a canned result; <code>handled(string \$class)</code> filters
Firefly\Cqrs\Query\QueryBus	StubQueryBus	<code>\$asked;</code> <code>willReturn(string \$queryClass, mixed \$result)</code> programs a canned result
Firefly\Cqrs\Metrics\CQRSMetrics	RecordingCQRSMetrics	<code>\$commandSuccesses /</code> <code>\$commandFailures /</code> <code>\$querySuccesses /</code> <code>\$queryFailures</code>

Port	Double (Firefly\Testing\Double*)	Records
Firefly\Cqrs\Event\CommandEventPublisher	RecordingCommandEventPublisher	\$published (list of DomainEvent); an optional constructor ?Throwable \$throw exercises the bridge's failure path
Firefly\Messaging\MessageBrokerPort	RecordingMessageBroker	\$published — {topic, value, key, headers}; publishedTo(string \$topic) filters
Firefly\Scheduling\Lock\DistributedLock	RecordingDistributedLock	\$acquired / \$released ; constructor (bool \$available = true) ; setAvailable() toggles lock grant
Firefly\Actuator\Health\HealthIndicator	FakeHealthIndicator	Defaults Status::Up ; constructor takes a Health , setHealth() reprograms it
Firefly\Observability\Tracing\Tracer	RecordingTracer	\$spans — every traced span name, in call order; still faithfully invokes and returns the traced callback

Port	Double (Firefly\Testing\Double*)	Records
Firefly\Context\Event\ApplicationEventPublisher	RecordingAuthenticationEvents	\$events , plus the security family by name: successes() , interactive() , failures() , logouts() , denials() ; an optional constructor ? ApplicationEventPublisher \$forwardTo re-publishes every recorded event for real

Two rows name the same port, and the difference between them is what a security test needs. `RecordingApplicationEventPublisher` records objects; `RecordingAuthenticationEvents` also knows the five security event kinds by name, so a sign-in flow test reads `$this->events->failures()[0]->username` instead of filtering a mixed list by class. It is the double every listing in Chapter 10's sign-in, logout and denial sections calls `$this->events`.

Its docblock carries the two rules that make it work, and both are easy to get wrong:

```
/**
 * // ...
 * Bind it BEFORE boot — ` $app->instance(ApplicationEventPublisher::class, $events)` from
 * defineFireflyEnvironment() — so firefly/security's AuthenticationEventPublisher bean
 * wraps it; the
 * framework's own default is a bound()-guarded provider binding, which is why an instance()
 * wins there.
 * // ...
 */
final class RecordingAuthenticationEvents implements ApplicationEventPublisher
```

```

{
    /** @var list<object> */
    public array $events = [];

    public function __construct(private readonly ?ApplicationEventPublisher $forwardTo =
    null) {}

```

The first rule is the binding moment: an `instance()` registered from `defineFireflyEnvironment()` is already there when the security provider's `bound()` - guarded default would have run, so the spy wins — bind it after boot and the framework's own publisher is what the filters hold.

The second is `$forwardTo`, and it exists because binding the spy **replaces** the port. Nothing registered against the dispatcher hears a security event any more: an `#[AsEventListener]` on `InteractiveAuthenticationSuccessEvent` stays silent for the whole suite, which is how a suite whose pipeline includes such a listener (`firefly/security-oauth2-server` stamps the sign-in instant that way) breaks without an error. Hand the framework's own publisher in — `new RecordingAuthenticationEvents(new DispatcherEventPublisher($app))` — and every event is recorded *and* published for real. The default, `null`, keeps the spy a pure recorder, which is right for a suite that only reads.

`RecordingEventPublisher`, in full, shows the shape every double in the table follows — a real port implementation, plus a plain array that just remembers what happened:

```

final class RecordingEventPublisher implements EventPublisher
{
    /** @var list<array{destination: string, eventType: string, payload:
    array<string,mixed>, headers: array<string,string>}> */
    public array $published = [];

    public function subscribe(string $eventTypePattern, callable $handler): void {}

```

```

    public function publish(string $destination, string $eventType, array $payload, array
$headers = []): void
    {
        $this->published[] = compact('destination', 'eventType', 'payload', 'headers');
    }

    public function start(): void {}

    public function stop(): void {}

    // ...

    public function publishedOf(string $eventType): array
    {
        return array_values(array_filter($this->published, static fn (array $e): bool =>
$e['eventType'] === $eventType));
    }
}

```

```

<?php

use Firefly\Testing\Double\RecordingEventPublisher;

$publisher = new RecordingEventPublisher;
$publisher->publish('accounts.events', 'AccountOpened', ['owner' => 'alice'], ['x' => '1']);

expect($publisher->published)->toHaveCount(1)
    ->and($publisher->published[0]['eventType']->toBe('AccountOpened');

```

Firefly-flavored Pest expectations

`Firefly\Testing\Pest\FireflyExpectations::register()` installs a small set of custom `expect()` matchers, called once from the monorepo's own root `tests/Pest.php`:

```
use Firefly\Testing\Pest\FireflyExpectations;
// ...
FireflyExpectations::register();
```

There are exactly **five**:

- **toHavePublished(string \$eventType, array \$payloadContains = [])** — on a `RecordingEventPublisher`; at least one published event of the given type whose payload contains the given key/value subset.
- **toHaveHandledCommand(string \$class)** — on a `RecordingCommandBus`; at least one sent command is an instance of `$class`.
- **toBeUp()** — on a `Health` or a `HealthIndicator` (narrowed via `->health()` automatically); asserts `Status::Up`.
- **toHaveRecordedMetric(string \$name, array \$tags = [])** — on anything exposing `meters(): list<Meter>` (e.g. a `MeterRegistry`); at least one recorded meter with the given name and tag subset.
- **toBeProblemDetails(int \$status)** — on a `TestResponse` (checks the `application/problem+json` content type and decodes the body) or a plain array; asserts the RFC-7807 shape has `type` / `title` / `status` keys and that `status` matches.

```
<?php
use Firefly\Testing\Double\RecordingCommandBus;

// @phpstan-ignore method.notFound
expect($commandBus)->toHaveHandledCommand(PlaceOrder::class);
```

THE @PHPSTAN-IGNORE METHOD.NOTFOUND COMMENTS, EXPLAINED

Pest installs these via `expect()->extend(...)` **at runtime** — PHPStan's static reflection of the vendor `Pest\Expectation` class has no way to see a method added this way, so every call is flagged `method.notFound` even though it genuinely exists once `tests/Pest.php` has run. Keep each custom expectation on its own `expect()` statement rather than chaining with `->and(...)`: chaining degrades the precisely-ignorable `method.notFound` into the unignorable `method.nonObject`.

For a plainer, PHPUnit-flavored assertion instead of the fluent expectations, two procedural helpers live alongside `bootFireflyApp()` in the same autoloader `functions.php`:

```
function assertEventPublished(object $publisher, string $eventType, array $payloadContains = []): void
// ...
function assertNoEventsPublished(object $publisher): void
{
    expect($publisher->published)->toBeEmpty('Expected no events to have been published.');
```


TOBEPROBLEMDetails() **AND THE** **TYPE** **FIELD: A REAL LIMITATION, AND HOW IT WAS CLOSED**

This box used to describe a genuine gap, and the gap is worth keeping in view because it is the shape these things usually have. `toBeProblemDetails()` asserts the response body has a `type` key — and the framework's real renderer, `Firefly\Web\Exception\ProblemDetailsRenderer`, published no `type` at all. It was an optional member emitted only when something explicitly passed one, which the exception-to-response path never did, so calling the shipped expectation against a genuine `404 / 422 / 403 / 409` failed every time. A matcher and a renderer that ship in the same framework disagreed about the shape of the same document.

RFC 9457 §3.1.1 settled it: an absent `type` is `about:blank`, so writing it out costs a conformant document nothing and saves every client from having to know the rule. The renderer now emits it — Spring's `ProblemDetail` makes the same choice — and `firefly.web.problem.type-uri` either points at a base URI, deriving a dereferenceable type from the stable `code` (`https://api.example.test/problems/resource-not-found`), or is set to `''` to drop the member again for a deployment that wants the pre-9457 document byte for byte. So the expectation works against the framework's own error responses, and `samples/lumen` exercises it on the 404:

```
it('returns RFC-7807 problem+json for an unknown wallet', function () {
    // ...
    $response = $this->getJson('/api/v1/wallets/nope/balance')
    ->assertStatus(404)
    ->assertHeader('Content-Type', 'application/problem+json')
    ->assertJsonPath('status', 404)
    ->assertJsonPath('code', 'RESOURCE_NOT_FOUND')
    ->assertJsonPath('title', 'Not Found');
    // ...
    expect($response)->toBeProblemDetails(404);
});
```


The field assertions stayed, and that is the part worth copying: `toBeProblemDetails()` says the response is a problem document of *that* status, while `assertJsonPath('code', 'RESOURCE_NOT_FOUND')` says which failure it is. The second is the one a regression trips over, so a test that can afford both should have both.

Slice testing: `WebSliceTestCase` and `DataSliceTestCase`

A slice test boots only the beans a PSR-4 scan discovers for one narrow vertical, plus whatever explicit port overrides you supply — Spring's `@WebMvcTest` / `@DataJpaTest` idea, ported. Both bases expose a method, callable straight from a Pest closure test:

```
abstract class WebSliceTestCase extends FireflyTestCase
{
    // ...
    public function webSlice(array $scan = [], array $overrides = []): ApplicationContext
    {
        if ($this->sliceRequested && ($scan !== $this->sliceScan || $overrides !== $this->sliceOverrides)) {
            throw new LogicException('webSlice() is per-test-class; call it once with a single slice definition.');
```

`DataSliceTestCase::dataSlice()` is the same method for the persistence vertical.

