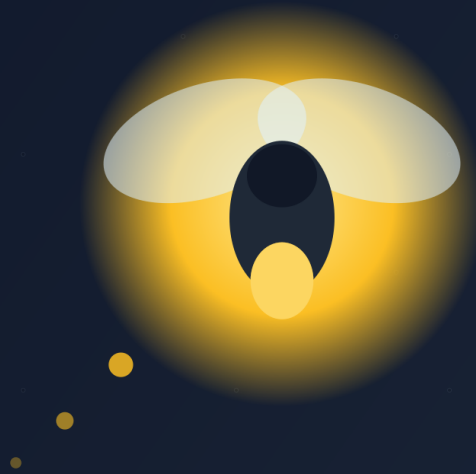


FIREFLY SOFTWARE SOLUTIONS INC.



LaraFly

by Example

Hexagonal PHP Microservices
on Laravel 13 with the Firefly Framework

LaraFly by Example

Microservicios PHP Hexagonales en Laravel 13 con el Firefly Framework

Firefly Software Solutions Inc.

Copyright © 2026 Firefly Software Solutions Inc.

Bajo licencia Apache License, Version 2.0 (la "Licencia"); no puedes usar este material salvo en cumplimiento de la Licencia. Puedes obtener una copia de la Licencia en <https://www.apache.org/licenses/LICENSE-2.0>.

Salvo que lo exija la ley aplicable o se acuerde por escrito, el software y la documentación distribuidos bajo la Licencia se distribuyen "TAL CUAL", SIN GARANTÍAS NI CONDICIONES DE NINGÚN TIPO, ni expresas ni implícitas. Consulta la Licencia para conocer el idioma específico que rige los permisos y las limitaciones bajo la Licencia.

Primera edición.

Cada listado de código de este libro procede del propio código fuente del framework LaraFly o de `samples/lumen`, la aplicación de monedero digital y libro mayor que se distribuye junto al monorepo del framework como acompañante, y se ha verificado que compila de forma limpia con un compilador PHP 8.3+ real: nada aquí es pseudocódigo ilustrativo.

"LaraFly" es el nombre de trabajo del port a PHP-sobre-Laravel del Firefly Framework. "Laravel" es una marca registrada de Taylor Otwell; esta es una obra independiente y no está afiliada ni respaldada por el proyecto Laravel.

Publicado por Firefly Software Solutions Inc.

Para todos los desarrolladores PHP que alguna vez conectaron a mano un contenedor de servicios, un enrutador, un validador y un despachador de eventos procedentes de cuatro paquetes sin relación entre sí — y luego lo volvieron a hacer, ligeramente distinto, en el siguiente proyecto.

Este es el único camino. Que os aproveche.

Prefacio

El PHP empresarial ha significado durante mucho tiempo ensamblar un contenedor de servicios de un paquete, un enrutador de otro, un validador de un tercero y un despachador de eventos de un cuarto — cada uno con su propio idioma de configuración, sus propias convenciones, su propia manera de decir "inyecta esto". Laravel resolvió gran parte de esa fragmentación para las aplicaciones web, pero el PHP hexagonal y orientado a servicios — inyección de dependencias basada en atributos, transacciones declarativas, CQRS, arquitectura orientada a eventos, seguridad a nivel de método, observabilidad de producción — ha seguido siendo cuestión de gustos, convenciones de nombres de carpetas y pegamento hecho a mano.

LaraFly cambia eso. Lleva a PHP 8.3+ y Laravel 13 la experiencia cohesionada de convención sobre configuración que Spring Boot dio al mundo Java, construida *sobre* Laravel en lugar de en su lugar: el mismo Eloquent, el mismo contenedor de servicios, el mismo `artisan`, ahora organizados mediante atributos de PHP 8 en componentes con estereotipo, un grafo de dependencias compilado y una tubería de arranque determinista.

Este libro enseña LaraFly **con ejemplos**. Cada listado de estas páginas no es pseudocódigo ilustrativo: está tomado del propio código fuente distribuido del framework o de `samples/lumen`, una aplicación real y en funcionamiento de monedero digital y libro mayor que se distribuye en el monorepo del framework, que compila, arranca y supera su propia batería de pruebas. Lo que lees es lo que realmente funciona.

Para quién es este libro

Este libro es para desarrolladores PHP de nivel intermedio que se sienten cómodos con los fundamentos de Laravel — modelos Eloquent, enrutamiento, el contenedor de servicios, `artisan` — y que quieren ver cómo se construye una arquitectura hexagonal de nivel empresarial *sobre* esa base, en lugar de *en* su lugar. No necesitas experiencia previa con Spring, CQRS ni arquitectura orientada a eventos: cada idea se presenta desde sus fundamentos antes del código que la implementa.

Los desarrolladores que llegan desde Spring Boot, Micronaut o Quarkus se sentirán especialmente como en casa. Allí donde un concepto de LaraFly refleja uno nativo de Laravel — un `#[Service]` junto a una clase corriente vinculada en un proveedor de servicios, un `#[RestController]` junto a un controlador Laravel normal — una nota de **paridad con Laravel** traza la comparación de forma explícita, para que puedas mapear lo que ya sabes sobre lo que es nuevo.

Lo que vas a construir

Cada capítulo hace avanzar **Lumen**, un servicio de monedero digital y libro mayor: un `Wallet` puede abrirse, recibir depósitos, sufrir retiradas y transferirse a otros monederos, protegiendo un único invariante por encima de todos los demás — **el saldo nunca es negativo** — y registrando cada cambio de estado como un evento de dominio que un escuchador proyecta en un libro mayor de solo anexado. Es un sistema pequeño, pero tiene exactamente la forma de uno real: una capa de dominio sin dependencia alguna del framework, un puerto hexagonal y su adaptador Eloquent, manejadores de comando y consulta de CQRS, eventos de dominio conectados a un bus de eventos, seguridad a nivel de método en una operación sensible, y un controlador REST delgado que no contiene lógica de negocio propia.

El recorrido empieza con suavidad. El **Inicio rápido** te lleva desde el instalador incluido `firefly new` hasta un endpoint en ejecución y consultable con curl, previendo en miniatura los estereotipos, el contenedor y la ruta de arranque compilada antes de que ningún capítulo te pida razonar sobre ellos. El **Capítulo 1** da un paso atrás y argumenta el enfoque completo — qué problema resuelve LaraFly y sobre qué pilares se sostiene. El **Capítulo 2** abre la sala de máquinas: el contenedor de inyección de dependencias, los atributos de estereotipo y el escaneo de componentes que compila tus clases anotadas en un manifiesto de arranque en caché, sin reflexión. Las partes posteriores de este libro — que llegarán en los capítulos siguientes — construyen hacia afuera desde esa base hacia la configuración, HTTP, la persistencia, el modelado de dominio, CQRS, la arquitectura orientada a eventos, la seguridad y la observabilidad, siempre a través de la misma base de código de `Lumen`, siempre con código que puedes ejecutar.

Cuando hayas terminado de trabajar todo el libro, tendrás un modelo mental de cada capa de un servicio LaraFly de producción, y una aplicación real y probada que lo demuestre.

Cómo usar este libro

Lee de forma secuencial. Cada capítulo se apoya en el vocabulario y el código del anterior.

Escribe tú mismo los listados. Leer y teclear el código a la vez es la manera de que los patrones se asienten. Resiste la tentación de copiar y pegar hasta que hayas escrito cada listado al menos una vez a mano.

Ejecútalo. Todos los listados de este libro son reales, y la mayoría vive, sin cambios, en el propio paquete `samples/lumen` del framework. Clona el repositorio del framework y podrás ejecutar la aplicación terminada — `composer test -- samples/lumen/tests` ejercita toda la batería de pruebas del ejemplo — y comparar tu propio trabajo con ella en cualquier momento.

Cada capítulo cierra con un breve **Resumen** de lo aprendido y, cuando corresponde, una sección de **Ponlo en práctica** que te lleva un paso más allá por tu cuenta.

Convenciones en breve

Las convenciones tipográficas y estructurales —el estilo de los listados de código, los tipos de recuadro y la numeración de las figuras— se demuestran, con ejemplos en vivo, en la sección **Convenciones** que sigue a este prefacio.

El código de acompañamiento

La aplicación `Lumen` terminada vive en el monorepo del framework, en `samples/lumen`. Es el destino al que este libro te conduce: un proyecto por capas — `Domain`, `Application`, `Infrastructure`, `Web` — que los capítulos posteriores de este libro hacen crecer, funcionalidad a funcionalidad. Úsala para comparar tu propio código, para ponerte al día si un capítulo avanza más rápido de lo esperado, o simplemente para ejecutar las partes sobre las que estás leyendo en cada momento.

Convenciones

Esta página explica las convenciones tipográficas y estructurales que se usan a lo largo del libro.

Listados de código

El código real se presenta en un bloque delimitado plano etiquetado `php`, y **cada uno de esos listados dice de dónde viene**. Justo encima de cada delimitador hay un comentario HTML — invisible en la página renderizada, pero la razón por la que puedes fiarte de lo que le sigue — y hay exactamente dos clases:

`<!-- source: <ruta> -->` marca **código del framework**: el bloque es un extracto *literal* de ese fichero del repositorio de LaraFly — las mismas líneas, en el mismo orden, con la misma indentación relativa. `tests/DocsCodeIsRealTest.php` compara cada bloque de esa clase contra el fichero que nombra en cada ejecución, así que un listado no puede desviarse cuando el código se mueve; y el fichero que cita ya está sujeto a las propias puertas del repositorio — PHPStan en nivel máximo, Pint y la suite de Pest del paquete. Nada dentro de uno de estos bloques está parafraseado, aseado para la página ni inventado.

`<!-- illustrative: <por qué> -->` marca **tu código**: la clase que una persona lectora escribe en su propia aplicación, que por definición no puede existir en este repositorio, así que no hay fichero contra el que compararla. Esos bloques son los que se pasan por el CLI real de PHP (`php -l`), y cada `use Firefly\...` que importan y cada `#[Attribute]` que usan se comprueba contra las clases que el framework declara de verdad.

Las dos clases se comprueban además por lo que *afirman*: cada clave de configuración `firefly.*`, cada comando `php artisan firefly:*` y cada `composer <script>` que un listado nombre tiene que ser uno que exista de verdad.

Elisiones (`// ...`)

Un extracto `source:` puede **cortar** líneas enteras que no necesita — una lista de imports, un docblock largo, el centro de un método. Cada corte se marca con una línea que es exactamente `// ...` (o `# ...` en un fichero, como YAML, cuyos comentarios empiezan por almohadilla), y la comparación se reanuda después de lo que ya ha casado, así que lo que queda sigue siendo, en orden, lo que dice el fichero. Tres cosas que un corte nunca puede hacer, y que la guarda rechaza: tragarse la declaración a la que pertenece — dejando una `{` sin nada encima que diga *qué* se está declarando —, vaciar un método hasta dejarlo con cuerpo vacío, lo que imprimiría un método que parece no hacer nada, o dejar un docblock en pie por su cuenta. Un docblock es una afirmación *sobre* una declaración, así que un extracto que muestre uno muestra su `/` `**`, su línea de cierre, y al menos una línea de la clase, el método o la constante que describe; empezar a mitad de un docblock imprimiría un párrafo de comentario sobre código que la página nunca enseña.

Por ejemplo, este es el objeto de valor `Money` de `samples/lumen`, entero hasta `zero()` y cortado ahí:

```
<?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);
    }
    // ...
}

```

Las referencias de código en línea dentro de la prosa usan fuente `monoespaciada`, como en "el atributo `#[Service]` registra la clase en el contenedor de LaraFly".

Recuadros

A lo largo del libro aparecen cuatro estilos de recuadro:

NOTA

Las notas aportan contexto complementario o aclaran un matiz del texto principal: merece la pena leerlas, pero no son bloqueantes.

CONSEJO

Los consejos comparten un atajo, un idioma o una buena práctica que te ahorrará tiempo en proyectos reales.

ADVERTENCIA

Las advertencias señalan un error habitual o un punto delicado que puede provocar problemas difíciles de depurar si se ignora.

PARIDAD CON LARAVEL

Los recuadros de paridad con Laravel relacionan un concepto de LaraFly directamente con su equivalente nativo en Laravel — ideales si llegas desde Laravel puro en lugar de desde Spring Boot u otro framework de convención sobre configuración.

Figuras

Los diagramas llevan un pie debajo de la imagen y se incrustan como SVG en línea, de modo que se renderizan con nitidez a cualquier nivel de zoom tanto en la edición en pantalla como en la impresa. Te encontrarás con la primera en el Capítulo 2.

Contenido

Inicio rápido

Construye Lumen paso a paso	16
-----------------------------------	----

PARTE I Fundamentos

1. ¿Por qué LaraFly?	31
2. Inyección de Dependencias y Auto-Configuración.....	40
3. Configuración, Perfiles y Secretos	60
4. Tu Primera API HTTP	78
4A. Documentar la API: OpenAPI 3.1 desde los Manifiestos.....	110

PARTE II Modelar y Persistir el Dominio

5. Persistencia y el Patrón Repositorio	156
6. Diseño Guiado por el Dominio	201

PARTE III Coordinar y Asegurar la Aplicación

7. CQRS: Comandos y Consultas.....	233
8. Arquitectura Guiada por Eventos y el Outbox Transaccional.....	253
9. Transacciones y el Proxy #[Transactional].....	283
10. Seguridad: Autenticación y Autorización	316
10A. OAuth2 y OpenID Connect: Iniciar Sesión, y Ser el Proveedor.....	371

PARTE IV Observabilidad, Pruebas y Entrega

11. Observabilidad: Salud, Métricas y el Actuador	417
12. Probar Aplicaciones LaraFly	482

13. La CLI y la Caché Sin Reflexión	521
---	-----

Apéndices

A. Hoja de Referencia Laravel → LaraFly.....	544
Glosario	556

Inicio rápido

INICIO RÁPIDO

Construye Lumen paso a paso

Bienvenido. Antes de la inmersión profunda, este capítulo te lleva desde una *terminal vacía* hasta una aplicación LaraFly *en ejecución y consultable con curl* — instalada, compilada y servida — en bastante menos de diez minutos. Cada comando y cada listado de este capítulo son reales: es exactamente lo que genera `firefly new`, sin modificar.

Esto es un *recorrido*, no la inmersión profunda. El Capítulo 1 argumenta el enfoque completo; el Capítulo 2 abre la sala de máquinas y explica, en profundidad, todo lo que aquí solo mencionas de pasada — el contenedor, los estereotipos y el manifiesto de arranque compilado. El objetivo de este capítulo es el impulso inicial: al terminarlo tendrás un servicio real en ejecución, y una primera sensación informal de lo que significa "convención sobre configuración" en LaraFly.

NOTA

Todos los listados de este capítulo se copian literalmente de `skeleton/`, el andamiaje de proyecto que instala `firefly new`, y de `samples/lumen`, la aplicación más completa de monedero digital y libro mayor que este libro construye a lo largo de los capítulos siguientes.

Paso 1 — Requisitos previos

LaraFly es un framework PHP construido sobre Laravel 13. Necesitas:

- **PHP 8.3 o más reciente**, con las extensiones que el propio Laravel requiere (`mbstring`, `openssl`, `pdo`, `tokenizer`, `xml`, `ctype`, `json`).

- **Composer 2**, el gestor de dependencias de PHP.

Confirma que ambos están disponibles:

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

Paso 2 — Instalación

El instalador `firefly new` entrega a Composer la plantilla incluida en la biblioteca:

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

`composer create-project` hace más que copiar archivos: ejecuta un script `post-create-project-cmd` que termina de preparar el proyecto por ti —

```
"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"
]
```

Para cuando ese comando termina, `.env` ya existe, se ha creado el archivo de base de datos SQLite, `APP_KEY` está configurada y — el paso que más importa para este libro — **firefly:cache ya ha compilado los manifiestos de tu aplicación**. Todavía no has escrito una sola línea de PHP, y la ruta de arranque sin reflexión sobre la que se construye todo este framework ya está en su sitio.

💡 EL INSTALADOR INCLUIDO

`composer global require fireflyframework/larafly` te da un comando `firefly` en tu PATH. `firefly new my-app` entrega a Composer la plantilla incluida en la biblioteca, y luego ejecuta `git init` y un commit inicial por ti — el equivalente en LaraFly de `laravel new`.

Qué acabas de instalar

El `composer.json` del andamiaje requiere la biblioteca principal y declara un requisito del componente CLI que esa biblioteca satisface:

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

`firefly/cli` te da los comandos `artisan firefly:*` que usarás a lo largo de este libro. `fireflyframework/larafly` es la **biblioteca completa del framework**, que contiene toda la familia Firefly (contenedor, contexto, configuración, web, datos, cqrs, eda, seguridad, validación, resiliencia, programación, observabilidad, actuador y más) en una sola línea `require`, de modo que tu propio `composer.json` nunca tiene que enumerarlos uno a uno.

📦 PARIDAD CON LARAVEL

`firefly new` es el equivalente en LaraFly de `laravel new` — y `fireflyframework/laraflly` es el equivalente de instalar el propio `laravel/framework`: una línea de dependencia que trae una pila completa y coherente en lugar de una colección de piezas versionadas de forma independiente.

Paso 3 — Un vistazo por dentro

Abre el proyecto generado. Hay dos cosas que merece la pena notar de inmediato, porque son la idea completa de este framework en miniatura.

Primero, `bootstrap/providers.php` está vacío:

```
<?php

declare(strict_types=1);

return [];
```

No hay ningún proveedor de servicios que registrar a mano. Los propios proveedores de `firefly/cli` y `fireflyframework/laraflly` se descubren automáticamente mediante el descubrimiento de paquetes de Composer/Laravel — nunca añades una línea aquí por un paquete de Firefly.

Segundo, `routes/web.php` también está casi vacío:

```
// 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 withRouter(web: ...) requires the path.
```

Aquí no se declara ninguna ruta. Como dice el comentario, vienen de un

RouteManifest compilado — construido a partir de atributos en tus controladores, no de un archivo de rutas que mantienes a mano. Enseguida conocerás la clase a la que apunta ese comentario.

El único archivo que sí importa ahora mismo es `config/firefly.php`. La copia que entrega el esqueleto es una **referencia** larga y anotada — cada clave `firefly.*` que el framework lee, agrupada por el paquete que la lee, con su valor por defecto real al lado — así que aquí se muestran los dos bloques que importan hoy y se recorta el resto del archivo. Cada `// ...` de abajo representa líneas enteras eliminadas del archivo real; la página de convenciones explica el marcador:

```
<?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` le dice al escáner de componentes de LaraFly qué raíces PSR-4 mirar — aquí, solo tu propio namespace `App\` bajo `app/`. `cache.path` es el lugar donde `firefly:cache` (que ya ejecutaste, en el Paso 2) escribió los manifiestos compilados que encontró allí.

Paso 4 — Conoce el primer bean

`app/GreetingProperties.php` es un pequeño DTO de configuración tipada — la respuesta de LaraFly a `@ConfigurationProperties` de Spring:

```
<?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` es una clase PHP corriente que lleva un único atributo, `#[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);
|     }
| }
```

Nada aquí registra `GreetingService` en un contenedor a mano, y nada conecta `GreetingProperties` en su constructor tampoco. Basta con `#[Service]`: el escaneo de componentes encuentra la clase, el contenedor la construye, y como el constructor pide un `GreetingProperties`, el contenedor resuelve e inyecta uno automáticamente. Esto es **inyección de dependencias por constructor**, y es el tema de todo el Capítulo 2.

Paso 5 — Exponlo por HTTP

`app/Http/GreetingController.php` le pone un borde web al servicio:

```
<?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]` es en sí mismo un estereotipo — está construido sobre la misma base `#[Component]` que `#[Service]` — así que `GreetingController` se autoconecta exactamente igual que `GreetingService`: el contenedor lo construye, ve que el constructor quiere un `GreetingService`, y le proporciona uno. `#[GetMapping]` y `#[PathVariable]` son lo que el escaneo de rutas lee, por separado, para construir el `RouteManifest` al que se refería el comentario de `routes/web.php`.

Paso 6 — Ejecútalo

Arranca el servidor de desarrollo:

```
php artisan firefly:serve
```

`firefly:serve` es un envoltorio delgado: llama a `artisan serve` (o a `octane:start`, si `laravel/octane` está instalado) — no reimplementa nada por su cuenta. En otra terminal, prueba la ruta que acabas de leer:

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

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

"Hello" es el valor por defecto de `$salutation` en `GreetingProperties` — nada en `config/greeting.php` lo sobrescribe todavía, así que lo que ves es el valor por defecto del constructor. Cambia ese valor por defecto, o vincula `greeting.salutation` en tu propia configuración, y cada respuesta lo reflejará, sin ningún cambio de código ni en el servicio ni en el controlador.

/ no es una ruta de este controlador, y el docblock de arriba dice qué clase la posee: `App\Http>WelcomeController`, un `#[Controller]` y no un `#[RestController]`, cuyo `index()` devuelve una vista. Abre `localhost:8000/` en un navegador y lo que vuelve es la página de bienvenida HTML del esqueleto, no JSON — esa diferencia entre los dos estereotipos es precisamente de lo que te avisa el docblock.

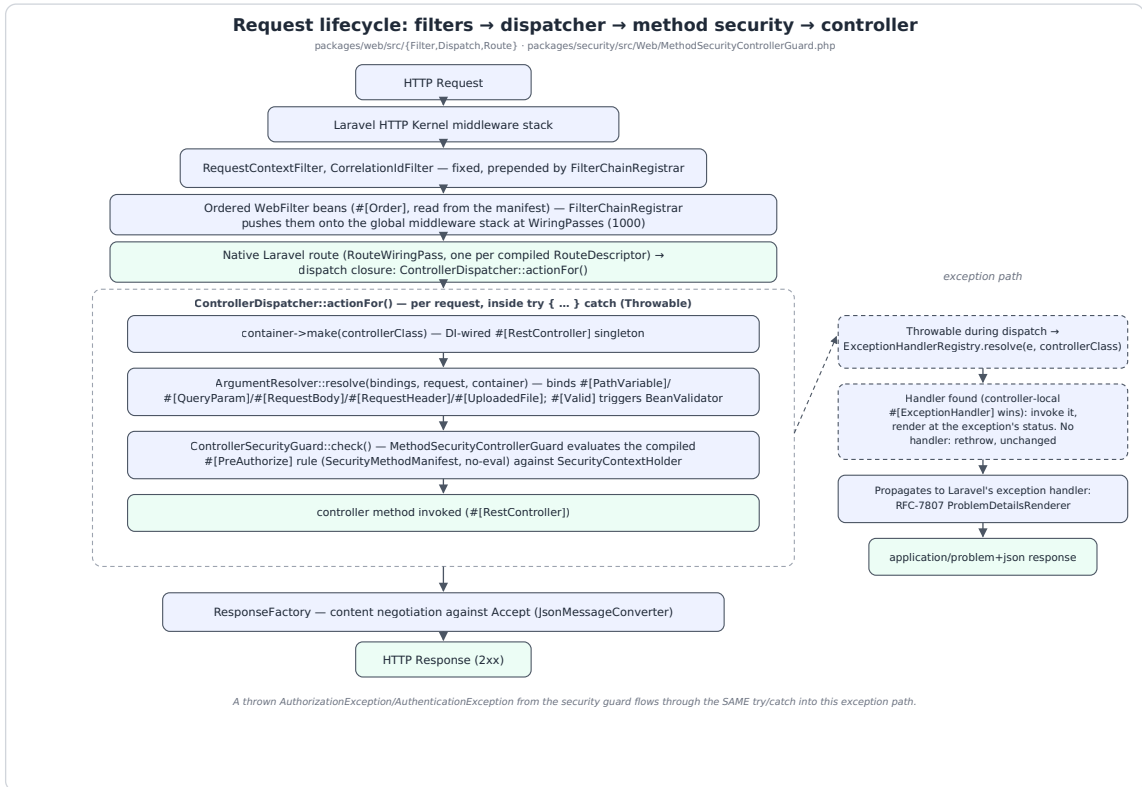


Figura 0.1 — Una petición atraviesa la cadena de filtros web y el despachador del controlador antes de que tu método manejador se ejecute.

DOS COMANDOS, UN SOLO TRABAJO

Ya ejecutaste `firefly:cache` una vez, de forma indirecta, durante `composer create-project`. Si añades o cambias un atributo — un nuevo `#[Service]`, una nueva ruta — vuelve a ejecutar `php artisan firefly:cache` para recompilar los manifiestos, o `php artisan firefly:clear` para eliminarlos y volver a la ruta de escaneo de desarrollo (más lenta, basada en reflexión). El Capítulo 2 explica exactamente qué produce cada comando.

Paso 7 — Un vistazo a hacia dónde va esto

`GreetingService` es deliberadamente la porción más pequeña posible: un bean, una dependencia, una ruta. La aplicación que este libro realmente construye, capítulo a capítulo, es más grande — **Lumen**, un servicio de monedero digital y libro mayor que ya existe, completamente construido y probado, en el propio paquete `samples/lumen` del framework. Expone:

Método	Ruta	Propósito
POST	<code>/api/v1/wallets</code>	Abrir un monedero
POST	<code>/api/v1/wallets/{id}/deposit</code>	Depositar fondos
POST	<code>/api/v1/wallets/{id}/withdraw</code>	Retirar fondos — protegido
POST	<code>/api/v1/wallets/transfers</code>	Transferir fondos entre dos monederos
GET	<code>/api/v1/wallets/{id}/balance</code>	Consultar solo el saldo
GET	<code>/api/v1/wallets/{id}/ledger</code>	Consultar el libro mayor proyectado del monedero

Cada uno de esos endpoints está construido a partir del puñado de ideas que acabas de conocer en miniatura — una clase con estereotipo, inyección por constructor, un atributo que el framework lee en el momento del escaneo — más muchas cosas más: un puerto de repositorio hexagonal y su adaptador Eloquent, manejadores de comando y consulta de CQRS, eventos de dominio proyectados en un libro mayor, y seguridad a nivel de método en el endpoint de retirada. Puedes ejecutar toda su batería de pruebas ahora mismo, antes de leer una página más, si quieres verla funcionar:

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

El resto de este libro construye exactamente esa aplicación contigo, una idea cada vez, empezando por la idea que ya has usado dos veces sin una explicación completa: el contenedor.

✓ Lo que construiste

Pasaste de una terminal vacía a una aplicación LaraFly compilada, en ejecución y consultable con curl, y conociste — sin necesitar todavía la explicación completa — un `#[Service]`, un DTO `#[ConfigProperties]`, un `#[RestController]`, la inyección de dependencias por constructor y el paso de compilación `firefly:cache` que convierte todo eso en un manifiesto de arranque sin reflexión.

En este Inicio rápido...	Se profundiza en
Instalaste con <code>firefly new</code>	Capítulo 1 — ¿Por qué LaraFly?
Viste cómo <code>#[Service]</code> , <code>#[ConfigProperties]</code> y <code>#[RestController]</code> registran beans sin conexión manual	Capítulo 2 — Inyección de Dependencias y Auto-Configuración
Ejecutaste <code>firefly:cache</code> y viste rutas servidas desde un manifiesto compilado, no desde <code>routes/web.php</code>	Capítulo 2 — Inyección de Dependencias y Auto-Configuración

Cuando estés listo para el *porqué* de todo esto, pasa la página al Capítulo 1.

✎ Ponlo en práctica

1. **Cambia el saludo por defecto.** Edita el valor por defecto del constructor de `GreetingProperties` de `'Hello'` a otra cosa, vuelve a ejecutar `php artisan firefly:cache`, reinicia `firefly:serve` y confirma que la respuesta cambia sin tocar ningún otro código.

2. **Añade una tercera ruta.** Dale a `GreetingController` un método `farewell(string $name)` mapeado a `#[GetMapping('/farewells/{name}')]` que devuelva `['message' => "Goodbye, {$name}!"]` directamente — todavía no necesitas ningún servicio nuevo.
3. **Lee el `README.md` del ejemplo Lumen** (`samples/lumen/README.md`) antes del Capítulo 1 — presenta, en prosa, todo lo que el resto de este libro construye en código.

PARTE I

Fundamentos

¿Por qué LaraFly?

Al terminar este capítulo entenderás el problema que LaraFly existe para resolver, los siete pilares sobre los que descansa su arquitectura, y cómo cada uno se construye sobre Laravel en lugar de sustituirlo — de modo que cuando el Capítulo 2 abra el contenedor de inyección de dependencias, ya sabrás *por qué* funciona como funciona.

El problema de la cohesión

Imagina el primer día de un nuevo microservicio PHP. El propio Laravel responde a la mayoría de las preguntas obvias — enrutamiento, el ORM, el contenedor de servicios, la cola — y las responde bien. Pero en el momento en que el servicio necesita ser algo más que una aplicación CRUD-y-Blade, aparece una segunda capa de preguntas, y Laravel las deja deliberadamente en tus manos.

¿Cómo mantienes las reglas de negocio fuera de tus modelos Eloquent y de tus controladores? ¿Dónde vive un manejador de comando, y cómo se despacha? ¿Cómo garantizas que una escritura en base de datos y el evento de dominio que provoca ocurran ambos o ninguno? ¿Cómo publicas ese evento a otros servicios sin fabricar a mano un trabajo de cola para cada uno? ¿Cómo bloqueas una operación concreta a un rol concreto, independientemente del middleware de ruta? ¿Cómo sabes, en producción, si el servicio está realmente sano?

Cada equipo responde estas preguntas de forma distinta — una convención de nombres de carpetas por aquí, una clase base de repositorio por allá, una capa de "servicio" hecha en casa sin forma compartida de un proyecto al siguiente. Seis meses después, un segundo equipo empieza un segundo servicio y toma decisiones

completamente distintas. Ahora dos bases de código comparten un lenguaje y un framework pero nada más: estrategias de pruebas distintas, convenciones de manejo de errores distintas, ningún modelo mental compartido de cómo está conectado nada.

Laravel le da a PHP una base excelente. Por sí mismo, no te da una arquitectura compartida para la capa que va encima de esa base.

Los desarrolladores Java resolvieron el problema equivalente hace años con Spring Boot: un único framework cohesionado y con opiniones, construido *sobre* la JVM en lugar de sustituirla, que toma las decisiones arquitectónicas difíciles una sola vez y deja que cada equipo comparta la misma respuesta. **LaraFly lleva esa misma disciplina a PHP, construida sobre Laravel 13.**

Qué es LaraFly

LaraFly no es un sustituto de Laravel — es un framework construido *sobre* él, de la misma manera que Spring Boot está construido sobre la JVM y las convenciones planas de Java EE. Toda aplicación LaraFly es una aplicación Laravel real: los mismos modelos Eloquent, el mismo contenedor de servicios por debajo, la misma línea de comandos `artisan`, la misma tubería de middleware y kernel HTTP. Lo que LaraFly añade es una segunda capa, de nivel más alto, de **atributos** de PHP 8 que describen *intención* — "esta clase es un servicio", "este método debe ejecutarse dentro de una transacción", "este manejador de comando requiere el rol `ADMIN`" — y un motor en tiempo de arranque que lee esos atributos una sola vez, los compila en un manifiesto y conecta la aplicación resultante de forma determinista.

Ya viste una pequeña muestra de esto en el Inicio rápido: `#[Service]` en una clase corriente bastó para hacerla inyectable, sin ninguna conexión manual. Ese único atributo es el ejemplo más pequeño de un patrón que se repite, con mayor profundidad, a lo largo del resto de este libro: **declara qué es una clase, y deja que el framework averigüe qué hacer con ella.**

⬆ PARIDAD CON LARAVEL

Nada en LaraFly te pide que abandones lo que ya sabes. Un bean `#[Service]` sigue siendo, al final, un objeto corriente que resuelve el contenedor de Laravel; la ruta de un `#[RestController]` sigue ejecutándose a través del propio kernel HTTP y middleware de Laravel; `#[Transactional]` (Capítulo 8) sigue llamando por debajo a los propios `DB::beginTransaction() / commit() / rollBack()` de Laravel. Los atributos de LaraFly describen *qué conectar*; el trabajo real de la conexión lo sigue haciendo Laravel.

Siete pilares

La arquitectura de LaraFly descansa sobre siete ideas. Cada una es un paquete real e independiente en el monorepo del framework — ninguna de ellas requiere las demás — y cada una es el tema de uno o más capítulos posteriores.

1. DI y auto-configuración basadas en atributos

`#[Service]`, `#[Repository]`, `#[Component]` y `#[Configuration]` / `#[Bean]` marcan una clase o un método fábrica como parte gestionada de la aplicación. Un **escaneo de componentes** — un único paso de reflexión — descubre cada uno de ellos y compila el resultado en un manifiesto PHP en caché. En tiempo de ejecución no se vuelve a escanear nada: el contenedor se construye a partir de ese manifiesto congelado, que es lo que hace que LaraFly sea seguro y rápido bajo workers de larga duración, no solo bajo el clásico PHP-FPM. Este es el tema completo del Capítulo 2.

2. `#[Transactional]` — demarcación declarativa de transacciones

Un método que lleva `#[Transactional]` obtiene un proxy generado, construido en tiempo de compilación, que abre una transacción de base de datos real de Laravel antes de que el método se ejecute y la confirma o la revierte después — usando `DB::beginTransaction() / commit() / rollBack()` manuales en lugar de `DB::transaction($closure)`, específicamente para que una excepción capturada pueda confirmarse-y-relanzarse cuando tus propias reglas de reversión digan que debe hacerlo, en lugar de revertirse siempre. Tú escribes el método de negocio; el proxy lleva el límite transaccional.

3. CQRS — comandos, consultas y un bus

Las escrituras y las lecturas viajan por dos caminos separados. Un `#[CommandHandler]` maneja una intención de escritura; un `#[QueryHandler]` responde una lectura; ambos se despachan a través de un bus en lugar de llamarse directamente. Un manejador de comando `#[Transactional]` obtiene su semántica transaccional gratis del mismo proxy que describe el pilar 2 — CQRS no añade lógica de interceptación propia.

4. Arquitectura orientada a eventos, con un outbox genuino

Un agregado emite eventos de dominio que describen lo que ocurrió; tras una confirmación, esos eventos pueden proyectarse en el mismo proceso o conectarse hacia afuera al `EventPublisher`, respaldado por bróker, de `firefly/eda` — en memoria para las pruebas, una cola de Laravel para entrega asíncrona, o un bróker de mensajes real (Kafka, RabbitMQ, o `LISTEN / NOTIFY` de Postgres) para entrega entre servicios. El adaptador de Postgres en particular escribe la fila del outbox en la *misma* transacción de base de datos que el propio agregado, de modo que un evento nunca puede registrarse sin que la escritura que describe se haya confirmado realmente.

5. Seguridad

Un modelo de principal con la forma de Spring Security 6 — una `Authentication` inmutable, un `SecurityContext`, `GrantedAuthority` — respalda una configuración de URL `HttpSecurity` que deniega por defecto y guardas a nivel de método como `#[PreAuthorize]`, aplicadas en el bus de CQRS y en el despacho del controlador, independientemente del middleware de ruta.

6. Actuator — una superficie de gestión de producción

`/actuator/health`, `/actuator/info` y similares, modelados directamente sobre Spring Boot Actuator: un SPI `HealthIndicator`, grupos de sondas de liveness/readiness, y un paso de registro de rutas que monta estos endpoints sin cambios de código en tu propia aplicación — solo la configuración decide qué se expone.

7. Arranque sin reflexión

Cada uno de los seis pilares anteriores se descubre por reflexión **exactamente una vez** — en el momento de `firefly:cache`, no en el momento de cada petición. Los manifiestos compilados que produce ese escaneo son de donde realmente arranca una aplicación en ejecución. Este es el hilo que une todo el framework, y es donde empieza el Capítulo 2.

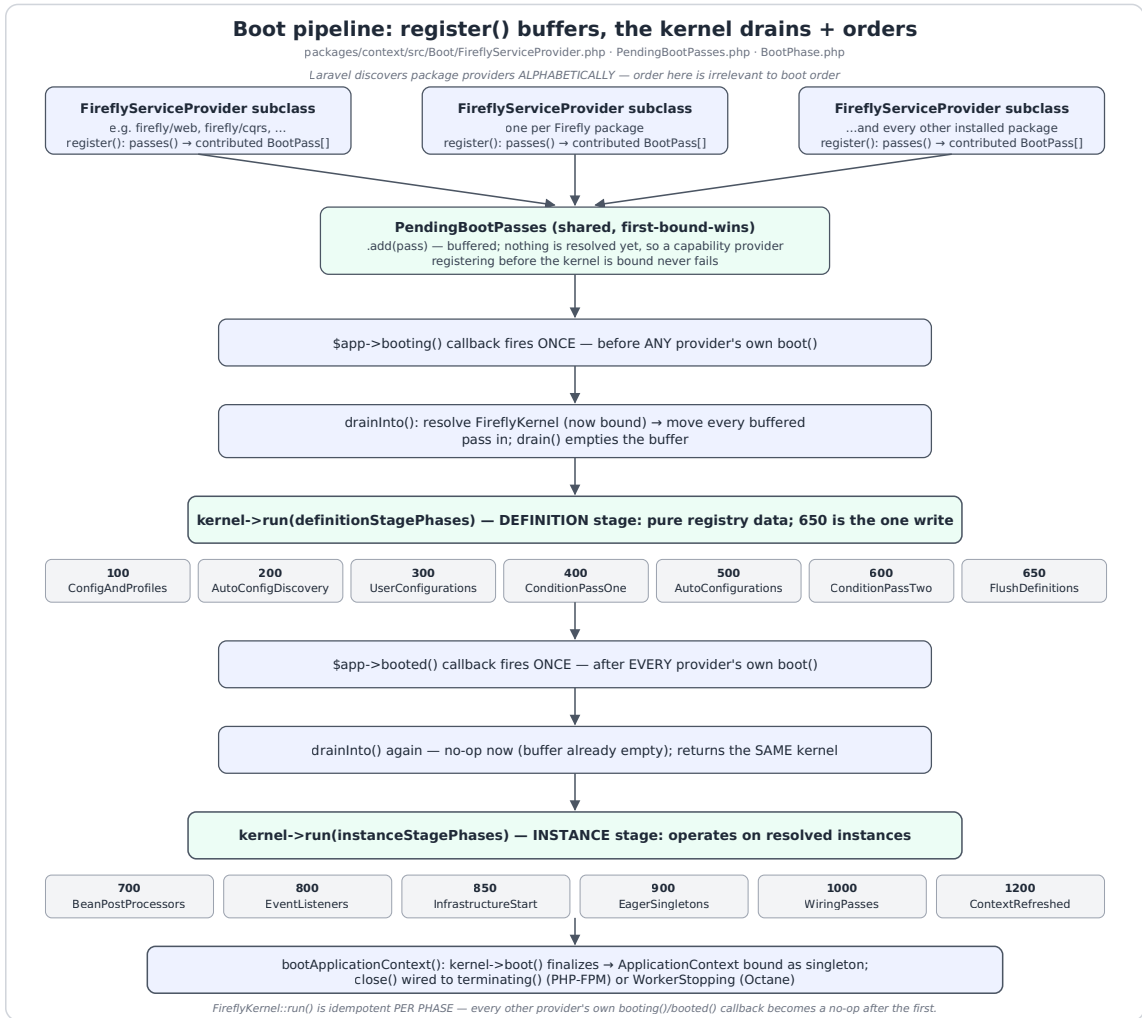


Figura 1.1 — Cada paquete de Firefly aporta un paso de arranque; solo el kernel decide el orden en que se ejecutan.

Por qué construir Lumen

El resto de este libro enseña estos siete pilares construyendo una única aplicación real, **Lumen**, un servicio de monedero digital y libro mayor — el mismo que entrevistaste al final del Inicio rápido. Un monedero puede abrirse, recibir depósitos, sufrir retiradas y transferirse a otros monederos, y protege exactamente una regla por encima de todas las demás: **el saldo nunca es negativo**. Esa única regla es lo bastante pequeña como para retenerla por completo en la cabeza, lo que deja tu atención libre para cómo se construye cada pilar, en lugar de para qué significa el dominio.

Cada listado que leerás a partir de ahora no es ilustrativo: está copiado del propio paquete `samples/lumen` que se distribuye en el monorepo del framework, y se valida con el linter, compila y supera su propia batería de pruebas exactamente como aparece impreso.

NOTA

Lumen ya existe, completamente construido, en `samples/lumen` — este libro no lo modifica. En cambio, cada capítulo *explica* una porción más de él, en el orden que hace más claras las ideas subyacentes, de modo que cuando hayas leído todo el libro entiendas cada línea de un servicio real con forma de producción.

✓ Lo que aprendiste

Pilar	Qué te da	Dónde se cubre
DI y auto-configuración basadas en atributos	Beans con estereotipo, inyección por constructor, un manifiesto compilado	Capítulo 2
<code>#[Transactional]</code>	Límites transaccionales declarativos mediante un proxy generado	Capítulos posteriores
CQRS	Un bus de comandos/consultas que separa escrituras de lecturas	Capítulos posteriores
EDA + outbox genuino	Eventos de dominio conectados a entrega en memoria, cola o bróker	Capítulos posteriores
Seguridad	Seguridad HTTP que deniega por defecto + guardas a nivel de método	Capítulos posteriores
Actuator	Una superficie de salud/información/gestión de producción	Capítulos posteriores
Arranque sin reflexión	Un paso de compilación, y después un manifiesto congelado en tiempo de ejecución	Capítulo 2

Ponlo en práctica

1. **Lee la propuesta de un tirón.** Repasa `samples/lumen/README.md` de principio a fin. Para cada punto bajo "What the sample shows", identifica a cuál de los siete pilares anteriores pertenece.
2. **Encuentra la costura.** Abre de nuevo `skeleton/config/firefly.php` y localiza `scan.paths`. Cada uno de los siete pilares depende, en última instancia, de que ese único array sea correcto — escribe, con tus propias palabras, qué pasaría si apuntara al namespace equivocado.
3. **Compara arquitecturas.** Si alguna vez has construido un servicio en Laravel puro, enumera tres decisiones arquitectónicas que tomaste por convención (dónde viven los comandos, cómo manejaste las transacciones, cómo aseguraste un endpoint sensible). Revisarás cada una de ellas, construida a la manera LaraFly, en un capítulo posterior.

Inyección de Dependencias y Auto-Configuración

Al terminar este capítulo conocerás los cuatro atributos de estereotipo, cómo la inyección por constructor resuelve las dependencias, cómo desambiguar entre varias implementaciones de una interfaz con `#[Primary]` y `#[Qualifier]`, cómo controlar el orden de las listas con `#[Order]`, cómo inyectar configuración y expresiones con `#[Value]`, qué significan los tres ámbitos de componente y — atando todo junto — qué ocurre exactamente entre `php artisan firefly:cache` y que tu aplicación sirva su primera petición.

❗ TÉRMINO NUEVO: CONTENEDOR

Un **contenedor de inyección de dependencias** es un objeto que sabe cómo construir otros objetos, y te entrega una instancia ya construida en lugar de que tú llames a `new` y conectes sus dependencias a mano. El contenedor de LaraFly es

`Illuminate\Container\Container` — el mismo contenedor que ya usa una aplicación Laravel corriente — con una incorporación: los atributos de PHP 8 le dicen *qué* construir, así que rara vez llamas tú mismo a `$container->bind(...)`.

Estereotipos: declarar un bean

Un **bean** es cualquier objeto que el contenedor gestiona en tu nombre — construido una vez (por defecto) y entregado cuando se solicita. Marcas una clase como bean con un atributo de **estereotipo**. LaraFly distribuye cuatro:


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

`#[Component]` es el atributo base; `#[Service]`, `#[Repository]` y `#[Configuration]` son todas especializaciones de él — literalmente, cada uno `extends Firefly\Container\Attributes\Component` en PHP. Un escaneo de componentes encuentra cada uno de ellos con una única llamada de reflexión:

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

`ReflectionAttribute::IS_INSTANCEOF` es lo que hace que esto funcione: coincide con `#[Component]` en sí mismo y con cualquier atributo que extienda de él, así que el escáner nunca tiene que enumerar los estereotipos por nombre. Esta es también la razón por la que `#[RestController]` de `firefly/web` — el atributo que conociste en `GreetingController` en el Inicio rápido — participa en ese mismo escaneo: él también `extends Component`.

Ya conociste el estereotipo más sencillo en el Inicio rápido:

```
<?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]` es el estereotipo para lógica de aplicación/negocio — el equivalente en LaraFly del `@Service` de Spring. No hay nada más que escribir: ningún `$this->app->bind(GreetingService::class, ...)` en ningún sitio, ningún cierre fábrica. El atributo es todo el registro.

`#[Repository]` sobre un puerto hexagonal

`#[Repository]` marca un bean orientado a la persistencia, y es donde los estereotipos empiezan a mostrar su valor real: desambiguar entre una interfaz y la clase que la implementa. `samples/lumen` define el *puerto* — la interfaz de la que dependen las capas de dominio y de aplicación — sin ningún atributo del framework encima, porque una interfaz nunca es un bean en sí misma:

```
<?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;
}

```

El *adaptador* — la clase que realmente habla con Eloquent — es el que lleva `#[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 findByIdOwnerId(string $ownerId): array
    {
        /** @var list<Wallet> $result */
        $result = $this->dispatchQuery('findByIdOwnerId', [$ownerId]);

        return $result;
    }
}
```

Nada en `samples/lumen` vincula nunca `WalletRepository` a `EloquentWalletRepository` a mano. Cuando el constructor de un bean pide `WalletRepository`, el contenedor necesita saber a qué clase *concreta* entregarle — y esto es la **auto-conexión de interfaces**: `ContainerRegistrar::wireInterfaces()`

observa las interfaces implementadas por cada componente escaneado y, para cada interfaz con exactamente un componente que la implemente (o exactamente uno marcado `#[Primary]`, que veremos más adelante en este capítulo), la vincula automáticamente.

🏠 PARIDAD CON LARAVEL

En una aplicación Laravel corriente escribirías tú mismo `$this->app->bind(WalletRepository::class, EloquentWalletRepository::class);`, normalmente en el método `register()` de un proveedor de servicios. `#[Repository]` más la auto-conexión de interfaces es exactamente esa vinculación, generada para ti a partir del hecho de que solo una clase escaneada implementa la interfaz.

Métodos fábrica `#[Configuration]` y `#[Bean]`

No todo lo que necesitas inyectar es una clase de tu propiedad. `#[Configuration]` marca una clase como una **fuentes de métodos fábrica** `#[Bean]` — el tipo de retorno de cada método se convierte en el tipo registrado del bean, y sus parámetros se resuelven e inyectan exactamente igual que los de un constructor. `#[Configuration]` es en sí mismo un `#[Component]`, así que la clase de configuración también es un bean gestionado.

Los propios paquetes del framework están contruidos con ellas, y la más corta es la que `firefly/validation` usa para ensamblar el validador que el Capítulo 4 pondrá a trabajar:

```
#[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);
    }
}

```

Lee los dos métodos `#[Bean]` y tienes el mecanismo entero. `validationSettings()` declara `ValidationSettings` como tipo de retorno, así que `ValidationSettings` es el tipo bajo el que se registra su bean. `validator()` declara `Validator`, y sus dos parámetros — la `Factory` de validación propia de Illuminate y el `ValidationSettings` que produce el método de encima — se resuelven del contenedor exactamente igual que se resolverían los de un constructor. Tipo de retorno adentro, clave de registro afuera; tipos de parámetro adentro, inyección afuera.

Los dos atributos extra pertenecen al Capítulo 3 y se nombran aquí solo para que no se lean como magia. `#[Order(1000)]` ordena esta clase *después* de cualquier cosa que declare tu aplicación, y `#[ConditionalOnMissingBean]` hace que cada valor por defecto se aparte en el momento en que una aplicación vincula un `Validator` propio — la regla de retirada que convierte un `#[Configuration]` corriente en una **auto-configuración**, que es como cada paquete Firefly distribuye un valor por defecto funcional que eres libre de reemplazar.

Nada de la forma cambia cuando la clase es tuya en lugar del framework: un `#[Configuration]` bajo tu propio `app/` se escanea, se ordena y se cablea en la misma pasada. Es la forma que volverás a ver cada vez que este libro presente un paquete que necesita entregarte un objeto ya construido en lugar de una clase que construyes directamente.

📦 PARIDAD CON LARAVEL

`#[Configuration]` + `#[Bean]` es el equivalente directo de que el método `register()` de un proveedor de servicios Laravel llame a `$this->app->singleton(SomeType::class, fn () => ...)` — salvo que el *tipo* que devuelve el cierre se lee de la propia declaración de tipo de retorno del método, así que no hay nada que mantener sincronizado a mano.

Inyección por constructor

📖 TÉRMINO NUEVO: INYECCIÓN

Inyección de dependencias significa que un bean recibe sus colaboradores a través de su constructor en lugar de construirlos él mismo o obtenerlos de un global. El contenedor inspecciona el constructor de un bean, resuelve el tipo de cada parámetro y pasa la instancia construida (o ya construida).

Ya has visto esto dos veces sin una explicación completa. `OpenWalletHandler`, el manejador de comando que abre un nuevo monedero en `samples/lumen`, es el ejemplo más claro — todo su trabajo es recibir el *puerto*, no el adaptador:

```
<?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;
    }
}

```

El constructor de `OpenWalletHandler` pide `WalletRepository` — la *interfaz*, no `EloquentWalletRepository`. El contenedor resuelve esa petición en tres pasos: busca `WalletRepository` en el manifiesto compilado, encuentra la interfaz auto-conectada a `EloquentWalletRepository` (la única implementación escaneada), y construye — o reutiliza — esa instancia. `OpenWalletHandler` nunca importa

`EloquentWalletRepository`, nunca menciona Eloquent, y seguiría compilando sin cambios si `samples/lumen` sustituyera mañana el adaptador de almacenamiento por otro distinto. Ese es todo el sentido de programar contra un puerto: el propio código fuente del manejador es la prueba de que no sabe, y no necesita saber, qué hay al otro lado de `WalletRepository`.

 CONSEJO: LA INYECCIÓN POR CONSTRUCTOR ES LA ÚNICA QUE HAY

LaraFly no admite inyección por propiedad ni por setter — toda dependencia que necesite un bean debe declararse como un parámetro de constructor tipado. Esto es deliberado: la firma del constructor de una clase es una lista completa y honesta de todo aquello de lo que depende, legible sin abrir el cuerpo de la clase.

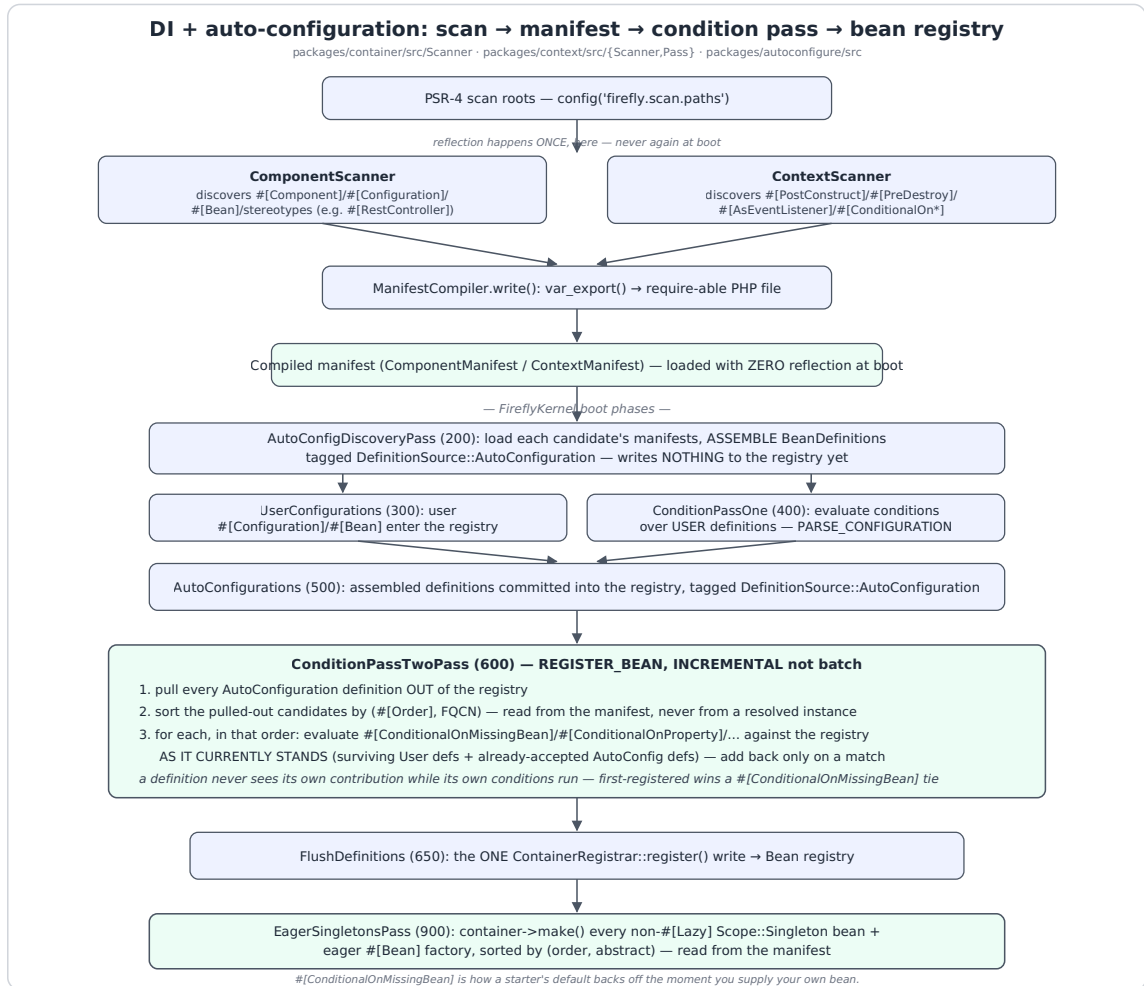


Figura 2.1 — Del escaneo de componentes al manifiesto compilado, al paso de condiciones, al registro de beans, a los singletons eager.

Ámbitos

Todo componente es un **singleton** por defecto: el contenedor lo construye una vez, y cada resolución posterior devuelve la misma instancia. Elige un ciclo de vida distinto con el argumento **scope** que acepta cualquier atributo de estereotipo:

```
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` es un enum de PHP corriente con tres casos:

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

Caso	Tiempo de vida
<code>Singleton</code>	una instancia durante toda la vida de la aplicación — el valor por defecto
<code>Transient</code>	una instancia nueva en cada resolución
<code>Scoped</code>	una instancia por petición o ámbito de Laravel

El ámbito de sesión está deliberadamente ausente: pertenece a `firefly/session`, no al contenedor.

Usa `Scope::Transient` para cualquier cosa que nunca deba compartirse — un id de correlación por operación, un constructor mutable. Usa el `Scope::Singleton` por defecto para todo lo demás, que es casi todo: los servicios, los repositorios y los DTO de configuración son todos naturalmente compartibles, y un singleton es más barato de resolver.

⚠ ADVERTENCIA

`Scope::Scoped` vincula una instancia por petición de Laravel bajo el clásico PHP-FPM. Si despliegas bajo Octane, la vida de un bean con ámbito es de una *petición*, no de un *worker* — no lo uses como sustituto de `Scope::Singleton` solo porque suene más seguro; los dos tienen ciclos de vida genuinamente distintos.

#[Primary] y #[Qualifier] : desambiguar implementaciones

La auto-conexión de interfaces, descrita antes en este capítulo, tiene una regla sencilla para el caso común: exactamente una implementación escaneada de una interfaz se vincula a ella automáticamente. Pero las aplicaciones reales a menudo tienen *más* de una implementación de la misma interfaz — un adaptador de producción y uno de prueba, o dos estrategias genuinamente distintas. `#[Primary]` y `#[Qualifier]` son la manera de decirle al contenedor cuál quieres:

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

Con ambos beans registrados, el contenedor resuelve la interfaz de tres maneras distintas según lo que le pidas:

```
$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]` desempata cuando un constructor pide la interfaz a secas: el contenedor vincula `Greeter::class` a la implementación que lleve `#[Primary]`, exactamente igual que `wireInterfaces()` vinculó antes `WalletRepository` a `EloquentWalletRepository` — salvo que ahora la elección es explícita en lugar de ser la única opción disponible. `#[Qualifier('spanish')]` le da a `SpanishGreeter` un *nombre* que puedes pedir directamente, independiente de la resolución por tipo, cuando específicamente necesitas la que no es la principal.

⚠ ADVERTENCIA

Con cero, o con más de una implementación `#[Primary]` de la misma interfaz, el contenedor no tiene un valor por defecto bien definido — resolver la interfaz a secas se vuelve ambigua. Dale a toda interfaz con varias implementaciones exactamente un `#[Primary]`, o resuelve cada implementación por su nombre `#[Qualifier]`.

`#[Order]`

`getAll(Greeter::class)`, arriba, devuelve todas las implementaciones — y las devuelve **ordenadas por** `#[Order]`, la más baja primero, exactamente como la convención `@Order` de Spring:

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

El orden por defecto es `0` cuando se omite el atributo, así que cualquier cosa que ordene explícitamente con un número positivo se ordena después de cualquier bean sin ordenar, y cualquier cosa que ordene con un número negativo se ordena antes

que ellos. `#[Order]` también se aplica a métodos fábrica `#[Bean]`, no solo a clases — el atributo funciona de forma idéntica allí donde una lista de beans necesite una secuencia determinista.

Inyección con `#[Value]`

A veces lo que necesitas inyectar no es un bean en absoluto, sino un único escalar — un valor de configuración o una pequeña expresión calculada. `#[Value]` apunta directamente a un parámetro de constructor:

```
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,
    ) {}
}
```

Se admiten dos formas de expresión. `${NAME:default}` lee la variable de entorno `NAME`, recurriendo a `default` cuando no está definida; `#{expr}` evalúa un pequeño lenguaje de expresiones en un entorno aislado (sandbox) — `#{25 * 2}` se resuelve como el entero `50`. (`firefly/config`, que se cubre en un capítulo posterior, extiende esa misma resolución para leer configuración de la aplicación, no solo el entorno.)

NOTA

`#[Value]` apunta exactamente a un parámetro de constructor — no es un estereotipo de clase, y una clase que solo tenga parámetros anotados con `#[Value]` sigue necesitando un atributo de estereotipo como `#[Service]` si quieres que el contenedor la gestione como un bean por derecho propio.

Del escaneo de componentes al arranque sin reflexión

Cada estereotipo, cada `#[Bean]`, cada `#[Primary]` / `#[Qualifier]` / `#[Order]` que acabas de leer es *metadato de reflexión* — inerte hasta que algo lo lee. Ese algo es `ComponentScanner`, y se ejecuta **exactamente una vez**, cuando invocas `firefly:cache`, nunca más en ninguna petición que sirva la aplicación en ejecución.

`ComponentScanner::scan()` recorre cada raíz PSR-4 que tu `config/firefly.php` nombra bajo `scan.paths`, refleja cada clase que encuentra y — para cada clase que lleve un atributo instancia-de `#[Component]` — registra su estereotipo, su ámbito, si es `#[Primary]`, su `#[Order]`, su nombre `#[Qualifier]`, las interfaces que implementa y (para una clase `#[Configuration]`) la lista de sus métodos `#[Bean]`. El resultado es un array plano de `ComponentDescriptor` — sin objetos, sin cierres, nada que no se pueda serializar.

`ManifestCompiler` toma ese array y lo escribe a disco sin nada más exótico que el propio `var_export()` de PHP:

```
"<?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";
```

El resultado — `bootstrap/cache/firefly/component.php` en tu proyecto — es un archivo PHP corriente, `require`-able: un array literal, sin reflexión, sin análisis de atributos y sin necesidad de recorrer el sistema de archivos para volver a leerlo.

`ContainerRegistrar::register()` carga ese archivo y hace la conexión real del contenedor que describe este capítulo — vinculando cada componente bajo su propia clase, conectando interfaces (incluyendo la regla `#[Primary]` /única-implementación) y registrando la salida de cada fábrica `#[Bean]` bajo su tipo de retorno.

`php artisan firefly:cache` es lo que impulsa toda esta tubería para tu aplicación:

```
php artisan firefly:cache
```

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

Ya ejecutaste esto una vez, de forma indirecta — `firefly new` lo llama por ti en su script `post-create-project-cmd`, que es la razón por la que la aplicación del Inicio rápido arrancó correctamente sin que ejecutaras el comando a mano ni una sola vez. De aquí en adelante, cada vez que añadas o cambies un `#[Service]`, `#[Repository]`, `#[Configuration]` o cualquier otro atributo de Firefly, vuelve a ejecutar `firefly:cache` para recompilar el manifiesto; `php artisan firefly:clear` elimina la caché compilada y vuelve a la ruta de escaneo de desarrollo (más lenta, basada en reflexión).

 CONSEJO: `FIREFLY:CACHE` NO ES SOLO PARA CONTENEDORES

El mismo comando también compila los manifiestos de todos los demás pilares que presentó el Capítulo 1 — rutas, proxies transaccionales, manejadores de CQRS, escuchadores de eventos, reglas de seguridad y más. La inyección de dependencias es simplemente el primero y más fundamental de ellos; la misma idea de "reflexionar una vez, congelar en un manifiesto, arrancar desde la copia congelada" se repite para cada uno.

✓ Lo que aprendiste

Concepto	Qué hace
<code>#[Component]</code> / <code>#[Service]</code> / <code>#[Repository]</code> / <code>#[Configuration]</code>	Marcan una clase como bean gestionado; los tres últimos especializan el atributo base
Inyección por constructor	El contenedor resuelve y pasa automáticamente cada parámetro de constructor tipado
<code>Scope::Singleton</code> / <code>Transient</code> / <code>Scoped</code>	Una instancia compartida (por defecto), una instancia nueva por resolución, o una por petición
<code>#[Primary]</code>	Desempata cuando varios beans implementan la misma interfaz
<code>#[Qualifier]</code>	Nombra un bean concreto para su búsqueda, independiente de la resolución por tipo
<code>#[Order]</code>	Ordena una lista de <code>getAll()</code> de implementaciones, la más baja primero
<code>#[Bean]</code> en un método <code>#[Configuration]</code>	Registra el valor de retorno del método bajo su tipo de retorno
<code>#[Value]</code>	Injecta un valor de entorno resuelto o una expresión en un parámetro de constructor
<code>firefly:cache</code>	Ejecuta el escaneo de componentes una vez y lo compila en un manifiesto sin reflexión

Ponlo en práctica

1. **Añade una segunda implementación.** Dale a

`Lumen\Infrastructure\WalletRepository` una segunda implementación en memoria propia (en un proyecto de prueba — no modifiques el paquete `samples/lumen` distribuido) y márcala `#[Repository]` sin `#[Primary]`. Ejecuta `firefly:cache` y observa qué le ocurre a la auto-conexión de interfaces ahora que existen dos implementaciones.

2. **Traza un constructor.** Abre de nuevo `OpenWalletHandler` y, sin consultar nada, anota cada bean que el contenedor debe resolver — directa o transitivamente — para construir una instancia de él.

3. **Lee el manifiesto compilado.** Después de ejecutar `php artisan firefly:cache` en una aplicación de andamiaje, abre `bootstrap/cache/firefly/component.php` y busca la entrada de `App\GreetingService`. Relaciona cada clave del array con el campo de `ComponentDescriptor` que describe este capítulo.

Configuración, Perfiles y Secretos

Al terminar este capítulo sabrás cómo `Config` de LaraFly envuelve el propio repositorio de configuración de Laravel con accesores tipados y de fallo rápido, cómo `#[ConfigProperties]` vincula todo un subárbol de configuración sobre un DTO tipado e inyectable, cómo `#[Value]` resuelve un escalar individual una vez instalado `firefly/config`, cómo se resuelven los perfiles activos y cómo se puede condicionar un bean a uno de ellos y — cerrando el círculo con el Capítulo 2 — exactamente cómo viaja un DTO `#[ConfigProperties]` desde el atributo de origen hasta un manifiesto compilado, sin reflexión.

❗ TÉRMINO NUEVO: SUBÁRBOL DE CONFIGURACIÓN

Un **subárbol** es todo lo anidado bajo una clave con puntos en el array de configuración de Laravel — `mail.host`, `mail.port` y `mail.tls` forman parte del subárbol `mail`. `#[ConfigProperties]` vincula un subárbol entero sobre un DTO en un solo paso, en vez de leer cada clave a mano.

La configuración sigue siendo la configuración de Laravel

LaraFly no sustituye el sistema de configuración de Laravel — se apoya en él. Cada archivo `config/*.php`, cada llamada a `env('...')`, cada variable de `.env` que ya conoces de una aplicación Laravel corriente funciona exactamente igual en una aplicación LaraFly. El trabajo de `firefly/config` es más concreto y acotado: te da un **accesor tipado y de fallo rápido** sobre ese mismo repositorio, una forma de **vincular un subárbol entero sobre un DTO**, y un **resolutor respaldado por configuración** para el atributo `#[Value]` que introdujo el Capítulo 2.

Ya viste el ejemplo más pequeño posible de esto en el Inicio rápido. `app/GreetingProperties.php`, generado por `firefly new`, es un DTO `#[ConfigProperties]` real y distribuido:

```
<?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') {}
}
```

Este capítulo explica, en profundidad, todo lo que ese pequeño archivo hace silenciosamente por ti.

Acceso tipado con `Config`

Antes de recurrir a un DTO, conviene ver la capa que hay justo debajo.

`Firefly\Config\Config` envuelve `Illuminate\Contracts\Config\Repository` — el mismo repositorio de configuración del que lee `config()` en una aplicación Laravel corriente — con cuatro getters tipados: `string()`, `int()`, `bool()` y `array()`. Cada

uno es de **fallo rápido**: una clave requerida (sin valor por defecto) que falta, o un valor que no se puede convertir al tipo solicitado, lanza `ConfigurationException` de inmediato, en vez de entregarte `null` o un tipo silenciosamente incorrecto tres marcos de llamada después.

```
$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` está registrado como singleton del contenedor, así que se inyecta por constructor en cualquier parte:

```
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');
    }
}
```

NOTA

`Config::has()` y `Config::get()` se comportan exactamente igual que `Repository::has()` / `Repository::get()` de Laravel — `Config` añade envoltorios tipados y de fallo rápido alrededor de ellos, no sustituye ni reinterpreta la resolución por notación de puntos de Laravel.

`string()` / `int()` / `bool()` / `array()` bastan para lecturas ocasionales. No escalan más allá de un puñado de llamadas dispersas: cada una es una consulta aislada sin información de tipo compartida, un error tipográfico en una cadena de clave solo aflora cuando se ejecuta esa línea exacta, y un servicio que necesita cinco ajustes relacionados termina con cinco llamadas separadas a `$config->...()` esparcidas por el cuerpo de su constructor. `#[ConfigProperties]` resuelve justamente esto.

`#[ConfigProperties]` : vincular un subárbol sobre un DTO tipado

`#[ConfigProperties(string $prefix)]` es un atributo de nivel de clase (`Firefly\Config\Attributes\ConfigProperties`). Nombra el prefijo de configuración con puntos del que se vincula la clase; el framework lee ese subárbol, empareja cada clave con un parámetro del constructor por nombre, convierte los escalares al tipo declarado, y registra la instancia resultante como singleton del contenedor — de modo que cualquier bean que pida el tipo del DTO recibe la configuración vinculada, completamente tipada, sin una sola llamada a `config('...')` en su propio cuerpo.

Aplicar el mismo patrón a un ajuste de negocio es tan corto como lo fue

`GreetingProperties`. Supongamos que Lumen quiere un límite diario de transferencia configurable:

```
<?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',
    ) {}
}

```

y el archivo de configuración correspondiente, `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` (los Capítulos 4 y 6 construyen el resto) ya puede depender directamente de `WalletProperties`:

```

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;
    }
}

```



```
    }
}
```

Nada aquí llama a `config('wallet.daily_transfer_limit_minor')`.

`WalletProperties` es un objeto tipado y autodocumentado — los propios valores por defecto del constructor (`1_000_000`, `'EUR'`) hacen las veces de documentación viva de lo que significa el ajuste cuando nada lo sobrescribe, y un error tipográfico en el nombre de un parámetro del constructor es un error de PHP en el momento del vínculo, no un campo silenciosamente `null` descubierto en producción.

📌 PARIDAD CON LARAVEL

`#[ConfigProperties]` es el equivalente en LaraFly de `@ConfigurationProperties` de Spring Boot. En una aplicación Laravel corriente, lo más parecido es escribir tu propio DTO y rellenarlo a mano desde `config('wallet')` en el método `register()` de un proveedor de servicios; `#[ConfigProperties]` es esa misma idea sin el código repetitivo — el prefijo es lo único que declaras, y el vinculador hace el resto.

Cómo funciona realmente el vínculo: `ReflectionConfigBinder`

El vinculador por defecto, `Firefly\Config\Binder\ReflectionConfigBinder`, resuelve un parámetro del constructor a la vez a partir del array del subárbol vinculado:

```
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);
}
// ...
}

```

De ese cuerpo de método se desprenden tres reglas directamente. Una clave ausente con un valor por defecto en el constructor usa silenciosamente ese valor por defecto — precisamente por qué el constructor de `WalletProperties` escribe explícitamente `1_000_000` y `'EUR'` en vez de dejarlos obligatorios. Una clave ausente **sin** valor por defecto y con un tipo no anulable lanza `ConfigurationException` de inmediato, en el momento del vínculo — no la primera vez que un manejador toca el campo. Y cuando un parámetro del constructor está tipado como otra clase (no un escalar) y el subárbol contiene un array bajo esa clave, el vinculador **recurre**: se llama a sí mismo sobre la clase anidada y el array anidado, de modo que un DTO `#[ConfigProperties]` puede componer DTOs más pequeños sin cableado adicional.

La conversión de escalares es igual de mecánica — los campos `int / float / bool` se convierten a partir de lo que sea que el valor de configuración realmente sea (útil cuando un valor llegó desde una variable de entorno como cadena), y `string` acepta cualquier cosa escalar:

```

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;
    }
}

```

💡 EL VINCULADOR ES UN PUNTO DE EXTENSIÓN, NO LA ÚNICA IMPLEMENTACIÓN

`ReflectionConfigBinder` implementa una interfaz de un solo método, `Firefly\Config\Binder\ConfigBinder`. Una aplicación (o un futuro paquete de Firefly) puede vincular un vinculador más rico sobre el contenedor — uno que entienda enums, u objetos de valor con su propia validación — sin tocar una sola clase `#[ConfigProperties]`. Los DTOs que escribes nunca saben, ni necesitan saber, qué vinculador los rellenó realmente.

Del atributo al manifiesto compilado — la misma historia que el Capítulo 2

`#[ConfigProperties]` sigue exactamente la misma forma "reflexionar una vez, congelar en un manifiesto, arrancar desde la copia congelada" que el Capítulo 2 recorrió para el contenedor. `ConfigPropertiesScanner::scan()` recorre tus raíces PSR-4, reflexiona cada clase no abstracta, y registra un `ConfigPropertiesDescriptor` — solo el nombre de la clase y el prefijo — por cada una que lleve `#[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` escribe esa lista en `bootstrap/cache/firefly/config-properties.php` con la misma técnica basada en `var_export()`, sin reflexión, que usó el manifiesto de componentes. `php artisan firefly:cache` ejecuta este escaneo como uno de los doce pares escáner/compilador que orquesta — nunca invocas `ConfigPropertiesScanner` tú mismo.

En el arranque, `Firefly\Cli\Boot\FireflyCacheServiceProvider` carga ese manifiesto compilado y se lo entrega directamente a `Firefly\Config\Registrar\ConfigRegistrar::register()`, que vincula el DTO de cada descriptor como singleton del contenedor en un solo paso:

```

    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);
        });
    }
}

```

Esta es la razón completa por la que `GreetingProperties` y `WalletProperties` no necesitan ningún cableado de proveedor de servicios en ninguna parte de tu aplicación: el manifiesto compilado ya conoce cada clase `#[ConfigProperties]` y su prefijo, y `ConfigRegistrar` los vincula todos, incondicionalmente, en el momento en que existe la caché.

⚠ LA CACHÉ TAMBIÉN IMPORTA AQUÍ

Exactamente como explicó el Capítulo 2 para los beans, el vínculo de `#[ConfigProperties]` se impulsa desde el manifiesto **compilado** — ejecuta `php artisan firefly:cache` después de añadir o renombrar una clase `#[ConfigProperties]`, igual que harías tras añadir un nuevo `#[Service]`.

#[Value] desde configuración real

El Capítulo 2 introdujo `#[Value]` sobre un parámetro de constructor desnudo — `#[Value('${MAIL_HOST:localhost}')]` — y mencionó que el propio resolutor de `firefly/container` solo entiende variables de entorno y expresiones pequeñas. Instalar `firefly/config` mejora esa resolución sin tocar un solo sitio `#[Value]` en tu código:

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

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

`Firefly\Config\Value\ConfigValueResolver` implementa el mismo puerto `ValueResolver` que define `firefly/container`, pero resuelve `${key:default}` contra la **configuración de la aplicación primero**, luego el entorno, luego el valor por defecto literal:

```
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}` sigue evaluando una pequeña expresión en sandbox, ahora con una función `config('key')` registrada sobre ella, así que una expresión también puede leer configuración — `#{config('mail.port') * 2}` es válido una vez instalado `firefly/config`.

`ConfigRegistrar::register()` vincula `ConfigValueResolver` por encima del valor por defecto del contenedor:

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

El propio registrador de `firefly/container` solo vincula su `DefaultValueResolver` **si no hay ninguno ya vinculado** — así que el resolutor respaldado por configuración, una vez que `firefly/config` lo registra, siempre gana, y `${mail.host:localhost}` ahora lee `config('mail.host')` primero, el entorno en segundo lugar, y solo recurre al literal `localhost` si ninguno de los dos está definido.

📦 PARIDAD CON LARAVEL

`#[Value('${mail.host:...}')]` leyendo primero la configuración de la aplicación es lo más parecido en LaraFly a `@Value("${...}")` de Spring leyendo `application.yaml`. En una aplicación Laravel corriente, el idioma equivalente es simplemente `config('mail.host')` llamado en el punto de uso — `#[Value]` te da el mismo valor resuelto, pero declarado una sola vez sobre el parámetro del constructor, junto al resto de las dependencias de la clase.

Perfiles

Un **perfil** es un entorno con nombre — `local`, `testing`, `staging`, `prod` — y LaraFly te da un vocabulario pequeño y concreto para leer cuáles están activos en cada momento, con independencia de cualquier clave de configuración concreta.

`Firefly\Config\Profile\ProfileResolver::resolve()` devuelve un objeto de valor `Profiles`, usando una precedencia fija: la variable de entorno `FIREFLY_PROFILES_ACTIVE` (separada por comas, para más de un perfil activo) si está definida; si no, el único valor `APP_ENV` de Laravel; si no, el perfil implícito `default`.

```
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);
    }
    // ...
}
```


📦 PARIDAD CON LARAVEL

Que `FIREFLY_PROFILES_ACTIVE` recurra a `APP_ENV` significa que cualquier aplicación Laravel existente ya tiene un perfil activo sin añadir una sola variable de entorno:

`APP_ENV=production` en tu `.env` ya es, hoy mismo, el perfil `prod` de LaraFly.

`FIREFLY_PROFILES_ACTIVE` existe para el caso que el único `APP_ENV` de Laravel no puede expresar por sí solo — activar más de un perfil a la vez.

Condicionar un componente a un perfil

Saber qué perfiles están activos se vuelve genuinamente útil en cuanto la propia presencia de un componente en el contenedor puede depender de ello.

`#[ConditionalOnProfile]` — de `firefly/context`, el paquete del motor de arranque que el Capítulo 2 mencionó de pasada — condiciona un `#[Component]` (o un `#[Configuration]` / `#[Bean]`) a uno o más perfiles:

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

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

`InMemoryMailer` solo entra en el contenedor cuando `local` o `testing` está entre los perfiles activos — la condición es una simple coincidencia "cualquier perfil listado está activo", evaluada por el mismo `ConditionEvaluator` que evalúa `#[ConditionalOnProperty]`, `#[ConditionalOnClass]` y `#[ConditionalOnMissingClass]`. En `prod`, la clase ni siquiera se instancia jamás: se filtra fuera del `BeanDefinitionRegistry` antes de construirse el contenedor, no simplemente se deja sin resolver.

⬆ PARIDAD CON LARAVEL

`#[ConditionalOnProfile]` es lo más cerca que LaraFly llega a `@Profile` de Spring. En Laravel puro, lo más parecido es una guarda `if (app()->environment('local', 'testing'))` dentro del método `register()` de un proveedor de servicios — `#[ConditionalOnProfile]` traslada esa misma decisión a la propia clase, de modo que el condicionamiento por entorno de un componente es visible justo al lado de su definición, en vez de en un archivo de proveedor aparte.

Mantener los secretos fuera de los archivos

Todo lo que ha mostrado este capítulo lee, en algún punto, a través de `env()` — `env('WALLET_DAILY_TRANSFER_LIMIT_MINOR', ...)` en `config/wallet.php`, `{mail.host:localhost}` cayendo al entorno, el propio `FIREFLY_PROFILES_ACTIVE`. Esto es deliberado, y es la misma disciplina que cualquier aplicación Laravel ya debería seguir: la configuración segura de subir al repositorio — puertos, indicadores de funcionalidad, monedas por defecto — pertenece a un archivo `config/*.php`; cualquier cosa que otorgue acceso — contraseñas de base de datos, claves de API, secretos de firma JWT — pertenece a `.env`, que nunca se sube al repositorio, y se suministra de forma distinta (archivos reales en local, secretos inyectados en CI/CD y producción) según el entorno.

⚠ NUNCA SUBAS SECRETOS AL REPOSITORIO

`#[ConfigProperties]` y `#[Value]` en última instancia leen lo que sea que devuelvan tus archivos `config/*.php` — si un archivo de configuración escribe una credencial real en vez de llamar a `env(...)`, esa credencial queda en el historial de tu repositorio. Mantén cada secreto detrás de una llamada `env()` sin valor por defecto comprometido al repositorio, e inyecta el valor real en el momento del despliegue.