Both are **per test class**: calling one twice with different slice definitions is a `LogicException` rather than a silent second boot, because the second definition would apply to every test in the class and not only to the ones written after it.

```
<?php
declare(strict_types=1);
```

```

use Firefly\Testing\Slice\WebSliceTestCase;

uses(WebSliceTestCase::class);

it('boots a web slice and serves a sliced controller route', function () {
    $this->webSlice(scan: [
        'App\Http\Slice\' => base_path('tests/Fixtures/Slice'),
    ]);

    $response = $this->get('/slice/ping');

    expect($response->status())->toBe(200);
});

```

`WebSliceTestCase` boots the real web + validation pipeline — including route wiring — over only the sliced controllers, so `$this->get(...)` serves a genuine `TestResponse` against them. `DataSliceTestCase` boots the real data pipeline over only the sliced beans, then eagerly resolves every one of them, so a mis-wired slice fails immediately at boot instead of passing by accident because nothing in the test happened to touch the broken bean:

```

<?php

declare(strict_types=1);

use Firefly\Testing\Slice\DataSliceTestCase;

uses(DataSliceTestCase::class);

it('boots ONLY the sliced data beans and resolves them', function () {
    $context = $this->dataSlice(
        scan: ['App\Domain\Widgets\' => base_path('tests/Fixtures/Widgets')],
        overrides: [PricingPort::class => new FakePricingPort],
    );
});

```

```
expect($context->get(WidgetRepository::class))->toBeInstanceOf(WidgetRepository::class);
});
```

Both methods are **per-test-class** — calling either a second time with different arguments throws a `LogicException`, so a slice's shape is fixed for the whole test class rather than mutable mid-test.

`#[FireflyTest]` / `#[WebSlice]` / `#[DataSlice]` attributes

For a hand-written, PHPUnit-style test **class** rather than a Pest closure file, three class-target attributes give the same three shapes declaratively, read by `setUp()` before Testbench's own boot ever runs:

```
<?php

declare(strict_types=1);

use Firefly\Testing\Attributes\WebSlice;
use Firefly\Testing\Slice\WebSliceTestCase;

#[WebSlice(scan: ['App\Http\Slice\' => __DIR__.'/Fixtures/Slice'])]
final class SliceControllerTest extends WebSliceTestCase
{
    public function test_web_slice_attribute_boots_the_sliced_route(): void
    {
        $response = $this->get('/slice/ping');

        $this->assertSame(200, $response->status());
    }
}
```

⚠ ATTRIBUTES VS. METHOD CALLS — DO NOT MIX THE TWO ON ONE TEST CLASS

`#[FireflyTest]` / `#[WebSlice]` / `#[DataSlice]` only work on a hand-written class extending the matching base — not on a Pest closure file. A Pest closure test calls `$this->webSlice(...)` / `$this->dataSlice(...)` directly instead, because the method internally calls `refreshApplication()` to re-boot with the slice values — Testbench has already performed an empty first boot in `setUp()` before the closure body ever runs. Pick one style per test class.

The fixture layer

Three small helpers round out the kit, each replacing a hand-rolled equivalent that used to be duplicated per package.

`FixtureRegistry` is a thin named-fixture registry over plain factory closures — the Firefly convenience sitting alongside Eloquent factories, not a replacement for them:

```
final class FixtureRegistry
{
    // ...
    public function register(string $name, Closure $factory): self
    // ...
    public function has(string $name): bool
    // ...
    public function make(string $name, array $overrides = []): object
    // ...
    public function load(string ...$names): array
}
```

```
<?php
```

```
use Firefly\Testing\Fixture\FixtureRegistry;
```

```

$registry = (new FixtureRegistry)->register('widget', function (array $overrides): Widget {
    return new Widget(
        $overrides['name'] ?? 'default',
        $overrides['qty'] ?? 1,
    );
});

$bolt = $registry->make('widget', ['name' => 'bolt', 'qty' => 5]);
$two = $registry->load('widget', 'widget'); // list<Widget>, 2 entries

```

AggregateSeeder replays an aggregate's domain events through an **ApplicationEventPublisher** port — useful for seeding a projection (Chapter 6's ledger, for instance) without re-running the whole command flow that originally produced those events:

```

<?php

use Firefly\Testing\Double\RecordingApplicationEventPublisher;
use Firefly\Testing\Fixture\AggregateSeeder;

$publisher = new RecordingApplicationEventPublisher;
(new AggregateSeeder)->publishEvents($publisher, new WidgetCreated('a'), new WidgetCreated('b'));

expect($publisher->events)->toHaveCount(2);

```

ListenerSpy records arbitrary string values a listener or handler fixture saw, in call order — the one shared replacement for what used to be a hand-rolled **Spy** class in every package that needed one:

```

<?php

use Firefly\Testing\Fixture\ListenerSpy;

```

```

$spy = new ListenerSpy;
$spy->record('order.placed');
$spy->record('order.shipped');

expect($spy->seen)->toBe(['order.placed', 'order.shipped']);

```

Real backends: `RequiresDocker` and `fireflyConfigFor()`

Some tests need a genuine backend — a real Postgres, a real Kafka broker — rather than an in-memory double. These are tagged `@group integration` and excluded from the default gate entirely: `vendor/bin/pest` (the whole-branch gate `composer test` runs) never touches one unless you explicitly ask with `vendor/bin/pest --group=integration`.

`RequiresDocker` is a trait (`@mixin PHPUnit\Framework\TestCase`) that turns "Docker is unavailable" into a clean skip rather than a hard failure, so a Docker-less machine that *does* run the integration group still gets a green result:

```

trait RequiresDocker
{
    protected function skipUnlessDocker(): void
    {
        if (! is_docker_available()) {
            $this->markTestSkipped('Docker is not available; skipping @group integration test.');
```

`is_docker_available()` runs `docker info` and returns `true` iff it exits `0` — it never throws, so calling it with no Docker installed at all is always safe. A real Kafka-backed integration test base shows the pattern end to end:

```
<?php

declare(strict_types=1);

use Firefly\Testing\Integration\RequiresDocker;
use PHPUnit\Framework\TestCase;

abstract class KafkaIntegrationTestCase extends TestCase
{
    use RequiresDocker;

    protected function setUp(): void
    {
        $this->skipUnlessDocker();
        parent::setUp();
    }
}
```

`fireflyConfigFor(object $container, string $prefix = 'database.connections.testing'): array` — the `@ServiceConnection` analogue — maps a started testcontainer into a flat, dot-keyed config array, duck-typed against whichever of `getHost()` / `getMappedPort()` / `getUsername()` / `getPassword()` / `getDatabase()` the container object actually exposes, so it works against any testcontainers-php container class with no hard dependency on the library's types:

```
/**
 * Map a started testcontainer into a flat Firefly/Laravel config array (the
 * @ServiceConnection analog).
 */
```

```

* @return array<string,mixed>
*/
function fireflyConfigFor(object $container, string $prefix =
'database.connections.testing'): array

```

Feed the result straight into a `FireflyTestCase` subclass's `configOverrides()`, or set each key on the config repository from `defineFireflyEnvironment()`.

Proof from Lumen: how the sample dogfoods it all

`samples/lumen` builds its own test base, `LumenTestCase`, on top of `FireflyDatabaseTestCase` — the pattern every chapter of this book has ultimately been resting on:

```

abstract class LumenTestCase extends FireflyDatabaseTestCase
{
    // ...
    protected function fireflyProviders(): array
    {
        return [
            ValidationServiceProvider::class,
            WebServiceProvider::class,
            DataServiceProvider::class,
            CqrsServiceProvider::class,
            CqrsWiringProvider::class,
            EdaServiceProvider::class,
            EdaWiringProvider::class,
            SecurityServiceProvider::class,
            SecurityWiringProvider::class,
            ActuatorServiceProvider::class,
            ActuatorWiringProvider::class,
        ];
    }
}

```



```
// ...
protected function configOverrides(): array
{
    return [
        // ...
        'firefly.scan.paths' => $this->scanPaths(),
        // ...
        'firefly.security.enabled' => true,
        // ...
        'firefly.management.enabled' => true,
        'firefly.management.endpoints.web.exposure.include' => 'health,info',
        // ...
    ];
}
// ...
}
```

Every one of Chapter 4 through Chapter 11's HTTP calls in this book — every `$this->postJson(...)`, every RFC-7807 assertion — runs against a `LumenTestCase` subclass. It deliberately binds the *real* `InMemoryEventBus` as the `EventPublisher` (never a `RecordingEventPublisher`), because Lumen's own tests need the domain-to-integration bridge from Chapter 8 to genuinely fire, not merely to be recorded as if it had.

📦 LARAVEL PARITY

Laravel's own test kit (`RefreshDatabase`, `Event::fake()`, `Bus::fake()`) is excellent for Laravel's own primitives, but it has no visibility into a Firefly port — a `CommandBus::send()` or an `EventPublisher::publish()` call is invisible to `Event::fake()`, because nothing about it goes through Laravel's own event dispatcher. `firefly/testing`'s recording doubles and Pest expectations are the parity layer: the same "arrange a fake, assert on what it saw" idiom you already know from Laravel, aimed at the ports this book's other chapters actually introduced.

A real browser over the real app

Everything so far in this chapter asserts on objects and on HTTP responses. Neither can tell you whether the welcome page actually rendered, whether the admin dashboard's charts drew, or whether Swagger UI threw a JavaScript error while building its operation list. For that you need a browser — and the reason most projects do not have browser tests is that the usual setup is miserable: a second process serving the app, a second database, a second container, and a suite that drifts from the one the unit tests exercise.

LaraFly's browser suite has none of that, because of one property worth stating plainly: **the browser drives the same booted application the rest of the suite does.**

Pest 4 plus `pestphp/pest-plugin-browser` serves the Testbench-booted application to Chromium **in process**. There is no `php -S`, no separate document root, no second connection — the same container, and the same in-memory SQLite database the test seeded:

```
/**
 * The application every browser scenario drives: the shipped skeleton, compiled by the real
 firefly:cache
 * writer and booted under Testbench with the provider set a created app gets – served to
 Chromium by the
 * plugin's in-process server, so the test keeps the container and the SQLite connection.
 // ...
 */
abstract class BrowserTestCase extends SkeletonExampleTestCase
{
    use SeedsOrders;

    /** @return list<class-string<ServiceProvider>> */
    protected function fireflyProviders(): array
    {
```

```

    return DiscoveredProviders::forSkeleton();
}

```

And it is the **skeleton** that gets driven, compiled by the real `firefly:cache` writer — the same project `composer create-project` produces. A browser test failing is therefore evidence about what a new user's application does, not about a fixture built to pass.