✓ Lo que aprendiste

Concepto	Qué hace
<code>Config</code>	Un accesor tipado y de fallo rápido (<code>string()</code> / <code>int()</code> / <code>bool()</code> / <code>array()</code>) sobre el propio repositorio de configuración de Laravel
<code>#[ConfigProperties(prefix)]</code>	Vincula un subárbol de configuración sobre un DTO plano, registrado como singleton del contenedor
<code>ReflectionConfigBinder</code>	El <code>ConfigBinder</code> por defecto: empareja parámetros del constructor por nombre, convierte escalares, recurre en DTOs anidados
<code>ConfigPropertiesScanner</code> / <code>ConfigManifestCompiler</code>	Reflexiona las clases <code>#[ConfigProperties]</code> una sola vez, en el momento de <code>firefly:cache</code> , en un manifiesto sin reflexión
<code>ConfigRegistrar</code>	Vincula cada entrada del manifiesto como singleton, e instala <code>ConfigValueResolver</code> sobre el valor por defecto del contenedor
<code>#[Value('\${key:default}')]</code>	Con <code>firefly/config</code> instalado, resuelve contra la configuración de la aplicación primero, luego el entorno, luego el valor por defecto
<code>ProfileResolver</code> / <code>Profiles</code>	Resuelve los perfiles activos desde <code>FIREFLY_PROFILES_ACTIVE</code> , recurriendo a <code>APP_ENV</code> , recurriendo a <code>default</code>
<code>#[ConditionalOnProfile(...)]</code>	Condiciona la presencia de un componente en el contenedor a uno o más perfiles activos

Ponlo en práctica

1. **Añade `WalletProperties` de verdad.** Crea `config/wallet.php` y `Lumen\Domain\WalletProperties` exactamente como se muestra en este capítulo (en una copia de trabajo del proyecto — no modifiques el paquete distribuido `samples/lumen`), inyéctalo por constructor en un nuevo servicio, y confirma que `php artisan firefly:cache` seguido de leer el DTO inyectado refleja un valor que fijaste vía `WALLET_DAILY_TRANSFER_LIMIT_MINOR` en `.env`.
2. **Provoca un fallo de clave ausente.** Quita el valor por defecto del constructor de uno de los parámetros de `WalletProperties`, borra la clave correspondiente de `config/wallet.php`, y observa `ConfigurationException` en el punto en que se resuelve el DTO — luego restaura el valor por defecto y confirma que la excepción desaparece.
3. **Condiciona un bean a un perfil.** Añade una clase `#[Component]` `#[ConditionalOnProfile('testing')]` a tu proyecto de pruebas, ejecuta `php artisan firefly:cache` con `FIREFLY_PROFILES_ACTIVE` sin definir, y confirma que la clase nunca aparece en el manifiesto de componentes compilado; luego vuelve a ejecutar la caché con `FIREFLY_PROFILES_ACTIVE=testing` y confirma que sí aparece.

Tu Primera API HTTP

Al terminar este capítulo sabrás cómo `#[RestController]` y `#[RequestMapping]` convierten una clase corriente en una superficie HTTP enrutada, cómo cada uno de los cinco atributos de vinculación de parámetros extrae una pieza de una petición hacia un argumento tipado del método, cómo `#[Valid]` condiciona un cuerpo de petición a las restricciones de Bean-Validation antes de que tu manejador siquiera se ejecute, y cómo cada error — un fallo de validación, un monedero inexistente, una comprobación de autorización denegada — se renderiza con la misma forma predecible `application/problem+json`. Este capítulo cierra la Parte I convirtiendo el dominio del monedero de Lumen en una API REST real y validada.

TÉRMINO NUEVO: DESPACHO

El **despacho** (*dispatch*) es el acto de enrutar una petición HTTP entrante hacia el método correcto del controlador correcto, con los argumentos correctos ya extraídos y convertidos. En LaraFly esto ocurre a través de rutas nativas de Laravel — no hay un enrutador de LaraFly aparte compitiendo con el propio de Laravel.

`#[RestController]` y `#[RequestMapping]`

Ya sabes, desde el Capítulo 2, que `#[RestController]` es en sí mismo un estereotipo `#[Component]` — `extends Firefly\Container\Attributes\Component`, así que un controlador se descubre con el mismo escaneo de componentes que un `#[Service]` o un `#[Repository]`, y obtiene inyección de dependencias por constructor completa de forma gratuita. Lo que el Capítulo 2 dejó pendiente es el *otro* recorrido en el que

participa un `#[RestController]`: una reflexión aparte, `RouteScanner`, que lee un conjunto distinto de atributos sobre la misma clase para construir la tabla de enrutamiento.

Aquí está toda la API de monedero de Lumen — el ejemplo que recorre el resto de este capítulo — en un único listado real y distribuido:

```
<?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 overdraw'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
urrency)));

        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_i
d));

        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];
    }

```

```
    }
}
```

Cuatro cosas de esta clase merecen destacarse antes que nada. Primero, `#[RequestMapping('/api/v1/wallets')]` es de nivel de clase y antepone su ruta a cada mapeo de método debajo — `open()` mapea `POST /api/v1/wallets`, `deposit()` mapea `POST /api/v1/wallets/{id}/deposit`. Segundo, el constructor del controlador pide `CommandBus` y `QueryBus` — la misma historia de inyección por constructor que cubrió el Capítulo 2, aplicada a un controlador en vez de a un servicio. Tercero, cada método es una traducción fina: construye un DTO de comando o consulta, se lo entrega al bus, da forma al array que el framework serializará como JSON. Aquí no hay lógica de negocio — ninguna comprobación de sobregiro, ninguna comparación de moneda — porque esa lógica vive en el agregado `Wallet` y sus manejadores de comando, que construyen los Capítulos 5 y 6. Cuarto, el docblock no es decoración: te dice, en el mismo archivo donde viven las rutas, exactamente qué fallos puede producir cada endpoint y cómo se renderizarán — el tema de la sección final de este capítulo.

📌 PARIDAD CON LARAVEL

`#[RestController]` + `#[RequestMapping]` + los cinco atributos de mapeo de verbo de abajo son el equivalente en LaraFly de `@RestController` + `@RequestMapping` + `@GetMapping` / `@PostMapping` de Spring. En una aplicación Laravel corriente, el artefacto correspondiente es una entrada en `routes/api.php` que apunta a un método de controlador; aquí, la ruta es el atributo, declarado sobre el método al que despacha, así que hay exactamente un sitio donde mirar para ambas cosas.

Los atributos de mapeo de verbo

Cada método lleva exactamente un atributo de verbo, y los cinco implementan la misma interfaz marcadora `Mapping`, así que `RouteScanner` los descubre polimórficamente — con `ReflectionAttribute::IS_INSTANCEOF`, el mismo truco que el Capítulo 2 usó para los estereotipos — en vez de necesitar código de escaneo separado por verbo:

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

Cada uno de los cinco acepta los mismos tres argumentos de constructor — una `path` relativa (unida a la base de nivel de clase), un `status` entero (el código de respuesta por defecto: 200 para los verbos de lectura, y `status: 201` en `WalletController::open()` arriba, ya que abrir un monedero es una creación), y un `name` de ruta opcional:

```
#[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` y `DeleteMapping` tienen la misma forma, cada una simplemente devolviendo su propia cadena de método HTTP desde `method()`.

Vinculación de parámetros: cinco atributos, dos convenciones

El Capítulo 2 explicó cómo el contenedor resuelve el constructor de un *bean*. Los parámetros de un *método* de controlador se resuelven de forma distinta — por `ArgumentResolver`, que lee un atributo de vinculación por parámetro y extrae exactamente una pieza de la petición hacia él:

Atributo	Origen	Notas
<code>#[PathVariable(name?)]</code>	segmento de ruta	siempre obligatorio; por defecto usa el propio nombre del parámetro
<code>#[QueryParam(name?, default?, required?)]</code>	cadena de consulta	no obligatorio por defecto
<code>#[RequestBody]</code>	cuerpo de la petición	decodificado vía el <code>MessageConverter</code> negociado; combínalo con <code>#[Valid]</code> para condicionar sobre validación
<code>#[RequestHeader(name?, default?)]</code>	cabecera HTTP	no obligatorio
<code>#[UploadedFile(name?)]</code>	archivo <code>multipart/form-data</code>	un <code>Firefly\Web\Http\UploadedFile</code> neutral al framework (filename/mimeType/size + <code>contents() / store()</code> perezosos)

`WalletController::balance()`, en el listado completo de arriba, muestra el más sencillo de estos: `#[GetMapping('/{id}/balance')]` emparejado con `#[PathVariable] string $id`, que lee el segmento de ruta `{id}` directamente en un parámetro de cadena tipado.

`open()` y `deposit()` muestran la combinación que usarás en cada endpoint de escritura — `#[Valid] #[RequestBody]` apilados sobre el mismo parámetro, cubierta en detalle en la siguiente sección.

Dos formas más de parámetro no necesitan **ningún atributo en absoluto**, y ambas son convenciones genuinas, no omisiones. Un parámetro tipado como una clase sin atributo de vinculación — una interfaz de servicio, por ejemplo — se resuelve como un simple **servicio del contenedor**, exactamente como lo haría `$container->make($type)`; esto le permite a un método alcanzar un colaborador adicional sin añadirlo al propio constructor del controlador. Un parámetro **escalar** sin atributo de vinculación por defecto se convierte en un parámetro de consulta con clave igual a su propio nombre, obligatorio a menos que el propio parámetro tenga un valor por defecto o sea anulable.

Los valores de ruta y de consulta se convierten al tipo escalar declarado del parámetro — `int`, `float` o `bool` — y un valor no convertible, o uno obligatorio ausente, lanza `InvalidRequestException` (HTTP 400, con códigos `MISSING_PARAMETER` / `TYPE_CONVERSION_ERROR`), renderizado a través del mismo flujo de detalles de problema con el que termina este capítulo.

Los DTOs de petición

Cada endpoint de escritura en `WalletController` acepta un DTO pequeño y real — al estilo Pydantic, aquí en PHP puro. Los tres viven en `samples/lumen/src/Web/Dto/` y son clases puras con parámetros promovidos por constructor y restricciones de validación por propiedad:

```
<?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,
    ) {}
}

```

Que `AmountRequest` sirva a la vez para depositar y retirar es una decisión de diseño deliberada, no pereza — ambas operaciones mueven una cantidad positiva de dinero en unidades menores, y el id del monedero ya vive en la ruta, no en el cuerpo, así que un único DTO compartido mantiene idénticos los contratos de ambos endpoints sin una clase base compartida.

`#[Valid]` → `BeanValidator` → 422

`#[Valid]`, de `firefly/validation`, es lo que convierte esos atributos por propiedad de metadatos inertes en una barrera efectivamente aplicada. Apilarlo sobre un parámetro `#[RequestBody]` — como hace cada método de escritura de `WalletController` — le dice al despachador: valida el *cuerpo decodificado en bruto* contra las reglas de restricción compiladas de esta clase DTO **antes** de siquiera construir el DTO.

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

Dos destinos, y los dos importan. Sobre un **parámetro** es la barrera que acabamos de describir. Sobre una **propiedad** de un DTO significa *cascada* — validar el objeto que ese miembro contiene, bajo claves con prefijo de punto, de modo que un `shipTo` de tipo clase reporta sus fallos como `shipTo.postcode`, y un miembro tipado `array` o `iterable` desciende en cascada hasta cada elemento y reporta `lines[0].sku`. Ese segundo caso es el que necesita el parámetro de constructor que el listado conserva: `#[Valid(each: Line::class)]` nombra la clase de elemento de un miembro de lista

cuyo docblock no lo dice ya. Sin el argumento del atributo y sin una etiqueta `@var list<Line>` / `@param list<Line> $lines` que `ConstraintScanner` pueda leer, una lista `#[Valid]` se **rechaza en tiempo de escaneo** en vez de omitirse en silencio — una colección sin validar que parece validada es exactamente el fallo que ese rechazo existe para evitar.

La validación en sí corre a través de

`Firefly\Validation\Constraint\BeanValidator`, que delega en un puerto `Validator` (`IlluminateValidator` por defecto) usando reglas que un `ConstraintManifest` compiló de antemano a partir del método `toRules()` de cada atributo de restricción:

```
#[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');
    }
}
```

Cada atributo de restricción se reduce, de esta forma, a una o más reglas de validación de Illuminate corrientes — `#[NotBlank]` rechaza `null`, `''`, `[]`, y una cadena compuesta solo por espacios; `#[Positive]` exige un valor numérico estrictamente mayor que cero; `#[CurrencyCode]` (usado en `OpenWalletRequest::$currency`) delega en un objeto de regla `Currency` dedicado en vez de en una cadena de regla. Si falla, `BeanValidator` lanza el `ValidationException` del kernel — el mismo tipo de excepción que la sección final de este capítulo mapea a HTTP 422 — llevando un `FieldError` por cada propiedad fallida. Si tiene éxito, el DTO se construye a partir del cuerpo decodificado **en bruto**, no de un subconjunto producido por el validador, así que una propiedad sin ningún atributo de restricción nunca se descarta silenciosamente.

❗ NOTA

Las restricciones validan el array de petición en bruto, no el DTO ya construido. Este orden importa: `#[Valid]` garantiza que tu método manejador solo recibe jamás un DTO cuyos campos restringidos ya han pasado todas las comprobaciones — no existe ningún camino en el que `WalletController::open()` se ejecute con un `OpenWalletRequest` cuyo `owner_id` esté en blanco.

📦 PARIDAD CON LARAVEL

`#[Valid]` `#[RequestBody]` es el equivalente en LaraFly del par `@Valid @RequestBody` de Spring. En una aplicación Laravel corriente, lo más parecido es una clase Form Request con un método `rules()`; los atributos por propiedad de `firefly/validation` trasladan esas mismas reglas a los propios campos del DTO, así que la restricción y el campo que restringe se declaran en el mismo sitio.

`firefly/validation` distribuye dieciséis restricciones del dominio financiero además de las de propósito general (`#[NotBlank]`, `#[NotEmpty]`, `#[NotNull]`, `#[Size]`, `#[Min]` / `#[Max]`, `#[Pattern]`, `#[Email]`, y más): `#[Iban]`, `#[Bic]`, `#[Swift]`, `#[Isin]`, `#[Cusip]`, `#[RoutingNumber]`, `#[Luhn]`, `#[CurrencyCode]`, `#[CountryCode]`, `#[LanguageTag]`, `#[UuidValue]`, `#[Phone]`, `#[PostalCode]`, `#[Percentage]`, `#[Money]` y `#[DecimalScale]` — más un escape `#[Rules]` para cualquier cadena de regla de Illuminate en bruto u objeto `ValidationRule` que necesites y que ningún atributo dedicado cubra todavía.

Negociación de contenido

Una vez que un manejador retorna, algo todavía tiene que decidir el formato sobre el cable. `firefly/web` distribuye hoy un solo `MessageConverter` — `JsonMessageConverter`, que coincide con `application/json` y cualquier tipo con

sufijo `+json` — elegido a partir de la cabecera `Accept` de la petición (analizada por valores q, con desempate por orden de cabecera) y recurriendo a JSON cuando nada coincide o la cabecera está ausente. Los cuerpos de petición se leen de la misma forma, según `Content-Type`. `MessageConverterRegistry` es un enlace de contenedor corriente y protegido, así que una aplicación puede registrar convertidores adicionales — el soporte de XML es un punto de extensión deliberadamente aplazado, no una funcionalidad distribuida.

Errores en los que los clientes pueden confiar

Una API bien diseñada nunca filtra una traza de pila en bruto, y nunca devuelve un 500 desnudo por algo que quien llama podría haber evitado. La jerarquía de excepciones de `firefly/kernel` es la columna vertebral que hace esto automático: cada excepción del framework extiende `FireflyException`, que lleva un `errorCode()` estable, un `httpStatus()`, una `category()` y una `severity()` — así que lanzar el *tipo* correcto de excepción es la única preocupación HTTP que tu código manejador jamás tiene que pensar.

Excepción	Código	Estado
<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()` lanza directamente la primera de estas, exactamente como debería hacerlo cualquier manejador:

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

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

El agregado `Wallet` del Capítulo 6 lanza `ConflictException` por un sobregiro o un desajuste de moneda, y el `#[PreAuthorize]` de `WithdrawHandler` (el Capítulo 6 lo toca brevemente, la seguridad completa llega más adelante) puede denegar una petición antes de que el cuerpo del manejador siquiera se ejecute, envolviendo una `AuthorizationException`. Las tres llegan al mismo renderizador.

El renderizador RFC-7807

`Firefly\Web\Exception\ProblemDetailsRenderer` es donde cada una de esas excepciones realmente se convierte en bytes sobre el cable:

```
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` se fija aquí, en cada respuesta que este renderizador produce — ese tipo de medio es el contrato del RFC-7807, y un cliente tiene todo el derecho a ramificar sobre él. A su lado viajan dos identificadores, y son dos a propósito.

`X-Correlation-Id` lleva el identificador que `CorrelationIdFilter` acuña o lee en el borde de la petición, y el documento repite ese mismo valor en `correlationId`, de modo que quien llama puede casar el cuerpo que tiene delante con su propio registro de peticiones. El `traceId` del documento es lo que responda

`TraceContext::referenceFor()` — el identificador de traza W3C de la petición cuando el trazado está activado y esta petición trae uno válido, y el identificador de correlación cuando no lo trae —, así que la referencia que alguien cita nunca está vacía, y activar el trazado es lo único que cambia cuál de los dos lleva. Cuando de verdad hay un identificador de traza, además se hace eco de él en una cabecera propia (`X-Trace-Id` por defecto; un nombre de cabecera vacío desactiva ese eco). Ambos identificadores pasan a través de `ErrorResponse` en vez de escribirse sobre el

array después, porque la lista de miembros del DTO es de donde se genera el componente OpenAPI publicado — un miembro añadido aquí sería uno que ningún cliente generado sabe decodificar.

El corte que hay encima del `return` oculta el resto del trabajo con cabeceras: las cabeceras que una `HttpExceptionInterface` ya trae se copian sobre la respuesta — el `Allow` de un `405`, entre ellas — y un `503` gana además `Retry-After: 5`.

`$this->encode()` es el cuerpo, y es total a propósito. Un byte que no es UTF-8 válido — el mensaje de un driver que cita una columna latin-1, una cabecera de la petición copiada a un miembro de extensión — se sustituye en vez de elevarse. Cualquier cosa que `json_encode` siga rechazando, como un `INF` que una aplicación puso en una extensión en el punto del throw, cae a un documento mínimo; y también cae ahí cualquier cosa que lance el código *propio* de un objeto codificado, porque codificar un objeto llama a ese código — un `jsonSerialize()`, un accesor de Eloquent — y lo que lance no llega a ser una `JsonException` siquiera, razón por la cual el respaldo captura `Throwable` y no sólo la de JSON. Ese documento conserva todos los miembros que la forma de arriba define — el estado, el título, el código, la categoría, la severidad, la frase y los identificadores de referencia — porque cada uno es una cadena que escribió esta clase y ninguno puede ser lo que `json_encode` rechazó. Conserva también los errores de campo, con su nombre y su frase intactos — un `422` que siguiera diciendo `category: validation` sin llevar miembro `errors` mandaría a un cliente generado a la rama de errores de campo sin nada que pintar — y conserva el *nombre* de cada miembro de extensión: un valor que no puede llevar se escribe como el nombre de su tipo (`"balance": "App\Models\Balance"`) en vez de borrarse, que es como el endpoint de configuración de `firefly/actuator` lleva respondiendo siempre a esta misma pregunta. Sólo se pierden dos cosas: el valor ilegible, sustituido por su

tipo, y el `rejectedValue` de cada error de campo, que se descarta del todo porque el contrato entero de ese miembro es *esto es lo que enviaste* y la palabra `float` no la envió nadie.

Y la degradación queda registrada. El throwable que capturó el respaldo es una excepción cualquiera de la aplicación, así que se registra — por el `Psr\Log\LoggerInterface` opcional que el provider le pasa a este renderizador, con el `traceId` del propio documento en el contexto para que la línea de log y el cuerpo que tiene quien llama se puedan juntar, y dentro de su propio `try / catch` para que un canal de log que también esté caído no pueda lanzar fuera del manejador de errores. Un subsistema de manejo de errores que falla de forma invisible es justo lo que toda esta superficie existe para eliminar, y un renderizador que se tragara una excepción en silencio mientras publica un documento indistinguible de uno sano sería exactamente eso. Un renderizador que lanzara aquí, por su parte, fallaría mientras atiende el fallo, y quien llama no recibiría documento alguno.

Lo que `render()` deliberadamente **no** decide es en qué `FireflyException` se convierte un throwable cualquiera. Esa regla vive una clase más allá, en `Firefly\Web>Error\ProblemMapper`, porque la página de error HTML del Capítulo 10 necesita la respuesta idéntica y dos copias de ella acabarían dándole a un navegador y a un cliente de API códigos distintos para el mismo fallo:

```
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,
    ),
    };
}

```

Cinco brazos que cubren cuatro casos — el `405` tiene un brazo propio solo para que los verbos que permite puedan sacarse de la frase del enrutador y llevarse a un miembro `allowed` y a la cabecera `Allow` — y todos ellos terminan de vuelta en la misma llamada `ErrorResponse::fromException(...)->toArray()`, así que la forma de la carga útil nunca depende de qué brazo la produjo.

Una `FireflyException` — o una de sus subclases tipadas, como `ResourceNotFoundException` — se devuelve intacta y se renderiza con su propio `httpStatus()`, porque su mensaje lo escribió tu aplicación *para* el cliente; ese es justamente el sentido de la taxonomía. El propio límite de tiempo de ejecución de PHP se nombra aparte y responde `503` con una cabecera `Retry-After`, porque «el servidor detuvo esta petición a los N segundos» es algo sobre lo que quien llama puede actuar y un `500` desnudo no lo es. Una excepción HTTP de Laravel/Symfony — una URL que no coincide con ninguna ruta, un verbo que una ruta no acepta — conserva su **código de estado real**, así que una ruta no coincidente sigue respondiendo `404` y nunca un `500` engañoso; solo se reemplaza la redacción del

propio enrutador por una frase escrita para una persona, y los verbos permitidos de un 405 pasan a un miembro `allowed` y a la cabecera `Allow`, donde un cliente puede leerlos sin analizar inglés. Cualquier otra cosa es un accidente, y `$disclose` — `firefly.web.problem.disclose`, por defecto `false` — decide si su mensaje puede publicarse siquiera: con él apagado el cuerpo lleva una frase fija que nombra la misma referencia que lleva el `traceId` del documento, y el mensaje real se queda en el log, que es donde el SQL de una `QueryException` y sus enlaces deben estar.

Pedir un monedero que nunca se abrió se renderiza así. Fíjate en `instance`: el RFC 9457 §3.1.5 lo define como una **referencia** URI, y una referencia relativa se resuelve contra la URI base del documento — así que el `api/v1/wallets/wlt-999` pelado que devuelve `$request->path()`, servido desde `/api/v1/wallets/wlt-999`, identificaría `/api/v1/api/v1/wallets/wlt-999`. LaraFly publica la forma relativa a la raíz, que es el único trabajo de `ProblemMapper::instanceFor()`, y un cliente puede compararla con la ruta que pidió. Fíjate también en `type`: el RFC 9457 §3.1.1 dice que un `type` ausente es `about:blank`, y LaraFly lo escribe en vez de dejar que el lector tenga que saberlo — apunta `firefly.web.problem.type-uri` a una URI base y el `code` estable deriva uno real y abrible.

```
{
  "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"
}
```

Un fallo de comprobación `#[Valid]` en `POST /api/v1/wallets` — un `owner_id` vacío — lleva además un array `errors`, una entrada por cada **restricción** fallida:

```
{
  "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": ""
    }
  ]
}
```

Cuatro miembros, y el emparejamiento es deliberado: `message` es la frase — la redacción que Bean Validation da a la restricción, o la que `#[NotBlank(message: '...')]` haya suministrado — y `constraint` es el **nombre corto del atributo**, que es el miembro sobre el que un cliente ramifica cuando la frase está escrita para una persona y puede traducirse o reformularse. `rejectedValue` es lo que el cliente envió de verdad, devuelto como eco para que un formulario pueda resaltar el campo sin adivinar. Aparece una entrada por restricción fallida, no por regla fallida: `#[NotBlank]` compila a tres reglas de Laravel y aun así reporta una sola vez.

Un intento de retiro se rechaza por dos vías distintas, y las dos **no dan el mismo estado**. Sin ningún principal autenticado, el `#[PreAuthorize]` de `WithdrawHandler` — aplicado en el bus, antes de que el cuerpo del manejador se ejecute — responde `401`: *autentícate primero*.

```
{
  "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"
}
```

El `403` queda reservado para un principal que **sí** ha iniciado sesión y aun así no tiene `ROLE_ADMIN` ni `ROLE_WALLET_OWNER` — *volver a autenticarse no ayudaría* — y ese documento nombra las autoridades que la expresión pedía:

```
{
  "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"]
}

```

Los dos se producen sin ningún cambio de código en

`WalletController::withdraw()`: el bus envuelve la excepción de seguridad que haya levantado el evaluador en una `CommandProcessingException` que copia el código, el estado y la categoría de la causa, así que el cable lleva la identidad de la causa y no un error genérico del bus. El Capítulo 10 vuelve sobre este par — y sobre las dos pruebas distribuidas que lo fijan — en detalle.

Manejar una excepción tú mismo

Antes de que una excepción llegue al renderizador genérico, LaraFly le da a tu aplicación la oportunidad de responderla directamente.

`#[ExceptionHandler(SomeException::class)]` marca un método como el renderizador de una clase de excepción concreta (o cualquier subclase) — declarado bien **en el propio controlador que la lanzó**, bien en un bean `#[ControllerAdvice]` dedicado para un manejador **global** compartido entre todos los controladores:

```

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()];
    }
}

```



```
}  
}
```

Un manejador local del controlador siempre gana sobre uno global

`#[ControllerAdvice]` para la misma excepción; dentro del ámbito que gane, se elige la clase coincidente más específica en la jerarquía. El valor de retorno de un manejador que coincide se negocia por contenido exactamente igual que cualquier otro retorno de controlador, pero se renderiza con el propio `HttpStatus()` de la excepción en vez del estado por defecto de la ruta. Si nada coincide en ninguno de los dos ámbitos, la excepción cae hacia `ProblemDetailsRenderer` — así que cada error sin manejar, empiece como empiece, sigue terminando como `application/problem+json`, nunca como una página de error de framework sin capturar.

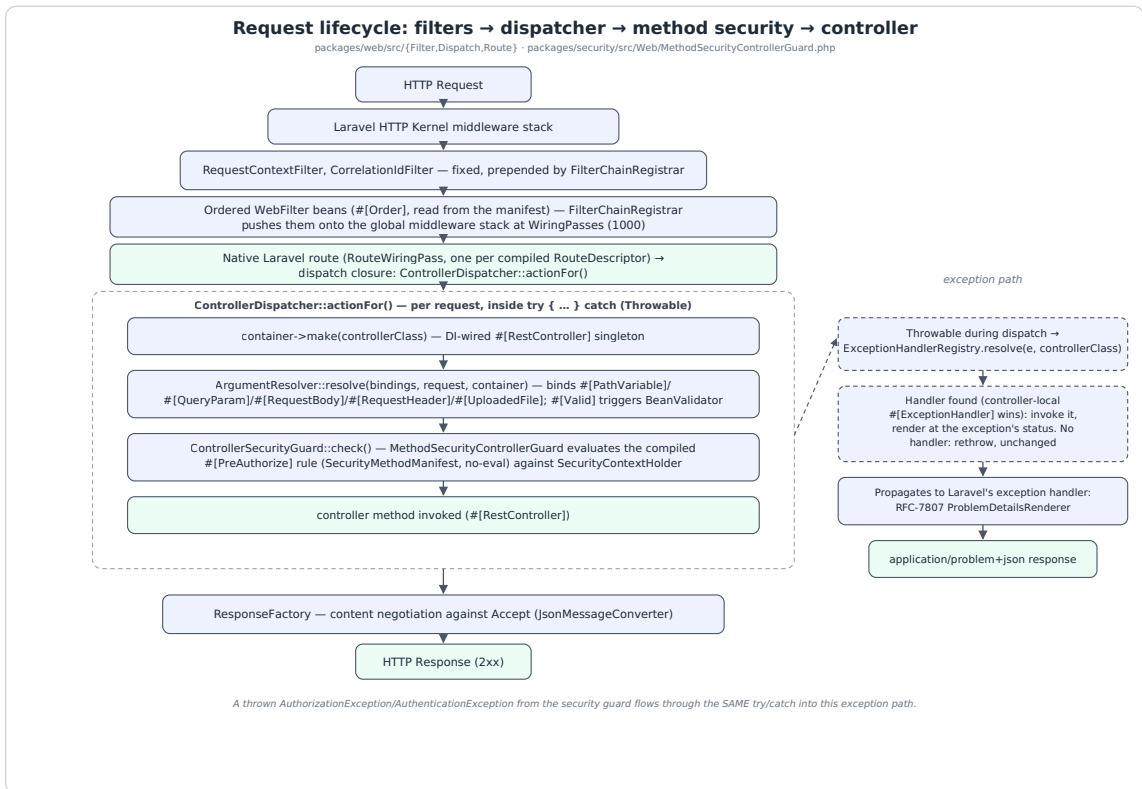


Figura 4.1 — Una petición atraviesa la cadena de filtros web, el despacho de rutas, la vinculación de argumentos y la validación y — ante un fallo en cualquier punto — el mismo renderizador RFC-7807, antes de que tu método manejador siquiera se ejecute.

El **RouteManifest** compilado

Ya has visto la misma forma dos veces — el manifiesto de componentes del Capítulo 2, el manifiesto de config-properties de este capítulo — y el enrutamiento la sigue por tercera vez. **RouteScanner** es el único sitio de reflexión en **packages/web/src**: recorre las clases **#[RestController]**, y por cada método mapeado con un verbo compila un **RouteDescriptor** — el método HTTP, la ruta unida, el nombre del controlador y del método, el estado por defecto, un nombre de ruta opcional, y un plan

de vinculación de parámetros en array puro. `RouteManifestCompiler` vuelca el resultado con `var_export()` a `bootstrap/cache/firefly/routes.php`; `RouteManifest::load()` lo vuelve a leer sin ninguna reflexión en absoluto.

En el arranque, `RouteWiringPass` registra una **ruta nativa de Laravel** por cada descriptor y le entrega a cada una un cierre de despacho a través de `ControllerDispatcher` — así que el listado de rutas, la generación de URLs, y el propio `route:cache` de Laravel se aplican todos a las rutas de `WalletController` exactamente igual que a cualquier entrada escrita a mano en `routes/api.php`, porque eso es genuinamente lo que son, una vez que se ha ejecutado `firefly:cache`.

✓ Lo que construiste

La Parte I está completa. Lumen ahora **arranca** (Capítulo 1), está **conectado** (la inyección por constructor del Capítulo 2 sin código de cableado), está **configurado** (los ajustes tipados y conscientes de perfil del Capítulo 3), y **sirve** — una API REST validada en `/api/v1/wallets` con:

Pieza	Qué hace
<code>#[RestController]</code> + <code>#[RequestMapping]</code>	Registra el controlador como bean y fija su prefijo de URL
<code>#[GetMapping]</code> / <code>#[PostMapping]</code> / ...	Un atributo de verbo por método, compilado en un <code>RouteDescriptor</code>
<code>#[PathVariable]</code> / <code>#[QueryParam]</code> / <code>#[RequestBody]</code> / <code>#[RequestHeader]</code> / <code>#[UploadedFile]</code>	Cinco formas de extraer una pieza de la petición hacia un parámetro tipado
<code>#[Valid]</code>	Condiciona un DTO <code>#[RequestBody]</code> a sus reglas de restricción compiladas, lanzando <code>ValidationException</code> si falla
La jerarquía <code>FireflyException</code>	Excepciones tipadas que llevan su propio código de error, estado HTTP, categoría y severidad
<code>ProblemDetailsRenderer</code>	Convierte cualquier excepción en una respuesta <code>application/problem+json</code> consistente
<code>#[ExceptionHandler]</code> / <code>#[ControllerAdvice]</code>	Un escape para renderizar tú mismo una excepción concreta, local o globalmente

El Capítulo 5 no reemplaza nada en `WalletController` — el controlador que acabas de leer se queda exactamente igual — y en cambio construye la capa de repositorio a través de la cual persisten los manejadores de comando detrás de `CommandBus` / `QueryBus`.

Ponlo en práctica

1. **Añade un endpoint `DELETE`**. En un proyecto de pruebas (no en el paquete distribuido `samples/lumen`), añade `#[DeleteMapping('/{id}', status: 204)]` a un nuevo método de `WalletController` que envíe un hipotético comando `CloseWallet` y devuelva `null`. Confirma que la ruta compila y que un `204` sin cuerpo es lo que produce una petición coincidente.
2. **Traza un fallo de validación de principio a fin**. Envía `POST /api/v1/wallets` con `{"owner_id": "", "currency": "XYZ"}` contra una instancia de Lumen en marcha y lee el cuerpo `application/problem+json` resultante. Empareja cada entrada de `errors` con el atributo de restricción en `OpenWalletRequest` que la produjo.
3. **Escribe un `#[ControllerAdvice]`**. Añade un manejador de excepción global para `Firefly\Kernel\Exception\Business\ConflictException` que devuelva una forma personalizada `{"code": "...", "hint": "..."}` en vez de la carga útil de detalles de problema por defecto, y confirma que se aplica a cada controlador de tu proyecto de pruebas, no solo a uno.

Documentar la API: OpenAPI 3.1 desde los Manifiestos

Al terminar este capítulo sabrás cómo `firefly/openapi` convierte el `RouteManifest` y el `ConstraintManifest` que el Capítulo 4 acaba de construir en un documento OpenAPI 3.1 válido **sin ningún dialecto de anotaciones propio** — cómo `firefly:openapi` convierte ese documento en un artefacto de compilación que un job de CI puede diferenciar, cómo cada `kind` de enlace del plan de ruta compilado se convierte en un Parameter Object o en un Request Body, cómo un `#[NotBlank]` o un `#[Positive]` que ya escribiste se convierte en un `pattern` o en un `exclusiveMinimum`, por qué cada DTO se registra una sola vez y se alcanza por `$ref` en lugar de incrustarse, por qué cada operación lleva el mismo componente de error `problem+json` que produce el renderizador del Capítulo 4, y por qué una ruta HTML `#[Controller]` queda fuera del documento por defecto. Cierra con la consola de navegador que el paquete sirve sobre ese documento — tres estilos, de los cuales el predeterminado es el **Swagger UI oficial servido desde tu propio origen**, sin paso de npm y sin petición a CDN, y de los cuales solo uno habla alguna vez con un tercero.

❗ TÉRMINO NUEVO: EXTENSIÓN DE ESPECIFICACIÓN

OpenAPI 3.1 permite que un documento lleve miembros cuyos nombres empiezan por `x-`, llamados **extensiones de especificación**. Las herramientas conformes deben ignorarlas, de modo que una extensión puede registrar algo que el vocabulario estándar no sabe expresar sin invalidar el documento. `firefly/openapi` usa exactamente una, `x-firefly-constraints`, y este capítulo muestra qué acaba en ella y por qué nunca se descarta nada en silencio.

No hay nada que anotar

Todas las cadenas de herramientas OpenAPI en PHP anteriores a esta te piden escribir el documento dos veces: una como el código que ejecuta el servidor, y otra como anotaciones, atributos o un fichero YAML que *describen* el código que ejecuta el servidor. Las dos divergen la primera vez que alguien añade un campo con prisa, y la divergencia es invisible — el documento sigue validando, simplemente ya no coincide con el servidor.

`firefly/openapi` no tiene ese modo de fallo disponible, porque no tiene una segunda fuente. El Capítulo 4 terminó con el `RouteManifest`: la tabla compilada de cada `RouteDescriptor` desde la que sirve el dispatcher, que lleva el verbo, la ruta, el estado declarado, el nombre de ruta, la clase y el método del controlador, y el *plan de enlace* por parámetro. El Capítulo 4 también presentó el `ConstraintManifest`: la lista de reglas compilada que `BeanValidator` ejecuta sobre un cuerpo `#[Valid]`. Esos dos artefactos, más el `ErrorResponse` de `firefly/kernel`, son toda la entrada:

```
composer require fireflyframework/laraflly
```

La biblioteca principal incluye el componente OpenAPI; las aplicaciones generadas con LaraFly ya lo tienen. Arranca la app y `GET /openapi.json` queda servido; `GET /openapi` renderiza una consola de referencia sobre él. No se anotó nada, y nada puede divergir, porque cada hecho del documento se lee del mismo artefacto compilado que lee el dispatcher.

`firefly:openapi`, y rutas que no son rutas por atributo

El documento también es un fichero que puedes versionar:

```
php artisan firefly:openapi --output=docs/openapi.json # escribe el fichero e imprime una
línea de resumen
php artisan firefly:openapi > openapi.json # escribe el documento en crudo a
stdout
```

El comando existe para que el documento pueda ser un **artefacto de compilación** y no solo un endpoint vivo. Versionar el fichero generado es lo que permite a un job de CI diferenciarlo y hacer fallar un pull request que cambió la API pública sin decirlo, y lo que permite a un repositorio de front-end regenerar su cliente tipado desde una especificación versionada sin arrancar la aplicación PHP en absoluto. Es además la única forma de obtener un documento de un despliegue que mantiene `firefly.openapi.enabled` apagado en producción.

El modo stdout está escrito con la bandera `OUTPUT_RAW` de Symfony, y ese detalle importa más de lo que parece: la salida de consola pasa normalmente por el formateador de Symfony, que trata `<...>` como marcado. Una `description` que mencione un tipo genérico — cualquier cosa que lleve un ángulo y haya llegado al documento desde un valor de configuración — sería o bien engullida o bien lanzaría una excepción ante una etiqueta desconocida. El sentido del modo stdout es canalizar directamente hacia un generador de clientes, así que los bytes deben ser exactamente los bytes del documento. Por eso también la línea de confirmación se imprime **solo** en modo `--output`, donde stdout no es el documento.