A scenario reads the way you would describe it out loud:

```

it('renders the welcome page with the live route table', function (): void {
    /** @var BrowserTestCase $this */
    visit('/')
        ->assertSee('Hello, LaraFly')
        ->assertSee('Your routes')
        ->assertSee('/orders/{id}')
        ->assertSee('OrderController')
        ->assertSee('GreetingController')
        ->assertNoJavaScriptErrors()
        ->screenshot(filename: 'welcome');
});
// ...

it('serves the OpenAPI viewer, and Swagger UI draws the operations without a JavaScript error', function (): void {
    /** @var BrowserTestCase $this */
    visit('/openapi')
        ->assertPresent('#swagger-ui')
        ->wait(2)
        ->assertSee('/orders/{id}')
        ->assertNoJavaScriptErrors()
        ->screenshot(filename: 'openapi-viewer');
});

```

`assertNoJavaScriptErrors()` is on almost every scenario, and it is the assertion that pays for the suite. A page that renders and quietly throws in the console is a page that looks fine in a screenshot and is broken for the person using it.

Why it does not slow your gate down

Browser tests need Node and a Chromium download. The default gate must not:

```
<testsuite name="unit">
  <directory>tests</directory>
  <directory>packages/*/tests</directory>
  <directory>samples/lumen/tests</directory>
  <exclude>tests/Browser</exclude>
</testsuite>
<testsuite name="browser">
  <directory>tests/Browser</directory>
</testsuite>
```

Two suites, and `tests/Browser` is excluded from the default one. So `composer check` — Pint, PHPStan, Pest and Deptrac — stays entirely Node-free, and `composer test:browser` runs the browser suite alone. CI has a separate `Browser` (Pest + `Playwright`) job that runs it and uploads every screenshot as an artifact, so a failure comes with a picture of what the page looked like.

The **suite split** is what does the work here, not the group label:

```
// Browser tests: everything under tests/Browser drives a real Chromium through
// pestphp/pest-plugin-browser against the Testbench-booted app (the plugin serves
// it in-process). tests/Browser is its own `browser` testsuite in phpunit.xml.dist
// and is excluded from the default `unit` suite, so the default gate never needs
// Node; `composer test:browser` runs this suite alone. The `browser` group is a
// label only (for --group=browser filtering): the suite split in phpunit.xml.dist
// is what keeps these files out of the default run – Pest includes every file of
// the configured suite before it filters groups, and the plugin boots Playwright
```

```
// the moment a file under tests/Browser is included.
pest()->group('browser')->in('Browser');

pest()->browser()
  ->inChrome()
  ->inLightMode()
  ->timeout(10000);
```

That comment is worth reading twice, because the distinction it draws catches people out: Pest **includes** every file of the configured suite before it filters by group, and the plugin boots Playwright the moment a file under `tests/Browser` is included. A `--exclude-group=browser` would therefore still pay the Playwright cost. Only the suite split avoids it.

⚠ THE PLUGIN FORCES `APP.DEBUG=FALSE`

Two framework keys default to `app.debug` — `firefly.web.error-page.trace` and `firefly.admin.enabled` — and the plugin forces `app.debug=false` while it handles each browser request. A suite that relied on those defaults would render the *production* error page while meaning to test the debug one. So `BrowserTestCase` switches every surface it means to look at **on by its own key**, and subclasses override `trace()` and `securityOverrides()` — both read before boot, which is why they are hooks rather than per-test calls.

And a browser authenticates the way a person does: `SecuredBrowserTestCase` puts the framework's own form login and two in-memory users in front of the application, so the login scenarios drive the same `FormLoginFilter` Chapter 10 described rather than a testing shortcut.

Signing a test in

Which brings us to the helpers the security chapters lean on. Three methods, a fourth attribute beside the three in `Firefly\Testing\Attributes\`, and one switch that turns the whole stack off. The three methods are built one on the next:

```
public function actingAsAuthentication(Authentication $authentication): static
{
    SecurityContextHolder::setContext(new SecurityContext($authentication));
    $this->app()->instance(ActingPrincipal::class, new ActingPrincipal($authentication));

    $kernel = $this->resolveHttpKernel();
    if ($kernel instanceof FoundationHttpKernel && ! $kernel-
>hasMiddleware(ActingPrincipalMiddleware::class)) {
        $kernel->prependMiddleware(ActingPrincipalMiddleware::class);
    }

    return $this;
}
```

`actingAsAuthentication()` is the seam: it sets the holder immediately, binds the principal, and prepends a middleware **once**, so a second call swaps who the next request is rather than stacking another copy on the kernel. The middleware is the outermost link on purpose — `SecurityContextPersistenceFilter` leaves an already-authenticated holder alone, and the framework filters that clear the holder in their own **finally** run inside it, so the principal is back on the holder when the request returns.

`actingAsPrincipal('ada', ['ROLE_USER'])` is the everyday form, built on it. And for the OAuth2 chapters there is Spring Security's `oidcLogin()`, as a method:

```
/**
 * Run the rest of this test as a person who signed in through OpenID Connect (Spring
 * Security's
```

```

    * `oidcLogin()` test support, as a method): the principal is a real DefaultOidcUser
over an id token whose
    * claims are `{iss: https://idp.test, sub: user, aud: firefly-app, iat, exp}` overlaid
by $claims – and
    * NO raw token value, exactly the shape the session hands a controller after a real
login – so
    * `#[AuthenticationPrincipal] OidcUser $user`, `getClaim()`, `getEmail()` and the rest
all answer. Its
    * authorities are OIDC_USER (an OidcUserAuthority carrying the claims) and SCOPE_x per
$scopes; the
    * Authentication carries those plus $authorities (where a GrantedAuthoritiesMapper
would have put ROLE_*),
    * and $registrationId on its attributes, which the authorized-client manager and RP-
initiated logout read.
    * The principal is named by $nameAttributeKey read from the claims (`sub` by default),
so a claim for it
    * must exist. Nothing here talks to a provider: no registration, no discovery, no
token.
    // ...
    */

    public function actingAsOidcUser(array $claims = [], array $authorities = [], string
$registrationId = 'oidc', array $scopes = ['openid'], string $nameAttributeKey = 'sub'):
static

```

`actingAsOidcUser()` builds a **real** `DefaultOidcUser`, with the authorities a real login would have granted — `OIDC_USER` plus a `SCOPE_x` per scope — and the registration id on the token's attributes, which is what the authorized-client manager and RP-initiated logout read. Nothing here talks to a provider: no registration, no discovery, no token request. And it deliberately carries **no raw token value**, because that is exactly the shape the session hands a controller after a real sign-in; a test that gets a raw token would be testing something the application never sees.

`#[WithMockUser]` is Spring's `@WithMockUser`, and it is the declarative form of `actingAsPrincipal()` — the fourth attribute in `Firefly\Testing\Attributes\`, beside the three slice attributes above:

```

/**
 * Spring's @WithMockUser: run the test (or every test of the class) as a signed-in
principal. `roles` are
 * prefixed with `ROLE_` unless already so; `authorities` are taken verbatim. A method-level
attribute wins
 * over a class-level one. Honoured by FireflyTestCase::setUp() through actingAsPrincipal(),
so it covers
 * direct calls, HTTP requests through the filters, the dispatcher guard and proxied beans
alike.
 */
#[Attribute(Attribute::TARGET_CLASS | Attribute::TARGET_METHOD)]
final readonly class WithMockUser
{
    /**
     * @param list<string> $roles
     * @param list<string> $authorities
     */
    public function __construct(
        public string $name = 'user',
        public array $roles = ['USER'],
        public array $authorities = [],
    ) {}
}

```

Unlike the three slice attributes, this one is read in `setUp()` on any `FireflyTestCase`, method-level beating class-level, so a hand-written test class can sign its whole suite in once and name the exception on the one method that needs a different principal.

And for the other direction there is `withoutSecurity()`:

```

/**
 * Switch the security stack off for the rest of this test: the URL filter, the CSRF
filter, every
 * authentication filter, the proxy's method-security link and the CQRS bus authorizers
(through
 * MethodSecurityMessageEnforcer, which DefaultCommandBus/DefaultQueryBus hold by

```



```
constructor) read their
    * flags live, and the dispatcher guard is rebound to the no-op default. Beans already
built stay built;
    * only their gates change – so a controller test that dispatches a command whose
handler carries a
    * #[PreAuthorize] gets the handler's answer, not a 401.
    */
    public function withoutSecurity(): static
```

Read the last sentence twice, because it is the reason this method exists at all rather than a config override: the flags are read **live**, on every request, so flipping them mid-test changes the gates on beans that were already built. A test that only wants to exercise the thing *behind* the gate does not have to re-boot the application to get past it.