Las rutas HTTP — la especificación, la consola y los propios assets de la consola — se montan de forma nativa sobre el `Router` de Illuminate desde un `BootPass`, no se declaran con `#[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');
}
}

```

Esta es la misma forma — y el mismo idiom de `BootPass` — que el Capítulo 11 te mostrará para las rutas propias del actuador, y se elige por dos razones independientes. Primera, **una ruta por atributo no puede ser configurable**: `#[GetMapping('/openapi.json')]` hornea su literal dentro de un `RouteDescriptor` compilado en tiempo de `firefly:cache`, de modo que un operador nunca podría mover la especificación de una ruta que choca con una suya, ni podría quitarla de una superficie pública sin borrar el paquete. Segunda, una ruta por atributo entraría en el `RouteManifest` de la aplicación — y el generador lee ese manifiesto, así que **el paquete se documentaría a sí mismo**. Registrarlas de forma nativa deja ambos problemas fuera de la existencia: las rutas salen de la configuración en el arranque, y nunca aparecen en la especificación que sirven. Fíjate en que la ruta de assets se monta *bajo* la ruta del visor, así que mover la consola mueve consigo su hoja de estilos y sus scripts.

`firefly.openapi.enabled` (por defecto `true`) se aplica *aquí*, sobre las rutas, y no sobre los beans — el generador y sus colaboradores son inertes sin rutas, así que cerrar las rutas es todo el interruptor. Apagarlo deja todas esas rutas genuinamente sin enrutar, de modo que devuelven 404 a través de la propia `NotFoundHttpException` del router, que el `ProblemDetailsRenderer` del Capítulo 4 renderiza entonces como un cuerpo de problem-details `404` en condiciones, no como un `500`.

❗ LAS ACCIONES SE RESUELVEN POR PETICIÓN, DENTRO DE LA CLAUSURA

Construir una `OpenApiSpecAction` en el arranque y capturarla en la ruta congelaría un `OpenApiGenerator` dentro de la ruta durante toda la vida del proceso — exactamente la forma que se rompe bajo Octane, donde el contenedor de una petición posterior es un sandbox distinto. `$container->make(...)` dentro de la clausura es la regla, aquí y en cualquier otro paquete del framework que monte una ruta nativa.

Del `RouteManifest` a los `Operation Objects`

`OpenApiGenerator::generate()` es el punto de entrada — memoiza una única pasada privada `build()` (`$this->document` `??=` `$this->build()`), y `toJson()` la envuelve para un fichero o un cuerpo HTTP. Esa pasada recorre el manifiesto una vez, salta las rutas excluidas, y entrega cada superviviente a `OperationFactory`. Todo lo relativo al resultado es determinista a propósito:

```
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];
}
// ...
}

```

Las rutas se ordenan, los verbos dentro de una ruta se ordenan en el orden canónico en que la propia especificación OpenAPI los enumera, y el registro de esquemas ordena los componentes por nombre. Esto no es pulcritud por la pulcritud. El orden de descubrimiento de rutas depende de la iteración del sistema de ficheros, así que un documento *sin ordenar* se rebarajaría entre máquinas y convertiría cada regeneración en un diff irrevisable — que es exactamente lo que hace que los equipos dejen de versionar el fichero generado, que es lo que hace que se quede obsoleto.

Dentro de una operación, el plan de enlace hace el trabajo de verdad.

`OperationFactory` despacha sobre el mismo discriminador `kind` que usa `ArgumentResolver` en tiempo de petición:

```

    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;
}

```

Los dos `// ...` cortan las partes sobre las que el plan de enlace no decide nada: el bloque que compone la prosa de cara al humano de la operación — su `summary`, su `description`, su bandera `deprecated` y sus `tags`, los cuatro cubiertos justo debajo — y el bloque `security` a nivel de operación, cuyos requisitos provienen de las reglas de seguridad de la aplicación y no de la firma del método. Todo lo demás que se imprime es lo que decide el *plan de enlace*. Leer el plan en lugar de releer la firma del método es lo que hace inequívoco el mapeo. `#[PathVariable]`, `#[QueryParam]` y `#[RequestHeader]` se convierten en Parameter Objects; `#[UploadedFile]` se convierte en una parte `multipart/form-data` tipada como `format: binary`; `#[RequestBody]` se convierte en el Request Body Object; y el sexto `kind`, `service` — el colaborador inyectado por el contenedor sin atributo que presentó el Capítulo 4 — no forma parte del contrato HTTP en absoluto y nunca aparece en el documento. Derivar esa lista de forma independiente tendría que volver a decidir cada uno de esos casos y podría discrepar del dispatcher; leer el plan no puede.

Cuatro decisiones menores rematan una operación:

- **Plantilla de ruta.** La grafía de parámetro opcional de Laravel, `{id?}`, no tiene equivalente en OpenAPI — allí un parámetro de ruta es obligatorio, punto — así que el marcador se elimina y el parámetro queda obligatorio. Emitir dos Path Items en su lugar describiría una superficie de router que no existe y duplicaría cada operación así en un cliente generado.
- **operationId.** El `name` de la ruta cuando lo tiene, y si no, derivado como el nombre corto del controlador menos un `Controller` final, en minúscula inicial, más el nombre del método: `Lumen\Web\WalletController::balance()` se convierte en `walletBalance`. Como `operationId` debe ser único en todo el documento — y un duplicado es el único fallo que hace que la mayoría de generadores de clientes aborten en lugar de degradar — una reclamación repetida se sufija (`walletBalance_2`) en lugar de permitirse que sobrescriba.
- **tags.** El mismo nombre corto, de modo que cada operación de `WalletController` se agrupa bajo "Wallet" en un visor.
- **summary.** Separado del nombre del método: `getBalance` se lee como "Get balance". El nombre de un método es la única etiqueta escrita por un humano que lleva una ruta — el `name` de un `#[Mapping]` es un nombre de ruta de Laravel, no prosa — así que es la fuente honesta.

Cuerpos de petición: un componente por DTO, alcanzado por `$ref`

El `OpenWalletRequest` del Capítulo 4 vuelve a ser el ejemplo conductor:

```
final class OpenWalletRequest
{
```

```

    public function __construct(
        #[NotBlank]
        public readonly string $owner_id,
        #[NotBlank]
        #[CurrencyCode]
        public readonly string $currency,
    ) {}
}

```

`POST /api/v1/wallets` lo enlaza con `#[Valid] #[RequestBody]`, y la operación generada lo referencia en lugar de repetirlo:

```

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

```

`SchemaRegistry` es lo que hace posible ese `$ref`, y resuelve dos problemas que tiene un generador ingenuo del tipo "incrusta el esquema en cada sitio de uso". El primero es la **duplicación**: un DTO usado por seis operaciones se emitiría seis veces, y cada cliente generado acuñaría seis tipos anónimos estructuralmente idénticos con seis nombres distintos. Registrarlo una vez y referirlo por puntero es lo que hace que `openapi-generator`, `orval` y `kiota` produzcan *un* tipo con nombre por DTO — que es todo el sentido de generar el documento en primer lugar. El segundo es la **recursión**: un DTO con un miembro `#[Valid] ?self $parent` no puede incrustarse en absoluto, porque la expansión no termina. Por eso el registro reserva el nombre del

componente *antes* de invocar al constructor del esquema, de modo que una llamada anidada para la misma clase encuentra el nombre ya tomado y devuelve el puntero de inmediato, cerrando el ciclo. Un ciclo de `$ref` es legal y útil en un documento donde una lista de reglas aplanada sería infinita.

Los nombres de componente son el nombre corto de la clase, porque es lo que lee un humano en un visor y lo que un generador convierte en un nombre de tipo. Dos DTOs que compartan nombre corto entre espacios de nombres — `Order\Dto\Address` y `Billing\Dto\Address` — chocarían y uno sobrescribiría al otro en silencio, así que el **segundo** reclamante de un nombre cae a su nombre completamente cualificado con puntos: feo, inequívoco y raro. Gana el primer reclamante, de modo que añadir un segundo `Address` en otro punto de la app nunca renombra el que ya estaba publicado.

Dos propiedades del esquema emitido merecen decirse explícitamente porque ambas son decisiones y no omisiones. La lista de miembros es la **lista de parámetros del constructor en orden de declaración**, porque es exactamente de donde hidrata `ArgumentResolver` — pero una restricción asociada a un miembro sin parámetro de constructor se documenta igualmente, porque `BeanValidator` valida el array decodificado en bruto y por tanto la exige en la entrada de todos modos. Y **nunca se emite `additionalProperties: false`**: el servidor genuinamente ignora las claves fuera de la lista del constructor, así que una especificación que afirmara lo contrario haría que clientes conformes rechazaran peticiones que el servidor habría servido.

Las restricciones se convierten en palabras clave de esquema

Ninguna de las dos mitades de un DTO basta por sí sola. El `ConstraintManifest` conoce el contrato de *validación* pero nada sobre tipos, porque una lista de reglas es, por construcción, sin tipos. El constructor conoce el contrato de *tipos* — `?int`, un enum respaldado, un DTO anidado, un valor por defecto — pero nada sobre las restricciones. Solo con los tipos, `#[NotBlank] string $name` se documentaría como una cadena sin límites; solo con las restricciones, `int $quantity` se documentaría como una cadena. `DtoSchemaFactory` las fusiona, y `ConstraintSchemaMapper` mapea la segunda mitad.

Mapea el **manifiesto compilado**, nunca los atributos `#[Constraint]`. Leer los atributos de vuelta desde el DTO sería la ruta obvia hacia "`#[Email]` → `format: email`", y documentaría un validador que no existe: el manifiesto ya ha aplicado el contrato de nulos Jakarta de `ConstraintScanner`, ya ha expandido `#[Size]` en un *objeto* regla de primera parte en vez de las cadenas polimórficas `min: / max:` de Laravel, y ya ha aplanado un nivel de `#[Valid]` en claves con puntos. Generar desde los atributos volvería a derivar todo eso a mano y divergiría la primera vez que cambiara un cuerpo de `toRules()`. Generar desde el manifiesto no puede divergir, porque el manifiesto es el contrato.

Este es el mapeo real, restricción a restricción, con las reglas compiladas en la columna central para que veas que el mapeador lee reglas y no atributos:

Restricción	Compila a	JSON Schema
<code>#[NotBlank]</code>	<code>required</code> , <code>string</code> , <code>regex:/\S/</code>	miembro añadido a <code>required</code> ; <code>type: string</code> ; <code>pattern: \S</code>
<code>#[NotEmpty]</code>	<code>required</code>	miembro añadido a <code>required</code>
<code>#[NotNull]</code>	<code>present</code> + un objeto regla <code>NotNull</code>	miembro añadido a <code>required</code> y <code>null</code> retirado de la unión de tipos
<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> , con los delimitadores PCRE y las banderas inocuas <code>D</code> / <code>u</code> retiradas
<code>#[Digits(i, f)]</code>	<code>numeric</code> , un <code>regex:</code> anclado	<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>

Restricción	Compila a	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> (la mitad <code>after:now</code> se registra, no se mapea)
<code>#[Size(min, max)]</code>	un objeto regla <code>Size</code>	<code>minLength</code> / <code>maxLength</code> , o <code>minItems</code> / <code>maxItems</code> cuando el tipo es un array
<code>#[UuidValue]</code>	un objeto regla <code>Uuid</code>	<code>format: uuid</code> + el <code>pattern</code> de UUID
<code>#[Phone]</code>	un objeto regla <code>E164</code>	<code>format: phone</code> + <code>pattern: ^\+[1-9]\d{1,14}\$</code>
<code>#[CurrencyCode]</code> / <code>#[CountryCode]</code>	objetos regla <code>Currency</code> / <code>CountryCode</code>	<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>	objetos regla	<code>format: bcp47</code> / <code>format: postal-code</code> , cada uno con su patrón
<code>#[Iban]</code> / <code>#[Swift]</code> / <code>#[Bic]</code>	objetos regla	<code>format: iban</code> / <code>swift</code> / <code>bic</code> , patrón retenido
<code>#[Cusip]</code> / <code>#[Isin]</code> / <code>#[Luhn]</code> / <code>#[RoutingNumber]</code>	objetos regla	solo <code>format</code> ; el dígito de control se registra, no se mapea
<code>#[Percentage]</code>	un objeto regla <code>Percentage</code>	<code>type: number</code> , <code>minimum: 0</code> , <code>maximum: 100</code>

Restricción	Compila a	JSON Schema
<code>#[Money]</code>	un objeto regla <code>PositiveMoney</code>	<code>type: number ,</code> <code>exclusiveMinimum: 0 ,</code> <code>multipleOf: 0.01</code>
<code>#[DecimalScale(n)]</code>	un objeto regla <code>DecimalScale</code>	<code>multipleOf</code> — la escala 2 se convierte en <code>0.01</code>

Tres filas de esa tabla compensan una mirada más detenida.

El patrón se retiene allí donde la regla normaliza primero. `Iban` quita espacios y pasa a mayúsculas antes de comparar; `Bic`, `Swift`, `Cusip` e `Isin` pasan a mayúsculas; `Luhn` y `RoutingNumber` quitan separadores. Publicar el patrón posterior a la normalización rechazaría cargas que el servidor acepta encantado, lo cual es peor que subespecificar — así que se emite el `format` y el patrón no.

`format` es un vocabulario abierto. En JSON Schema 2020-12 un `format` desconocido es una anotación, no un error. IBAN, BIC, ISIN, CUSIP y E.164 no tienen nombre de formato registrado, así que se emiten unos autodescriptivos (`iban`, `bic`, ...) en lugar de nada.

La nulabilidad se escribe a la manera de 3.1. OpenAPI 3.1 es JSON Schema 2020-12, que eliminó la palabra clave `nullable: true` de 3.0 en favor de una unión de tipos. Un miembro nutable es por tanto `"type": ["string", "null"]`, y un `enum` gana además un miembro `null` — ensanchar solo el tipo dejaría a `null` fallando la enumeración, y la propiedad quedaría, en la práctica, indocumentable como nula.

La única regla que decide toda la fusión es **gana quien escribe primero**, aplicada primero al tipo declarado y luego a las reglas en orden de declaración — el mismo orden en que las aplica el validador:

```

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

```

Sembrar el tipo PHP declarado *antes* de ver ninguna regla es la razón de que `#[Min(1)] int $quantity` se quede en `type: integer` en vez de ensancharse a `number` por la cadena de regla `numeric` que emite `#[Min]` — un ensanchamiento que documentaría erróneamente `1.5` como aceptable.

Aquí está toda esa tubería sobre un DTO real. Es el propio fixture de pruebas del generador, elegido porque abarca deliberadamente cada ruta de mapeo que tiene el generador — una cadena con límite de longitud, un nutable del contrato de nulos Jakarta, un entero con límites numéricos, un decimal con escala, un enum respaldado, un DTO anidado con `#[Valid]`, un patrón PCRE y un objeto regla:

```

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,
    ) {}
}

```

```
    ) {}
}
```

...y este es el componente que genera, literalmente:

```
{
  "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"]
}
```

Cada miembro de ese objeto es rastreable hasta algo de la clase de arriba: el enum vino de los propios casos del enum respaldado, el `$ref` de `#[Valid]`, las dos uniones nulables de `?string`, el `multipleOf` de `#[DecimalScale(2)]`, y `required` de las reglas de presencia — `coupon` e `idempotencyKey` están ausentes de él porque nada afirma su presencia.

Los DTOs de Lumen muestran la misma maquinaria a menor escala, y uno de ellos muestra un caso que la tabla anterior no puede: una propiedad que lleva **dos** patrones.

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

`#[NotBlank]` `#[CurrencyCode]` `string $currency` produce genuinamente dos patrones — `\\S` de la regla de no-blanco y `^[A-Z]{3}$` de la regla de divisa — y JSON Schema tiene exactamente una ranura `pattern` por objeto de esquema. Colapsarlos quedándose con el último descartaría en silencio la garantía de no-blanco, así que varios patrones se convierten en un `allOf` de subesquemas de un patrón cada uno. Un solo patrón sigue siendo un `pattern` liso; el `allOf` aparece únicamente cuando hace falta.

Nada se descarta en silencio

Algunas restricciones no tienen equivalente alguno en JSON Schema, y unas pocas se mapean solo de forma aproximada. Descartarlas calladamente produciría un documento que promete *menos* validación de la que realiza el servidor — un cliente enviaría una carga que la especificación llama válida y recibiría un `422` de vuelta. El caso por defecto del mapeador dice qué ocurre en su lugar:

```
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);
    }
  }
  // ...
}

```

Todo lo no reconocido — `after:now` y `before:now`, `exists:` y `unique:`, un dígito de control Luhn o CUSIP sin más, un límite que nombra a otro campo (`gte:other_field`) en vez de a un número, y cualquier `ValidationRule` de terceros — se registra bajo `x-firefly-constraints`. También un patrón que el mapeador solo pudo aproximar: `D` y `u` se descartan por ser inocuas de verdad, pero cualquier otra bandera — `i` sobre todo, para la que ECMA-262 no tiene sintaxis en línea dentro de una cadena de patrón — no puede trasladarse, así que el patrón se emite igualmente (es la afirmación verdadera más próxima disponible) y la regla original se registra, de modo que un lector puede ver que el patrón publicado es más estricto que el del servidor.

Tres propiedades, tres desenlaces:

```

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

```

Las herramientas conformes ignoran las tres extensiones y ven un documento válido. Un humano, o un generador que escribas tú, puede leer la verdad completa.

Respuestas: un componente de problema, y un conjunto de errores derivado

Cada operación termina en el mismo componente de error, y ese componente describe lo que LaraFly *realmente* devuelve, no lo que la RFC 9457 describe en abstracto:

```
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]]],
        ];
    }
    // ...
}
```

La distinción importa porque las dos formas difieren. El `ErrorResponse::toArray()` del Capítulo 4 emite `status`, `title`, `code`, `category` y `severity` incondicionalmente, luego `detail` / `type` / `instance` / `traceId` / `timestamp` solo cuando no son nulos, y luego `errors` solo cuando no está vacío. Así que `code`, `category`, `severity` y `errors` son miembros de Firefly encima de los cinco de la RFC; `type` se escribe en todos los documentos como `about:blank` — el RFC 9457 §3.1.1 dice que un `type` ausente es `about:blank`, y LaraFly toma la decisión del

`ProblemDetail` de Spring y lo dice, mientras que `firefly.web.problem.type-uri` o bien apunta a una URI base de la que derivar un tipo real a partir del `code` estable, o bien se pone a `'` para omitir el miembro; e `instance` es una **referencia URI relativa a la raíz**, construida por `ProblemMapper::instanceFor()`, y no la *ruta* de petición pelada que llevaba hasta la pasada del RFC 9457. Documentar la forma de la RFC en vez de esta le entregaría a cada cliente generado un decodificador que descarta en silencio los tres miembros sobre los que un llamante realmente ramifica.

Las dos enumeraciones se leen directamente de los propios enums del kernel, de modo que un caso añadido en `firefly/kernel` aparece en la especificación en la siguiente generación sin ninguna edición en `firefly/openapi`:

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

Qué estados enumera una operación es algo **derivado, no adivinado**. Compara dos operaciones reales de Lumen. `GET /api/v1/wallets/{id}/balance` toma una variable de ruta `string` y ningún cuerpo:

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

`POST /api/v1/wallets/{id}/deposit` toma la misma variable de ruta más un `#[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" }
}
```

El `400` aparece exactamente cuando la operación tiene algo que `ArgumentResolver` pueda rechazar *antes* de que corra el controlador — un cuerpo que decodificar y enlazar, una subida que validar, una query o cabecera obligatoria que el cliente puede omitir, o un parámetro no-`string` que hay que coercion desde la cadena del cable. Está deliberadamente ausente de `balance`: nada de esa petición puede fallar el enlace, porque un segmento de ruta ausente no casa con la ruta en absoluto, y un `400` documentado que el endpoint no puede producir es ruido que un cliente generado convierte en una rama de error muerta. El `404` se deriva de un parámetro de ruta con patrón. El `422` se deriva de cuerpos tipados con `#[Valid]`, siguiendo la rama que llama a `BeanValidator`; los parámetros de consulta, servicios y cuerpos sin tipo no la activan. Y `default` cubre todo lo que el propio manejador pueda levantar — un `404` de una `ResourceNotFoundException`, un `409` de una `ConflictException`, un `403` de un `#[PreAuthorize]` denegado — que no puede enumerarse desde el manifiesto de rutas sin leer el cuerpo del controlador, y que de todos modos se renderiza todo a través del mismo `ProblemDetailsRenderer`.

Un `204`, o un retorno `void / never`, no obtiene contenido alguno, porque emitir un mapa de contenido para un estado que no lleva cuerpo es exactamente lo que un generador de clientes estricto convierte en un tipo de retorno fantasma. El cuerpo de éxito de todo lo demás es el asunto de la siguiente sección.

El cuerpo de éxito: lo que un endpoint devuelve de verdad

Mira otra vez las dos operaciones de arriba. Ambas respuestas correctas son

```
{"type": "object"}
```

 — un objeto sin miembros.

Esa era toda respuesta correcta en todo documento que este generador producía, y es la que más importa: un visor la dibuja como un panel en blanco y `openapi-generator` la convierte en `any`, así que la frase más útil que contiene un documento de API — *esto es lo que recibes de vuelta* — era la única que faltaba, en todos los endpoints de todas las aplicaciones.

El razonamiento había sido que un `@return array{...}` es texto de comentario que ninguna otra parte del framework trata como vinculante. Eso ya había dejado de ser cierto. `RouteScanner` lee `@param list<X>` para compilar la tabla desde la que `ArgumentResolver` hidrata, así que una expresión de tipo en un docblock es exactamente igual de vinculante que un tipo declarado a la entrada. Y hay un argumento aún más fuerte: **PHPStan en nivel max ya comprueba estas expresiones contra el código en cada build**, que es lo que hace seguro leerlas. Un `@return` desactualizado es una puerta que falla, no una mentira silenciosa.

Así que el cuerpo de éxito sale ahora de tres fuentes, de la más específica a la menos. Tres métodos de una de las pruebas de respuesta del propio paquete muestran lo que lleva cada una de ellas:

```

/**
 * A page of consignments.
 *
 * @return array{page: positive-int, size: positive-int, total: int, items:
list<Consignment>}
 */
#[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 [];
}

```

La expresión `@return` de `index()` es la primera fuente, y produce este esquema — `positive-int` llevando su propio `minimum`, y `list<Consignment>` resolviendo el nombre corto a través de los propios imports del controlador hacia una referencia de componente:

```
{
  "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()` no necesita ningún tipo de docblock — su tipo de retorno declarado ya es una clase, que es la segunda fuente — y `shipments()`, cuyo tipo declarado es un `array` pelado, lo sostiene por completo su `@return list<Shipment>`. En orden, entonces:

1. La **expresión de tipo de `@return`** — el único sitio donde un `array` de PHP puede decir qué lleva dentro. La prosa escrita después del tipo se convierte en la `description` de la respuesta, que es la única descripción de respuesta que alguien escribe de verdad.
2. El **tipo de retorno declarado** — una clase se convierte en un `$ref` a un componente, un enum respaldado en su conjunto de valores, un escalar en sí mismo.
3. **Ninguno de los dos** — `type: object`, el comportamiento anterior, conservado como *reserva* para un retorno `array` sin nada dicho sobre él. Un `@return array<string, mixed>` se analiza sin problema y no significa nada, así que se trata

como que no dice nada en lugar de dejar que suprima lo que el tipo declarado sí sabía.

Un analizador, no otra expresión regular

El paquete ya tenía dos regex para la única forma que manejaba, `list<X>` y `X[]`, y no se pueden extender al resto. `array{items: list<array<string, Order>>}` necesita `<>` y `{}` balanceados y una coma que solo separa en el nivel exterior. Eso es una gramática, y una gramática quiere un analizador — unas doscientas líneas de descenso recursivo en `DocType`, frente a las cuatro dependencias transitivas que `phpstan/phpdoc-parser` metería en toda aplicación que instale este paquete.

Escrito	Se convierte en
<code>list<Order>, Order[], array<int, Order></code>	<code>type: array</code> con <code>items: {\$ref: Order}</code>
<code>array<string, Money></code>	<code>type: object</code> con <code>additionalProperties</code>
<code>array{a: int, b?: string}</code>	un objeto, <code>required: [a]</code> , <code>additionalProperties: false</code>
<code>array{a: int, ...}</code>	lo mismo, abierto — el <code>...</code> es lo único que lo levanta
<code>array{int, string}</code>	<code>prefixItems</code> — una tupla
<code>'draft'\ 'sent'</code>	<code>type: string</code> con <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</code> , <code>callable</code> , un nombre irresoluble	nada — quien llama vuelve a lo que ya sabía

Un `?` sobre una **clave** de shape significa «puede estar ausente» y se convierte en `required`; un `?` sobre el **valor** significa «puede ser null». Confundir ambos documenta como obligatorio un miembro omitible. Los nombres de clase se resuelven a través de los imports del fichero donde se escribió la expresión, porque la reflexión no expone las sentencias `use` de un fichero — sin eso solo funcionarían los nombres completamente cualificados, que es la única forma que nadie escribe.

Una clase devuelta se construye desde su forma de cable

`ResponseSchemaFactory` no es `DtoSchemaFactory`, y la diferencia es el asunto. Esa fábrica deriva los miembros del **constructor** y las reglas del `ConstraintManifest` — las dos fuentes correctas para una carga que el servidor enlaza y valida, y las dos equivocadas a la salida. Una respuesta nunca se valida, y sus miembros son lo que `json_encode` emite.

Que PHP escribe de dos maneras. Una clase que implementa `JsonSerializable` se serializa como lo que `jsonSerialize()` **devuelve**; todo lo demás como sus **propiedades públicas**. El `App\Orders\Order` del esqueleto es el caso que decide el diseño:

```
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` es un **método** derivado, no una propiedad. Reflejar solo las propiedades publicaría cinco de los seis miembros que la API envía de verdad. El array shape enuncia los seis, PHPStan lo comprueba contra el método, y el generador lo lee — borra la anotación y `total` desaparece en silencio del documento mientras la API sigue enviándolo.

Una forma declarada solo gana cuando dice algo: `@return array<string, mixed>` en `jsonSerialize()` significa «un objeto, miembros desconocidos», que es estrictamente menos que la lista de propiedades que habría suprimido, así que se ignora en favor de la reflexión.

Una regla se invierte a la salida. **Ser nullable no es ser opcional aquí.** Un miembro de respuesta está presente o ausente, y `?int $id` está siempre *presente* y a veces es null — así que los miembros de respuesta siguen siendo `required` y los nullable ensanchan su tipo. La regla del lado de la petición habría dicho a todo cliente que esperase una ausencia que nunca ocurre.

`#[ApiResponse]` también acepta una expresión de tipo

```

/**
 * 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 [];
    }

```

(El cuerpo vacío no es una omisión: esta es una de las pruebas de respuesta del propio paquete, y lo único que el generador lee son la firma, el docblock y los atributos.)

`type` es una expresión completa, no solo el nombre de una clase o de un escalar, y un nombre corto se resuelve a través de los propios imports del controlador — `Consignment::class` y la cadena `'list<Shipment>'` los maneja el mismo analizador. Fíjate también en lo que hacen juntos los tres estados: el `201` de `#[PostMapping]`, más un `#[ApiResponse]` por cada desenlace alternativo, cada uno con su propio esquema.

Las rutas HTML `#[Controller]` no son operaciones

El Capítulo 4 presentó `#[RestController]` junto a su hermano HTML `#[Controller]`, y solo el primero es una API JSON. El generador honra esa distinción por defecto:

```

    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;
    }

```

Lee el nombre antes que el cuerpo: `excluded()` responde *deja esta ruta fuera*, así que cada `return true` de arriba es una ruta que **no** llega al documento. El primer brazo es la regla de `#[Controller]`, el segundo es `#[ApiIgnore]`, y el tercero es la lista configurada de prefijos de ruta.

Las rutas `#[Controller]` quedan excluidas por defecto. Activa `firefly.openapi.include-html` para incluirlas. El medio de respuesta se deriva del retorno: las vistas y los valores `Htmlable` producen `text/html`, los arrays producen JSON y las redirecciones llevan `Location`. El estereotipo no determina el medio de respuesta.

La segunda mitad de ese método es el instrumento romo para todo lo demás: `firefly.openapi.exclude` es un CSV de prefijos de ruta — `'/internal,/admin'` — para rutas que son JSON pero no son la API pública de nadie.

El visor: tres estilos, y solo uno de ellos llama fuera

`GET /openapi` renderiza una consola de navegador sobre el documento. Cuál de ellas la decide `firefly.openapi.viewer.style`, y la elección es una decisión de cadena de suministro disfrazada de preferencia:

style	Se sirve desde	¿Petición a un tercero en cada visita?
<code>swagger</code> (por defecto)	tu propio origen, desde el paquete de composer <code>swagger-api/swagger-ui</code>	no
<code>builtin</code>	en línea en la respuesta	no
<code>cdn</code>	<code>cdn.jsdelivr.net</code>	sí, en cada visita

Un valor no reconocido cae de vuelta a `swagger` en lugar de renderizar una página en blanco — una errata en un fichero de configuración no debería costarte nada.

Por qué el valor por defecto es el Swagger UI oficial, desde tu propio origen

Todos los visores de estantería — Swagger UI, Redoc, Elements — son aplicaciones JavaScript empaquetadas, y durante años eso dejaba a un paquete PHP exactamente dos opciones. Incrustar un bundle minificado de varios megabytes en el historial git del propio paquete, de modo que cada clon de cada proyecto dependiente lo pague para siempre y el framework quede atado a un tren de releases que no puede parchear sin publicar una versión propia. O traerlo de una CDN en cada visita, lo que es una dependencia de cadena de suministro y una cuestión de protección de datos, y lo que *no renderiza en absoluto* en los entornos aislados y de CSP estricta donde más se quiere una consola de API interna.

Hay una tercera opción, y este paquete la toma. `swagger-api/swagger-ui` publica su `dist` en Packagist bajo Apache-2.0, así que **composer** puede traerlo y fijarlo — es un `require` duro de `firefly/openapi`, de modo que los ficheros ya están en disco en `vendor/` cuando llegas por primera vez a la ruta — y `SwaggerAssetAction` sirve esos ficheros desde el propio origen de la aplicación:

```
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;
    }
}
```

Obtienes la consola byte a byte tal y como la publica Swagger — el conjunto completo de funciones, deep linking, try-it-out, el popup de redirección OAuth2 — sin petición a CDN, sin paso de npm, y sin nada en el historial de este repositorio que un `composer`

`update` no pueda sustituir. La cabecera de caché larga es segura precisamente porque los bytes son inmutables para una versión fijada: composer solo los cambia cuando cambia la versión fijada, y el ETag cambia con ellos.

❗ EL PATH TRAVERSAL SE DEFIENDE CON UNA LISTA BLANCA, NO CON UN SANEADOR

Solo siete nombres de fichero son servibles siquiera — `swagger-ui.css`, `swagger-ui-bundle.js`, `swagger-ui-standalone-preset.js`, `oauth2-redirect.html`, dos favicons e `index.css` — y cada ruta resuelta se comprueba con `realpath()` para estar dentro del directorio `dist`. La propia ruta restringe `{file}` a `[A-Za-z0-9._-]+`, así que ni siquiera puede *expresar* un traversal. Cotejar contra una lista fija en lugar de limpiar la entrada es la elección deliberada: una lista blanca no puede vencerse con un truco de codificación que se le escapó a un saneador. Cualquier otra cosa es un 404 `text/plain` simple — no `problem+json`, porque quien llama aquí es un navegador pidiendo una hoja de estilos, no un cliente de API.

El directorio `dist` se localiza preguntando a **los metadatos de versiones instaladas del propio Composer** por la raíz del paquete, en lugar de subir directorios desde `__DIR__`. La profundidad de `src/Web/` a `vendor/` difiere entre un paquete instalado (`vendor/firefly/openapi/src/Web`) y este monorepo (`packages/openapi/src/Web`), así que un recorrido relativo funcionaría exactamente en uno de los dos; ese recorrido se conserva solo como respaldo para un runtime cuyo autoloader no pueda responder.

Y si la distribución falta de verdad — un `vendor/` recortado, un phar, un runtime sin composer — `render()` cae de vuelta en lugar de servir una página cuyos assets dan 404:

```
final class ViewerPage
{
    // ...
    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),
    };
}
// ...
}

```

Para qué sirve **builtin**

Una referencia escrita a mano y sin dependencias: un script en línea, unos cientos de bytes de CSS, un solo `fetch` a la ruta de la especificación, y una paleta que sigue a `prefers-color-scheme`. Hace las dos cosas que quien lee realmente necesita de una especificación generada y que el JSON en crudo no le da — agrupa las operaciones por etiqueta con verbos y rutas visibles de un vistazo, y **resuelve los punteros `$ref` en el cliente**, de modo que quien lee ve los miembros de un DTO y no un puntero a `#/components/schemas`. Try-it-out, flujos OAuth y ejemplos de código están deliberadamente ausentes; para eso está `swagger`.

Elígelo cuando la regla del despliegue sea *nada de JavaScript de terceros en la respuesta*, y no meramente *nada de hosts de terceros*.

POR QUÉ ESA PÁGINA ES UN NOWDOC

El visor integrado incrusta una aplicación JavaScript, y un **heredoc** de PHP interpola variables. Cada `$ref`, `$schema` y `$1` de ese script se leía por tanto como una variable PHP — `$ref` se convertía calladamente en la cadena vacía, y la resolución de `$ref`, que es todo el sentido de la página, dejaba de funcionar. Un nowdoc toma el script literalmente y las dos sustituciones reales se hacen explícitamente después. Es la clase de bug que no produce ningún error en ninguna parte: la página renderiza, y sencillamente muestra punteros en lugar de esquemas.

Lo que cuesta `cdn`

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

Cada visita carga entonces Swagger UI desde `cdn.jsdelivr.net`. La versión está fijada exactamente, y **no se declara ningún hash de Subresource Integrity** — deliberadamente: un hash que el framework no puede verificar en el momento de publicar es teatro de seguridad, y uno equivocado simplemente rompería la página.

Sopesa el intercambio con honestidad. A cambio de una petición a un tercero en cada visita, de una Content-Security-Policy que tiene que permitir ese host, y de una consola que no renderiza nada en un despliegue aislado, obtienes... el mismo Swagger UI que `swagger` ya te servía desde tu propio origen. El estilo se mantiene porque es lo que muestran la mayoría de los tutoriales, y porque algunas organizaciones prefieren genuinamente que sus bytes vengan de una caché en la que ya confían — no porque sea el mejor valor por defecto.

⚠ `VIEWER.CDN` SIGUE GANANDO SOBRE `VIEWER.STYLE`

`firefly.openapi.viewer.cdn` (por defecto `false`) es la gráfica booleana antigua de esta opción, de antes de que `style` existiera. Sigue **forzando** la página de CDN y anula a `style`, de modo que una aplicación que la fijó conserva el comportamiento que configuró en lugar de que una actualización del framework la mueva calladamente a otra consola. Prefiere `style` en configuración nueva; borra `cdn` cuando lo adoptes.

El visor trae la especificación desde la ruta hermana en lugar de tener el documento incrustado en la página, de modo que una especificación regenerada aparece con un simple refresco del navegador, y de modo que las dos rutas puedan exponerse de

forma independiente — un despliegue puede muy bien querer el documento legible por máquina público y la consola apagada, o al revés. La URL de la especificación se resuelve a través del `UrlGenerator` en lugar de concatenarse, porque una app montada bajo un subdirectorio o detrás de `APP_URL` obtendría si no un enlace que da 404 desde cualquier página que no sea la raíz, y un visor cuya única llamada de red es incorrecta es un visor que no muestra nada en absoluto.