✓ What you learned

Concept	What it does
<code>FireflyTestCase</code>	Testbench base; three hooks (<code>fireflyProviders</code> / <code>configOverrides</code> / <code>defineFireflyEnvironment</code>); config seeded before boot
<code>FireflyDatabaseTestCase</code> / <code>UsesSqliteMemory</code>	Adds a shared sqlite <code>:memory:</code> connection + <code>createSchema()</code>
<code>bootFireflyApp()</code> / <code>fireflyApplication()</code>	Bare, no-Testbench boots; a <code>\$needs</code> menu fills in missing cache/validation/http fallbacks
11 recording doubles	Real port implementations that record what they saw — the parity layer <code>Event::fake()</code> can't reach
5 Pest expectations	<code>toHavePublished</code> , <code>toHaveHandledCommand</code> , <code>toBeUp</code> , <code>toHaveRecordedMetric</code> , <code>toBeProblemDetails</code>
<code>WebSliceTestCase</code> / <code>DataSliceTestCase</code>	Boot only a PSR-4-scanned slice + explicit overrides; per-test-class, fail-fast on data
<code>#[FireflyTest]</code> / <code>#[WebSlice]</code> / <code>#[DataSlice]</code>	Class-attribute analog of the same three shapes, for hand-written test classes
<code>FixtureRegistry</code> / <code>AggregateSeeder</code> / <code>ListenerSpy</code>	Named fixtures; domain-event replay; a shared call-order spy
<code>RequiresDocker</code> / <code>fireflyConfigFor()</code>	<code>@group integration</code> opt-in; skip-not-fail without Docker; testcontainer → config mapping

Concept	What it does
The <code>browser</code> suite	Pest 4 + <code>pest-plugin-browser</code> serving the Testbench-booted skeleton in process — same container, same SQLite, no second server
<code>phpunit.xml.dist</code> 's two suites	<code>tests/Browser</code> excluded from <code>unit</code> , so <code>composer check</code> stays Node-free; <code>composer test:browser</code> runs it alone
<code>assertNoJavaScriptErrors()</code>	The assertion that earns the suite: a page that renders and throws in the console looks fine in a screenshot
<code>actingAsAuthentication()</code>	The seam under <code>actingAsPrincipal()</code> and <code>actingAsOidcUser()</code> ; middleware prepended once, so a second call swaps the principal
<code>actingAsOidcUser()</code>	Spring's <code>oidcLogin()</code> as a method: a real <code>DefaultOidcUser</code> , real authorities, no raw token — the shape a session really hands a controller
<code>RecordingAuthenticationEvents</code>	The security-aware double: the security event family by name (<code>successes</code> / <code>interactive</code> / <code>failures</code> / <code>logouts</code> / <code>denials</code>); bind it before boot, and pass <code>\$forwardTo</code> when real listeners must still hear
<code>#[WithMockUser]</code>	Spring's <code>@WithMockUser</code> : the declarative <code>actingAsPrincipal()</code> , read in <code>setUp()</code> , method-level beating class-level

Concept	What it does
<code>withoutSecurity()</code>	Every security flag off for the rest of the test — read live, so beans already built change gates without a re-boot

Try it yourself

- Write a slice test for one of Lumen's own controllers.** Using `WebSliceTestCase`, scan only `Lumen\Web\` and confirm you can serve one wallet endpoint without booting CQRS, EDA, or security at all — then note which requests fail, and why, once you understand exactly which beans the slice left out.
- Watch `type` move under `toBeProblemDetails()`.** Call `toBeProblemDetails(404)` against a real `404` from Lumen's own REST API and confirm it passes — the `type` key this chapter's note is about is `about:blank`. Then set `firefly.web.problem.type-uri` to `https://api.example.test/problems`, re-run, and confirm the member became `https://api.example.test/problems/resource-not-found` while the expectation still passes. Finally set the key to `''` and watch the same expectation fail with the missing-`type`-key message — the failure the note describes, now something a deployment chooses rather than something the framework does to you.
- Run the browser suite, then break a page.** Run `composer test:browser` and look at `tests/Browser/Screenshots`. Then add a `<script>` that throws to the welcome view and re-run: confirm the page still renders, still passes every `assertSee`, and that `assertNoJavaScriptErrors()` is the only assertion that catches it.
- Prove the suite split matters.** Time `composer test` and confirm nothing downloads Chromium. Then try `vendor/bin/pest --exclude-group=browser` and

watch Playwright boot anyway — the group label filters tests, the suite split is what stops the files being included at all.

5. **Add a `RequiresDocker` -gated integration test.** Pick a repository backed by Eloquent, write an `@group integration` test against a real Postgres testcontainer using `fireflyConfigFor()`, and confirm `vendor/bin/pest` (no flags) skips it entirely while `vendor/bin/pest --group=integration` runs it for real.
6. **Watch `$forwardTo` matter.** Bind a `RecordingAuthenticationEvents` with no `$forwardTo` from `defineFireflyEnvironment()`, register an `#[AsEventListener]` on `InteractiveAuthenticationSuccessEvent` that writes a row, and sign in: confirm the spy records the event and the listener never ran. Then re-bind it wrapping the framework's own publisher and confirm both happen. Finally move the `instance()` call to *after* boot and confirm the spy records nothing at all — the binding moment is the whole trick.
7. **Reach past a gate two ways.** Take a controller test that gets a `401`, and make it pass twice: once with `#[WithMockUser(roles: ['ADMIN'])]` on the method, once with `withoutSecurity()`. Then put `#[WithMockUser]` on the class and a different one on a single method, and confirm which of the two the framework honours.

The CLI and the Zero-Reflection Cache

By the end of this chapter you will know exactly what `php artisan firefly:cache` does — the thirteen scanner-and-compiler steps it runs, in order, over your application's own PSR-4 roots, and the `bootstrap/cache/firefly/` manifests and generated proxy classes (one per class the proxy plan names) it writes — how `FireflyCacheServiceProvider` loads that cache back in at boot with zero reflection, `firefly:clear`'s exact inverse, the actuator-over-CLI commands (`firefly:about` / `:routes` / `:health` / `:metrics`) that reuse Chapter 11's endpoints in-process with no HTTP round trip, the eight `make:firefly-*` generators, and how `firefly/installer`'s `firefly new` gets you from nothing to a cached, running application in one command.

NEW TERM: ZERO-REFLECTION BOOT

Every chapter before this one has mentioned, in passing, that a compiled manifest replaces "scan the filesystem with PHP reflection at every boot." A **zero-reflection boot** is what you get once `firefly:cache` has actually run: `FireflyCacheServiceProvider` loads pre-computed manifests straight from PHP files under `bootstrap/cache/firefly/`, so no attribute, no class, and no method is ever reflected on at request time — the scan happened once, at deploy time, not on every request.

firefly:cache : the zero-reflection payoff

`CacheCommand` is a thin wrapper — it reads `firefly.scan.paths` (the same PSR-4 map every earlier chapter's `configOverrides()` seeded) and hands it to `ManifestCacheWriter`:

```
final class CacheCommand extends Command
{
    /** @var string */
    protected $signature = 'firefly:cache';
    // ...
    public function handle(): int
    {
        // ...
        $psr4 = $this->laravel->make('config')->get('firefly.scan.paths', []);
        if ($psr4 === []) {
            $this->warn('firefly.scan.paths is empty – nothing to compile.');
        }

        return self::SUCCESS;
    }

    $dir = FireflyCachePaths::dir($this->laravel);
    $report = (new ManifestCacheWriter)->write($psr4, $dir);

    $this->info(sprintf(
        'firefly:cache – wrote %d manifest(s) + %d proxy(ies) to %s',
        count($report->files),
        $report->proxyCount,
        $dir,
    ));

    $this->reportStaleEntries();

    return self::SUCCESS;
}
```



```
// ...
}
```

`ManifestCacheWriter::write()` is the real orchestration, and it is worth reading end to end: it is nothing more than every capability package's *existing* scanner-and-compiler pair, called in sequence, over the same PSR-4 map — no new abstraction, zero edits to any settled package:

```
final class ManifestCacheWriter
{
    // ...
    public function write(array $psr4, string $dir): CacheReport
    {
        $report = $this->writeManifests($psr4, $dir);
        $count = $this->writeProxies($psr4, $dir);

        return new CacheReport([...$report->files, $dir.'/'.FireflyCachePaths::PROXY_MAP],
            $count);
    }
    // ...
    public function writeManifests(array $psr4, string $dir): CacheReport
    {
        if (! is_dir($dir)) {
            mkdir($dir, 0o755, true);
        }

        $files = [];

        // container + context + autoconfigure (folded): one call emits both component +
        // context manifests.
        (new AutoConfigManifestCompiler)->write(
            $psr4,
            $files[] = $dir.'/'.FireflyCachePaths::COMPONENT,
            $files[] = $dir.'/'.FireflyCachePaths::CONTEXT,
        );
        // ...
    }
}
```

```

    (new ConfigManifestCompiler)->write(
        (new ConfigPropertiesScanner)->scan($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::CONFIG_PROPERTIES,
    );

    // web
    $routeScanner = new RouteScanner;
    (new RouteManifestCompiler)->write(
        $routeScanner->scan($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::ROUTES,
    );
    // ...
    (new ExceptionHandlerManifestCompiler)->write(
        $routeScanner->scanExceptionHandlers($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::EXCEPTION_HANDLERS,
    );
    // ...
    (new ConstraintManifestCompiler)->write(
        (new ClassEnumerator)->enumerate($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::CONSTRAINTS,
    );
    // ...
    $handlers = (new HandlerScanner)->scan($psr4);
    (new HandlerManifestCompiler)->write(
        $handlers['handlers'],
        $handlers['destinations'],
        $files[] = $dir.'/'.FireflyCachePaths::HANDLERS,
    );

    // eda
    (new EventListenerManifestCompiler)->write(
        (new EventListenerScanner)->scan($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::EVENT_LISTENERS,
    );

    // messaging
    (new MessageListenerManifestCompiler)->write(
        (new MessageListenerScanner)->scan($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::MESSAGE_LISTENERS,
    );

```

```

    );
}

    // scheduling
    (new ScheduledManifestCompiler)->write(
        (new ScheduledScanner)->scan($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::SCHEDULED,
    );

    // security
    (new SecurityMethodManifestCompiler)->write(
        (new MethodSecurityScanner)->scan($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::SECURITY_METHODS,
    );
    // ...
    (new TransactionalManifestCompiler)->write(
        (new TransactionalScanner)->scan($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::TRANSACTIONAL,
    );
    // ...
    (new ProxyPlanCompiler)->write(
        $this->planner()->plan($psr4),
        $files[] = $dir.'/'.FireflyCachePaths::PROXY_PLAN,
    );

    return new CacheReport($files);
}
}

```

That is twelve `scan()` -then- `write()` calls — every capability from Chapter 2's component scan through Chapter 10's method-security manifest, each already introduced in an earlier chapter as an in-process scan — plus one more step, `writeProxies()`, that generates the proxy classes Chapter 9 taught you about, using the exact same reflection-free `ProxyClassGenerator` the runtime itself uses when no cache exists yet:

```

final class ManifestCacheWriter
{
    // ...

    public function writeProxies(array $psr4, string $dir): int
    {
        $proxyDir = $dir.'/'.FireflyCachePaths::PROXY_DIR;
        if (! is_dir($proxyDir)) {
            mkdir($proxyDir, 0o755, true);
        }

        $planner = $this->planner();
        $plan = $planner->plan($psr4);
        $generator = new ProxyClassGenerator;
        $map = [];

        foreach ($planner->proxyMethods($plan) as $targetClass => $methods) {
            $source = $generator->generate($targetClass, $methods);
            $proxyClass = $plan->proxyClassFor($targetClass);
            $file = $proxyDir.'/'.str_replace('\\', '_', $targetClass).'.php';
            file_put_contents($file, $source);
            $map[$proxyClass] = $file;
        }

        file_put_contents(
            $dir.'/'.FireflyCachePaths::PROXY_MAP,
            "<?php\n\ndeclare(strict_types=1);\n\n// Generated by firefly/cli. Do not edit.\n\nreturn ".var_export($map, true).";\n",
            );

        return count($map);
    }

    // ...

    private function planner(): ProxyPlanner
    {
        return new ProxyPlanner([
            new ObservabilityAdviceSource,
            new MethodSecurityAdviceSource,
            new ResilienceAdviceSource,
        ]);
    }
}

```

```

        new TransactionalAdviceSource,
    );
}
// ...
}

```

Thirteen steps in total land fourteen artifacts under `bootstrap/cache/firefly/`:

```

bootstrap/cache/firefly/
├─ component.php      # DI component manifest (container + context + autoconfigure)
├─ context.php        # application-context manifest
├─ config-properties.php # #[ConfigProperties] DTOs
├─ routes.php         # compiled route table
├─ exception-handlers.php # #[ControllerAdvice]/#[ExceptionHandler] manifest
├─ constraints.php    # validation constraint manifest
├─ handlers.php       # #[CommandHandler]/#[QueryHandler] manifest
├─ event-listeners.php # #[EventListener] manifest
├─ message-listeners.php # #[MessageListener] manifest
├─ scheduled.php      # #[Scheduled] manifest
├─ security-methods.php # #[PreAuthorize]/#[Secured]/#[RolesAllowed] manifest
├─ transactional.php  # #[Transactional] method manifest
├─ proxy-plan.php     # which beans get a proxy, and which advice each method runs
├─ proxies.php        # FQCN => file classmap for the generated proxies
└─ proxies/           # one generated proxy class file per class the plan names

```

FireflyCacheServiceProvider : loading the cache back in

Writing the manifests is only half the story — something has to notice they exist and bind them *instead of* running the in-process scan. `FireflyCacheServiceProvider` (auto-discovered) is that something, and its `register()` runs before any bean resolution at all:

```

final class FireflyCacheServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $dir = FireflyCachePaths::dir($this->app);
        // ...
        if (is_file($path = $dir.'/'.FireflyCachePaths::ROUTES)) {
            $this->app->instance(RouteManifest::class, RouteManifest::load($path));
        }
        // ...
        if (is_file($path = $dir.'/'.FireflyCachePaths::HANDLERS)) {
            $this->app->instance(HandlerManifest::class, HandlerManifest::load($path));
        }
        // ...
        // Proxies: register a classmap autoloader so TransactionalBeanPostProcessor's
class_exists($proxyClass)
        // is satisfied BEFORE it throws. register() runs at provider-register, before any
boot-pass bean resolution.
        $proxyMap = $dir.'/'.FireflyCachePaths::PROXY_MAP;
        if (is_file($proxyMap)) {
            // ...
            $map = require $proxyMap;
            spl_autoload_register(static function (string $class) use ($map): void {
                if (isset($map[$class]) && is_file($map[$class])) {
                    require $map[$class];
                }
            });
        }
    }
}

```

Each `$app->instance(...)` call is **unconditional**, and every capability's own default is `if (! bound(X)) singleton(X, empty)` — so the loaded manifest always wins, regardless of which provider Laravel happened to register first. The `spl_autoload_register` call at the end is what actually makes the cached proxies reachable: it installs a classmap loader keyed by the generated proxy's fully-qualified

name, so the very first time `TransactionalBeanPostProcessor` checks `class_exists($proxyClass)`, the class is already loadable — no reflection, no filesystem scan, just a map lookup.

None of this is mandatory. An application with no cache directory at all still boots — every capability's manifest binding falls back to the in-process scanner exactly as it has in every earlier chapter — just without the zero-reflection guarantee. `firefly:cache` is a pure performance-and-determinism upgrade you opt into by running it, not a requirement for the framework to function.

firefly:clear

The exact inverse: it recursively deletes `FireflyCachePaths::dir()` — the configured `firefly.cache.path`, or `bootstrap/cache/firefly` by default — and nothing else:

```
final class ClearCommand extends Command
{
    /** @var string */
    protected $signature = 'firefly:clear';
    // ...
    public function handle(): int
    {
        $dir = FireflyCachePaths::dir($this->laravel);
        if (! is_dir($dir)) {
            $this->info('firefly:clear – nothing to remove.');
        }

        return self::SUCCESS;
    }
    // ...
    $it = new RecursiveIteratorIterator(
        new RecursiveDirectoryIterator($dir, FilesystemIterator::SKIP_DOTS),
        RecursiveIteratorIterator::CHILD_FIRST,
```

```

    );
    foreach ($it as $file) {
        $file->isDir() ? rmdir($file->getPathname()) : unlink($file->getPathname());
    }
    rmdir($dir);

    $this->info('firefly:clear - removed ' . $dir);

    return self::SUCCESS;
}
}

```

Safe to run at any time: the very next boot simply falls back to the in-process scanner, the same as an application that never ran `firefly:cache` at all.

When the cache names a class you deleted

Both commands above have to boot the application before they can do their work — `firefly:cache` cannot write the manifests without first booting the application whose manifests they are, and `firefly:clear` cannot delete a directory it has not booted past. That circularity turns uncomfortable the moment the cache is *wrong*, and the ordinary way to make it wrong is to delete a `#[Component]` and not recompile: `component.php` still names the class, and no autoloader can find it.

That state used to be unrecoverable by any single command. The deleted class was still in the manifest `ContainerRegistrar` read, so `wireInterfaces()` bound the interface it implemented — and tagged it as an implementation — to a class that was not there. The failure therefore surfaced nowhere near its cause: `Target class [App\Security\ControlPlaneJwksProvider] does not exist` came out of resolving a bean nobody had touched, one that merely injects that interface.

`EagerSingletonsPass` skips a definition whose class is gone, and it did not help,

because the throw was never raised on the deleted definition. `composer dump-autoload` could not help either — its `package:discover` boots the application too. The recovery was to delete `bootstrap/cache/firefly/*.php` by hand, then `composer dump-autoload`, then `firefly:cache`, in that order.

`Firefly\Context\Scan\AppScan::repairing()` is the seam that closes it: while `firefly:cache` or `firefly:clear` is the running command, `BeanDefinitionRegistry` — the one door every definition enters through — drops a definition whose class cannot be found, so the registrar, the bean-post-processor pass, the lifecycle pass and the event-listener pass never see it. `firefly:cache` then names what it dropped:

```
php artisan firefly:cache
```

```
firefly:cache - wrote 14 manifest(s) + 3 proxy(ies) to bootstrap/cache/firefly
firefly:cache - skipped 1 stale manifest entry naming a class that no longer exists:
App\Security\ControlPlaneJwksProvider
```

The manifest it writes no longer mentions the class, so the second run is an ordinary clean one and the line goes away.

⚠️ ONLY THOSE TWO COMMANDS DROP ANYTHING

Under every other command the manifest is trusted exactly as it was, and a class that has gone missing still stops the boot with the container's own error. That is the right answer there: a missing class in a process about to serve traffic is not a stale cache, it is a broken deployment — a truncated artifact, a classmap built from a different tree — and dropping the definition would hand the application an interface quietly rebound to whichever implementation survived. A wrong answer nobody is told about is worse than the boot failure. And only a MISSING class is ever tolerated: a class that exists and cannot be constructed still fails fast, in a repair command as much as anywhere else.

Actuator-over-CLI: `firefly:about`, `firefly:routes`, `firefly:health`, `firefly:metrics`

Chapter 11 built a management surface reachable over HTTP. These four commands render the **same** endpoints at the terminal, in-process — no HTTP request is made, and none of them reimplement any actuator logic:

```
final class AboutCommand extends Command
{
    /** @var string */
    protected $signature = 'firefly:about';
    // ...
    public function handle(ActuatorCliRenderer $renderer): int
    {
        $this->line('LaraFly '.Version::VERSION);
        foreach (['info', 'env', 'beans', 'conditions', 'mappings', 'scheduledtasks'] as
$id) {
            $this->newLine();
            $this->info('# '.$id);
            $renderer->render($this, $id);
        }
    }
}
```

```

    return self::SUCCESS;
}
}

```

`firefly:about` resolves the compiled `ActuatorRegistry` (the same registry `ActuatorRouteRegistrar` populates at boot), calls each endpoint's `handle()`, and prints the response — the full Spring Boot `--debug /actuator` introspection story at a glance, in one command. `firefly:routes` renders just the `mappings` endpoint (your compiled route table), `firefly:health` renders `health`, and `firefly:metrics` renders `metrics` — the last one requires `firefly/observability` to be installed, and prints a warning and exits successfully rather than erroring if it is not.

Scaffolding: the `make:firefly-*` generators

Every framework stereotype this book has introduced has a matching Artisan generator — pyfly's `generate` command family, ported to Laravel's own `GeneratorCommand` base:

Command	Generates
<code>make:firefly-controller</code>	Two files: a <code>#[RestController]</code> REST resource (<code>index</code> / <code>show</code> / <code>store</code> / <code>update</code> / <code>destroy</code> under one class-level <code>#[RequestMapping]</code> whose path is derived from the resource name) <i>and</i> the <code>#[Valid]</code> <code>#[RequestBody]</code> DTO its <code>store</code> / <code>update</code> bind — <code>--plain</code> gives the single-action shape with no DTO.
<code>make:firefly-service</code>	A <code>#[Service]</code> bean.
<code>make:firefly-component</code>	A <code>#[Component]</code> bean.
<code>make:firefly-handler</code>	Two files: a <code>#[CommandHandler]</code> <i>and</i> the command class its <code>handle()</code> takes — <code>#[QueryHandler]</code> plus its query with <code>--query</code> .
<code>make:firefly-listener</code>	An <code>#[EventListener]</code> method by default, or a <code>#[MessageListener]</code> with <code>--message</code> .
<code>make:firefly-entity</code>	A DDD entity extending <code>Firefly\Domain\Entity</code> (there is no <code>#[Entity]</code> attribute).
<code>make:firefly-repository</code>	A concrete <code>#[Repository]</code> class extending <code>Firefly\Data\Repository\EloquentRepository</code> , with a <code>\$model</code> to repoint.
<code>make:firefly-config-properties</code>	A <code>#[ConfigProperties]</code> -bound configuration DTO.