Configuración, y asegurar la superficie

Todo lo que el documento y sus rutas necesitan vive bajo una sola clave de configuración. El bloque de abajo está escrito como PHP vivo para que se lea mejor; `skeleton/config/firefly.php` entrega esas mismas claves **comentadas**, con su valor por defecto al lado, porque `firefly/openapi` es un paquete opcional y un bloque sin comentar sería una decisión tomada por ti:

```
<?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` acepta las dos grafías que usa un fichero de configuración real — una lista de URLs sueltas, y la propia forma de objeto de OpenAPI con una `description` — y una entrada que no sea ninguna de las dos se descarta en lugar de emitirse, porque un Server Object sin `url` es inválido bajo el esquema 3.1 y envenenaría un documento por lo demás correcto.

Un despliegue público se asegura como cualquier otra ruta. Las reglas de `HttpSecurity` del Capítulo 10 — más adelante, en la Parte III — cubren las rutas de especificación y visor sin ningún borde de código, porque `HttpSecurityFilter` es un middleware global y corre para las rutas registradas nativamente exactamente igual que corre para tus controladores:

```

<?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'],
            ],
        ],
    ],
];

```

```
    ],
};
```

Fíjate en los tres patrones. `openapi` a secas no casa con `openapi/assets/swagger-ui.css`, y `openapi.json` es un literal aparte — un conjunto de reglas que cubre la consola pero no sus assets produce una página autenticada cuya hoja de estilos responde 401, que es peor resultado que cualquiera de los dos extremos.

La alternativa, para un despliegue que no quiere ninguna superficie de documentación en producción, es `enabled => false` más un paso `firefly:openapi --output=` en CI.

Reemplazar una pieza de la tubería

Cada colaborador del paquete — `OpenApiProperties`, `ConstraintSchemaMapper`, `DtoSchemaFactory`, `OperationFactory`, `OpenApiGenerator` y `ViewerPage` — es un `#[Bean]` tras un `#[ConditionalOnMissingBean]`, el mecanismo del Capítulo 2.

Enseñarle al generador tu propia `ValidationRule` es por tanto una `#[Configuration]` corta en tu aplicación y nunca un fork:

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

❗ LA REFLEXIÓN DE AQUÍ ES REAL, Y NO ESTÁ EN EL CAMINO DE PETICIÓN

`DtoSchemaFactory` refleja el constructor de un DTO para conocer los tipos de sus propiedades, lo que parece una violación de la regla que el Capítulo 13 enunciará por completo: nada en el camino de petición cacheado refleja. No lo es. Esa reflexión corre cuando `firefly:openapi` genera un fichero, o ante un impacto en la ruta de la especificación — cuyo resultado el generador memoiza durante toda la vida del proceso — y nunca mientras se despacha una petición de la aplicación. Es la misma categoría de trabajo que `RouteScanner` y `ConstraintScanner`, que ambos reflejan solo en tiempo de compilación. La alternativa, enseñar a `RouteScanner` a emitir tipos por propiedad dentro de cada `RouteDescriptor`, se rechazó porque haría crecer el manifiesto de rutas compilado de *todas* las apps en beneficio de un único paquete opcional.

🏠 PARIDAD CON LARAVEL

Laravel de serie no trae soporte OpenAPI. Las respuestas habituales son un paquete de terceros gobernado por su propio dialecto de anotaciones (los bloques `@OA\` de `zircote/swagger-php`, o las clases de atributos de `vyuldashev/laravel-openapi`) o un fichero YAML mantenido a mano — ambos son una *segunda* descripción de la API, situada junto a las rutas y los FormRequests que realmente la imponen, y ambos divergen. `firefly/openapi` no tiene dialecto que aprender porque no tiene segunda descripción: lee el mismo `RouteManifest` desde el que despacha el dispatcher y el mismo `ConstraintManifest` con el que valida el validador. El análogo más cercano fuera de PHP es `springdoc-openapi`, y lo más parecido dentro del propio Laravel es `php artisan route:list` — exacto por la misma razón, e incapaz por la misma razón de decirte nada sobre un cuerpo de petición.

✓ Lo que aprendiste

Concepto	Qué hace
<code>OpenApiGenerator</code>	Ensambla un documento OpenAPI 3.1 desde <code>RouteManifest</code> + <code>ConstraintManifest</code> ; memoizado por instancia, ordenado de forma determinista para que las regeneraciones diferencien limpiamente
<code>firefly:openapi</code>	Escribe el documento en <code>--output=</code> o en crudo a stdout, convirtiendo la especificación en un artefacto versionable que un job de CI puede diferenciar
<code>OpenApiRouteRegistrar</code>	Monta <code>/openapi.json</code> , <code>/openapi</code> y <code>/openapi/assets/{file}</code> nativamente desde un <code>BootPass</code> — rutas configurables que una ruta por atributo nunca habría podido tener, y sin autodocumentación
<code>OperationFactory</code>	Mapea cada <code>kind</code> de enlace — <code>path</code> / <code>query</code> / <code>header</code> / <code>file</code> / <code>body</code> — a su forma OpenAPI; los enlaces <code>service</code> nunca aparecen
<code>SchemaRegistry</code>	Un componente por DTO, alcanzado por <code>\$ref</code> : sin tipos generados duplicados, y un nombre reservado cierra un ciclo recursivo de <code>\$ref</code>
<code>DtoSchemaFactory</code>	Fusiona los tipos declarados del constructor con las restricciones compiladas; no emite <code>additionalProperties: false</code> , porque el servidor ignora las claves extra

Concepto	Qué hace
<code>ConstraintSchemaMapper</code>	Mapea la lista de reglas compilada — no los atributos — a palabras clave; gana quien escribe primero, así que <code>#[Min(1)] int</code> se queda en <code>integer</code>
<code>x-firefly-constraints</code>	Registra lo que JSON Schema no sabe enunciar (<code>after:now</code> , un dígito de control, un PCRE con banderas, una regla de terceros) en lugar de descartarlo
<code>ProblemSchema</code>	La única respuesta compartida <code>application/problem+json</code> ; documenta <code>code / category / severity / errors</code> de Firefly, con los enums leídos de los propios casos del kernel
Conjunto de errores derivado	<code>400</code> solo cuando algo es rechazable antes de que corra el controlador, <code>404</code> para patrones de ruta, <code>422</code> para cuerpos tipados con <code>#[Valid]</code> , <code>default</code> siempre
<code>DocType</code>	Compila una expresión de tipo PHPDoc a un fragmento de JSON Schema — shapes, genéricos, tuplas, uniones de literales, pseudo-tipos de PHPStan — y devuelve <i>nada</i> en lugar de adivinar cuando no puede leer una
<code>ResponseSchemaFactory</code>	Construye una clase devuelta desde su forma de CABLE: el <code>@return</code> declarado de <code>jsonSerialize()</code> cuando lo hay, las propiedades públicas si no. Los miembros de respuesta siguen siendo <code>required</code> y los nullable ensanchan su tipo

Concepto	Qué hace
<code>#[ApiResponse(type:)]</code>	Una expresión de tipo completa (<code>'list<Shipment>'</code>), resuelta con los propios imports del controlador
<code>\$route->html</code>	Las rutas <code>#[Controller]</code> quedan excluidas por defecto; <code>firefly.openapi.include-html</code> las incluye con el medio derivado de su retorno
<code>firefly.openapi.viewer.style</code>	<code>swagger</code> (por defecto) <code>builtin</code> <code>cdn</code> . Solo <code>cdn</code> hace una petición a un tercero en cada visita; un valor no reconocido cae de vuelta a <code>swagger</code>
<code>SwaggerAssets</code>	Sirve el Swagger UI OFICIAL desde tu propio origen, desde el paquete de composer <code>swagger-api/swagger-ui</code> — siete nombres de fichero en lista blanca, cada uno comprobado con <code>realpath()</code> dentro del directorio <code>dist</code>
<code>ViewerPage::render()</code>	Cae de vuelta a <code>builtin</code> cuando falta la <code>dist</code> de Swagger, en lugar de renderizar una consola cuyos assets dan 404

Ponlo en práctica

1. **Genera el documento de Lumen y léelo.** Ejecuta `php artisan firefly:openapi --output=openapi.json` en el sample y abre `/openapi` en un navegador. Busca `walletBalance` y confirma que no tiene respuesta `400`; luego busca

`walletDeposit` y confirma que tiene tanto un `400` como un `422` — y convéncete, con las reglas de este capítulo, de por qué difieren.

2. **Convierte la especificación en una puerta de CI.** Versiona el fichero generado y añade un job que lo regenere y ejecute `git diff --exit-code` sobre él. Cambia un DTO — añade un `#[Size(max: 32)]` a `OpenWalletRequest::$owner_id` — y observa al job fallar con un diff que nombra la palabra clave de esquema exacta que cambió.
3. **Demuestra que la consola por defecto no hace ninguna petición saliente.** Abre `/openapi` en el sample con el panel de red del navegador grabando, y confirma que todas las peticiones son del mismo origen: la página, `openapi/assets/swagger-ui.css`, los dos bundles y `openapi.json`. Luego pon `firefly.openapi.viewer.style` a `cdn`, recarga, y observa aparecer `cdn.jsdelivr.net` en ese mismo panel — esa petición es toda la diferencia, y es lo que una CSP estricta o un host aislado bloquearía.
4. **Borra una anotación y mira cómo el documento pierde un miembro.** En el esqueleto, quita el `@return array{...}` de `App\Orders\Order::jsonSerialize()`, regenera, y encuentra `total` ausente del esquema `Order` mientras `GET /orders/1` lo sigue devolviendo. Vuelve a ponerlo, luego cambia `total: float` por `total: string` y ejecuta PHPStan: la puerta que mantiene honesto al documento es la que falla.
5. **Observa a una restricción caer hasta la extensión.** Añade `#[Future]` a una propiedad `string` de un DTO de petición, regenera, y encuentra el array `x-firefly-constraints` de la propiedad llevando `after:now` junto a un `format: date-time` perfectamente corriente. Luego añade `#[Pattern('/^[a-z]+$/i')]` a otra propiedad y compara: el patrón *sí* se publica, y la regla original se registra a su lado porque la bandera `i` no pudo sobrevivir a la traducción.

PARTE II

Modelar y Persistir el Dominio

Persistencia y el Patrón Repositorio

Al terminar este capítulo conocerás los dos puertos que implementa todo repositorio de LaraFly, cómo `EloquentRepository` te da CRUD completo por el precio de una sola asignación a `$model`, cómo el nombre de un método de consulta derivada se compila en una consulta Eloquent real sin cuerpo propio, el escape `#[Query]` para cualquier cosa que un nombre no pueda expresar con limpieza, cómo las `Specification` componen predicados reutilizables, cómo la **consulta por ejemplo** convierte una entidad a medio rellenar en una consulta sin escribir un solo predicado, qué dice de un método cada uno de los cuatro atributos de método (`#[Modifying]`, `#[Projection]`, `#[Lock]`, `#[EntityGraph]`), los objetos de valor `Page` / `Pageable` / `Sort` que transportan la paginación de principio a fin y el `Slice` que deliberadamente no cuenta, la familia `DataAccessException` a la que se traduce todo error de driver — y dónde esa traducción deliberadamente no ocurre — y el puente de eventos de dominio que conecta la llamada `save()` de un repositorio con el agregado del Capítulo 6.

❗ TÉRMINO NUEVO: CONSULTA DERIVADA

Una **consulta derivada** es un método de repositorio **sin cuerpo** — solo un nombre como `findByOwnerId` — cuyo SQL compila el framework analizando el propio nombre del método, en el momento en que se llama por primera vez. Tú escribes el nombre; el framework escribe la cláusula `WHERE`.

Los puertos: **CrudRepository** y **PagingAndSortingRepository**

Todo repositorio de LaraFly implementa en última instancia dos pequeñas interfaces de **firefly/data**. PHP no tiene genéricos en tiempo de ejecución, así que las firmas de los puertos son deliberadamente **object / mixed / array** — los parámetros **@template** del docblock le devuelven a PHPStan los tipos precisos de entidad e id:

```
/**
 * // ...
 * @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 sobrecarga un único `findAll(Pageable)` / `findAll(Sort)` ; PHP no tiene sobrecarga de métodos, así que `PagingAndSortingRepository` divide los dos en métodos distintos, tipados con limpieza:

```

/**
// ...
 * @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;
}

```

No implementarás ninguna de las dos interfaces directamente.

`Firefly\Data\Repository\EloquentRepository` ya implementa ambas sobre `Model::query()`, y cada repositorio concreto de tu aplicación extiende eso — la misma relación que describió el Capítulo 2 entre una interfaz y su único bean implementador, ahora un nivel más abajo en la pila.

PARIDAD CON LARAVEL

`CrudRepository / PagingAndSortingRepository` son los equivalentes en LaraFly de `CrudRepository / PagingAndSortingRepository` de Spring Data. Si has escrito una `interface OrderRepository extends JpaRepository<Order, UUID>` de Spring, `EloquentRepository` es la misma idea trasladada a Eloquent: obtienes la superficie CRUD completa con un solo `extends`, y añades solo las consultas específicas de tu entidad.

El puerto y el adaptador: `WalletRepository`

El Capítulo 2 introdujo la división hexagonal entre un *puerto* del que dependen tus capas de dominio y aplicación, y el *adaptador* que realmente habla con el almacenamiento. La persistencia es donde esa división demuestra su valor. Aquí está el puerto completo, sin cambios desde el Capítulo 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 findByIdOwnerId(string $ownerId): array;
}
```

Y aquí está el adaptador — la clase que extiende `EloquentRepository` y hace el trabajo real:


```

<?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;
    }
}

```

Fíjate en la línea `protected string $model = Wallet::class;` — esa única asignación es lo que le dice a `EloquentRepository` contra qué modelo debe correr `query()`, y es la razón completa por la que la superficie CRUD entera (`save`, `saveAll`, `findById`, `findAll`, `findAllById`, `existsById`, `count`, `delete`, `deleteById`, `deleteAll`, más paginación y ordenación) aparece en `EloquentWalletRepository` de forma gratuita. `$this->model::query()` es la única línea al final de cada método heredado.

⚠ UNA RESTRICCIÓN DE VARIANZA DE PHP, NO UNA DECISIÓN DE DISEÑO

Una clase que a la vez `extends EloquentRepository` e `implements WalletRepository` debe dar a cada método de la interfaz un cuerpo explícito — el despacho mágico `__call()` por sí solo no satisface una cláusula `implements`. Y como `WalletRepository::save(Wallet $wallet): Wallet` restringe el `save(object $entity): object` del padre, sobrescribir con el tipo de parámetro *más estrecho* es una sobrescritura inválida en PHP (los tipos de parámetro deben permanecer iguales o ensancharse, nunca estrecharse); `EloquentWalletRepository::save()` mantiene la firma `object $entity` del padre y solo estrecha el tipo de *retorno*, protegiendo el requisito real de `Wallet` con una comprobación `instanceof` dentro del cuerpo del método en su lugar.

Consultas derivadas: el nombre del método es la consulta

`findByOwnerId` de arriba no tiene lógica de consulta en su propio cuerpo — `dispatchQuery()` es el motor compartido debajo tanto de él como del despacho mágico `__call()`. Un método de repositorio cuyo nombre sigue la gramática de abajo no necesita ningún cuerpo cuando se llama a través de `__call()`;

`EloquentRepository::dispatchQuery()` analiza el nombre del método con `DerivedQueryParser::parse()` — análisis de cadenas puro, sin reflexión, el nombre del método es la única entrada — y conduce un `Builder` de Eloquent en consecuencia.

La gramática

Pieza	Valores	Notas
Prefijo	<code>findBy</code> / <code>countBy</code> / <code>existsBy</code> / <code>deleteBy</code>	<code>find</code> también acepta <code>findFirstBy...</code> (límite 1), <code>findTop{N}By...</code> (límite N), <code>findDistinctBy...</code> (añade <code>DISTINCT</code>) antes del <code>By</code> .
Conector	<code>And</code> / <code>Or</code>	Divide grupos de predicados; solo se reconoce antes de un nuevo campo en CamelCase.
Operador	ver abajo	Se empareja de más largo a más corto contra el final de cada grupo de predicados; <code>Equals</code> (sin token) es el implícito por defecto.
Modificador de mayúsculas	sufijo <code>IgnoreCase</code>	Encamina la familia de igualdad/LIKE a través de <code>LOWER(col) op LOWER(?)</code> .
Ordenación	<code>OrderBy{Field}{Asc Desc}</code> final	Encadenable; la dirección por defecto es <code>asc</code> si se omite.

Los operadores, en el orden exacto de coincidencia-más-larga-primero que recorre el analizador:

Operador	Efecto SQL	Argumentos consumidos
GreaterThanOrEqualTo	<code>>=</code>	1
LessThanOrEqualTo	<code><=</code>	1
GreaterThan	<code>></code>	1
LessThan	<code><</code>	1
Between	<code>BETWEEN ? AND ?</code>	2
NotLike	<code>not like</code>	1
Like	<code>like</code>	1
NotIn	<code>NOT IN (?)</code>	1 (array)
In	<code>IN (?)</code>	1 (array)
Containing	<code>like '%value%'</code>	1
StartingWith	<code>like 'value%'</code>	1
EndingWith	<code>like '%value'</code>	1
IsNotNull	<code>IS NOT NULL</code>	0
IsNull	<code>IS NULL</code>	0
Not	<code>!=</code>	1
True	<code>= true</code>	0
False	<code>= false</code>	0
(ninguno)	<code>Equals</code> → <code>=</code>	1

Cada segmento de campo en CamelCase se mapea a una columna `snake_case`, y cada columna derivada se valida contra un patrón de identificador desnudo antes de poder llegar a una consulta — el nombre del método es una entrada no confiable, alcanzable a través del `__call` público, así que esta validación es lo que mantiene libre de inyección el fragmento `LOWER(col)` en bruto de la vía `IgnoreCase`. Los argumentos se vinculan a los predicados en el orden de declaración, de izquierda a derecha, avanzando un cursor tantos valores como consume cada operador.

El propio `RecordRepository` de prueba del framework — contra el que corre la suite de consultas derivadas de `packages/data` — muestra la gramática de un vistazo. Las etiquetas `@method` son cómo PHPStan ve un retorno tipado para lo que, en tiempo de ejecución, es despacho dinámico `__call`:

```
/**
 * // ...
 * @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;
```

Ninguno de esos seis métodos está declarado en ninguna parte de la clase: `__call()` analiza cada nombre la primera vez que se usa y el manifiesto compilado responde a partir de entonces.

`EloquentWalletRepository::findByOwnerId()` es la versión real y distribuida de exactamente este patrón — la única diferencia es que `implements` `WalletRepository`, así que su cuerpo llama a `dispatchQuery()` explícitamente en vez de depender de `__call()`, por la razón de varianza explicada arriba. De cualquier forma, `findByOwnerId('alice')` se compila a `WHERE owner_id = ?` sin SQL escrito a mano.

💡 CUANDO UN NOMBRE SE VOLVERÍA ABSURDO, USA `#[Query]`

Los nombres derivados se leen bien hasta dos o tres predicados. Pasado eso, recurre al escape de consulta explícita de abajo en vez de a un nombre de método de cincuenta caracteres.

El escape de consulta explícita `#[Query]`

Un método puede declarar su SQL directamente, saltándose por completo la gramática de consulta derivada:

```
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;
}
```

El cuerpo es la misma línea única que lleva todo método de repositorio declarado: entregar el nombre propio del método y sus argumentos al despachador compartido, que encuentra el método en el conjunto `#[Query]` del manifiesto y ejecuta el SQL. El `assert()` y el `@var` están ahí por PHPStan, que no puede ver a través de `mixed`.

Los marcadores de posición nombrados `:param` se reescriben a `?` posicionales en el orden en que aparecen, y los propios argumentos del método se vinculan posicionalmente. Los métodos `#[Query]` los descubre el mismo `TransactionalScanner` que compila el manifiesto `#[Transactional]` (la sección sobre transacciones de un capítulo posterior cubre ese escáner en detalle), así que el SQL de un método declarado se resuelve desde el manifiesto compilado con cero reflexión por petición — tanto la vía de consulta derivada como un método `#[Query]` explícito confluyen en el único método `dispatchQuery()` compartido por `__call()` y cualquier cuerpo de método que lo llame directamente.

Specification : predicados componibles y reutilizables

Las consultas derivadas responden preguntas fijas. Una **Specification** es un predicado reutilizable que puedes nombrar una vez y componer libremente en el punto de llamada — "monederos con al menos este saldo", combinado con otras condiciones según haga falta:

```
interface Specification
{
    // ...
    public function toBuilder(Builder $query): Builder;
}
```

Specifications es la fábrica estática — las interfaces de PHP no pueden llevar cuerpos de fábrica estáticos:

```
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'));
```

Aplicar una especificación construida corre a través de dos métodos de repositorio que ya provee cada **EloquentRepository**:

```
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));
}
```

Cuatro ayudantes cargan con todo lo que esos dos cuerpos no deletrean, y la división de su visibilidad es el sentido del asunto. `reading()` y `translating()` son **protected** — son la costura a la que recurre un repositorio subclase, y es lo que pone a tu alcance la receta de «anotar una sobrescritura» de más abajo — mientras que `pageOf()` y `narrow()` son fontanería **private**. `reading(__FUNCTION__)` abre la consulta y aplica el `#[EntityGraph]` que el manifiesto compilado guarde *para esta clase de repositorio y este método*, que es por lo que anotar una sobrescritura es la receta entera. `narrow()` es el que el cuerpo paginado de arriba no llama: se queda solo con las filas que realmente son `TModel` y devuelve un `list<TModel>`. `pageOf()` cuenta sobre un clon del builder y luego lo recorta, así que la cuenta y el recorte ven el mismo predicado. Y `translating()` envuelve la llamada terminal para que un fallo del driver salga del repositorio como una `DataAccessException` tipada y no como una `QueryException` en bruto — el asunto de la última sección de este capítulo.

Una especificación para monederos por encima de un saldo mínimo, compuesta con un filtro de moneda, se lee exactamente como la regla que expresa:

```
$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))
```

ejecuta ese predicado compuesto, cuenta las coincidencias, ordena y recorta — devolviendo una `Page` sin SQL propio más allá de los dos cierres `where()`.

Consulta por ejemplo

A veces el predicado que quieres es sencillamente «*otra fila que se parezca a esta*». Spring Data lo llama un `Example`, y `firefly/data` lo porta de la manera que no cuesta nada aprender: **un `Example` es una `Specification`**. No tiene una vía de llamada aparte, ni un método de repositorio especial por el que sea la única forma de alcanzarlo, ni reglas de composición propias. Todo lo que mostró la sección anterior se le aplica sin cambios.

```
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());
    }
}
```

La **sonda** es aquello que se parece a lo que quieres. Puede ser un modelo Eloquent con algunos atributos fijados, cualquier objeto plano (sus propiedades públicas), o un array pelado de `columna => valor`. Una sonda de modelo aporta exactamente los

atributos que se *fijaron* — `new Record(['status' => 'open'])` es una sonda de una sola propiedad, no una sonda de todas las columnas de la tabla — porque los atributos se leen como los originales en bruto del modelo superpuestos con su conjunto sucio.

Cada clave de la sonda se valida como identificador pelado ahí mismo, en `of()`, y una inválida es una `InvalidArgumentException` antes de que se construya consulta alguna. Eso no es decoración defensiva: una clave llega hasta un fragmento de SQL en bruto cuando se ignora la caja o cuando un `LIKE` necesita su cláusula `ESCAPE`, así que se comprueba en el único sitio donde comprobarla una vez basta — la misma disciplina que el analizador de consultas derivadas aplica a los nombres de campo.

Como un `Example` es una `Specification`, los métodos de repositorio que ya conoces lo aceptan directamente, y hay cuatro métodos de conveniencia nombrados como un lector de Spring Data espera:

```
$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);
```

Una sonda vacía coincide con todo, que es la respuesta honesta en vez de un error: sin propiedades no hay predicado. `findOneByExample()` es la variante de como mucho uno — `null` si no hay ninguna fila, la fila si hay una, y `IncorrectResultSizeDataAccessException` si hay más de una, porque una sonda ambigua es un error de programación y no un resultado:

```
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);
```

El emparejador lleva las reglas

Una sonda por sí sola dice *qué* comparar. Un `ExampleMatcher` dice *cómo*. Es un objeto de valor inmutable cuyos `with*` devuelven todos una instancia nueva, así que un emparejador se puede construir una vez y compartir sin riesgo:

```
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]);
}
```

Las reglas que esos cinco combinadores expresan, al completo:

Regla	Por defecto	Cómo cambiarla
Cómo se combinan las propiedades	<code>AND (matching() = matchingAll())</code>	<code>ExampleMatcher::matchingAny()</code> para <code>OR</code>
Un valor <code>null</code> en la sonda	se omite por completo	<code>withIncludeNullValues()</code> lo convierte en <code>IS NULL</code>
Una propiedad que no quieres comparar	se compara	<code>withIgnorePaths('id', 'created_at')</code>
Cómo se compara un valor de cadena	<code>StringMatcher::EXACT (=)</code>	<code>withStringMatcher(StringMatcher::CONTAINING)</code> y compañía
Sensibilidad a mayúsculas	sensible a la caja	<code>withIgnoreCase()</code> para todas las cadenas, <code>withIgnoreCase('email')</code> para una
Una propiedad que necesita sus propias reglas	sigue los valores por defecto de arriba	<code>withMatcher('email', GenericPropertyMatcher::startsWith()->caseSensitive())</code>

Solo los valores de **cadena** reciben emparejamiento de cadena. Un `int`, un `float` o un `bool` se compara siempre con `=`, diga lo que diga el emparejador de cadenas por defecto — un emparejador `CONTAINING` no convierte calladamente `amount => 5` en `LIKE '%5%'`.

Un emparejador por ruta gana a los ajustes generales del emparejador para esa ruta y deja intacta cualquier otra:

```
$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` tiene cuatro casos — `EXACT`, `CONTAINING`, `STARTING`, `ENDING` — y los tres de `LIKE` escapan el valor antes de que se acerque siquiera a un patrón:

```
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,
    };
}
```


Así que una sonda de `under_` busca un guion bajo literal, no «cualquier carácter». El fragmento que `Example` emite lleva `ESCAPE '!'`, la única forma de escape que se lee igual en `sqlite`, `mysql`, `pgsql` y `sqlsrv`, cosa que una contrabarra no consigue (`mysql` se la come dentro de la cadena antes).

El emparejador `REGEX` de Spring está deliberadamente ausente: no hay SQL portable para él.

Compone, porque es una especificación

El sentido de portar `Example` como una `Specification` en vez de al lado de una es este — un ejemplo puede ser una hoja de un predicado mayor, y pagina exactamente igual que cualquier otro:

```
$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);
```

🔍 DÓNDE SE GANA EL SUELDO LA CONSULTA POR EJEMPLO

Un formulario de búsqueda de administración con ocho campos opcionales es el caso canónico. Construye la sonda con lo que la persona haya rellenado, ignora el resto, y no habrás escrito ni una línea de construcción condicional de consultas — los campos vacíos sencillamente no están en la sonda.

Decir más sobre un método: `#[Modifying]`, `#[Projection]`, `#[Lock]`, `#[EntityGraph]`

Cuatro atributos permiten que un método de repositorio diga algo que el *nombre* del método no puede. Antes de cualquiera de ellos, una regla que atrapa a todo lector exactamente una vez:

⚠ ESTOS ATRIBUTOS NECESITAN UN MÉTODO DECLARADO

PHP no puede adjuntar un atributo a un método que nunca se declara, y una consulta derivada alcanzada a través de `__call()` nunca se declara — existe solo como una etiqueta `@method` en un docblock. Así que un método que lleva uno de estos cuatro atributos tiene un cuerpo real, y ese cuerpo es siempre la misma línea: `return $this->dispatchQuery(__FUNCTION__, func_get_args());` (más el `assert()` que PHPStan necesite). Declarar el método no cambia nada de cómo se resuelve; solo le da al atributo un sitio donde vivir.

El escáner registra los cuatro en el manifiesto compilado en tiempo de construcción, y `EloquentRepository` los honra en el despacho **sin reflexión en tiempo de petición** — el constructor del DTO de un `#[Projection]`, por ejemplo, se refleja exactamente una vez, cuando `firefly:cache` compila el manifiesto.

`#[Modifying]` — este método escribe

```
/** 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;
}

```

El SQL se ejecuta como sentencia y no como consulta, y el método devuelve el número de filas afectadas. Dos cosas se rechazan en tiempo de *escaneo*, así que te enteras de ellas por `firefly:cache` o por el primer arranque y nunca por la petición de una persona usuaria: `#[Modifying]` sin `#[Query]` (un `deleteBy...` derivado ya es una sentencia y no necesita atributo), y un `#[Query]` cuyo SQL es un `SELECT`, `WITH` o `VALUES`. En tiempo de ejecución la sentencia se niega a ejecutarse fuera de una transacción activa en la conexión del modelo — una `TransactionRequiredException` — salvo que renuncies a ello con `requiresTransaction: false`.

El `clearAutomatically` de Spring se acepta por compatibilidad de código fuente y no hace nada. Eloquent no tiene contexto de persistencia que limpiar, y fingir lo contrario sería la peor clase de paridad.

`#[Projection]` — selecciona menos, hidrata un 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;
}

```

En un método **derivado** la lista del `SELECT` se infiere de los parámetros del constructor del DTO (o se da explícitamente con `columns:`); en un método `#[Query]` el SQL es dueño de su propia lista de selección. Cada fila se hidrata a través del constructor: columna en `snake_case` a parámetro en `camelCase`, escalares convertidos desde lo que el driver haya devuelto, un enum respaldado desde su valor de respaldo, un `DateTimeImmutable` desde la cadena, y un parámetro opcional cuya columna está ausente toma su valor por defecto.

La conversión es **sin pérdida o se rechaza**. `'150.75'` hacia un `int`, un valor de respaldo para el que el enum no tiene caso, una cadena que no es una fecha — cada uno es una `ConfigurationException` que nombra el DTO, la columna, el valor y el parámetro, nunca un `TypeError` pelado surgido de algún lugar profundo de PHP. Una columna obligatoria que falta y un `NULL` que llega a un parámetro no nutable son la misma clase de error, y las columnas se comprueban contra la tabla antes de que la consulta se ejecute (sqlite leería si no un identificador desconocido entre comillas dobles como un literal de cadena y devolvería un sinsentido).

Las proyecciones por interfaz de Spring (`interface RecordSummary { String getEmail(); }`) no se ofrecen. PHP no tiene un proxy en tiempo de ejecución que pudiera implementar una interfaz por nombre de columna; declara la clase DTO en su lugar.

#[Lock] — toma el bloqueo de fila que la lectura necesita

```
/** @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` compila a `lockForUpdate()`, `PESSIMISTIC_READ` a `sharedLock()`, escritos por la gramática de la propia conexión. Ambos se niegan a ejecutarse fuera de una transacción activa — el comportamiento de Spring, y el correcto: un bloqueo de fila se libera cuando la transacción termina, así que fuera de una no protege nada. `$repo->findByIdForUpdate($id)` es el gemelo programático del atributo, y sqlite, que no tiene bloqueos de fila en absoluto, compila la cláusula a nada y sencillamente tiene éxito.

El escáner rechaza `#[Lock]` sobre un método `#[Query]`: el SQL en bruto lleva su propia cláusula de bloqueo, y dos de ellas se pelearían.

#[EntityGraph] — carga ansiosa, de forma declarativa

```

/** @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;
}

```

Se mapea al `with()` de Eloquent, y es el único atributo que alcanza métodos que no escribiste. Cada lectura heredada que **abre un builder** — `findById`, `findAll`, `findAllById`, `findPaged`, `findSorted`, `findSlice`, `findBySpecification`, `findByExample`, `findOneByExample`, `findByIdForUpdate` y las demás — parte de `reading(__FUNCTION__)`, que pide al manifiesto el grafo registrado contra *esta clase de repositorio y ese nombre de método*. Así que anotar una sobrescritura que no hace más que `return parent::findAll();` es la receta entera, exactamente igual que en Spring. Las cuatro lecturas que responden una pregunta *sobre* las filas en vez de devolverlas — `existsById()`, `count()`, `existsByExample()`, `countByExample()` — van directas a `query()` y deliberadamente no tocan nunca `reading()`: un grafo de entidades no significa nada para un `COUNT(*)`, y cargar relaciones con ansia para tirarlas sería coste puro. Un grafo con nombre que el repositorio nunca declaró en `$entityGraphs` es una `ConfigurationException` en el primer uso.

⚠ DOS COMBINACIONES QUE NO FUNCIONAN COMO TE GUSTARÍA

Un `#[Projection]` y un `Pageable` final **no se combinan** — la proyección devuelve el resultado entero, sin paginar. Pagina las entidades, o pon un `LIMIT` en el `#[Query]`. Y `#[Query]` vincula **posicionalmente**: los `:marcadores` con nombre se reescriben a `?` por orden de primera aparición, así que un marcador usado dos veces necesita que el argumento se pase dos veces.

Paginación: `Page`, `Pageable` y `Sort`

Tres pequeños objetos de valor inmutables transportan de principio a fin una petición de página y devuelven un resultado de página. `Pageable` es la petición — un número de página basado en 1, un tamaño, y un `Sort` opcional:

```
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 y **Order** componen una ordenación a partir de piezas pequeñas e inmutables — cada combinador devuelve un **Sort** nuevo, nunca muta aquel con el que empezaste:

```

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')` empareja un nombre de propiedad con un enum `Direction::Asc / Direction::Desc` cuyo *valor* de cadena subyacente **es** la dirección de `orderBy()` de Eloquent — `$order->direction->value` no necesita ninguna tabla de traducción en absoluto.

`Page` es lo que vuelve — el recorte de filas más todo lo que un cliente necesita para renderizar un paginador:

```

/**
 * // ...
 * @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()` construye una `Page` en el borde de Eloquent: una consulta `count()` para `total`, luego `skip($pageable->offset())->take($pageable->size)` para el recorte, con `applySort()` plegando cada orden de `Sort` en una llamada `orderBy()` en medio. `Page::map()` es a lo que recurre un manejador de consulta para convertir una página de entidades en una página de DTOs **sin perder los metadatos de paginación** — los mismos números `total` / `page` / `size` sobreviven intactos a la transformación.

`Slice` : la página que no cuenta

`total` es caro. Sobre una tabla grande, un `SELECT COUNT(*)` sobre el mismo predicado puede costar más que traer la página misma, y una lista de scroll infinito no muestra el número de todas formas — solo necesita saber si mantener el botón de «cargar más». Eso es un `Slice`: todo lo que tiene `Page` salvo `total`, más el único hecho que ese botón necesita.

```

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` no se adivina y no se cuenta. El repositorio trae **una fila más** que el tamaño de página, y si esa fila extra volvió se descarta y se recuerda como «hay una página siguiente»:

```

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;

```

Así que un recorte es una consulta barata donde una página es una consulta barata más una cara. `$repo->findSlice(Pageable::of(3, 20, Sort::by('id')))` es la página sin cuenta del propio puerto; `Slice::nextPageable()` te entrega la petición de la siguiente, y `Slice::map()` transforma los elementos exactamente igual que `Page::map()`.

Un método derivado también puede devolver un recorte — y aquí las dos mitades de la regla importan por separado. Un **argumento Pageable final** es lo que hace que un método derivado pague siquiera; el **tipo de retorno declarado** es lo que lo convierte en un `Slice` y no en una `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;
}

```

Una consulta derivada `@method` sin declarar no tiene tipo de retorno declarado que leer, así que un `Pageable` sobre una produce siempre una `Page`. Es la misma regla que obedecen los cuatro atributos, llegando desde otra dirección.

Borrado suave, auditoría y bloqueo optimista

Tres bloques constructivos más viven en `EloquentRepository`, apoyados sobre los propios mecanismos de Eloquent en vez de reinventados. Ninguno lo ejerce `Wallet` en este libro, pero cada uno está a una sola declaración `use` de distancia en cualquier modelo que lo necesite.

El **borrado suave** reutiliza tal cual el trait nativo `SoftDeletes` de Eloquent:

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

Una vez que un modelo lo usa, `delete()` / `deleteById()` a través del repositorio borra suavemente la fila, y cada lectura ordinaria excluye de forma transparente las filas eliminadas mediante el propio ámbito global de Eloquent.

`findAllIncludingDeleted()` y `restore(mixed $id): ?object` completan el ciclo de vida.

La **auditoría** es un trait `Auditable` opcional que estampa `created_by` / `updated_by` al escribir, impulsado por un puerto `AuditorAware`:

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

El **bloqueo optimista** protege escrituras concurrentes con una columna entera `version` vía `HasOptimisticLock`, que sobrescribe el `performUpdate()` interno de Eloquent para añadir `WHERE version = <versión cargada>` a cada `UPDATE` — una escritura contra una versión obsoleta no coincide con ninguna fila y lanza `OptimisticLockException` en vez de sobrescribir silenciosamente el cambio de otro. Esa excepción extiende la `OptimisticLockingFailureException` del kernel, así que un `catch` sobre cualquiera de los dos nombres funciona y la regla de la sección siguiente también se le aplica.

Cuando el driver falla: la familia `DataAccessException`

Todo lo anterior ha dado por supuesto que la base de datos dice que sí. Esta sección trata de lo que llega a tu código cuando dice que no — y la respuesta del framework es la de Spring: **un fallo del driver nunca sale de `firefly/data` como un fallo del driver.**

La respuesta propia de Laravel es `QueryException`. Es una sola clase para todo fallo posible, y dos cosas la convierten en algo pobre que entregar a una aplicación. La primera es que no puedes ramificar sobre ella: una dirección de correo duplicada y un servidor de base de datos inalcanzable llegan como el mismo tipo, así que distinguirlas significa comparar códigos de error del driver en el punto de llamada, una vez por cada driver que soportes. La segunda es peor. El mensaje de una `QueryException` es la sentencia que falló **con sus enlaces interpolados** — es decir, la dirección de correo, el token o el identificador de inquilino que se estaba escribiendo. Cualquier cosa que renderice ese mensaje a un cliente, o que lo registre a un nivel que otra persona pueda leer, ha filtrado datos de usuario.

`PersistenceExceptionTranslator` arregla las dos de golpe:

```

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]);
}

```

Lee lo que *no* está en ese método. No se copia ningún `$cause->getMessage()` a la excepción nueva. Cada uno de esos constructores lleva su propia **frase fija**, escrita una vez, sin sentencia y sin enlaces dentro. El texto propio del driver se queda exactamente donde corresponde — en `previous`, para el log y la traza de pila — y la única pieza de la respuesta del driver que es a la vez inofensiva y útil, el SQLSTATE, viaja al lado como el miembro de extensión `sqlState` del documento de problema.

El tipo es el diagnóstico, y es sobre lo que ramifica quien llama:

Se lanza	Extiende	HTTP	Cuándo
<code>DuplicateKeyException</code>	<code>DataIntegrityViolationException</code>	409	una violación de unicidad o de clave primaria
<code>DataIntegrityViolationException</code>	<code>DataAccessException</code>	409	cualquier otra restricción: not-null, clave foránea, check
<code>DeadlockLoserDataAccessException</code>	<code>CannotAcquireLockException</code>	409	esta transacción fue elegida como víctima del interbloqueo
<code>CannotAcquireLockException</code>	<code>DataAccessException</code>	409	la espera de un bloqueo agotó su tiempo
<code>QueryTimeoutException</code>	<code>DataAccessException</code>	504	la sentencia sobrepasó su tiempo límite
<code>TransientDataAccessResourceException</code>	<code>DataAccessException</code>	503	la conexión se cayó — reintentar puede funcionar
<code>DataAccessResourceFailureException</code>	<code>DataAccessException</code>	503	el origen de datos no está ahí en absoluto
<code>BadSqlGrammarException</code>	<code>DataAccessException</code>	500	la sentencia es incorrecta; reintentar no puede ayudar

Se lanza	Extiende	HTTP	Cuándo
<code>DataAccessExcepti on</code>	<code>InfrastructureExc eption</code>	500	un fallo que las tablas no reconocen

La jerarquía es el sentido de todo esto. Un manejador al que solo le importa «la escritura entró en conflicto» captura `DataIntegrityViolationException` y se lleva también las claves duplicadas; uno al que le importa «cualquier cosa que la base de datos rechazó» captura `DataAccessExcepti` y se lleva las nueve. Como todas ellas son una `FireflyException`, el renderizador de detalles de problema del Capítulo 4 ya conoce el código de estado y el código de error sin una sola línea de mapeo — `DUPLICATE_KEY`, `QUERY_TIMEOUT`, `BAD_SQL_GRAMMAR` y los demás viajan al cliente como `type / code`, y la sentencia no viaja a ninguna parte.

Dos miembros más de la familia vienen del repositorio y no de un driver:

`getId($id)` lanza `EmptyResultDataAccessException` (404) exactamente donde `findById()` habría devuelto `null`, y `findOneByExample()` lanza `IncorrectResultSizeDataAccessException` cuando la sonda coincidió con más de una fila.

Dónde ocurre la traducción, y dónde deliberadamente no

```
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);

```

Hay exactamente dos costuras por las que un error de base de datos puede salir de `firefly/data`, y las dos traducen: cada método de `EloquentRepository` (eso es lo que hace la envoltura `translating()` que viste alrededor de `save()` y `findBySpecification()`), y `TransactionTemplate::execute()` — lo que significa cada método `#[Transactional]` del capítulo siguiente, ya que el proxy delega en la plantilla. Una llamada `DB::` en bruto que hagas fuera de ambas sigue lanzando la `QueryException` de Laravel exactamente igual que siempre. No se parchea nada globalmente, y no se parchea nada en caliente.

Tres guardas mantienen eso estrecho. Un throwable que **ya** está en la familia se devuelve intacto, así que una llamada a repositorio que traduce dentro de una plantilla que vuelve a traducir es idempotente en vez de quedar doblemente envuelta. Cualquier cosa que no sea una `PDOException` pasa de largo — con una excepción para los cuatro fallos que el propio Laravel ya ha clasificado (`UniqueConstraintViolationException`, `SQLiteDatabaseDoesNotExistException`, `LostConnectionException`, `DeadlockException`), cuya respuesta es al menos tan buena como la de cualquier tabla, y que se buscan **también en la causa**, porque Laravel envuelve lo que sea que lance el callback de la consulta. Y todo el mecanismo está a una clave de apagarse: `firefly.data.exception-translation.enabled`, `true` por defecto.

Qué tabla de códigos se aplica lo decide el nombre del driver. Las tablas en sí — códigos de error de driver para sqlite, mysql, mariadb y sqlsrv, SQLSTATEs exactos, y *clases* de SQLSTATE como recurso final — viven en un único fichero, `DriverErrorTable`, con una fila de test por cada fila de tabla. sqlite necesita un paso extra, y está en el código fuente en vez de escondido: sqlite reporta cada violación de

restricción como código 19, así que la familia de unicidad/clave primaria solo es visible en el texto del mensaje, y ese es el único sitio donde el traductor lee un mensaje del driver.

💡 LA REGLA QUE MERECE LA PENA RECORDAR

Captura el **tipo**, registra el **previous**, renderiza la **frase fija**. Si alguna vez te encuentras leyendo `$e->getMessage()` para decidir qué pasó, el tipo que necesitabas ya existe.

El puente de eventos de dominio

`EloquentRepository::save()` hace algo más, además de persistir la fila, que importa enormemente para el Capítulo 6: si la entidad guardada también implementa `RecordsDomainEvents` — que es el caso de `Wallet` — y hay una transacción activa en ese momento sobre la propia conexión de esa entidad, `save()` registra la entidad en `AggregateTracker`, el registro de unidad de trabajo de `firefly/data`:

```
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;
    });
}
```

```
    });  
}
```

Ese registro es lo que permite al framework drenar y publicar los eventos de dominio lanzados por un `Wallet` después de que confirme el método `#[Transactional]` que lo rodea — y nunca si hace rollback. El Capítulo 6 construye el agregado que lanza esos eventos; este es el punto de conexión, en el lado de la persistencia, que los captura.

✓ Lo que aprendiste

Concepto	Qué hace
<code>CrudRepository</code> / <code>PagingAndSortingRepository</code>	Los puertos agnósticos de persistencia que implementa en última instancia todo repositorio
<code>EloquentRepository</code>	La base respaldada por Eloquent: fija <code>\$model</code> , hereda CRUD + paginación + ordenación completos
Consulta derivada (<code>findByOwnerId</code>)	Un nombre de método sin cuerpo analizado por <code>DerivedQueryParser</code> y conducido contra el <code>Builder</code> de Eloquent
<code>#[Query('sql')]</code>	Un escape de SQL explícito, descubierto en tiempo de escaneo, resuelto con cero reflexión por petición
<code>Specification</code> / <code>Specifications</code>	Predicados componibles (<code>allOf</code> / <code>anyOf</code> / <code>not</code> / <code>where</code>) aplicados vía <code>findBySpecification(Paged)</code>
<code>Example</code> / <code>ExampleMatcher</code>	Consulta por ejemplo, portada <i>como</i> una <code>Specification</code> , así que compone y pagina como cualquier otro predicado
<code>#[Modifying]</code> / <code>#[Projection]</code> / <code>#[Lock]</code> / <code>#[EntityGraph]</code>	Lo que un método declarado puede decir y su nombre no: escrituras, hidratación de DTO, bloqueos de fila, carga ansiosa
<code>Page</code> / <code>Pageable</code> / <code>Sort</code> / <code>Order</code>	Objetos de valor inmutables que transportan una petición de página y devuelven un resultado de página (con metadatos)
<code>Slice</code>	La misma página sin la consulta de cuenta: una fila traída de más responde a <code>hasNext</code>

Concepto	Qué hace
<code>SoftDeletes</code> / <code>Auditable</code> / <code>HasOptimisticLock</code>	Traits opcionales para borrado suave, estampado <code>created_by</code> / <code>updated_by</code> , y escrituras protegidas por versión
Familia <code>DataAccessException</code>	Un fallo del driver se convierte en una excepción tipada con una frase fija; la sentencia nunca sale del log
El puente de eventos de dominio	<code>save()</code> registra una entidad <code>RecordsDomainEvents</code> en <code>AggregateTracker</code> mientras hay una transacción activa

Ponlo en práctica

- Añade un contador derivado.** En una copia de trabajo del proyecto, declara `public function countByCurrency(string $currency): int` sin cuerpo en un repositorio que extienda `EloquentRepository`, y confirma que llamarlo se compila a `SELECT COUNT(*) ... WHERE currency = ?` sin SQL propio.
- Compón dos especificaciones.** Construye una `Specification` para "saldo de al menos N" y otra para "en la moneda C" con `Specifications::where(...)`, combínalas con `Specifications::allOf(...)`, y ejecuta el compuesto a través de `findBySpecificationPaged` contra una pequeña tabla poblada de prueba.
- Convierte una página en un recorte.** Toma la lectura paginada del ejercicio anterior y pide `findSlice(Pageable::of(1, 2))` en su lugar. Activa el registro de consultas y confirma lo que la sección afirma: el recorte ejecuta **una** consulta donde la página ejecutaba dos, y `hasNext` sigue siendo correcto.

4. **Haz que la base de datos diga que no.** Añade un índice único a la tabla de monederos, guarda dos monederos con el mismo identificador de propietario, y captura el resultado. Comprueba tres cosas: el tipo es `DuplicateKeyException`, `getMessage()` no contiene el identificador de propietario que escribiste, y `getPrevious()` es la `QueryException` de Laravel con la sentencia todavía encima.
5. **Lee una página de monederos.** Llama a la contraparte de paginación de `EloquentWalletRepository::findAll(...)` (añade un llamador de `findPaged` a través de `WalletRepository` en tu proyecto de pruebas) con `Pageable::of(1, 2, Sort::by('created_at')->descending())` contra tres monederos de prueba poblados, y confirma que `total`, `totalPages` y `hasNext` coinciden todos con lo que esperas antes y después de `Page::map()`.

Diseño Guiado por el Dominio

Al terminar este capítulo conocerás la diferencia que traza `firefly/domain` entre una `Entity` y un `ValueObject`, cómo `AggregateRoot` (y su gemelo en forma de trait, `HasDomainEvents`) convierte a un objeto en la única fuente de sus propios eventos de dominio, cómo se construye e identifica un `DomainEvent`, cómo el agregado `Wallet` hace cumplir sus propias reglas de sobregiro y de moneda sin que nada externo pueda saltárselas, y — cerrando el círculo abierto en el Capítulo 5 — exactamente cómo un evento lanzado viaja desde un método del agregado hasta un hecho publicado al otro lado de una transacción confirmada.

TÉRMINO NUEVO: INVARIANTE

Una **invariante** es una regla que siempre debe cumplirse, sin importar qué camino de código llegó hasta ahí — para `Wallet`, "el saldo nunca es negativo" es una invariante, no una convención. Una invariante vive *dentro* del objeto que protege; una regla aplicada solo por un servicio que quienes la llaman pueden saltarse no es realmente una invariante en absoluto.

Entidades y objetos de valor

`firefly/domain` es deliberadamente el paquete más pequeño de todo el framework: `Entity`, `ValueObject`, `AggregateRoot` y `DomainEvent` — cuatro bloques constructivos de PHP puro con **cero dependencia del framework y cero reflexión**. (Una prueba dedicada busca en el propio código fuente del paquete `ReflectionClass` / `ReflectionMethod` / `getAttributes` y afirma que no hay coincidencias — tu modelo de dominio nunca tiene que saber que LaraFly existe.)

Entity traza la primera de las dos distinciones fundacionales de DDD: la identidad, no el valor, decide si dos entidades son la misma cosa.

```
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;
    }
}
```

Dos entidades son `equals()` solo cuando son la misma clase concreta **y** tienen ids iguales y no nulos. Una entidad **transitoria** — aquella cuyo id todavía es `null`, aún no persistida — es igual solo a sí misma, por pura identidad de objeto; dos instancias transitorias distintas nunca son iguales aunque el resto de su estado sea idéntico. PHP no tiene genéricos en tiempo de ejecución, así que `id()` está tipado `int|string|null` — lo bastante amplio para cubrir tanto una clave entera autoincremental como un id de cadena como el propio `wlt-...` de `Wallet`.

`ValueObject` es la segunda distinción: ninguna identidad en absoluto, inmutable por convención, igual puramente por los valores que contiene.

```
interface ValueObject {}
```

Es una interfaz marcadora desnuda a propósito — la *igualdad* de un objeto de valor se suministra por separado, mediante un pequeño 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)`, no `ReflectionProperty` — deliberadamente, para que `firefly/domain` se mantenga libre de reflexión — compara cada propiedad visible en el ámbito. Esto es exactamente correcto para un objeto de valor **readonly plano**; un

objeto de valor que anida otro objeto de valor debería componer la igualdad delegando en el propio `equals()` del objeto anidado, en vez de confiar en que `==` recorra correctamente el array de propiedades exterior.

TÉRMINO NUEVO: AGREGADO

Un **agregado** es un pequeño cúmulo de objetos que debe permanecer consistente como grupo, al que se entra y se muta a través de exactamente un objeto: su **raíz de agregado**. La siguiente sección construye uno.

Money : el objeto de valor de manual

`Money` es el `ValueObject` de `firefly/domain` aplicado al manejo real de divisas de Lumen — código real y distribuido de `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}"
            );
        }
    }
}

```

Tres decisiones de diseño aquí recompensan una lectura atenta. `$minorUnits` se almacena como un **entero** — céntimos, no un `float` — precisamente para que la aritmética financiera nunca acumule error de redondeo; `Money(1050, Currency::EUR)` son exactamente 10,50 €, y `add / subtract` operan directamente sobre ese entero. `add()` y `subtract()` devuelven ambos una instancia **nueva** de `Money` — `final readonly class` hace de esto la única opción, no una disciplina que debas recordar — así que un depósito nunca muta el importe del que partió; nada más en el código puede estar silenciosamente reteniendo una referencia obsoleta a un `Money` que acaba de cambiar por debajo. Y `assertSameCurrency()` es un ayudante privado por el que ambos métodos públicos pasan primero, así que combinar importes en EUR y USD es una `ConflictException` lanzada en el punto exacto de llamada del error, nunca una suma silenciosamente incorrecta descubierta durante la conciliación.

`final readonly class Money implements ValueObject { use ValueObjectEquality; }` es toda la historia de la igualdad — dos instancias de `Money(1050, Currency::EUR)` construidas independientemente son `equals()` entre sí, porque `ValueObjectEquality` compara su estado público, no su identidad de objeto.

⬆ PARIDAD CON LARAVEL

`ValueObject` + `ValueObjectEquality` se corresponden con la idea `@ValueObject / @Embeddable` del mundo JPA, y con la propia característica de clase `readonly` de PHP haciendo la mitad de inmutabilidad del trabajo. En una aplicación Laravel/Eloquent corriente, lo más parecido a `Money` es un objeto de casteo de Eloquent — salvo que aquí `Money` no tiene ni idea de que Eloquent existe, y se ejercita en una prueba unitaria con dos argumentos de constructor y nada más.

La raíz de agregado: la única fuente de sus propios eventos

`Money` resuelve la representación. Algo todavía tiene que *poseer* la decisión de si un depósito o un retiro está permitido siquiera — ese es el trabajo de la raíz de agregado.

`AggregateRoot` extiende `Entity` y añade exactamente una cosa: un búfer privado de eventos de dominio pendientes, y el único método protegido que añade a él.

```
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()` es `protected` — deliberadamente. Solo los métodos *propios* del agregado pueden llamarlo, que es lo que convierte a un agregado en la **frontera de consistencia** que prometió la introducción del capítulo: ningún llamador externo puede meterse a hacer que el agregado emita un evento en su nombre, del mismo modo que ningún llamador externo puede meterse a mutar su estado sin pasar por un método que aplique las reglas. `pendingEvents()` es una instantánea que no drena — segura de llamar desde una aserción de prueba o una línea de log sin efectos secundarios. `pullEvents()` drena el búfer y se lo entrega a quien lo pidió, que es lo que llama la maquinaria post-confirmación del framework. `clearEvents()` descarta el búfer sin publicar nada — se usa en un rollback.

`RecordsDomainEvents` es la interfaz que nombra exactamente este contrato del lado del drenaje, deliberadamente **sin** `raiseEvent()` — porque solo el propio agregado puede lanzar sus propios eventos, así que ese método permanece `protected` en cualquier clase que implemente la interfaz:

```
interface RecordsDomainEvents
{
    // ...
    public function pendingEvents(): array;
    // ...
    public function pullEvents(): array;
    // ...
    public function clearEvents(): void;
}
```


Por qué `Wallet` no extiende `AggregateRoot`

Aquí es donde la historia de agregados de LaraFly tiene que resolver un problema que PyFly y Java nunca enfrentan: PHP tiene herencia simple, y `Wallet` necesita ser a la vez un **modelo Eloquent persistido** y una raíz de agregado. No puede hacer `extends AggregateRoot` y `extends Model` al mismo tiempo. `HasDomainEvents` es el trait que cierra esta brecha — exactamente el mismo búfer y exactamente los mismos cuatro métodos que `AggregateRoot`, pero como un trait que cualquier clase puede `use`, sin importar qué ya extienda:

```
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 = [];
    }
}

```

Aquí está el agregado `Wallet` completo y real — `samples/lumen/src/Domain/Wallet.php` — combinando exactamente este trait con `extends Model` e `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->mi
norUnits
    ));
}
// ...
private function assertCurrency(Money $amount): void
{
    if ($amount->currency !== $this->currency()) {
        throw new ConflictException('currency mismatch');
    }
}
}
}

```

Esta es una forma genuinamente distinta de la que tendría un lenguaje con herencia múltiple de verdad — y es deliberada, no un rodeo. `Wallet` es la fila de Eloquent (`protected $table = 'wallets'`, `$fillable`, una clave de cadena no autoincremental) y la raíz de agregado, en un solo objeto. No hay una clase ORM `WalletEntity` separada ni un mapeador cruzando una frontera entre ellas para este agregado: `Wallet::open()` construye a la vez los atributos de la fila y lanza `WalletOpened` en el mismo método de fábrica estático, y `deposit()` / `withdraw()` mutan a la vez el atributo de Eloquent y lanzan su evento en la misma llamada. El búfer de `HasDomainEvents`, `$pendingEvents`, es una propiedad privada genuinamente

declarada, no un atributo de base de datos — la magia `__get` / `__set` de Eloquent nunca la intercepta, porque una propiedad declarada siempre eclipsa a los accesores mágicos.

Tres invariantes viven dentro de estos tres métodos, y en ningún otro sitio. `open()` rechaza un `owner_id` en blanco. `deposit()` y `withdraw()` rechazan ambos un importe no positivo y un desajuste de moneda (a través del ayudante compartido `assertCurrency()`). `withdraw()` además rechaza dejar que el saldo se vuelva negativo — la regla de sobregiro. Cada una de estas comprobaciones se ejecuta *antes* de que se llame a `setAttribute()`, así que una operación rechazada deja los atributos persistidos del monedero completamente intactos, y el evento correspondiente nunca se lanza para una operación que nunca ocurrió.

⚠ MANTÉN LAS INVARIANTES EN EL MODELO, NO EN EL SERVICIO

Si la comprobación de sobregiro viviera en un método de servicio en vez de dentro de `withdraw()`, cualquier cosa que llamara a `$wallet->save()` directamente — un trabajo en segundo plano, un script de administración, un futuro desarrollador con prisa — podría saltársela silenciosamente. Como la comprobación está dentro del propio método del agregado, no existe ningún camino de código hacia un `Wallet` sobregirado en absoluto: la regla y la única puerta hacia el estado que protege son el mismo trozo de código.

Eventos de dominio: `DomainEvent`, y los cuatro que distribuye Lumen

`DomainEvent` es la base plana e inmutable que extiende cada evento concreto.

Rellena automáticamente dos campos que nunca tienes que fijar tú mismo:

```

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` es un uuid-v4 recién generado construido desde `random_bytes()` — sin dependencia de `ramsey/uuid`, sin reflexión — y `occurredAt` por defecto es "ahora" en el momento de la construcción; ambos aceptan también un valor explícito, para

reconstitución o para una prueba que necesite una marca de tiempo fija. `eventType()` es el propio nombre corto de la clase concreta — `strrpos / substr` sobre `static::class`, de nuevo sin reflexión — y es esta cadena, no la clase PHP, contra la que compara un listener aguas abajo (verás exactamente esa regla de coincidencia en la sección de cierre de este capítulo).

Lumen distribuye cuatro eventos así, uno por cada transición de estado que puede producir el monedero o una transferencia. Aquí están los tres que lanza el propio `Wallet` — código real y distribuido de `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();
}
}

```

Un cuarto evento, `TransferCompleted`, no lo lanza el propio `Wallet` sino la capa de aplicación una vez que ambas piernas de una transferencia tienen éxito — `TransferHandler`, más adelante en este capítulo, muestra exactamente dónde. Cada uno lleva `#[PublishDomainEvent('wallet.events')]` — un atributo de `firefly/cqrs`, no de `firefly/domain`, que nombra el **destino** al que un evento se republica una vez que cruza al mundo de los eventos de integración; `firefly/domain` en sí no tiene ninguna opinión sobre destinos en absoluto, solo sobre lanzar y drenar.

Fíjate en lo que cada evento lleva y no lleva. `FundsDeposited` y `FundsWithdrawn` llevan ambos el saldo **posterior a la operación** (`$balanceMinor`), no solo el importe que se movió — un suscriptor que actualiza un modelo de lectura nunca tiene que recargar el monedero para conocer su nuevo saldo; todo lo que necesita ya está en el hecho que recibió. Ese diseño rinde frutos directamente en `LedgerProjector`, más adelante en este capítulo.

📦 PARIDAD CON LARAVEL

`AggregateRoot / HasDomainEvents + DomainEvent` se corresponden con el `AbstractAggregateRoot` de Spring Data y su mecanismo `registerEvent() / @DomainEvents / @AfterDomainEventPublication . raiseEvent()` es `registerEvent()`; `pullEvents()` es el drenaje que `@AfterDomainEventPublication` realiza automáticamente. En Laravel/Eloquent puro, el idioma incorporado más parecido es un modelo disparando un evento de Laravel directamente desde dentro de un mutador — salvo que eso se dispara **de inmediato**, gane o pierda, mientras que `raiseEvent()` solo almacena en búfer, y nada se publica hasta que una transacción que lo rodea realmente confirma.

El modelo de eventos post-confirmación

Almacenar un evento en búfer no es lo mismo que publicarlo — y la brecha entre ambas cosas es exactamente lo que te protege de que un listener reaccione alguna vez a un cambio que se deshizo. El camino completo desde un evento lanzado hasta un hecho publicado atraviesa tres paquetes trabajando juntos:

1. Un método mutador del agregado — `Wallet::deposit()`, por ejemplo — llama a `raiseEvent()`, añadiendo al búfer privado. Todavía no se publica nada.
2. Cuando la entidad se persiste a través de un repositorio de Firefly, `EloquentRepository::save()` (la sección de cierre del Capítulo 5) la registra en el `AggregateTracker` de `firefly/data` — pero **solo** mientras hay una transacción activamente abierta en la propia conexión de esa entidad.
3. Cuando el método `#[Transactional]` que lo rodea confirma, el framework drena el `pullEvents()` de cada recorder rastreado y publica cada uno, programado a través del propio `DB::afterCommit()` de Laravel sobre la misma conexión en la que corrió la transacción.
4. Laravel solo dispara los callbacks de `afterCommit()` después de la confirmación real **más externa**, y los **descarta** por completo en un rollback — así que un listener solo observa jamás un evento de una unidad de trabajo que genuinamente tuvo éxito.

`DepositHandler` — el manejador de comando real y distribuido detrás de `WalletController::deposit()` del Capítulo 4 — es todo este ciclo en cinco líneas:

```
<?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]` (tema completo de un capítulo posterior) es lo que hace posible el paso 2 de arriba: sin él, `save()` volcaría (*flush*) la escritura pero nunca vería `transactionLevel() > 0`, así que el agregado nunca se rastrearía y `FundsDeposited` nunca se publicaría, sin importar con cuánta corrección lo lanzara `Wallet::deposit()`. `$wallet->deposit(...)` ejecuta las comprobaciones de invariantes y encola el evento; `$this->wallets->save($wallet)` persiste la fila y registra el agregado para el despacho post-confirmación, en la misma llamada; el método retorna, la transacción confirma, y solo entonces `FundsDeposited` llega a algún listener.

`TransferHandler` muestra el mismo ciclo con dos agregados y una garantía genuina de todo-o-nada:

```
<?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()` es donde se lanza el hecho `TransferCompleted` prometido antes — deliberadamente la *última* línea, después de que ambas piernas ya han tenido éxito, sobre el monedero de origen (que `save()` ya registró con el tracker, de modo que un evento lanzado tras ese guardado igual se drena en el commit). Es un pequeño método de `Wallet` que solo llama a `raiseEvent(new TransferCompleted(...))`; como se ejecuta únicamente en el camino de éxito, una transferencia revertida nunca lo alcanza y nunca lo publica.

Si el `deposit()` del destino lanza — un desajuste de moneda — el `rollbackFor` por defecto de `#[Transactional]` (cualquier `Throwable`) revierte todo el método, deshaciendo también el débito del origen. Ninguno de `FundsWithdrawn`, `FundsDeposited` o `TransferCompleted` se había publicado todavía en ese punto — cada uno solo estaba almacenado en búfer en su agregado — así que un listener nunca ve la mitad de una transferencia que en realidad nunca ocurrió. El dinero no puede ni desvanecerse ni duplicarse: o ambas piernas confirman y sus eventos se publican, o ninguna de las dos cosas.

❗ REHIDRATACIÓN Y LA FÁBRICA

Cada llamada a `findById()` de arriba reconstruye un `Wallet` a partir de una fila almacenada mediante la propia hidratación de Eloquent — nunca a través de `Wallet::open()`. Eso es correcto: `open()` es para monederos **nuevos**, y volver a llamarlo sobre una fila ya persistida re-lanzaría `WalletOpened` para un monedero que lleva meses existiendo. Un `Wallet` cargado desde el almacenamiento es exactamente tan válido como uno recién abierto; simplemente no lleva ningún evento nuevo, porque no le ha ocurrido nada nuevo todavía.

Del evento de dominio al modelo de lectura:

LedgerProjector

El último tramo del viaje — un `FundsDeposited` publicado que realmente llega a algo útil — se cierra con un listener real y distribuido. `LedgerProjector` convierte cada evento de monedero confirmado en una fila de solo-anexión en `ledger_entries`, la tabla que `GetLedgerHandler` (el `WalletController::ledger()` del Capítulo 4) lee de vuelta:

```
<?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(),
        ]);
    }
}

```

Lee el docblock con atención — nombra cada salto que el evento realmente da: el comando `#[Transactional]` confirma; el despachador de `firefly/data` drena los eventos almacenados en búfer del agregado; el puente dominio-a-integración de `firefly/cqrs` republica cada uno sobre el bus de eventos de `firefly/eda` (el bus

en memoria por defecto); y `SubscriberRegistry` entrega el `EventEnvelope` resultante a cada `#[EventListener]` cuyo patrón coincida con el **nombre de tipo** del evento — `FundsDeposited`, no la cadena de destino `#[PublishDomainEvent('wallet.events')]`. Que `$envelope->payload['balanceMinor']` exista siquiera, sin necesidad de recargar el monedero, es el rédito directo de que `FundsDeposited` lleve el saldo posterior a la operación desde el momento en que `Wallet::deposit()` lo lanzó.

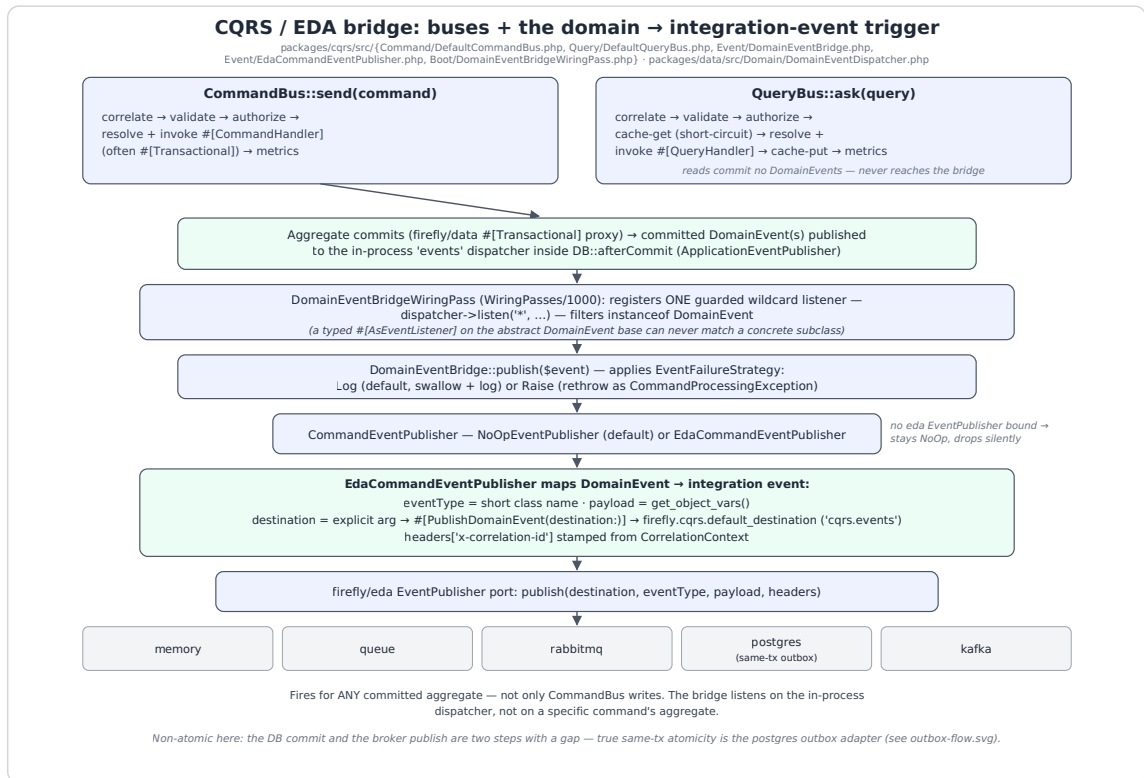


Figura 6.1 — El puente dominio-a-integración: un comando confirma, los eventos de dominio almacenados en búfer se drenan y se republiкан sobre el bus de eventos, y un listener como `LedgerProjector` reacciona al otro lado.

Un relato completo del bus de comandos de `firefly/cqrs` y del bus de eventos de `firefly/eda` es terreno de un capítulo posterior. Lo que importa aquí es la forma: `Wallet` nunca importa `firefly/eda`, nunca importa `LedgerProjector`, y no tiene ni idea de que existe un libro mayor. Solo lanza un hecho. Todo lo que ocurre aguas abajo de ese hecho es asunto de otro — que es todo el sentido de un evento de dominio.

✓ Lo que aprendiste

Concepto	Qué hace
<code>Entity</code>	Igualdad basada en identidad: misma clase, ids iguales y no nulos; una entidad transitoria es igual solo a sí misma
<code>ValueObject</code> / <code>ValueObjectEquality</code>	Un marcador para estado sin identidad e inmutable, más igualdad estructural vía <code>get_object_vars()</code>
<code>AggregateRoot</code>	La frontera de consistencia libre de persistencia: un búfer privado de eventos más un <code>raiseEvent()</code> protegido
<code>HasDomainEvents</code>	El mismo búfer como trait, para una clase (como un <code>Model</code> de Eloquent) que ya extiende otra cosa
<code>RecordsDomainEvents</code>	El contrato del lado del drenaje (<code>pendingEvents</code> / <code>pullEvents</code> / <code>clearEvents</code>) — deliberadamente sin <code>raiseEvent()</code>
<code>DomainEvent</code>	Una base plana e inmutable que rellena automáticamente <code>eventId</code> (uuid-v4) y <code>occurredAt</code> ; <code>eventType()</code> es el nombre corto de la clase
<code>Wallet::open</code> / <code>deposit</code> / <code>withdraw</code>	Las tres invariantes reales — propietario obligatorio, importe positivo, sin sobregiro, coincidencia de moneda — aplicadas antes de cualquier mutación

Concepto	Qué hace
El modelo post-confirmación	<code>save()</code> rastrea una entidad <code>RecordsDomainEvents</code> mientras hay una transacción abierta; los eventos se drenan y publican solo después de que esa transacción confirma
<code>LedgerProjector</code>	Un <code>#[EventListener]</code> real reaccionando a los hechos publicados, emparejado por nombre de tipo de evento

Ponlo en práctica

- Añade un comportamiento `freeze()`**. En una copia de trabajo del proyecto (no en el paquete distribuido `samples/lumen`), añade un evento `WalletFrozen` (`walletId: string`, `reason: string`), un método `freeze(string $reason): void` en `Wallet` que lo lance, y protege `deposit()` / `withdraw()` con una comprobación que lance `ConflictException` cuando el monedero esté congelado. Escribe una pequeña prueba que demuestre que el saldo de un monedero congelado nunca cambia ante un intento de depósito, y que nunca se encola ningún `FundsDeposited` para el intento rechazado.
- Demuestra que la invariante se cumple**. Abre un monedero, deposita una pequeña cantidad, y luego intenta retirar más que el saldo. Afirmar la `ConflictException` lanzada, luego afirmar que el `balanceMoney()` del monedero permanece completamente sin cambios y que `pendingEvents()` no muestra ningún `FundsWithdrawn` — la regla se disparó antes de cualquier mutación, exactamente como describe este capítulo.

3. **Traza la frontera de la transacción.** Quita temporalmente `#[Transactional]` de una copia de `DepositHandler`, deposita en un monedero, y confirma (mediante una lectura fresca en una nueva petición) que la fila nunca llegó a confirmarse realmente — demostrando que el *flush* por sí solo de `EloquentRepository::save()` no es durabilidad, y que la publicación de eventos de dominio depende de esa misma confirmación.

PARTE III

Coordinar y Asegurar la Aplicación

CQRS: Comandos y Consultas

Al terminar este capítulo conocerás la forma de los dos buses de `firefly/cqrs` (`CommandBus::send` / `QueryBus::ask`), cómo `#[CommandHandler]` / `#[QueryHandler]` registran un manejador como bean de DI y como destino de despacho a la vez — con el tipo de mensaje manejado *inferido* del propio parámetro `handle()` del manejador —, exactamente qué aspecto tiene la tubería acotada que ejecuta cada bus, cómo la excepción propia de un manejador llega hasta quien lo llama envuelta en una `CommandProcessingException` / `QueryProcessingException`, y cómo `HandlerScanner` compila el descubrimiento de manejadores en un manifiesto que `firefly:cache` escribe justo al lado de los manifiestos de componentes y rutas que ya conociste en los Capítulos 2 y 4.

TÉRMINO NUEVO: CQRS

Command Query Responsibility Segregation (segregación de responsabilidad entre comandos y consultas) trata las escrituras y las lecturas como dos tipos distintos de mensaje con dos caminos de despacho distintos — un **comando** es una intención de cambiar el modelo de escritura y devuelve como mucho un id o un recuento; una **consulta** es una lectura pura que nunca muta nada. `firefly/cqrs` le da a cada uno su propio bus, su propio atributo de manejador y su propia tubería, así que los dos nunca se mezclan por accidente en un mismo punto de llamada.

Dos buses, una misma forma

Ambos buses son puertos de un solo método:

```
interface CommandBus
{
    public function send(object $command): mixed;
}
```

```
interface QueryBus
{
    public function ask(object $query): mixed;
}
```

`send` es el verbo del lado de escritura; `ask` es el verbo del lado de lectura — elegido deliberadamente en vez del más obvio `query()`, que colisionaría con el propio `Model::query()` de Eloquent en cada punto de llamada dentro de un repositorio.

Ninguna de las dos interfaces exige que su mensaje implemente nada:

`Firefly\CQRS\Command\Command` y `Firefly\CQRS\Query\Query` existen puramente como marcadores documentales *opcionales* —

```
interface Command {}
```

— así que un comando es libre de ser exactamente lo que ya es

`Lumen\Application\Command\OpenWallet`: un DTO `final readonly` sin clase base ni interfaz alguna:

```
<?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,
    ) {}
}

```

Despachar uno es una única llamada, ya venga de un controlador, de una prueba o de otro servicio:

```

/** @var CommandBus $commands */
$walletId = $commands->send(new OpenWallet('owner-A', Currency::EUR));

```

`$commands->send()` devuelve lo que sea que devuelva el manejador — aquí, el id de monedero recién acuñado — porque `send()` / `ask()` están tipados `mixed`. El bus en sí no tiene ni idea de qué significa un `OpenWallet`; eso es enteramente trabajo del manejador, que es exactamente lo que construye la siguiente sección.

Manejadores: un atributo, dos trabajos

`#[CommandHandler]` y `#[QueryHandler]` **especializan** ambos `#[Component]` del Capítulo 2 — lo *extienden*, en vez de simplemente parecerse a él:

```
#[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();
    }
}
```

Gracias a ese `extends Component`, un solo atributo hace **dos** trabajos independientes a la vez: `ComponentScanner` (Capítulo 2) descubre la clase mediante la coincidencia `ReflectionAttribute::IS_INSTANCEOF`, exactamente como lo haría con un `#[Service]`, así que el manejador se convierte en un bean singleton inyectado por constructor **sin necesitar un `#[Service]` aparte** — y el propio `HandlerScanner` de `firefly/cqrs` lee ese mismo atributo para construir la tabla de despacho de comandos/consultas. Aquí está el manejador completo y real detrás de `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 S5'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;
    }
}

```

```
    }
}
```

Fíjate en que aquí `#[CommandHandler]` no lleva **ningún argumento** —

`OpenWallet::class` no se escribe en ningún sitio de este archivo. `HandlerScanner` — el único punto de reflexión de todo el paquete — resuelve el tipo de mensaje manejado leyendo el propio parámetro de `handle()` :

```
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();
    }
    // ...
}

```

Es la misma regla que el Capítulo 6 ya te mostró para el escáner gemelo de `#[EventListener]`. Un `#[CommandHandler(OpenWallet::class)]` explícito siempre gana si está presente — útil cuando `handle()` recibe un tipo base compartido o un parámetro `object` que el escáner no puede estrechar — pero para el caso común de un manejador, un mensaje concreto, un parámetro tipado, nunca escribes el nombre de la clase dos veces. Equivócate en la forma — sin método `handle()`, más de un parámetro, un tipo de parámetro escalar/unión que el escáner no puede resolver a una clase — y `HandlerScanner` lanza `CqrsConfigurationException` en **tiempo de caché**, no en alguna petición posterior.

El lado de consultas es la imagen especular, y lee una consulta sin efecto secundario alguno — `GetBalanceHandler`, código real y distribuido:

```

<?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` es `final` y no lleva ningún `#[Transactional]` — una consulta nunca muta estado, así que no hay ninguna unidad de trabajo que delimitar y nada que un proxy pueda interceptar. `GetLedgerHandler` (detrás del `WalletController::ledger()` del Capítulo 4) tiene la misma forma, leyendo directamente filas de `LedgerEntry`:


```

<?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()
        );
    }
}

```

⬆ PARIDAD CON LARAVEL

Laravel puro no tiene ninguna separación entre lectura y escritura en la capa de despacho — o bien haces `dispatch()` de un job en cola o llamas directamente a un método de servicio, y nada impide que un objeto "consulta" mute algo silenciosamente.

`#[CommandHandler]` / `#[QueryHandler]` convierten esa separación en un concepto de primera clase, descubierto por atributo: un manejador de consulta que intentara llamar a `$this->wallets->save(...)` seguiría compilando, pero sería una señal de alarma obvia en cualquier revisión de código, porque nada en la tubería de `QueryBus::ask()` (sin transacción, sin drenaje de eventos de dominio) lo respalda.

La tubería acotada

Ninguno de los dos buses es una simple llamada a función sobre el manejador.

`DefaultCommandBus::send()` — la implementación distribuida — ejecuta una secuencia pequeña y fija de colaboradores alrededor del despacho:

```
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);
}
}
}
}

```

Léelo como **correlacionar** → **validar** → **autorizar** → **resolver e invocar el manejador** → **métricas**, envuelto en un `try / catch / finally` que garantiza que el id de correlación se restaura pase lo que pase. `DefaultQueryBus::ask()` tiene la misma forma con un paso extra — un cortocircuito de caché — insertado entre autorizar e invocar:

```

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);
    }
}
}

```

Todos los colaboradores excepto el propio `HandlerRegistry` son, por defecto, un **no-op**, así que la tubería es una cadena de etapas que puedes ir activando de una en una, en vez de un framework de todo-o-nada:

Etapa	Por defecto	Qué la enciende
Validación	Se omite salvo que el mensaje implemente <code>Validatable</code>	El <code>Validator</code> distribuido de <code>firefly/validation</code>
Autorización	<code>AllowAllAuthorizer</code> — permite todo	<code>SecurityCommandAuthorizer</code> / <code>SecurityQueryAuthorizer</code> de <code>firefly/security</code> (Capítulo 10)
Correlación	Siempre activa — un id uuid4 por despacho, restaurado después	Nada que activar; se estampa en <code>x-correlation-id</code> cuando un evento de dominio llega al bus de eda (Capítulo 8)
Métricas	<code>NoOpCqrsMetrics</code>	<code>firefly/observability</code> (un capítulo posterior)
Caché de consultas	<code>NoOpQueryCache</code> — siempre falla	Un futuro adaptador de <code>firefly/cache</code>

Ninguno de los dos buses abre una transacción, y ninguno de los dos drena eventos de dominio. `#[Transactional]` sobre `OpenWalletHandler::handle()` de arriba hace ese trabajo enteramente por su cuenta, a través del proxy generado que el Capítulo 9 explica en detalle — el bus es completamente ajeno a él, y se comportaría idénticamente si `handle()` fuera un método plano, sin proxy, que llamara a `DB::beginTransaction()` a mano.

Cuando un manejador lanza: envoltura que preserva la categoría

El `handle()` de `WithdrawHandler` puede fallar de dos formas muy distintas — puede que el monedero no exista, o que la propia invariante de no-sobregiro del dominio rechace el importe. En cualquier caso, el bloque `catch` de `send()` envuelve lo que se haya lanzado:

```
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,
        );
    }
}
```

Tres cosas merecen destacarse. Primero, la causa siempre está disponible en `getPrevious()` — el quinto argumento del constructor hacia el padre — así que nada del fallo original se pierde, solo se envuelve. Segundo, esto **preserva la categoría**: cuando la causa es una `FireflyException` (todas las excepciones del kernel lo son), su propio `httpStatus()` / `category()` / `severity()` se copian sobre el envoltorio en vez de recaer en un genérico `500 / Internal`. Una `ResourceNotFoundException` (404, negocio) lanzada por un id de monedero desconocido sigue renderizándose como una respuesta de detalles de problema 404 incluso después de ser envuelta — el

bus añade contexto, nunca enmascara el fallo. Tercero, `send()` comprueba `$e instanceof CommandProcessingException` antes de envolver, así que un despacho de comando anidado (un manejador que a su vez llama a `$commands->send(...)`) nunca queda envuelto dos veces en dos capas de la misma excepción.

`QueryProcessingException` es el gemelo del lado de lectura, idéntico en forma.

`HandlerRegistry` lanza sus propias excepciones, más estrechas, para un error de **cableado** en vez de un fallo de negocio — un mensaje sin mapear enviado a través del bus:

```
final class HandlerRegistry
{
    // ...
    public function findCommandHandler(string $commandClass): callable
    {
        return $this->commandHandlers[$commandClass] ?? throw new
        CommandHandlerNotFoundException($commandClass);
    }
    // ...
}
```

`CommandHandlerNotFoundException` / `QueryHandlerNotFoundException` son ambas `ErrorCategory::Framework`, `500` — una señal genuinamente distinta de un `409` de `ConflictException`: la primera significa "nunca se registró ningún manejador para este mensaje", la segunda significa "algo sí se registró, y correctamente rechazó la petición". El registro también rechaza de plano un **segundo** registro para la misma clase de mensaje:

```
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;
    }
    // ...
}

```

Un comando, un manejador — aplicado en tiempo de cableado, no dejado a merced de cuál registro se haya ejecutado último.

🔍 DESENVOLVER LA CAUSA EN UNA PRUEBA

Una prueba que quiere afirmar *por qué* se rechazó un comando llega directamente más allá del envoltorio con `getPrevious()`. `samples/lumen/tests/Application/TransferSecurityTest.php` hace exactamente esto para un `#[PreAuthorize]` denegado (el Capítulo 10 explica la protección en sí):

```

$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);

```

El tipo exterior te dice *qué operación del bus* falló; el tipo interior — alcanzado vía `getPrevious()` — te dice *por qué*.

Compilado, no reflexionado: `HandlerScanner` → `HandlerManifest` → `firefly:cache`

`HandlerScanner` es el **único** punto de reflexión en `packages/cqrs/src` (la misma invariante forzada por grep que el Capítulo 6 describió para `firefly/domain`). Recorre una vez las raíces PSR-4 de la aplicación, resuelve cada `#[CommandHandler]` / `#[QueryHandler]` (ejecutando exactamente la inferencia de `messageType()` que acabas de leer) en un `HandlerDescriptor`, y entrega la lista completa a `HandlerManifestCompiler`, que la `var_export()` a en un archivo PHP de array plano — la misma técnica sin reflexión que el Capítulo 2 introdujo para el manifiesto de componentes y el Capítulo 3 para `#[ConfigProperties]`.

Nunca ejecutas `HandlerScanner` tú mismo. Es uno de los **doce** pares escáner/compilador que `php artisan firefly:cache` orquesta en un solo paso, junto al de componentes/contexto, rutas, propiedades de configuración y (del Capítulo 9) el transaccional:

```
$ php artisan firefly:cache
firefly:cache - wrote 14 manifest(s) + 1 proxy(ies) to /path/to/my-app/bootstrap/cache/
firefly
```

El archivo compilado aterriza en `bootstrap/cache/firefly/handlers.php`; `FireflyCacheServiceProvider` lo vincula directamente sobre la entrada del contenedor `HandlerManifest` — sobrescribiendo incondicionalmente el `HandlerManifest` **vacío** por defecto que `CqrsWiringProvider` vincula cuando todavía no existe ninguna caché (el mismo patrón de retirada al estilo `#[ConditionalOnMissingBean]` que el Capítulo 2 te mostró para los beans, aplicado aquí a todo un manifiesto compilado). `CqrsHandlerWiringPass` recorre entonces ese manifiesto en el arranque y llena el `HandlerRegistry` que conociste arriba con un

invocador por descriptor — cada invocador resuelve su bean manejador **de nuevo desde el contenedor en cada despacho**, nunca en caché en tiempo de cableado, así que un manejador `#[Transactional]` como `OpenWalletHandler` siempre se observa a través de su proxy generado (Capítulo 9), nunca a través de la clase desnuda, sin proxy.