`make:firefly-handler` is representative of the whole family — a `GeneratorCommand` that picks which stub to render off a single flag:

```

final class MakeHandlerCommand extends GeneratorCommand
{
    /** @var string */
    protected $name = 'make:firefly-handler';

    /** @var string */
    // ...
    protected $type = 'Firefly handler';
    // ...
    protected function getStub(): string
    {
        // ...
        return __DIR__.'/../../../../stubs/'.($this->option('query') ? 'query-handler.stub' :
'command-handler.stub');
    }
    // ...
    protected function getOptions(): array
    {
        return [
            ['query', null, InputOption::VALUE_NONE, 'Generate a #[QueryHandler] instead of
a #[CommandHandler].'],
        ];
    }
}

```

and the command-handler stub it renders — with its `{{ namespace }}` / `{{ class }}` placeholders substituted by `GeneratorCommand` the way `php artisan make:firefly-handler RegisterWidget` would — is exactly the shape Chapter 7 taught you to write by hand:

```

namespace App\Handlers;

use Firefly\CQRS\Attributes\CommandHandler;

#[CommandHandler]

```

```
final class RegisterWidget
{
    /**
     * Replace `object` with the concrete Command message class this handler dispatches for
     * (or pass it explicitly via #[CommandHandler(SomeCommand::class)]).
     */
    public function handle(object $command): mixed
    {
        return null;
    }
}
```

```
php artisan make:firefly-handler RegisterWidget
php artisan make:firefly-handler CountWidgets --query
```

Thin passthroughs: `firefly:serve`, `firefly:db`

Two commands exist purely for a consistent `firefly:*` naming convention over commands Laravel already ships — neither reimplements any Laravel behavior:

```
php artisan firefly:serve {--host=127.0.0.1} {--port=8000}
```

Delegates to `artisan serve`, or to `octane:start` when `laravel/octane` is installed — probed only via `class_exists()`, so Octane stays an optional runtime dependency `firefly/cli`'s own `composer.json` never requires.

```
php artisan firefly:db {action=migrate}
```

Delegates to Laravel's own database commands: `migrate` (default), `db:seed` (`firefly:db seed`), or `migrate:fresh` (`firefly:db fresh`).

From zero to running: `firefly/installer` and `firefly/skeleton`

`firefly/installer` ships the global `firefly new` command — a small Symfony Console binary, not itself a scaffolder. It invokes `composer create-project` with the template bundled in `fireflyframework/laraflly` as a local path repository:

```
use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;
// ...
#[AsCommand(name: 'new', description: 'Create a new LaraFly application')]
final class NewCommand extends Command
{
    // ...
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);
        $runner = $this->runner ?? new SymfonyProcessRunner($output);
        // ...
        $repository = json_encode([
            'type' => 'path',
            'url' => dirname(__DIR__, 3).'/skeleton',
            'options' => ['symlink' => false, 'versions' => ['firefly/skeleton' =>
Version::VERSION]],
            ], JSON_THROW_ON_ERROR | JSON_UNESCAPED_SLASHES);
        $create = ['composer', 'create-project', 'firefly/skeleton', $directory, '--no-
interaction', '--repository='.$repository];
        if ($input->getOption('dev')) {
            $create[] = '--stability=dev';
        }
        if ($runner->run($create) !== 0) {
```

```

        $io->error('composer create-project failed.');
```



```

        return self::FAILURE;
    }
    // ...
    if (! $input->getOption('no-git')) {
        $runner->run(['git', 'init', '-q'], $directory);
        $runner->run(['git', 'add', '.'], $directory);
        $runner->run(['git', 'commit', '-q', '-m', 'Initial commit'], $directory);
    }

    $io->success("LaraFly application ready at {$directory}");
    $io->writeln("  cd {$name}");
    // ...
    $io->writeln('  php artisan firefly:serve');
```



```

    return self::SUCCESS;
}
```

`#[AsCommand]` here is **Symfony's**, not Firefly's, and the `use` lines above say so. `firefly new` is a standalone Symfony Console application rather than an artisan command, because it has to run before there is a Laravel application to be a command of.

Every external process goes through the `ProcessRunner` seam — `composer`, `php artisan`, `git` — which is what makes the whole flow assertable without a network or a real `composer create-project`.

`firefly/skeleton` — a genuine, top-level Laravel 13 application template pre-wired with the Firefly family and a sample `#[RestController] / #[Service]` pair — is what the installer supplies to `composer create-project` from its own distribution. Its own `composer.json` closes the loop this whole chapter has been building toward: `firefly:cache` runs **automatically**, right after install, with no manual step:


```

"scripts": {
  "post-create-project-cmd": [
    "@php -r \"file_exists('.env') || copy('.env.example', '.env');\"",
    "@php -r \"file_exists('database/database.sqlite') || touch('database/database.sqlite');\"",
    "@php artisan key:generate --ansi",
    "@php artisan firefly:cache"
  ]
}

```

Every application `firefly new` scaffolds is therefore, from the very first `git commit`, already boot-cached — a genuinely zero-reflection application on disk, not merely a framework capable of becoming one. `bootstrap/cache/firefly/` itself is `.gitignore`d in the skeleton (the manifests are generated fresh per install, not committed), which is exactly what you would expect of a build artifact rather than source.

LARAVEL PARITY

`composer create-project laravel/laravel` gives you a bare Laravel app with nothing pre-wired; `firefly new` gives you the same Laravel app, plus the Firefly family pre-required, plus a sample stereotype pair, plus a boot cache already warmed — the equivalent of Spring Initializr's "generate a working starter project," aimed at `artisan` rather than a web form. `make:firefly-*` mirrors Laravel's own `make:controller / make:model` family exactly in spirit — an Artisan `GeneratorCommand`, a stub, a namespace placeholder — just one stereotype attribute deeper.

✓ What you learned

Concept	What it does
<code>firefly:cache</code>	Runs every capability's own scanner → compiler pair (13 steps, 14 artifacts) over <code>firefly.scan.paths</code> ; writes manifests + one generated proxy class file per class the proxy plan names to <code>bootstrap/cache/firefly/</code>
<code>FireflyCacheServiceProvider</code>	Binds each cached manifest unconditionally (wins over every empty default) + installs the proxy classmap autoloader — before any bean resolution runs
<code>firefly:clear</code>	Recursively deletes only the cache dir; the next boot falls back to in-process scanning
<code>firefly:about</code> / <code>:routes</code> / <code>:health</code> / <code>:metrics</code>	Renders Chapter 11's actuator endpoints at the terminal, in-process, no HTTP round trip
<code>make:firefly-*</code>	Eight generators, one per stereotype — <code>GeneratorCommand</code> + a stub, exactly like Laravel's own <code>make:*</code> family
<code>firefly:serve</code> / <code>firefly:db</code>	Thin passthroughs to <code>artisan serve</code> / <code>octane:start</code> and Laravel's own database commands
<code>firefly new</code> (<code>firefly/installer</code>)	Supplies the bundled skeleton to <code>composer create-project</code> ; the skeleton runs <code>firefly:cache</code> automatically post-install

Try it yourself

1. **Run `firefly:cache` against Lumen and inspect the output.** From the framework's monorepo, point `firefly.scan.paths` at `samples/lumen/src` and run the command by hand; open `bootstrap/cache/firefly/handlers.php` and confirm it lists `WithdrawHandler` and its siblings from Chapter 6 and 7, in a form you can read without any reflection at all.
2. **Break the cache on purpose.** Run `firefly:cache`, then rename a `#[CommandHandler]` class without re-running it. Confirm the cached `handlers.php` still points at the old name — and that only `firefly:cache` again (not `firefly:clear`) fixes it — proving the cache is a snapshot, not a live view.
3. **Scaffold a whole vertical with the generators.** Use `make:firefly-entity`, `make:firefly-repository`, `make:firefly-handler`, and `make:firefly-controller` in sequence to build a tiny new feature from scratch, then run `firefly:cache` once at the end and confirm every piece appears in the compiled manifests together.

Appendices

APPENDIX A

Laravel → LaraFly Cheat-Sheet

LaraFly is **additive**: it layers a Spring-Boot-shaped application model on top of Laravel 13 — it never forks, wraps, or replaces the framework underneath. Every LaraFly capability is a normal Composer package registering normal Laravel service providers; a LaraFly app is still, in every respect a Laravel developer would recognize, a Laravel app. This appendix is a quick-reference map from the Laravel idiom you already know to the LaraFly concept this book introduced it alongside, with the chapter that covers it in depth.