Equivócate en esto en una aplicación — añade un manejador, olvida volver a ejecutar `firefly:cache` — y el mensaje que debería haber manejado no resuelve nada: `HandlerRegistry::findCommandHandler()` nunca ve el nuevo descriptor, y despachar ese comando lanza `CommandHandlerNotFoundException` hasta que se regenere la caché. Falla ruidosamente en tiempo de caché (un manejador mal formado) o falla ruidosamente en tiempo de despacho (una caché obsoleta) — nunca un no-op silencioso.

✓ Lo que aprendiste

Concepto	Qué hace
<code>CommandBus::send(object): mixed</code> / <code>QueryBus::ask(object): mixed</code>	Los dos puertos de despacho; un comando puede mutar y devuelve como mucho un id, una consulta nunca muta
<code>#[CommandHandler]</code> / <code>#[QueryHandler]</code>	Especializan <code>#[Component]</code> — un solo atributo registra el manejador como bean de DI y como destino de despacho
Inferencia del tipo de mensaje	<code>HandlerScanner</code> lee el único parámetro de <code>handle()</code> a menos que se dé una cadena de clase explícita
La tubería acotada	correlacionar → validar → autorizar → (obtener de caché, solo consultas) → resolver e invocar → (guardar en caché, solo consultas) → métricas
<code>CommandProcessingException</code> / <code>QueryProcessingException</code>	Envoltorios que preservan la categoría; el fallo original siempre es alcanzable vía <code>getPrevious()</code>
<code>HandlerRegistry</code>	Una clase de comando/consulta se mapea a exactamente un manejador — un registro duplicado falla ruidosamente en tiempo de cableado
<code>HandlerScanner</code> → <code>HandlerManifest</code>	El único punto de reflexión; compilado por <code>firefly:cache</code> (uno de doce pares escáner/compilador) en un manifiesto sin reflexión

Ponlo en práctica

1. **Provoca `CommandHandlerNotFoundException`**. En una copia de trabajo del proyecto (no en el paquete distribuido `samples/lumen`), define una nueva clase de comando sin ningún `#[CommandHandler]` en toda la aplicación, despáchala a través de `CommandBus::send()`, y confirma que obtienes una `CommandHandlerNotFoundException` sin envolver en nada (la lanza directamente el registro, antes de que el propio bloque `catch` del bus pudiera envolver el fallo de un manejador).
2. **Fuerza un `handle()` ambiguo**. Dale a una clase `#[QueryHandler]` un método `handle()` con dos parámetros (o un parámetro tipado `object`) y ejecuta `php artisan firefly:cache`. Confirma que falla ruidosamente con `CqrsConfigurationException` nombrando la clase y el método exactos, en vez de omitir el manejador silenciosamente.
3. **Lee el manifiesto compilado**. Tras ejecutar `firefly:cache` en un proyecto con `OpenWalletHandler` y `GetBalanceHandler` presentes, abre `bootstrap/cache/firefly/handlers.php` y encuentra ambas entradas. Empareja el campo `kind` de cada array (`'command'` o `'query'`) con el enum `HandlerKind` que describió este capítulo.

Arquitectura Guiada por Eventos y el Outbox Transaccional

Al terminar este capítulo conocerás las dos superficies de eventos, deliberadamente distintas, que distribuye LaraFly y por qué confundirlas es el error más común de todos; el puerto `EventPublisher` y el `EventEnvelope` que transporta; cómo `#[EventListener]` compara con el **nombre de tipo** de un evento — no con su destino — un tropiezo real que sufrió la construcción de `samples/lumen` al conectar `LedgerProjector` en el Capítulo 6; los adaptadores en memoria y de cola; el modelo de reintento/carta-muerta en el que se envuelve todo listener; y — cerrando la brecha de durabilidad que deja abierta el puente dominio → integración — el genuino **outbox de la misma transacción** que `firefly/eda-postgres` escribe de forma atómica junto con el agregado que lanzó el evento.

📌 TÉRMINO NUEVO: EVENTO DE INTEGRACIÓN

El Capítulo 6 construyó **eventos de dominio** — `WalletOpened`, `FundsDeposited` — hechos que un agregado lanza para consumidores dentro del *mismo* proceso y el *mismo* desplegable. Un **evento de integración** es el mismo hecho reemitido a través de una frontera de proceso o de servicio: un envoltorio con forma de JSON con un nombre de tipo, una carga útil y encabezados de enrutamiento, transportado por un broker (o, en proceso, por ese mismo bus). Este capítulo trata de la maquinaria al otro lado de esa frontera, y del puente que lleva un evento de dominio hasta ella.

Dos superficies de eventos — no una

LaraFly distribuye dos mecanismos de eventos sin relación entre sí que resultan tener un nombre de atributo de aspecto parecido:

Superficie	Atributo	Se suscribe a	Entrega	Paquete
Eventos de aplicación en proceso	<code>#[AsEventListener]</code>	una clase de evento PHP	síncrona, mismo proceso, sin broker	<code>firefly/context</code>
Bus broker de EDA	<code>#[EventListener]</code>	un patrón de tipo de evento (glob al estilo fnmatch, p. ej. <code>'user.*'</code>)	en memoria (síncrona) o en cola (asíncrona)	<code>firefly/eda</code>

`#[AsEventListener]` se dispara cuando el código de la aplicación llama a `ApplicationEventPublisher::publish(object $event)` con un objeto PHP tipado — el mismo mecanismo sobre el que cabalga el despacho post-confirmación del Capítulo 6 (`DomainEventDispatcher` llama exactamente a esto para publicar un `DomainEvent` drenado). `#[EventListener]`, el tema de este capítulo, se dispara cuando llega al bus de `eda` un `EventEnvelope` cuya cadena `eventType` coincide con tu patrón — en memoria, a través de un worker de cola, o (la segunda mitad de este capítulo) a través de una fila del outbox de Postgres. Ningún paquete depende del otro; nada en `firefly/eda` importa siquiera el `Dispatcher` de Illuminate.

El puerto `EventPublisher` y el `EventEnvelope`

Todo en este capítulo pasa por un único puerto pequeño:

```

interface EventPublisher
{
    public function subscribe(string $eventPattern, 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()` toma un patrón al estilo fnmatch (`'user.*'` , `'order.created'` , un `'*'` desnudo); `publish()` toma una cadena de destino, una cadena de tipo de evento, una carga útil y encabezados opcionales, y construye el envoltorio que recibe cada suscriptor que coincida:

```

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` y `destination` son dos cosas genuinamente distintas, y el resto de este capítulo gira sobre esa distinción: `destination` es *hacia dónde* se enruta el envoltorio en el cable (un nombre de cola, la columna `destination` de una fila del outbox de Postgres); `eventType` es *qué hecho transporta*, y es el **único** campo contra el que se compara jamás un patrón de `#[EventListener]`.

`withHeaders(array $extra): self` devuelve una copia con los encabezados fusionados — usado internamente para estampar `x-correlation-id`, y (lo verás de nuevo más abajo) `x-original-topic` / `x-exception` en un envoltorio enviado a la carta muerta. `start()` / `stop()` son no-ops en los adaptadores en memoria y de cola; el propio consumidor del outbox (más adelante en este capítulo) es donde empiezan a hacer trabajo real.

`#[EventListener]` y el tropiezo de la coincidencia por `eventType`

```
#[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]` es **metadato inerte y nada más**, la misma regla que todo atributo de Firefly — un único patrón de cadena se normaliza a una lista de un elemento, y `IS_REPEATABLE` permite que un método lleve varias suscripciones, cada una con su propio orden. La suscripción en sí ocurre dentro del registro compartido que componen ambos adaptadores distribuidos:

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

Lee con atención esa llamada a `fnmatch()` : compara `$subscriber['pattern']` contra `$envelope->eventType` — **nunca** contra `$envelope->destination` . Este es exactamente el tropiezo que sufrió la construcción de `samples/lumen` al conectar `LedgerProjector` en el Capítulo 6, y merece la pena repetirlo aquí en su totalidad

porque es la forma más fácil de conectar un listener que jamás se dispara en silencio. Todo evento de dominio del monedero lleva

`#[PublishDomainEvent('wallet.events')]` — esa cadena es el **destino** al que se enruta el evento una vez que cruza al mundo de los eventos de integración. El propio docblock de `LedgerProjector` enuncia la regla como una advertencia exactamente por este motivo:

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

```
    }
}
```

Un tentador `#[EventListener(['wallet.*'])]` — razonando por analogía con la cadena de destino — compila, se despliega, y luego no invoca ni una sola vez `onWalletEvent()`, porque ningún `eventType` de un evento del monedero (`'FundsDeposited'`, `'WalletOpened'`, ...) empieza jamás por `'wallet.'`; esa palabra solo aparece en el *destino*. El Capítulo 6 te mostró el código fuente completo de `LedgerProjector` y sus consecuencias para el modelo de lectura; este capítulo es donde vive de verdad la regla de la que depende.

⚠ EVENTTYPE , NUNCA DESTINATION

Cada vez que escribas un patrón de `#[EventListener]`, pregúntate "¿este glob coincide con el nombre corto de clase del evento que quiero, y no con la cadena de cola/tema por la que resulta enrutarse?" Las dos son cadenas sin relación fáciles de confundir precisamente porque a menudo *parecen* relacionadas (el destino `'wallet.events'`, el tipo de evento `'WalletOpened'`) — `fnmatch()` solo ve jamás la segunda.

Una segunda ilustración mínima, tomada de la propia documentación de `firefly/eda`, concreta el sitio de llamada, patrón y publicación imperativa juntos:

```
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]);
    }
}

```

Aquí el destino (`'orders'`) y el tipo de evento (`'order.placed'`) son cadenas deliberadamente distintas, y el patrón `'order.*'` coincide con la **segunda** — precisamente para que este ejemplo no pueda confundirse con evidencia de que los patrones alguna vez miran el destino.

`EventListenerScanner` (`packages/eda/src/Scanner/EventListenerScanner.php`) es el único sitio de reflexión en `firefly/eda`, recorre una vez las raíces PSR-4 de la aplicación y compila cada método `#[EventListener]` en un `EventListenerDescriptor`; `EventListenerManifest` carga el resultado sin reflexión alguna, exactamente el mismo idioma de `HandlerManifest` que te mostró el Capítulo 7. `firefly:cache` ejecuta este escáner como uno de sus doce pares, escribiendo `bootstrap/cache/firefly/event-listeners.php`; `FireflyCacheServiceProvider` lo vincula, sustituyendo el valor por defecto vacío con el que arrancaría una aplicación sin caché. `EventListenerWiringPass` recorre después ese manifiesto al arrancar — en *todo* proceso, tanto en la petición web como en el worker de cola — envuelve cada invocación de destino en el decorador de reintento/DLQ de más abajo, y llama a `$bus->subscribe($pattern, $wrapped)` por cada patrón, resolviendo el bean de destino **fresco desde el contenedor en cada despacho**.

Reintento y carta muerta: todo listener se envuelve, siempre

`RetryingEventHandler::wrap()` decora *todo* manejador suscrito — deliberadamente no hay una ruta rápida sin envolver:

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

```

Un lanzamiento se reintenta hasta `firefly.eda.retries` veces más con retroceso **lineal** (el intento N duerme `retryDelay * N` segundos); en el fallo final, un `DeadLetterStore` vinculado recibe el envoltorio — enriquecido con `x-original-topic / x-exception` — y el fallo se traga; sin ningún almacén vinculado, la excepción original se relanza sin cambios. Con `retries === 0` y sin DLQ, el manejador igual se ejecuta a través del envoltorio, solo una vez, el mismo resultado observable que una llamada sin envolver. `InMemoryDeadLetterStore` — una lista en proceso, de solo-anexado, de `DeadLetterEntry` (envoltorio + clase de excepción + mensaje) — es el valor por defecto de `#[ConditionalOnMissingBean]`: nada se pierde en silencio, pero tampoco nada sobrevive al proceso.

❗ DOS CAPAS DE REINTENTO INDEPENDIENTES

En el adaptador `QueueEventBus`, la propia maquinaria de reintento a nivel de trabajo/ `failed_jobs` de Laravel gobierna el trabajo de **entrega del envoltorio** en sí. `RetryingEventHandler` gobierna si el **propio fallo del manejador** de un `#[EventListener]` individual se reintenta y se envía a la carta muerta — de forma completamente independiente, en ambos adaptadores. Un trabajo que se entrega con éxito pero cuyo listener lanza una excepción es asunto de `RetryingEventHandler`, no de reintento de cola.

En memoria y en cola: los dos adaptadores de M9

`InMemoryEventBus` es el valor por defecto del esqueleto (`firefly.eda.provider` sin fijar, o `memory`) — síncrono, cero servicios externos:

```
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()` construye el envoltorio y se lo entrega a

`SubscriberRegistry::deliver()` **de forma síncrona** — todo manejador que coincida ya se ha ejecutado para cuando `publish()` retorna, y un manejador que lance, lanza dentro de la llamada que publicó. Las dos llamadas a `EdaTracing` que lo envuelven son un no-op hasta que el Capítulo 11 instale un tracer real; lo que compran es que el bus en memoria emita el mismo par de spans productor-luego-consumidor que emitiría un broker, así que una traza no cambia de forma cuando cambias de proveedor.

`QueueEventBus` (`firefly.eda.provider=queue`) mantiene el contrato

`EventPublisher` idéntico pero hace que `publish()` sea de disparar-y-olvidar:

```

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));
}
// ...
}

```

Las mismas dos costuras, repartidas entre dos procesos: `publish()` es el lado productor en la petición web, `deliver()` el lado consumidor en el worker, y el `traceparent` que viaja en las cabeceras del envoltorio es lo único que los une.

`publish()` retorna antes de que ningún listener se haya ejecutado — la entrega ocurre en el worker de cola que sea que recoja el `DispatchEventJob`. El propio arranque de ese worker vuelve a poblar su `SubscriberRegistry` a partir del mismo manifiesto compilado que lee `EventListenerWiringPass` en todas partes, así que un worker que no comparte nada reconstruye el conjunto idéntico de suscriptores cada vez que arranca. Bajo el driver de cola `sync`, la entrega colapsa al mismo comportamiento síncrono que `InMemoryEventBus` — que es exactamente cómo una prueba ejerce la ruta asíncrona sin que haya ningún worker realmente en ejecución.

El puente de evento de dominio a evento de integración

El Capítulo 6 te mostró el *efecto* del puente — un `WalletOpened` confirmado llegando a `LedgerProjector` como un `EventEnvelope`. Aquí está el mecanismo. Todo `DomainEvent` confirmado a través del despacho post-confirmación de `firefly/data` (la sección de cierre del Capítulo 6) se publica también, en proceso, a `ApplicationEventPublisher` — la propia superficie `#[AsEventListener]` con la que este capítulo abrió al contrastarla con `#[EventListener]`. Como el propio dispatcher de Illuminate compara un evento objeto solo por su **clase concreta** (más las interfaces, nunca las clases padre), un único `#[AsEventListener]` tipado sobre la clase base abstracta `DomainEvent` nunca podría dispararse para ninguna subclase concreta. `DomainEventBridgeWiringPass` esquiva exactamente esta limitación con un único listener comodín (*wildcard*) protegido:

```
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()` mapea el `DomainEvent` sobre el puerto `EventPublisher` de eda a través de `EdaCommandEventPublisher`: `eventType = $event->eventType()` (el nombre corto de la clase — la misma cadena contra la que comparan los patrones de `#[EventListener]`), `payload` = los campos públicos del evento vía `get_object_vars()`, y `destination` = una anulación explícita, si no el `#[PublishDomainEvent(destination:)]` del evento (`'wallet.events'` para todo evento del monedero), si no `firefly.cqrs.default_destination` (`'cqrs.events'`). El id de correlación activo (el `CorrelationContext` del Capítulo 7) se estampa en el encabezado `x-correlation-id` del envoltorio.

```

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],
            );
        }
    }
}

```

```
    }  
}
```

Esto se ejecuta **después** de que la propia transacción del agregado ya ha confirmado — la escritura ya tuvo éxito para cuando se ejecuta este código.

`firefly.cqrs.event_failure_strategy` (por defecto `log`) decide qué ocurre si la publicación al broker falla a continuación: `log` traga el fallo y lo registra — el resultado del comando se mantiene; la publicación de integración fue de mejor esfuerzo. `raise` la relanza, envuelta en `CommandProcessingException`, exponiendo el fallo post-confirmación al llamador original aunque la escritura subyacente ya sea duradera. En cualquier caso, el puente se dispara **solo** para una unidad de trabajo genuinamente confirmada: una transacción revertida no lanza ningún `DomainEvent` en absoluto (Laravel descarta los callbacks de `afterCommit` en un rollback), así que jamás se publica nada para ella — la misma garantía que el modelo post-confirmación del Capítulo 6 ya te dio para los listeners en proceso, ahora extendida a través de la frontera del proceso.

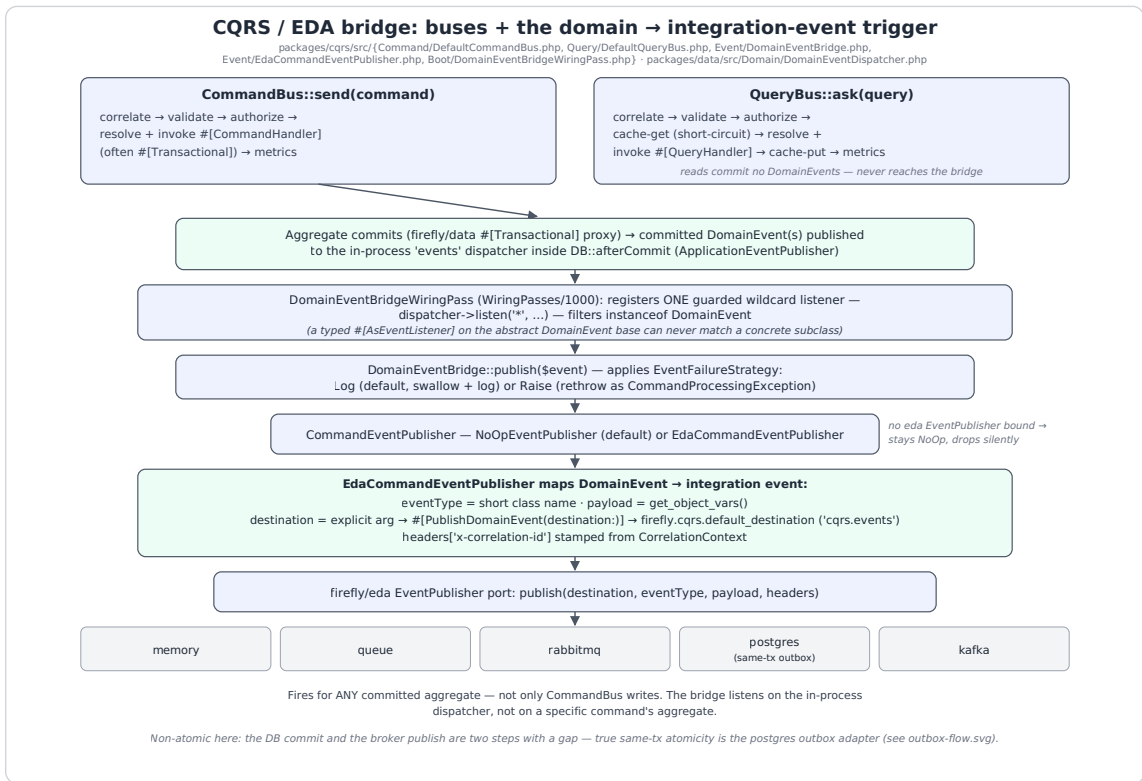


Figura 8.1 — El puente dominio-a-integración: un listener comodín protegido sobre el dispatcher en proceso captura todo DomainEvent confirmado y lo republica a través del CommandEventPublisher resuelto.

Cuando no hay ningún `EventPublisher` de `firefly/eda` vinculado en absoluto, `CqrsAutoConfiguration` recae en `NoOpEventPublisher`, que descarta el evento (registrándolo opcionalmente a nivel debug) — instalar `firefly/cqrs` sin `firefly/eda` sigue arrancando limpiamente y simplemente no emite eventos de integración.

La brecha que esto deja abierta. La escritura confirma; *después*, en un paso separado, se intenta la publicación al broker. Entre esos dos pasos hay una ventana real — un proceso caído, una caída del broker — en la que el propio estado de la base de datos del agregado ya avanzó pero el evento de integración que describía ese avance nunca se envió. Esta es una garantía honesta, al-menos-una-vez-ish, no

atómica, adecuada para muchísimas aplicaciones y explícitamente inadecuada para movimientos de dinero sobre los que otros servicios deben reaccionar de forma fiable. Cerrar esa brecha es exactamente lo que construye el resto de este capítulo.

El genuino outbox de la misma transacción (`firefly/eda-postgres`)

`firefly/eda-postgres` cierra la brecha escribiendo la fila del outbox **dentro** de la propia transacción abierta del agregado, en la propia conexión del agregado — así que confirma o revierte de forma atómica junto con el agregado, sin doble escritura y sin nada que "esperar" después del hecho.

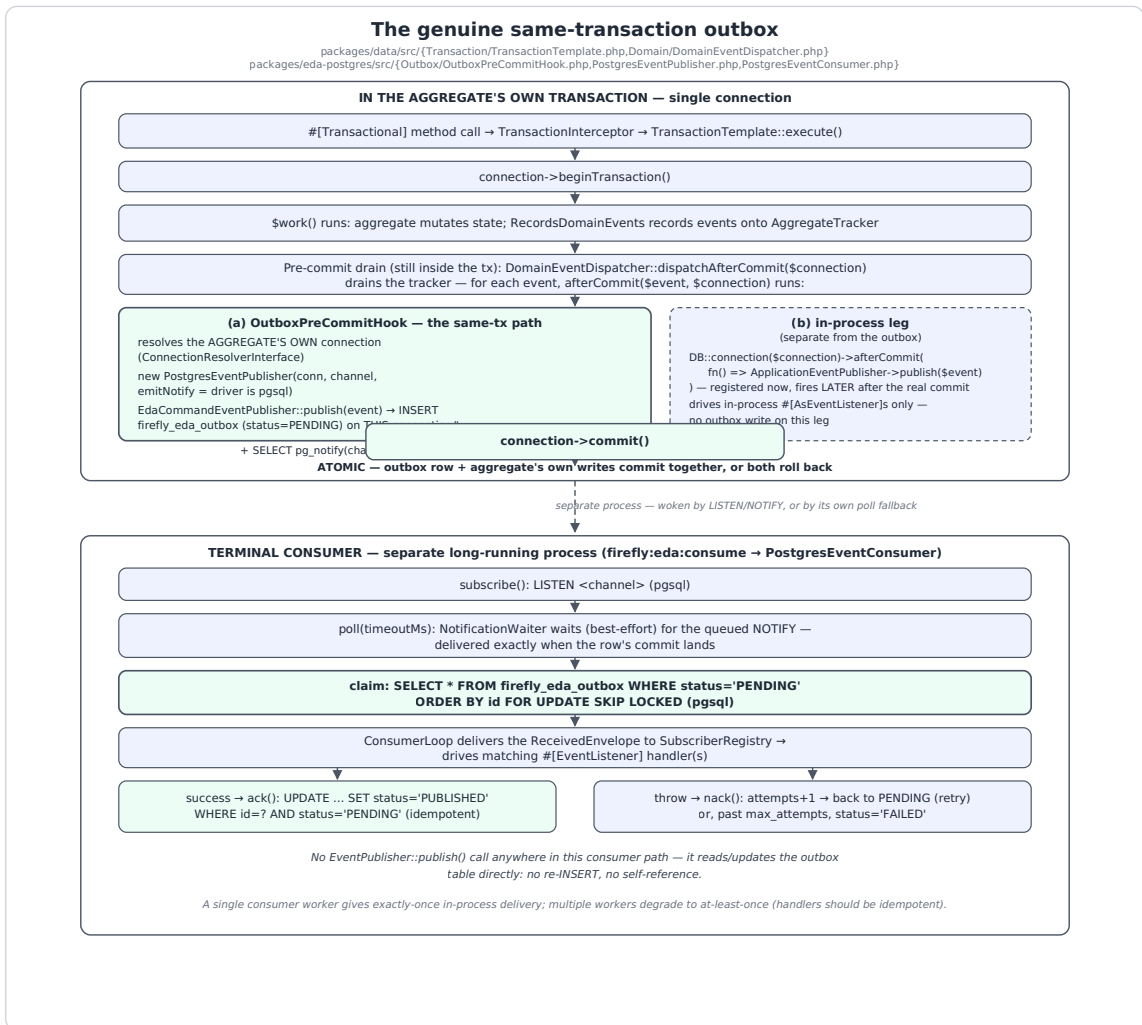


Figura 8.2 — El outbox de la misma transacción: la propia confirmación del agregado y el INSERT del outbox son una unidad atómica; un consumidor reclama filas PENDING de forma independiente.

El mecanismo: **PreCommitEventHook**

El Capítulo 6 mostró a `DomainEventDispatcher::dispatchAfterCommit()` drenando el `AggregateTracker` y planificando cada evento vía `DB::afterCommit()`. Lleva un colaborador más, opcional — un hook que se ejecuta **antes** de esa planificación, mientras la transacción sigue abierta:

```
interface PreCommitEventHook
{
    public function handle(object $event, ?string $connection = null): void;
}
```

Sin ningún hook vinculado — sin `firefly/eda-postgres`, o con cualquier proveedor que no sea `postgres` — esto es un no-op completo y `DomainEventDispatcher` se comporta exactamente como describió el Capítulo 6. `OutboxPreCommitHook` de `firefly/eda-postgres` es la implementación que lo activa:

```
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` se propaga sin cambios desde el dispatcher, así que la fila del outbox de un agregado `#[Transactional(connection: 'x')]` aterriza en la conexión `x` — nunca un valor por defecto fijo — y el hook reutiliza el *mismo* mapeo de

`EdaCommandEventPublisher` (tipo de evento, carga útil, destino, id de correlación) que usa el puente post-confirmación, así que la fila que esto escribe es idéntica byte a byte en forma a lo que habría producido el puente ordinario.

El INSERT, y `pg_notify` en la misma transacción

```
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]);
            }
        });
    }
}
```

```
// ...
}
```

Como `publish()` se ejecuta sobre **cualquiera que sea la conexión con la que se construyó**, y esa conexión está en medio de una transacción cuando `OutboxPreCommitHook` la llama, el `INSERT` se alista en la propia transacción del agregado: confirma cuando el agregado confirma, y revierte cuando el agregado revierte. Cuando el driver es `pgsql`, el mismo closure también dispara `SELECT pg_notify(?, ?)` — **dentro de la misma transacción**. Postgres encola un `NOTIFY` lanzado dentro de una transacción y solo lo entrega de verdad una vez que esa transacción confirma, así que un consumidor a la escucha se despierta con baja latencia en el instante exacto en que la fila se vuelve visible para otras conexiones — y nunca se despierta para una fila que termina revertida.

Ese cuerpo entero vive dentro de `EdaTracing::tracePublish()` — el puerto único por el que pasan ahora los tres publicadores de broker, `eda-rabbitmq` y `eda-kafka` junto a este. Como propagar significa *escribir* algo en el sobre, el puerto le entrega al closure de envío las cabeceras que debe llevar y el adaptador construye la fila a partir de ellas, así que el `traceparent` que estampa el span productor va en la fila que confirma con el agregado — este es el único adaptador donde el span productor y el registro duradero son genuinamente atómicos, en lugar de un span que termina en cuanto un socket aceptó el mensaje. Con el trazado apagado, el `EdaTracing` vinculado es `NoOpEdaTracing`, que llama al closure con las cabeceras del llamador sin cambios, así que el camino es byte a byte el que ya era; y `transaction_id` se sigue leyendo de `x-correlation-id`, nunca de un `traceparent`, así que la clave de idempotencia del relay no cambia solo porque una petición resultara estar trazada.

La tabla del outbox (`firefly_eda_outbox` , un único esquema compartido por la migración, el publicador, el consumidor y el relay) lleva: `id`, `destination`, `channel`, `event_type`, `payload`, `headers`, `transaction_id`, `status`, `attempts`, `error_message`, `created_at`, `processed_at`, `failed_at` , con `status` siendo uno de `PENDING` / `PUBLISHED` / `FAILED` .

Una prueba genuina y distribuida demuestra ambas mitades de la garantía directamente contra una transacción real:

```
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);
});
```

Ninguna cantidad de prosa sustituye a esa segunda prueba: la fila del outbox de la unidad de trabajo fallida simplemente no existe después, porque nunca fue una escritura separada para empezar — fue una sentencia más dentro de la misma transacción que se revirtió.

Evitar la doble publicación

Con `provider=postgres` vinculado, tanto `OutboxPreCommitHook` (en-tx) como el `DomainEventBridge` post-confirmación ordinario podrían, en principio, intentar escribir el mismo evento en el outbox. `PostgresOutboxAutoConfiguration` lo evita en `#[Order(900)]` — por delante de los valores por defecto `#[Order(1000)]` de `firefly/data` y `firefly/cqrs` — vinculando, en conjunto, como un único paquete:

- su **propio** `DomainEventDispatcher`, portando el `OutboxPreCommitHook`, que gana el `#[ConditionalOnMissingBean(DomainEventDispatcher::class)]` de `DataAutoConfiguration`, y
- un `NoOpEventPublisher` como el `CommandEventPublisher`, que gana el `#[ConditionalOnMissingBean(CommandEventPublisher::class)]` de `CqrsAutoConfiguration` — silenciando **solo** el tramo post-confirmación de eda.

El resultado: un evento de dominio se escribe en el outbox **exactamente una vez**, en-tx, de forma atómica junto con el agregado. El despacho post-confirmación genérico (no de eda) `#[AsEventListener]` con el que abrió este capítulo queda completamente intacto — esos listeners en proceso siguen disparándose exactamente como antes. Los cuatro beans están además protegidos por `#[ConditionalOnProperty('firefly.eda.provider', 'postgres')]`, así que instalar el paquete sin seleccionarlo como proveedor activo es totalmente inerte.

Reclamar filas: `PostgresEventConsumer`

Una vez que una fila es `PENDING`, `PostgresEventConsumer` es la ruta de entrega en proceso terminal, siempre disponible — `php artisan firefly:eda:consume` lo conduce como un worker de larga duración:

```
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()` primero espera brevemente (de mejor esfuerzo, protegido contra cualquier fallo) un `NOTIFY`, y luego — sin importar si llegó uno — reclama la fila `PENDING` más antigua con `FOR UPDATE SKIP LOCKED` en postgres (un `ORDER BY id` sencillo bajo sqlite, para las pruebas). Fíjate en que **no hay ninguna llamada a**

`EventPublisher::publish()` en ningún lugar de esta clase — lee y actualiza la tabla del outbox directamente, así que no hay un `INSERT` de vuelta ni riesgo de que un consumidor realmente su propia salida a la tabla de la que acaba de leer.

`ConsumerLoop` entrega el envoltorio reclamado al mismo `SubscriberRegistry` al que ya se suscribieron tus manejadores de `#[EventListener]`, exactamente igual que hacen los adaptadores en memoria y de cola; el guardián `WHERE status = 'PENDING'` de `ack()` hace que la confirmación sea idempotente, y un lanzamiento del manejador llama en cambio a `nack()`, que incrementa `attempts` y o bien reinicia la fila a `PENDING` (reintento) o, pasado `firefly.eda.postgres.max_attempts` (3 por defecto),

la marca `FAILED` in situ — no hay una tabla de carta muerta separada para Postgres; una fila `status='FAILED'` es la carta muerta, consultable en la misma tabla `firefly_eda_outbox`.

Esta es una ventana de estado `PENDING` → `PUBLISHED` **duradera**, no una marca de agua alta en memoria: un consumidor reiniciado retoma en la fila `PENDING` más antigua todavía pendiente y nunca reproduce una fila ya `PUBLISHED`. Un único worker consumidor da entrega en proceso exactamente-una-vez; ejecutar más de uno a la vez degrada a al-menos-una-vez (escribe tus listeners de forma idempotente si lo haces).

📦 PARIDAD CON LARAVEL

Laravel puro no tiene ningún patrón de outbox transaccional de primera parte en absoluto — el idioma que tendrías que construir a mano es una tabla `outbox` más un comando planificado que la sondea, sin ninguna ayuda del framework para mantener el `INSERT` dentro de la misma transacción que tu escritura de dominio. `firefly/eda-postgres` es exactamente ese idioma, pero con la atomicidad garantizada por construcción (`PreCommitEventHook` se ejecuta mientras la transacción sigue abierta) en vez de dejada a la disciplina del desarrollador.

✓ Lo que aprendiste

Concepto	Qué hace
Dos superficies de eventos	<code>#[AsEventListener]</code> (en proceso, <code>firefly/context</code>) frente a <code>#[EventListener]</code> (bus broker, <code>firefly/eda</code>) — mecanismos sin relación
<code>EventPublisher</code> / <code>EventEnvelope</code>	El puerto del bus broker; <code>destination</code> es <i>hacia dónde</i> , <code>eventType</code> es <i>qué hecho</i>
<code>SubscriberRegistry::deliver()</code>	Compara <code>fnmatch(\$pattern, \$envelope->eventType)</code> — nunca el destino; el tropiezo exacto que sufrió <code>LedgerProjector</code>
<code>RetryingEventHandler</code>	Envuelve siempre a todo listener: reintento con retroceso lineal, luego carta muerta (o relanzamiento sin almacén vinculado)
<code>InMemoryEventBus</code> / <code>QueueEventBus</code>	Adaptadores <code>EventPublisher</code> síncrono frente a respaldado-por-cola (disparar-y-olvidar), puerto idéntico
<code>DomainEventBridge</code>	El disparador post-confirmación que republica un <code>DomainEvent</code> confirmado como evento de integración — de mejor esfuerzo, no atómico
<code>OutboxPreCommitHook</code> / <code>PostgresEventPublisher</code>	Escribe la fila del outbox dentro de la propia transacción abierta del agregado — genuinamente atómico
<code>PostgresEventConsumer</code>	Reclama filas <code>PENDING (FOR UPDATE SKIP LOCKED + LISTEN / NOTIFY)</code> , conduce directamente los manejadores de <code>#[EventListener]</code> , sin <code>INSERT</code> de vuelta

Ponlo en práctica

1. **Reproduce el tropiezo, deliberadamente.** En una copia de trabajo del proyecto, cambia los patrones de `#[EventListener]` de `LedgerProjector` a `['wallet.*']` y vuelve a ejecutar su suite de pruebas. Confirma que el proyector jamás se dispara en silencio y que nunca se escribe ninguna fila en `ledger_entries` — luego restaura los patrones originales de nombre de tipo y confirma que vuelven a pasar.
2. **Fuerza una carta muerta.** Escribe un pequeño manejador `#[EventListener]` que siempre lance una excepción, configura `firefly.eda.retries=2` y vincula un `InMemoryDeadLetterStore`, publica un evento que coincida, e inspecciona `DeadLetterStore::all()` después. Confirma que el `exceptionMessage` de la entrada coincide con lo que lanzó tu manejador.
3. **Demuestra tú mismo el rollback del outbox.** Usando una conexión Postgres (`@group('integration')`, según las propias convenciones de prueba del framework), reproduce los dos casos de `OutboxSameTransactionTest` que citó este capítulo — publicar-y-luego-confirmar y publicar-y-luego-lanzar — y confirma que la presencia o ausencia de la fila coincide con la descripción de este capítulo.

Transacciones y el Proxy

#[Transactional]

Al terminar este capítulo sabrás exactamente qué hace `#[Transactional]` — sus siete modos de propagación, sus ajustes de aislamiento/solo-lectura/rollback, y el proxy generado que le da dientes — cómo la auto-invocación esquivaba ese proxy (y qué hacer en su lugar), cómo se hace cumplir de verdad un `timeout:` — un reloj de pared monótono que juzga el método después de que retorna, más un tiempo límite de sentencia del driver, en la medida de lo posible, capaz de interrumpir una consulta en marcha — cómo `#[TransactionalEventListener]` difiere *al listener* en lugar del evento hacia una de cuatro fases de la transacción, de las cuales `BEFORE_COMMIT` puede vetar la confirmación de plano, cómo `firefly:cache` y el escaneo en proceso dan juntos a toda aplicación un proxy funcional sin cableado propio, y el hecho más consecuente de todo este libro hasta ahora: **un evento de dominio se publica únicamente porque `TransactionTemplate` — la maquinaria detrás de `#[Transactional]` — es el único que llama al despacho post-confirmación.** Sin `#[Transactional]`, no hay publicación, sin importar con cuánta corrección un agregado haya lanzado su evento.

❗ TÉRMINO NUEVO: DEMARCACIÓN DECLARATIVA DE TRANSACCIONES

En vez de escribir `DB::beginTransaction()` / `DB::commit()` / `DB::rollback()` a mano dentro del cuerpo de un método, *declaras* la frontera con un atributo y dejas que un proxy generado la haga cumplir. Este es el modelo `@Transactional` de Spring, y es por lo que los Capítulos 6 y 7 ya pudieron mostrarte `#[Transactional]` sobre `OpenWalletHandler`, `DepositHandler`, `WithdrawHandler` y `TransferHandler` sin una sola llamada explícita a `DB::` dentro de ninguno de sus cuerpos `handle()`.

El atributo `#[Transactional]`

```
#[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,
    ) {}
}
```

Sobre una **clase**, fija el valor por defecto para cada método público. Sobre un **método**, *reemplaza* — nunca se fusiona con — el atributo de nivel de clase para ese único método (la misma semántica de Spring). `packages/data/tests/Fixtures/Capstone/AccountService.php` es código de prueba real y distribuido que pone en juego ambas formas a la vez:

```
<?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) {
    }
}
}
}

```

Cada método público aquí hereda el valor por defecto de `#[Transactional]` a nivel de clase (`REQUIRED`, rollback ante cualquier `Throwable`) **excepto** `logButKeep()`, cuyo propio atributo de nivel de método *reemplaza por completo* ese valor por defecto para este único método — no apila `noRollbackFor` encima del valor por defecto de la clase, es la configuración efectiva. Por defecto `rollbackFor = [Throwable::class]`: PHP no tiene una separación entre excepciones comprobadas y no comprobadas, así que por defecto *cualquier* throwable revierte la transacción, a menos que también coincida con `noRollbackFor`, que siempre gana.

Los siete modos de propagación

`Propagation` es un enum sin respaldo con todos los modos de Spring, incluyendo `NESTED` — posible sobre una conexión relacional corriente gracias a los propios savepoints automáticos de Laravel:

```

enum Propagation
{
    case REQUIRED;
    case REQUIRES_NEW;
    case NESTED;
    case SUPPORTS;
}

```

```

    case NOT_SUPPORTED;
    case MANDATORY;
    case NEVER;
    // ...
}

```

`TransactionTemplate::execute()` es la única fuente de verdad por la que pasan tanto el proxy generado como cualquier llamador programático directo — no hay un segundo camino de código que mantener sincronizado:

```

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());
    }
}

```

El método entero es un solo `match` sobre el modo de propagación y un `catch` que entrega lo que haya salido al traductor de excepciones del Capítulo 5. Todo lo demás — los savepoints, las sentencias de aislamiento, el tiempo límite, el drenaje posterior al commit — está dentro de `runInTransaction()`, al que solo llegan tres de los siete modos.