At a glance

Concern	Raw Laravel	LaraFly	Chapter
Entry point	Service providers registered in <code>bootstrap/providers.php</code> , wired by hand	The same providers, plus auto-discovered <code>AutoConfiguration</code> classes assembled by a kernel-decided <code>BootPass</code> pipeline	2
Dependency injection	<code>app()->bind()</code> / <code>app()->singleton()</code> in a provider's <code>register()</code>	<code>#[Component]</code> / <code>#[Service]</code> / <code>#[Repository]</code> / <code>#[Configuration]</code> stereotypes on the class itself; a compiled component scan resolves constructor dependencies	2
Configuration	<code>config('mail.host')</code> (array access, untyped)	<code>#[ConfigProperties]</code> DTOs bound from a config subtree — typed, fail-fast on a missing/mismatched key	3

Concern	Raw Laravel	LaraFly	Chapter
HTTP routing	<code>routes/web.php</code> / <code>routes/api.php</code> route files	<code>#[RestController]</code> + verb attributes (<code>#[GetMapping]</code> , ...), compiled to a <code>RouteManifest</code> , still dispatched through native Laravel routes	4
Validation	<code>FormRequest::rules()</code> (array rules)	<code>#[Valid]</code> parameter interception over a Bean-Validation- style constraint model, still backed by Laravel's validator	4
Persistence	Eloquent models + query builder directly in application code	The same Eloquent underneath a <code>CrudRepository</code> port/adaptor — application code depends on an interface, never on Eloquent directly	5
Domain modeling	Eloquent models carry both data and behavior	<code>Firefly\Domain\Entity</code> /aggregates with no framework dependency; Eloquent lives only in the adaptor	6

Concern	Raw Laravel	LaraFly	Chapter
Transactions	<code>DB::transaction(fn () => ...)</code> (closure-scoped)	<code>#[Transactional]</code> on a class/method — declarative propagation/ isolation/rollback rules, a generated proxy drives <code>beginTransaction /</code> <code>commit / rollback</code>	9
Events	<code>Event::listen() /</code> <code>#[AsEventListener]</code> <code>], in-process only</code>	Two distinct surfaces: the in- process bus <code>(#[AsEventListener])</code> and a broker- backed EDA bus <code>(#[EventListener])</code>	8
CQRS	Not a first-class concept — a Laravel "command" usually means an Artisan console command	<code>CommandBus /</code> <code>QueryBus</code> with a bounded pipeline (validate → authorize → invoke → metrics) and <code>#[CommandHandler] /</code> <code>#[QueryHandler]</code>	7

Concern	Raw Laravel	LaraFly	Chapter
Security	Auth guards + route middleware (- <code>>middleware('auth')</code> , <code>Gate::allows()</code>)	Spring-Security-6-shaped <code>SecurityContext</code> + deny-by-default <code>HttpSecurity</code> URL DSL + method security (<code>#[PreAuthorize]</code> / <code>#[Secured]</code> / <code>#[RolesAllowed]</code>)	10
Operations	Nothing built in — health checks and metrics are typically bespoke or a package	<code>firefly/actuator</code> (health/info/env/beans/conditions/mappings/loggers/scheduledtasks) + <code>firefly/observability</code> (Prometheus/Micrometer-JSON metrics)	11
Testing	<code>RefreshDatabase</code> , <code>Event::fake()</code> , <code>Bus::fake()</code> — see Laravel's own primitives only	<code>FireflyTestCase</code> / <code>FireflyDatabaseTest</code> , recording doubles for Firefly's own ports, Firefly-flavored Pest expectations	12

Concern	Raw Laravel	LaraFly	Chapter
Scaffolding	<code>php artisan make:controller / make:model</code>	<code>php artisan make:firefly- controller / make:firefly- handler /... — one generator per stereotype</code>	13
Boot performance	Laravel's own config/route caching (<code>config:cache</code> , <code>route:cache</code>)	<code>php artisan firefly:cache</code> — one command compiles every Firefly manifest and generates <code>#[Transactional]</code> proxies for a zero- reflection boot	13

Entry point: service providers vs. auto-configuration

Laravel boots by registering whatever service providers `bootstrap/providers.php` lists, in the order they appear (or, for package-discovered providers, roughly alphabetically). LaraFly's `firefly/context` sits on top of that: every capability package's provider still exists and is still discovered by Laravel exactly the same way — but its `register()` method does nothing except *buffer* its `BootPass` contributions into a `PendingBootPasses` collector. The shared `FireflyKernel` drains that buffer once it is actually resolved and runs each phase (scanning, condition evaluation, bean registration, wiring passes) in a kernel-decided order — so which provider Laravel happened to instantiate first is irrelevant.

Dependency injection: `app()->bind()` vs. stereotypes

Laravel's container is powerful but explicit — a binding lives in a provider, separate from the class it binds:

```
final class GreetingServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->bind(GreetingService::class, fn () => new GreetingService('Hello'));
    }
}
```

LaraFly's `firefly/container` layers PHP 8 attributes onto `Illuminate\Container` instead — mark the class itself with `#[Component]` (or `#[Service]` / `#[Repository]` / `#[Configuration]`, all specializations of it) and a compiled component scan registers it, resolving constructor dependencies by type:

```
#[Service]
final readonly class GreetingService
{
    public function __construct(private string $greeting = 'Hello') {}
}
```

Nothing stops you from also using `app()->bind()` directly — the two coexist, and Chapter 2 showed you exactly when each is the better fit.

Configuration: `config()` vs. `#[ConfigProperties]`

`config('mail.port')` is untyped array access — a typo or a missing key returns `null` silently. `firefly/config` adds `#[ConfigProperties('mail')]` DTOs that bind a whole config subtree onto a plain readonly class at once, typed and fail-fast on a missing required key. Laravel's `config/*.php` files remain the single source of truth; LaraFly reads them, it does not replace them.

Transactions: `DB::transaction()` vs. `#[Transactional]`

`DB::transaction(fn () => ...)` scopes a transaction to a closure — propagation and rollback rules are whatever you write inline. `#[Transactional]` declares propagation (`REQUIRED`, `REQUIRES_NEW`, `NESTED`, ...), isolation, read-only, and rollback/no-rollback exception lists on the class or method itself; a proxy generated at scan time (Chapter 9) drives the boundary through manual `beginTransaction()` / `commit()` / `rollback()`, so a caught exception can still be committed when it matches `noRollbackFor`.

Security: middleware vs. deny-by-default `HttpSecurity`

Laravel's default posture is permissive: a route is public unless you attach `>middleware('auth')` or an ability check. `firefly/security` inverts that — the `HttpSecurity` URL DSL builds an ordered rule list and denies anything unmatched (401 anonymous, 403 authenticated), the same deny-by-default model as Spring Security. Method security (`#[PreAuthorize]`, `#[Secured]`, `#[RolesAllowed]`) is enforced at the CQRS bus and the controller dispatcher via a closed-whitelist expression evaluator — no `eval()` anywhere in the chain (Chapter 10). Laravel's own auth guards and middleware still work underneath; `firefly/security` is a stricter layer atop them, not a fork.

Operations: actuator & observability

Laravel ships no health-check or metrics endpoint out of the box — most teams either hand-roll one or reach for a package. `firefly/actuator` (health/info/env/beans/conditions/mappings/loggers/scheduledtasks under `/actuator`, secured entirely by ordinary `HttpSecurity` config) and `firefly/observability` (a first-party Prometheus-0.0.4 + Micrometer-JSON `/metrics` exposition) are the Spring Boot Actuator and Micrometer analogues, respectively — both opt-in Composer packages, both secure-by-default (Chapter 11).

Scaffolding and boot performance: `make:* / config:cache` vs. `make:firefly-* / firefly:cache`

Laravel's own `make:controller / make:model` generators and its `config:cache / route:cache` boot-performance commands have direct LaraFly analogues, extended to cover every framework stereotype rather than only Eloquent models and routes: `make:firefly-*` (eight generators, one per stereotype) and `firefly:cache` (which compiles every Firefly manifest — components, routes, CQRS handlers, event listeners, scheduled tasks, security rules, transactional metadata — plus generates the `#[Transactional]` proxy classes, in one command) — see Chapter 13.

LARAVEL PARITY

Every row on this page is meant to be read in one direction only: LaraFly never asks you to give up a Laravel idiom, it asks you to additionally reach for a more structured one where the codebase benefits from it. A LaraFly project can freely mix `app()->bind()` and `#[Service]`, a plain `routes/api.php` route and a `#[RestController]`, in the same codebase — nothing in this book's samples requires an all-or-nothing migration.

APPENDIX

Glossary

Actuator — LaraFly's production-management surface: framework endpoints (`/actuator/health` , `/actuator/info` , `/actuator/env` , and others) mounted directly on the same Illuminate **Router** your own controllers use, secure-by-default and unreachable until explicitly exposed. The Spring Boot Actuator analogue, shipped by `firefly/actuator` (Chapter 11).

Adapter — A concrete class that implements a port by delegating to a specific technology — an Eloquent-backed repository, a Kafka-backed message broker. Adapters live at the edge of the hexagonal architecture and can be swapped without touching domain or application code (Chapters 5, 6).

Aggregate root — The single entry point to a cluster of domain objects that must stay consistent together. All state changes are routed through its methods, which enforce invariants and record domain events; external code loads and saves only the root, never its inner objects. `Wallet` is Lumen's aggregate root — its one invariant is that the balance never goes negative (Chapter 6).

ApplicationContext — The boot-engine facade: the object every test and every request-time consumer resolves to reach a booted bean, a compiled manifest, or the kernel's own state. `fireflyContext()` is how a `FireflyTestCase` subclass reaches it (Chapters 2, 12).

Attribute (PHP 8) — Native PHP 8 syntax (`#[Component]` , `#[CommandHandler]` , `#[Transactional]` , ...) that LaraFly reads at scan time to discover a class's architectural role. Attributes replace the annotation-driven configuration Spring Boot expresses in Java; a compiled manifest is what makes reading them a one-time, not a per-request, cost (Chapter 2).

Authentication — An immutable token answering "who is making this request?" — a principal, its granted authorities, and whether it has actually been authenticated.

`Authentication::authenticated()` / `unauthenticated()` are its only two constructors (Chapter 10).

Authorization — The question "is this principal allowed to do *this specific thing*?", answered independently of authentication by `HttpSecurity`'s deny-by-default URL rules and by method security (`#[PreAuthorize]` , `#[Secured]` , `#[RolesAllowed]`) (Chapter 10).

Autoconfiguration — A `#[Configuration]` class, ordered by `#[Order(N)]` , whose `#[Bean]` factory methods register a capability's default beans — always guarded by `#[ConditionalOnMissingBean]` so an application override wins.

`ObservabilityAutoConfiguration`'s `#[Order(500)]` winning the `CqrsMetrics` race before `CqrsAutoConfiguration` evaluates at `#[Order(1000)]` is the canonical example of ordering deciding which default survives (Chapters 2, 7, 10, 11).

Autowiring — Resolving a bean's constructor parameters from their declared types alone, with no factory code written by hand. LaraFly's compiled component scan performs this once, at cache time, rather than by reflection on every request (Chapter 2).

Bean — Any object the container creates, wires, and manages. A class becomes a bean by carrying a stereotype attribute (`#[Component]` , `#[Service]` , `#[Repository]` , `#[Configuration]`), or by a `#[Bean]` -attributed factory method inside a `#[Configuration]` class (Chapter 2).

BootPass — A single, ordered unit of boot-time work — scanning, condition evaluation, bean registration, route mounting — run by the `FireflyKernel` at a specific `BootPhase` . `ActuatorRouteRegistrar` and `MeterBindingsPass` are both `BootPass` es (Chapters 2, 11).

Circuit breaker — A resilience pattern that trips to an open state after a threshold of consecutive failures against a dependency, short-circuiting further calls until the dependency has had time to recover. `firefly/observability`'s `MeterBindingsPass` exposes each configured breaker's live state as a `resilience_circuit_breaker_state{name}` gauge (Chapter 11).

Command (CQRS) — An object expressing a single write intent — "withdraw funds", "open a wallet" — dispatched through the `CommandBus` to exactly one `#[CommandHandler]`. Commands may be denied before they ever reach their handler by a `CommandAuthorizer` (Chapters 7, 10).

CommandBus — The pipeline that receives a `Command`, runs it through a bounded sequence of stages (correlate → validate → authorize → invoke → metrics), and routes it to its one registered `#[CommandHandler]` (Chapter 7).

Component — The generic stereotype attribute for a managed bean that does not fit the more specific `#[Service]` / `#[Repository]` / `#[Configuration]` roles. All stereotypes are equivalent to the container; the distinction is for human readers and tooling (Chapter 2).

Condition (#[ConditionalOn*]) — An attribute (`#[ConditionalOnProperty]`, `#[ConditionalOnBean]`, `#[ConditionalOnMissingBean]`) evaluated during the boot pipeline's condition pass to decide whether a bean definition survives.

`DbHealthIndicator`'s `#[ConditionalOnProperty(matchIfMissing: true)]` is what keeps it on by default; a separate interface, `ConditionalHealthIndicator::available()`, is what keeps a database-less application from a surprise `DOWN` (Chapters 2, 11).

Deptrac — The static architecture-boundary linter (`deptrac/deptrac`) that enforces which package may depend on which, as declared layers in `deptrac.yaml` . A `0` violation count is part of this book's own definition of done for every chapter's code (Chapters 2, 11, 13).

Domain event — An immutable record of a business fact that has already happened — "wallet opened", "funds withdrawn" — recorded by an aggregate root and published, after a successful commit, through an `ApplicationEventPublisher` (Chapters 6, 8, 9).

DTO (Data Transfer Object) — A plain object carrying data across a layer boundary — a request body, a `#[ConfigProperties]` -bound configuration subtree — without exposing internal domain types (Chapters 3, 4).

Entity — A domain object with a stable identity that persists through state changes. LaraFly's `make:firefly-entity` scaffolds a class extending `Firefly\Domain\Entity` — there is no `#[Entity]` attribute (Chapters 6, 13).

EventListener — Either of two distinct, deliberately unrelated surfaces:
`#[AsEventListener]` for Laravel's own in-process event dispatch, and
`#[EventListener]` for `firefly/eda`'s broker-backed bus, which subscribes to an event-type *pattern* rather than a PHP class (Chapter 8).

Granted authority — A single permission or role string (`'ROLE_ADMIN'` , `'orders:read'`) carried by an `Authentication` . `hasRole('X')` normalises to a check against the granted authority `'ROLE_X'` (Chapter 10).

Health indicator — A one-method SPI (`HealthIndicator::health(): Health`) that a `#[Component]` implements to report whether some dependency or resource is up. `HealthEndpoint` aggregates every registered indicator to the single most-severe `Status` (Chapter 11).

Hexagonal architecture — An architectural style placing domain and application logic at the centre, surrounded by ports (interfaces) that adapters implement at the edges. Business code depends only on a port; which adapter satisfies it at runtime is a wiring decision, never a business-logic one (Chapters 1, 5, 6).

Integration event — A domain event that has crossed the boundary from the in-process domain model onto the broker-backed EDA bus, via the guarded after-commit bridge introduced alongside CQRS. Not the same object, and not the same surface, as the domain event that produced it (Chapters 7, 8, 9).

JWT (JSON Web Token) — A signed, expiring bearer token `JwtService` issues and validates. LaraFly's `JwtService` refuses to boot at all with a weak or placeholder signing secret, and refuses to accept any token missing its `exp` claim, however valid its signature (Chapter 10).

Manifest — A compiled, cacheable snapshot of everything a scanner found — routes, CQRS handlers, event listeners, security rules, transactional methods — written to `bootstrap/cache/firefly/` by `firefly:cache` and loaded back in at boot with no reflection at all (Chapters 2, 13).

Meter — A single named, tagged measurement — a `Counter`, a `Gauge`, or a `Timer` — held by a `MeterRegistry`. Registration is idempotent per `type|name|sorted-tags`; re-registering one name under a different type is a programming error the registry rejects loudly (Chapter 11).

Method security — Attribute-declared authorization (`#[PreAuthorize]`, `#[Secured]`, `#[RolesAllowed]`) enforced at the CQRS bus and the controller dispatcher by a closed-whitelist expression evaluator that never calls `eval()` (Chapter 10).

Outbox pattern — A technique for publishing a domain event reliably alongside the database write that produced it: both are written in the same local transaction, eliminating the need for a two-phase commit between the database and the message broker (Chapters 8, 9).

Pest expectation — A custom `expect()->extend(...)` matcher `firefly/testing` installs — `toHavePublished`, `toHaveHandledCommand`, `toBeUp`, `toHaveRecordedMetric`, `toBeProblemDetails` — registered once from the monorepo's root `tests/Pest.php` (Chapter 12).

Port — An interface a piece of business logic depends on, with no implementation detail attached. `EventPublisher`, `CommandBus`, `HealthIndicator`, and `MeterRegistry` are all ports; the concrete adapter satisfying each is a wiring decision (Chapters 1, 5, 11).

Principal — The "who" of a request — the `mixed $principal` an `Authentication` carries once authenticated, together with its granted authorities (Chapter 10).

Problem Details (RFC 7807) — The standard `application/problem+json` error shape (`type`, `title`, `status`, plus framework-specific extension keys like `code` / `category`) every unhandled exception renders as. `ProblemDetailsRenderer` is what produces it (Chapters 4, 12).

Projection — A read model built by consuming a stream of domain events, kept separate from the write model that produced them — Lumen's ledger is a projection over `Wallet`'s own domain events (Chapter 6).

Proxy (#[Transactional] proxy) — A generated subclass that wraps a `#[Transactional]`-attributed method with declarative transaction boundaries (`beginTransaction()` / `commit()` / `rollBack()`), produced by the same reflection-free `ProxyClassGenerator` whether at boot time (dev) or ahead of time by `firefly:cache` (production) (Chapters 9, 13).

Query (CQRS) — An object expressing a read intent — "get wallet balance" — dispatched through the `QueryBus` to exactly one `#[QueryHandler]`. Queries never mutate state (Chapter 7).

QueryBus — The read-side counterpart to `CommandBus`: receives a `Query`, routes it to its `#[QueryHandler]`, and (Chapter 7's pipeline) can serve a cached result transparently.

Recording double — A plain implementation of a real Firefly port that records what it saw, so a test can assert on behavior — `RecordingEventPublisher`, `RecordingCommandBus`, and eight siblings ship in `firefly/testing`, one per port. Laravel's own `Event::fake()` / `Bus::fake()` cannot see (Chapter 12).

Repository — A collection-like abstraction over persistence, letting application code load and save aggregates or entities with no SQL of its own. `CrudRepository` is the typed base every domain repository extends (Chapter 5).

Secure by default — LaraFly's operational posture wherever it applies: an actuator endpoint is unreachable (404) until explicitly exposed, an `HttpSecurity` URL rule denies anything unmatched, and a JWT secret that looks weak refuses to boot at all — the system fails closed, not open (Chapters 10, 11).

SecurityContext / SecurityContextHolder — An immutable snapshot of the current `Authentication`, held per-request by a static accessor backed by Laravel's own `Context` facade. Every authentication filter clears it in a `finally` block so one request's principal can never leak into the next (Chapter 10).

Service — A managed bean, `#[Service]`-attributed, that houses business logic and orchestrates calls to repositories, event publishers, and other services (Chapter 2).

Slice test — A test that boots only the beans a narrow vertical needs — the web pipeline over one controller (`WebSliceTestCase`), or the data pipeline over one repository (`DataSliceTestCase`) — rather than the whole application, so a mis-wired slice fails fast rather than passing by accident (Chapter 12).

Stereotype — An attribute that both registers a class as a bean and signals its architectural role — `#[Service]`, `#[Repository]`, `#[Component]`, `#[Configuration]`, `#[RestController]`. Every stereotype is a specialization of the base `#[Component]` (Chapter 2).

Testcontainers — A library that starts real Docker containers for integration tests and tears them down afterward. `firefly/testing`'s `RequiresDocker` trait and `fireflyConfigFor()` helper make a testcontainers-backed test opt-in and CI-safe even with no Docker present (Chapter 12).

Transactional (`#[Transactional]`) — A class/method attribute declaring propagation (`REQUIRED`, `REQUIRES_NEW`, `NESTED`, ...), isolation, read-only, and rollback rules — enforced by a generated proxy rather than a manual `DB::transaction()` closure (Chapter 9).

Value object — An immutable domain object identified by its value rather than by an identity field; two value objects are equal if all their fields are equal. `Money` — an integer amount in minor units plus a `Currency` — is Lumen's canonical value object (Chapter 6).

Zero-reflection boot — The state an application reaches once `firefly:cache` has run: every manifest is loaded from a pre-compiled PHP file under `bootstrap/cache/firefly/`, so no attribute, class, or method is ever reflected on again at request time (Chapter 13).