Modo	Comportamiento
REQUIRED <i>(por defecto)</i>	Se une a la transacción del llamador si hay una activa; si no, inicia una nueva transacción más externa.
REQUIRES_NEW	Siempre corre dentro de una transacción. Sin ninguna activa se convierte en la nueva transacción más externa; con una ya activa en la misma conexión , Laravel no tiene primitiva de suspensión, así que degrada a un <code>beginTransaction()</code> anidado — un savepoint, no una transacción verdaderamente independiente.
NESTED	La misma mecánica que REQUIRES_NEW en esta implementación: una transacción más externa nueva si ninguna está activa, o si no, un savepoint — así que un fallo en NESTED se deshace solo hasta su propio savepoint, nunca toda la unidad de trabajo.
SUPPORTS	Corre dentro de la transacción del llamador si hay una activa; si no, sin transacción alguna. Nunca inicia una.
NOT_SUPPORTED	Siempre corre sin transacción — pero en la <i>misma</i> conexión no hay primitiva de suspensión, así que una transacción ya activa simplemente no se pausa; el trabajo igual corre dentro de ella.
MANDATORY	Requiere una transacción activa; corre dentro de ella, o lanza <code>TransactionRequiredException</code> .

Modo	Comportamiento
NEVER	Prohíbe una transacción activa; lanza <code>TransactionNotAllowedException</code> si hay una activa, o si no, corre sin ninguna.

`outerWithNested()` de arriba es `NESTED` en acción: el método externo inserta `'outer'` bajo el valor por defecto `REQUIRED` de nivel de clase, después llama a `$this->template->execute(...)` directamente con `propagation: Propagation::NESTED`, inserta `'inner'`, y lanza. Una prueba capstone real y distribuida demuestra exactamente qué se deshace y qué sobrevive:

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

La fila `'inner'` desaparece con el savepoint; `'outer'` — insertada antes siquiera de que empezara la unidad de trabajo anidada — sobrevive y luego confirma normalmente cuando el método externo retorna.

Aislamiento, solo-lectura, y la decisión de rollback

`Isolation` es un enum respaldado por cadenas cuyo valor es la cláusula SQL:

```
enum Isolation: string
{
```

```

    case DEFAULT = 'DEFAULT';
    case READ_UNCOMMITTED = 'READ UNCOMMITTED';
    case READ_COMMITTED = 'READ COMMITTED';
    case REPEATABLE_READ = 'REPEATABLE READ';
    case SERIALIZABLE = 'SERIALIZABLE';
}

```

En la transacción más externa de una unidad de trabajo, un aislamiento distinto de `DEFAULT` emite `SET TRANSACTION ISOLATION LEVEL {value}`, y `readOnly: true` emite `SET TRANSACTION READ ONLY` — ambos **de mejor esfuerzo**: si cualquiera de las dos sentencias falla (un driver que la ignora, como SQLite) se captura y se ignora en silencio en vez de hacer fallar toda la unidad de trabajo.

```

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
}

```

Léelo en orden: una excepción que coincide con `noRollbackFor` siempre **confirma** (se comprueba primero, así que gana incluso sobre una coincidencia en `rollbackFor`); si no, una coincidencia en `rollbackFor` (por defecto, todo) **revierte**; si

no — solo alcanzable con un `rollbackFor` deliberadamente reducido — **confirma**. `AccountService::logButKeep()` ejercita exactamente la primera rama, y una prueba capstone real demuestra que la fila sobrevive a la excepción con la que se lanza:

```
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);
});
```

Nota que `TransactionTemplate` usa `beginTransaction()` / `commit()` / `rollback()` **manuales** en todo momento, nunca el propio `DB::transaction($closure)` de Laravel — solo el control manual permite que una excepción capturada sea *confirmada*-y-relanzada cuando `noRollbackFor` lo indica, en lugar del propio helper de Laravel, que siempre revierte ante cualquier excepción sin ningún punto de anulación de ese tipo.

El modelo de proxy

Un bean `#[Transactional]` nunca se llama directamente.

`TransactionalBeanPostProcessor` — un `BeanPostProcessor` descubierto exactamente como cualquier otro bean, instalado en la fase 700 — lo intercambia, en su segunda pasada (después de que `#[PostConstruct]` ya haya corrido sobre el

bean real), por una instancia de una `final class {Target}` `__FireflyTransactionalProxy` extiende `{Target}` generada. `ProxyClassGenerator` emite esa clase, sobrescribiendo cada método transaccional con:

```
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();
  }
}
```

— entregando la llamada a `MethodInvocation::proceed()`, que recorre los interceptores que el plan compilado nombró para ese método y termina en la closure terminal que llama a `parent::`. Para un bean cuyo único advice es

`#[Transactional]` esa lista tiene un solo eslabón:

`TransactionInterceptor::invoke()` lee el `TransactionalDescriptor` horneado desde la invocación y le pasa `fn () => $invocation->proceed()` a su inalterado `run()`, que delega directamente en `TransactionTemplate::execute()`. Una prueba capstone real confirma que el intercambio realmente ocurrió — la clase del bean resuelto **no** es en absoluto la clase de servicio plana:

```

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);
});

```

`$service::class` es el `AccountService_FireflyTransactionalProxy` generado, y sin embargo `$service instanceof AccountService` sigue siendo verdadero — el proxy es un `{Target}`, así que cada llamada del contenedor y `#[PreDestroy]` se resuelven contra él exactamente como lo harían contra el bean original. `ProxyFactory` lo instancia **preservando el estado**: `newInstanceWithoutConstructor()` (así `#[PostConstruct]` no se vuelve a ejecutar), y luego el estado ya inicializado del bean real se copia sobre el proxy ranura a ranura, cada ranura escrita por una closure vinculada a la clase que la *declara* — nunca `ReflectionProperty::setValue()`. Escribir desde la clase declarante no es un detalle: una closure vinculada solo a la clase declarada únicamente ve los privados de esa clase, así que escribir cada ranura desde donde fue declarada es lo que permite que la copia alcance un `private` de un padre (el traductor de `EloquentRepository`, bajo cada `#[Repository]`) e inicialice un `protected readonly` de un padre (el manifiesto y el tracker de `EloquentRepository`) en PHP 8.3 — el mínimo que el framework soporta — donde una propiedad `readonly` solo es inicializable desde el ámbito de la clase que la declara. El cast `(array)` es lo

que hace exactas las ranuras: mangla un privado como `"\00wner\0name"`, así que un privado que un padre y un hijo declaran bajo un mismo nombre sigue siendo dos ranuras, y una propiedad tipada que el bean nunca inicializó sencillamente no está y se queda sin inicializar en el proxy.

⚠ LA AUTO-INVOCACIÓN ESQUIVA EL PROXY

Un método que llama a `$this->otroMetodo()` desde *dentro* de la clase proxificada llama directamente a través de `parent::`, saltándose `__fireflyTxInterceptor` por completo — la misma limitación bien conocida de Spring. Esto es exactamente por qué `AccountService::outerWithNested()` de arriba no simplemente llama a algún hipotético método `$this->innerNested()` — en su lugar pasa por el `TransactionTemplate` **inyectado**, que es la vía de escape correcta para obtener semántica transaccional en una unidad de trabajo interna desde dentro de otro método de la misma instancia.

Un proxy, muchos advices

Ese proxy ya no va solo de transacciones. Cualquier paquete puede aportar un tipo de advice publicando un `#[Component]` que implemente `AdviceSource` — `advice()` nombra su bean interceptor, su clase descriptora y un **orden**; `scan()` devuelve las filas que reclama; `render()` convierte una fila de vuelta en el literal PHP que la clase generada hornea dentro. `ProxyPlanner` fusiona las filas de cada fuente en un único `ProxyPlan`, aplicando las fuentes en `Advice::order`, así que la lista de advices de cada método queda de fuera hacia dentro por construcción. La clase generada conserva el nombre que siempre ha tenido, `{Target}__FireflyTransactionalProxy`, pero ahora declara una propiedad interceptora privada y una fábrica estática de descriptores horneada **por cada tipo de advice** que la clase usa, y `MethodInvocation::proceed()` recorre esa lista antes de llegar a la closure terminal

`fn (array $__fireflyArgs) => parent::m(...$__fireflyArgs)`. Un Advice de orden menor se ejecuta *más afuera*: el advice de `MethodSecurityAdviceSource` es 100 y el transaccional es 1000, así que un rechazo se lanza antes de que llegue a abrirse una transacción. Un advice cuyo bean interceptor no está presente hace fallar el arranque con una `ConfigurationException` salvo que haya declarado `inertWhenUnbound` — el de seguridad lo hace, porque «las anotaciones son inertes hasta que `firefly.security.enabled` esté activo» es su estado documentado — y entonces `InterceptorRegistry` le entrega al proxy un `PassThroughInterceptor` en su lugar.

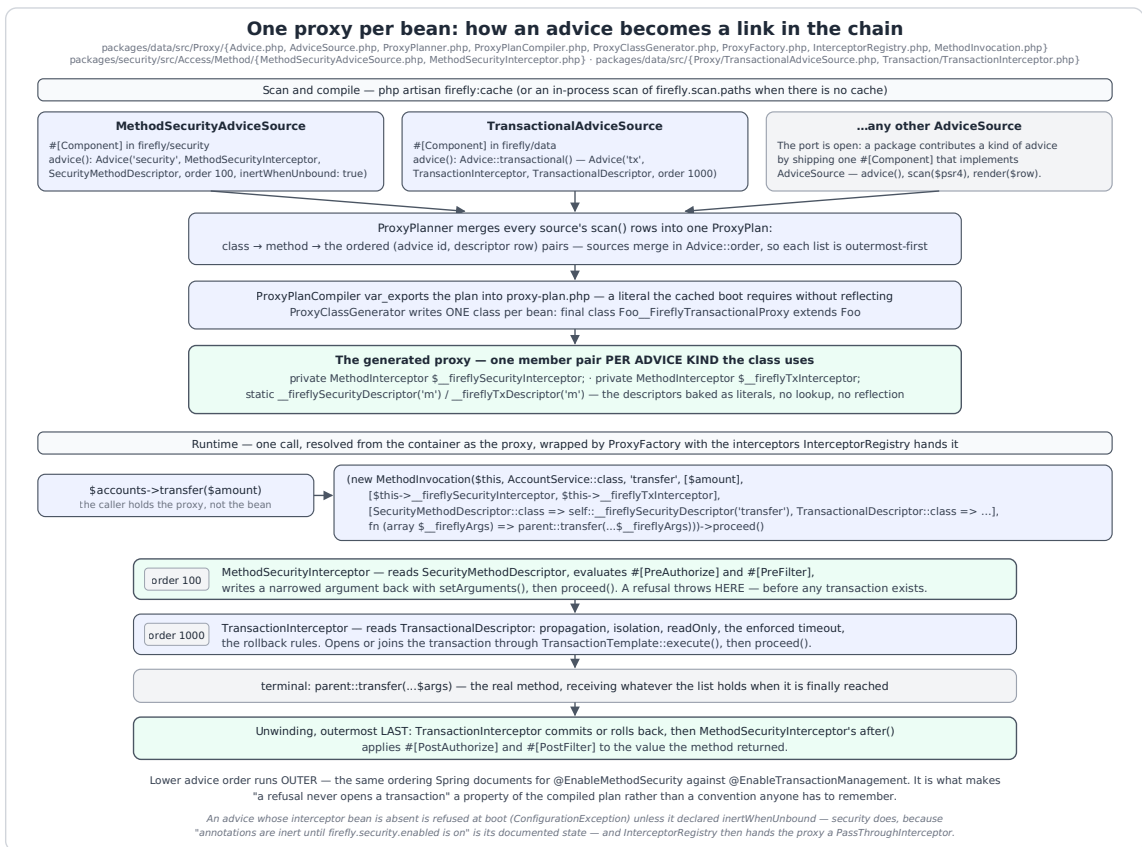


Figura 9.1 — Cada AdviceSource aporta filas a un único ProxyPlan compilado; una llamada ejecuta entonces la seguridad de método en el orden de advice 100 antes que la transacción en 1000.

El tiempo límite se hace cumplir, no solo se transporta

`#[Transactional(timeout: 5)]` era antes metadato que un lector podía fijar y que nadie leía. Ahora se hace cumplir, y merece la pena entender cómo, porque una mitad puede interrumpir una sentencia en curso y la otra no.

```
$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;
```

El reloj de pared es la mitad que siempre se aplica. Se toma un plazo monótono justo después de `beginTransaction()`, y cuando el trabajo *retorna* la plantilla compara el reloj contra él. Pasarse significa hacer rollback y lanzar

`TransactionTimedOutException` — 504, código de error `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,
    ));
}
```

Fíjate en lo que eso no puede hacer: PHP no va a interrumpir un `SELECT` que sigue dentro del driver. La comprobación del reloj de pared juzga al método *después* de que retorne, lo que atrapa un exceso pero no lo detiene.

El tiempo límite de sentencia del driver es la mitad que sí puede. Inmediatamente después del `beginTransaction()` de la transacción más externa, `StatementTimeoutApplier` le dice a la propia base de datos que se rinda, en el dialecto que esa base de datos hable — `SET LOCAL statement_timeout` en postgres (con ámbito de transacción, que es precisamente por lo que debe emitirse después del `BEGIN`), `max_execution_time` más `innodb_lock_wait_timeout` en mysql, el `max_statement_time` propio de mariadb en mariadb, y el tiempo de espera por ocupado en sqlite, que es el único mando que sqlite tiene. `apply()` devuelve el paso que lo deshace, y la plantilla lo ejecuta en su `finally` — porque las variables de mysql son variables de *sesión*, y en una conexión persistente la petición siguiente heredaría si no el presupuesto de esta.

Cada una de esas sentencias es de mejor esfuerzo, exactamente igual que el `SET` de aislamiento: un driver que rechace una no hace fallar la transacción, y la comprobación del reloj de pared sigue aplicándose. Puedes apagar la mitad del driver por completo con `firefly.data.transaction.statement-timeout=false` y quedarte con el reloj de pared.

Tres reglas deciden el número en sí, y un test fija cada una:

```
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:)]` gana a `firefly.data.transaction.default-timeout`; `0` en ambos significa que no hay plazo alguno; y una transacción **unida** nunca agota su tiempo por su cuenta — solo la transacción más externa toma un plazo, que es la regla de Spring y la única que tiene sentido cuando el presupuesto de un método interno truncaría si no un trabajo del que el método externo sigue siendo responsable.

`#[TransactionalEventListener]` : oír un evento en una fase

El `#[AsEventListener]` del Capítulo 8 oye un evento cuando se publica. Dentro de un método `#[Transactional]`, «cuando se publica» suele ser el momento equivocado: la fila está escrita pero no confirmada, así que un listener que envía un correo o encola un trabajo puede estar actuando sobre una transacción que está a punto de revertirse.

Hay dos respuestas distintas a eso, y merece la pena ser preciso sobre cuál quieres:

- `DomainEventDispatcher::publishAfterCommit()` — visto antes en este capítulo — aplaza **el evento**. Nadie lo oye en absoluto antes del commit.
- `#[TransactionalEventListener]` aplaza **el listener**. El evento se publica de inmediato (un `#[AsEventListener]` corriente sobre la misma clase sigue viéndolo dentro de la transacción); este método se encola en la transacción actual y se invoca en la fase que pidió.

```
enum TransactionPhase: string
{
    case BEFORE_COMMIT = 'BEFORE_COMMIT';
    case AFTER_COMMIT = 'AFTER_COMMIT';
    case AFTER_ROLLBACK = 'AFTER_ROLLBACK';
    case AFTER_COMPLETION = 'AFTER_COMPLETION';
}
```

Fase	Cuándo se ejecuta	Qué ve
BEFORE_COMMIT	dentro de la transacción, justo antes del commit	la fila sin confirmar, en nivel de transacción 1 — un throw aborta el commit
AFTER_COMMIT (la de por defecto)	tras un commit con éxito	la fila confirmada, en nivel 0
AFTER_ROLLBACK	tras un rollback	la fila desaparecida
AFTER_COMPLETION	tras cualquiera de los dos	el que de los dos haya ocurrido

La propia prueba del framework declara un listener por fase, que es la forma más clara de leerlos:

```
#[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));
}
}

```

Cuatro detalles, cada uno de los cuales acabarás necesitando:

- **La clase de evento se infiere** del tipo del primer parámetro del método, exactamente como lo hace `#[AsEventListener]`. `event:` está ahí para el caso en que quieras ser explícito.
- **order:** sigue la convención de `#[Order]` — menor primero — entre listeners transaccionales *de la misma fase*.
- **Sin transacción, no hay listener.** Sin nada activo el método se omite, en silencio y a propósito: pidió ejecutarse en una fase que no existe. `fallbackExecution: true` dice «ejecútame ya en su lugar», que es lo que quieres para un listener que debe dispararse de todos modos.
- **El atributo es metadato inerte.** `TransactionalScanner` lo compila en el mapa `listeners` del manifiesto, `TransactionalEventListenerWiringPass` lo registra, y `TransactionSynchronizationRegistry` lo encola contra la transacción en curso — sin reflexión en tiempo de publicación. Todo el mecanismo está detrás de `firefly.data.transactional-event-listeners.enabled`, `true` por defecto.



BEFORE_COMMIT ES UN VETO, Y ESA ES LA PARTE ÚTIL

Una comprobación de invariante de última hora va aquí: se ejecuta dentro de la transacción, así que lanzar desde ella aborta el commit y se lleva por delante toda la unidad de trabajo. Publicar desde ella también funciona — un segundo evento encolado mientras la cola de `BEFORE_COMMIT` se drena se atiende dentro del mismo commit, no se aplaza al siguiente.

Cómo una aplicación real consigue de verdad un proxy funcional

El Capítulo 2 te enseñó la forma de una clase `#[Configuration]` y sus métodos fábrica `#[Bean]`. Durante una versión de este framework, toda aplicación LaraFly tuvo que escribir una de ellas a mano — un `#[Configuration]` cuyo único `#[Bean]` cargaba el `TransactionalManifest` cacheado — o `#[Transactional]` no hacía absolutamente nada. **Ya no necesitas esa clase**, y merece una página: el fallo que aquella clase sorteaba fue de los más afilados que ha tenido este framework, y su arreglo es una pequeña lección sobre cómo se resuelve cada artefacto compilado.

`php artisan firefly:cache` emite **tres** artefactos separados que lee la maquinaria de proxies. Ejecuta `TransactionalScanner / TransactionalManifestCompiler` — uno de los doce pares scanner/compiler — para escribir los datos del manifiesto en `bootstrap/cache/firefly/transactional.php`. Ejecuta `ProxyPlanCompiler` sobre el `ProxyPlan` fusionado para escribir `bootstrap/cache/firefly/proxy-plan.php` — el artefacto que decide qué beans reciben siquiera un proxy y qué advice ejecuta cada uno de sus métodos. Y ejecuta `ProxyClassGenerator` para emitir un archivo fuente `{Target}__FireflyTransactionalProxy` **por cada clase que el plan nombra** — cada clase que reclama algún `AdviceSource`, así que un bean que solo lleva reglas de seguridad de método también recibe uno — en un directorio `proxies/`, más un classmap `proxies.php`.

Durante mucho tiempo, nada cargaba el primero de esos tres.

`DataAutoConfiguration` vinculaba un `TransactionalManifest` vacío incondicionalmente, el `transactional.php` compilado solo lo referenciaba su propia declaración, y por eso `hasProxyFor()` era siempre falso y `TransactionalBeanPostProcessor` devolvía cada bean sin envolver. `#[Transactional]` era un **no-op silencioso** en cualquier aplicación que no escribiera

a mano su propia configuración de manifiesto — que es exactamente lo que aquel `#[Configuration]` manuscrito existía para hacer, y exactamente el modo de fallo del que avisaba la sección anterior, llegando desde el propio lado del framework.

Está arreglado en el origen. Los dos beans resuelven ahora su artefacto como se resuelve en este framework todo artefacto de Categoría B — **archivo compilado primero, escaneo en proceso después, vacío al final**:

```
#[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);
}
```

Tres cosas de ese método merecen leerse despacio. El archivo compilado gana cuando existe, que es la ruta de producción cacheada. Cuando no existe, el escáner corre en proceso sobre `firefly.scan.paths`, que es la ruta de desarrollo — no hace falta `firefly:cache` para tener un proxy funcional mientras escribes código. Y `ProxyMaterializer::classmap()` se llama **antes** de devolver el manifiesto, así que las clases proxy generadas son cargables antes de que nada pueda pedir una;

`TransactionalBeanPostProcessor` lanza una `ConfigurationException` ante un manifiesto que promete una clase proxy que no encuentra, de modo que una caché a medio escribir falla ruidosamente al arrancar en lugar de correr sin proxy en silencio.

`proxyPlan()` se sienta a su lado con la misma forma y un peldaño extra: `proxy-plan.php` primero, luego — para una caché escrita por un `firefly:cache` anterior a la existencia de los planes de proxy — un plan solo-transaccional puenteado desde el manifiesto, luego el escaneo en proceso a través de cada bean `AdviceSource`, y por último, sin ninguna ruta de escaneo, un plan solo-transaccional a partir del `TransactionalManifest` que esté vinculado. Ese último peldaño es lo que permite que los propios fixtures capstone del framework compilen un manifiesto a mano y aun así envuelvan sus beans `#[Transactional]`.

Así que el skeleton **no distribuye** ningún `CachedTransactionalConfiguration`, y tu aplicación tampoco debería. Una aplicación LaraFly obtiene un proxy `#[Transactional]` funcional en desarrollo desde el escaneo y en producción desde la caché, sin cableado propio. Las suites de pruebas del framework sí suministran un `#[Configuration]` equivalente en línea — `packages/data/tests/Fixtures/Capstone/CapstoneTransactionalConfiguration` para el fixture `AccountService` de arriba, y `samples/lumen/tests/Support/LumenTransactionalConfiguration` para los manejadores que este libro toma de `samples/lumen` — porque esas suites no arrancan a través de una aplicación en absoluto y no tienen ningún `firefly.scan.paths` que escanear.

💡 SI ESCRIBISTE UNA DE ESTAS CLASES, BÓRRALA

Un `TransactionalManifest` vinculado a mano sigue funcionando — `#[ConditionalOnMissingBean]` hace que el framework se aparte ante cualquier definición de bean competidora — pero ahora ata tu aplicación a lo que haga esa clase, y nunca fue más que un apaño.

La lección clave: los eventos de dominio se publican solo a través de una frontera #[Transactional]

Todo en esta sección ha estado construyendo hacia un hecho, y es lo más importante que enseña este capítulo. Mira de nuevo desde dónde se llama en realidad a

`DomainEventDispatcher::dispatchAfterCommit()` dentro de `TransactionTemplate` — el brazo del `catch`, el brazo del tiempo límite y el brazo del éxito de un mismo método:

```

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()` se llama desde exactamente **tres** lugares, y los tres están dentro de ese único método de `TransactionTemplate` — no hay un cuarto punto de llamada en ningún lugar del framework. Dos de los tres están ahí para que el rastreador se *drene* en los caminos de fallo: drenarlo en un rollback lo vacía sin programar ninguna publicación, y eso es lo que impide que los eventos de una unidad de trabajo se filtren a la siguiente. Y `dispatchAfterCommit()` en sí solo tiene algo que despachar por un segundo hecho, igual de estructural: `EloquentRepository::save()` (Capítulo 5) solo registra una entidad en `AggregateTracker` cuando `$connection->transactionLevel() > 0` — y el *único* camino de código que alguna vez hace eso cierto es la propia llamada a `beginTransaction()` de `TransactionTemplate::runInTransaction()`, unas líneas más arriba.

Encadena esos dos hechos y la conclusión es ineludible: **un método sin #[Transactional] nunca eleva su nivel de transacción, así que `save()` nunca rastrea el agregado, así que no hay nada que `dispatchAfterCommit()` pueda drenar aunque de alguna forma se le llamara**. El propio docblock de `OpenWalletHandler` declara esto como la razón por la que el atributo está ahí, y no decoración:

```
/**
 * // ...
 * #[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
```

Recuerda del Capítulo 7 que `DefaultCommandBus::send()` no abre **ninguna** transacción propia — correlaciona, valida, autoriza e invoca al manejador, y punto. Cada pizca de comportamiento transaccional que has visto en `OpenWalletHandler`, `DepositHandler`, `WithdrawHandler` y `TransferHandler` proviene *enteramente* del atributo `#[Transactional]` sobre sus métodos `handle()`, a través del mecanismo de proxy exacto que este capítulo acaba de recorrer. Quítale el atributo a cualquiera de ellos y el comando igual "tiene éxito" — la fila igual se escribe mediante una llamada a `save()` plana, sin proxificar — pero `WalletOpened` / `FundsDeposited` / `FundsWithdrawn` se lanzan al búfer privado de eventos del agregado y luego se **descartan en silencio**, porque nada jamás drena ese búfer. `LedgerProjector` (Capítulo 6) simplemente nunca se dispararía, sin error, sin advertencia, y con una respuesta HTTP de apariencia perfectamente exitosa.

⚠ SIN `#[TRANSACTIONAL]`, NO HAY PUBLICACIÓN DE EVENTOS DE DOMINIO — EN SILENCIO

Esto no es un fallo lento ni un mensaje de error engañoso; es un **no-op completo y silencioso** para cada evento de dominio que el agregado del manejador haya lanzado. La escritura se confirma. El evento se desvanece. Nada en la respuesta, en los logs, ni en el esquema de la base de datos te dice que ocurrió — la única forma de notarlo es que un proyector o listener aguas abajo que esperabas que se disparara, nunca lo hace. Trata `#[Transactional]` sobre un manejador de comando que toca un agregado como estructuralmente necesario, no como un estilo.

El **Transfer** atómico: el dinero no puede desvanecerse

El Capítulo 6 ya te mostró el código completo de **TransferHandler** — una única frontera **#[Transactional(propagation: Propagation::REQUIRED)]** envolviendo un débito, un guardado, un crédito, y un segundo guardado. Lo que el Capítulo 6 no te mostró es la prueba rigurosa de que la afirmación de atomicidad realmente se sostiene ante un fallo real. **samples/lumen/tests/Application/TransferSecurityTest.php** despacha a través del **CommandBus / QueryBus** reales — los mismos puertos que introdujo el Capítulo 7 — y demuestra ambas direcciones:

```
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');

```

La segunda prueba es la que importa. `Wallet::withdraw()` sobre el origen corrió con éxito y lanzó `FundsWithdrawn` a su propio búfer; luego `Wallet::deposit()` sobre el destino lanzó un `ConflictException` de moneda no coincidente — *después* de que la llamada `save()` del débito ya se hubiera ejecutado dentro de la misma transacción, todavía abierta. Porque el `rollbackFor` por defecto de `#[Transactional]` captura cualquier `Throwable`, el método entero revierte: el débito del origen se deshace a nivel de base de datos, y — porque el rollback ocurre *antes* de que `dispatchAfterCommit()` fuera a alcanzarse en el camino de éxito, y `TransactionTemplate` también lo llama en la rama `catch` precisamente para que un drenado revertido igual vacíe el rastreador sin llegar jamás a programar una

publicación — ni `FundsWithdrawn` ni un `FundsDeposited` que ni siquiera llegó a lanzarse alcanza a ningún listener. El `CommandProcessingException` del Capítulo 7 envuelve el `ConflictException` subyacente, y las dos aserciones de saldo son todo el punto: no "la transferencia falló" en abstracto, sino que el propio `10000` del origen volvió **exactamente**, y el `0` del destino nunca se movió. No se creó valor; no se destruyó ninguno.

✓ Lo que aprendiste

Concepto	Qué hace
<code>#[Transactional]</code>	Valor por defecto a nivel de clase, <i>reemplazo</i> a nivel de método (no fusión); <code>rollbackFor = [Throwable::class]</code> por defecto
<code>Propagation</code> (7 modos)	<code>TransactionTemplate::execute()</code> es la única fuente de verdad para los siete, tanto para el proxy como para el llamador programático
<code>Isolation</code> / <code>readOnly</code>	Sentencias <code>SET TRANSACTION ...</code> de mejor esfuerzo, solo en la transacción más externa
<code>shouldRollBack()</code>	<code>noRollbackFor</code> gana sobre <code>rollbackFor</code> ; no coincidir con ninguna de las dos listas también confirma
<code>#[Transactional(timeout:)]</code>	Un plazo monótono tomado solo en la transacción más externa y juzgado cuando el trabajo retorna — pasarse revierte y lanza <code>TransactionTimedOutException</code> (504, <code>TRANSACTION_TIMED_OUT</code>); el atributo gana a <code>firefly.data.transaction.default-timeout</code> , <code>0</code> significa ninguno
<code>StatementTimeoutApplier</code>	La mitad que sí puede interrumpir una sentencia en curso: un <code>SET</code> de mejor esfuerzo por dialecto, emitido después del <code>BEGIN</code> y deshecho en el <code>finally</code> (las variables de mysql son de sesión); se apaga con <code>firefly.data.transaction.statement-timeout</code>

Concepto	Qué hace
<code>#[TransactionalEventListener]</code>	Aplaza el listener , no el evento: cuatro fases, <code>BEFORE_COMMIT</code> se ejecuta dentro de la transacción así que un throw desde ella aborta el commit; sin transacción activa el método se omite salvo que lleve <code>fallbackExecution: true</code>
El proxy generado	<code>{Target}__FireflyTransactionalProxy extends {Target} ;</code> enruta cada llamada a través de <code>TransactionInterceptor::run()</code> y luego <code>parent::</code>
Esquive por auto-invocación	<code>\$this->otro()</code> dentro de la clase proxificada se salta el interceptor por completo — usa el <code>TransactionTemplate</code> inyectado en su lugar
<code>DataAutoConfiguration::transactionalManifest() / proxyPlan()</code>	Artefacto compilado primero, escaneo en proceso después, vacío al final — la razón por la que ninguna aplicación necesita vincular un manifiesto a mano
La lección clave	<code>TransactionTemplate</code> es el único que llama a <code>dispatchAfterCommit()</code> — sin <code>#[Transactional]</code> , no hay publicación de eventos de dominio, en silencio
<code>Transfer</code>	Débito + crédito + ambos <code>save()</code> en una sola frontera — una rama de crédito fallida revierte también el débito ya ejecutado

Ponlo en práctica

1. **Reproduce la pérdida silenciosa de eventos.** En una copia de prueba del proyecto (no el paquete `samples/lumen` distribuido), quita `#[Transactional]` de una copia de `DepositHandler::handle()`, deposita en un monedero a través de la API HTTP, y confirma que el saldo *sí* se actualiza (la fila igual se escribe) mientras que el libro mayor (la tabla `ledger_entries` de `LedgerProjector`) **no recibe ninguna fila nueva en absoluto** — sin error en ningún lado.
2. **Demuestra que `NOT_SUPPORTED` no puede suspender.** Dale a un método `#[Transactional(propagation: Propagation::NOT_SUPPORTED)]`, llámalo desde dentro de otro método `#[Transactional(propagation: Propagation::REQUIRED)]` en la *misma* conexión (a través de `TransactionTemplate`, no por auto-invocación), y confirma — según la descripción "Known-latent" de este capítulo — que el trabajo interno igual corre dentro de la transacción externa en lugar de estar realmente fuera de una.
3. **Lee el código fuente del proxy generado.** Después de ejecutar `php artisan firefly:cache` en un proyecto con una clase `#[Transactional]`, abre el archivo emitido bajo `bootstrap/cache/firefly/proxies/` y busca los cuatro tipos de miembro que renderiza `ProxyClassGenerator`: la sobrescritura `(new \Firefly\Data\Proxy\MethodInvocation(...))->proceed()` que escribe para cada método aconsejado, la propiedad privada `$_fireflyTxInterceptor`, la fábrica privada estática `__fireflyTxDescriptor('m')` dentro de la cual se hornea el literal del descriptor, y la tabla pública estática `__fireflyAdvice()` que `ProxyFactory` lee para saber qué bean interceptor va en cada propiedad. Después dale a un *segundo* bean una regla `#[PreAuthorize]` y **ningún** `#[Transactional]`, vuelve a ejecutar `firefly:cache`, y confirma que el plan también lo nombró: recibe un proxy propio, con `$_fireflySecurityInterceptor` y

`__fireflySecurityDescriptor('m')` en su lugar. Por último pon ambos advices sobre un mismo método y lee el orden de la cadena directamente del fuente generado — el array de interceptores de la sobrescritura es `[$this->__fireflySecurityInterceptor, $this->__fireflyTxInterceptor]`, de fuera hacia dentro, exactamente el orden que dibuja la Figura 9.1.

4. **Haz saltar un tiempo límite, y luego quítale la mitad del driver.** Fija

`firefly.data.transaction.default-timeout` a `1`, dale a un método `#[Transactional]` un `sleep(2)` entre un `save()` y su retorno, y confirma que obtienes un `504` con código de error `TRANSACTION_TIMED_OUT` y **ninguna** fila. Después fija `firefly.data.transaction.statement-timeout` a `false` y ejecútalo otra vez: el mismo fallo, solo con el reloj de pared. Por último sube el presupuesto con `#[Transactional(timeout: 10)]` sobre el método y confirma que el atributo gana al valor configurado por defecto — y que llamar al método desde *dentro* de otro método `#[Transactional]` no le da plazo propio alguno.

5. **Veta un commit desde BEFORE_COMMIT.** Declara un

`#[TransactionalEventListener(phase: TransactionPhase::BEFORE_COMMIT)]` que lance cuando un evento lleve algún valor centinela, publica ese evento desde dentro de un método `#[Transactional]`, y confirma que toda la unidad de trabajo se revierte. Después cambia la fase a `AFTER_COMMIT` y confirma que el mismo throw deja ahora la fila confirmada. Por último, llama al mismo método publicador sin ninguna transacción y confirma que el listener se omite en silencio — luego añade `fallbackExecution: true` y míralo ejecutarse de inmediato.

Seguridad: Autenticación y Autorización

Al terminar este capítulo conocerás el modelo de principal inmutable de `firefly/security` (`Authentication`, `SecurityContext`, `SecurityContextHolder`), cómo `DaoAuthenticationProvider` autentica un par de usuario/contraseña mientras neutraliza ataques de enumeración, cómo `JwtService` se niega a arrancar con un secreto débil y se niega a aceptar un token sin caducidad, el DSL de URL `HttpSecurity` que deniega por defecto, cómo se identifica de verdad una persona a través de `SecurityContextPersistenceFilter` y `FormLoginFilter` — incluida la página de login que el framework entrega y la que se niega a reemplazar —, cómo funcionan cerrar la sesión y seguir dentro, y por qué una petición con *recuérdame* queda *marcada* en lugar de recién autenticada, cómo `firefly.security.http.entry_point` decide entre una redirección al login y un 401 pelado, cómo se inyecta el principal en la firma de un controlador, la familia de eventos de seguridad que publican un inicio de sesión, un cierre de sesión y una denegación, y — la pieza central de este capítulo — exactamente cómo se evalúa `#[PreAuthorize]` mediante una gramática de expresiones hecha a mano, de **lista blanca cerrada**, que nunca llama a `eval()`, que normaliza `hasRole('X')` a una comprobación contra la autoridad concedida `'ROLE_X'`, que se hace cumplir en el bus de CQRS que conociste en el Capítulo 7, y que ahora rige sobre *cualquier* bean estereotipado a través del proxy del Capítulo 9 y no solo en el bus.

❗ TÉRMINO NUEVO: AUTENTICACIÓN FRENTE A AUTORIZACIÓN

La **autenticación** responde "¿quién está haciendo esta petición?" — produce un principal. La **autorización** responde "¿le está permitido a ese principal hacer *esto en concreto*?" `firefly/security` mantiene ambas estrictamente separadas:

`Authentication / SecurityContextHolder` llevan la respuesta a la primera pregunta; `HttpSecurity`, `#[PreAuthorize]` y `AuthorizationChecker` responden todos a la segunda, evaluando una expresión **contra** lo que ya haya resuelto la primera pregunta.

El modelo de principal

`Authentication` es un token inmutable construido exclusivamente mediante dos fábricas con nombre, de modo que los dos estados en los que puede estar — una petición no autenticada que aún lleva credenciales en bruto, y un principal autenticado que lleva autoridades concedidas — nunca puedan confundirse:

```
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);
    }
    // ...
}

```

El constructor privado significa que la única forma de construir uno es `authenticated()` (las credenciales siempre `null`, `authenticated` siempre `true`) o `unauthenticated()` (las autoridades siempre `[]`, `authenticated` siempre `false`) — no existe ningún camino que permita construir un token "autenticado" con credenciales en bruto todavía pegadas.

`eraseCredentials()` es el mismo argumento repetido una vez más, un nivel más abajo, y sus dos líneas merecen una lectura atenta. Devuelve una instancia **nueva** en vez de limpiar un campo, así que una referencia que otra persona ya sostenga jamás puede observar un token que se limpió y luego se repobló. Y no se detiene en sus propias `credentials`: a un principal que sea a su vez un `CredentialsContainer` — el `User` distribuido, cuyo `getPassword()` es el hash codificado — se le pide su propia copia libre de credenciales, porque un hash de contraseña alcanzable a través de `$authentication->getPrincipal()` es un hash de contraseña sobre el cable en cuanto algo serializa el contexto. Esta es la forma que

`SessionSecurityContextRepository` escribe en la sesión. Fíjate en lo que *no* llama: el gestor de autenticación, al tener éxito. El token que recibe un filtro sigue llevando la contraseña codificada del principal, porque la firma de la cookie de *recuérdame* se calcula a partir de ella — así que solo un **almacén** necesita jamás la copia borrada. `GrantedAuthority` / `SimpleGrantedAuthority` envuelven una cadena de autoridad desnuda (`'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` es una instantánea inmutable de la `Authentication` actual (`anonymous()` es el valor cero no autenticado), y `SecurityContextHolder` es donde una petición lo encuentra:

```
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();
    }
}

```

Es `static` a propósito — la misma forma que tiene el propio `SecurityContextHolder` de Spring — porque es un accesor local a la petición, no un colaborador inyectado, respaldado por la propia fachada `Context` de Laravel (por petición, seguro bajo Octane). `getContext()` nunca exige una comprobación de nulos en el sitio de la llamada: un contexto no establecido resuelve automáticamente a `SecurityContext::anonymous()`. La garantía que sostiene todo lo demás, contra la fuga del principal de una petición hacia la siguiente, es que cada filtro de autenticación llama a `clearContext()` en un bloque `finally` al salir — PHP siempre ejecuta un `finally`, así que ningún camino de excepción puede saltárselo.

Autenticar una petición: **DaoAuthenticationProvider**

DaoAuthenticationProvider comprueba un par de usuario/contraseña contra un **UserDetailsService**, y está escrito específicamente para neutralizar ataques de temporización por enumeración de nombres de usuario:

```
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());
}
}

```

Un nombre de usuario desconocido se hace deliberadamente **indistinguible** de una contraseña incorrecta: ante `UsernameNotFoundException`, el proveedor igualmente ejecuta una verificación real de contraseña — contra un hash ficticio fijo, precalculado en el momento de la construcción con el *mismo* codificador y el mismo factor de coste que usan los usuarios reales — antes de lanzar exactamente la misma `BadCredentialsException` genérica, con el mismo mensaje, que produce una contraseña genuinamente incorrecta. Ninguna diferencia de temporización, ninguna diferencia de contenido. Los fallos de estado de cuenta (`locked` / `disabled`) se comprueban solo **después** de una contraseña correcta — el orden recomendado por OWASP — de modo que un atacante sin la contraseña correcta no aprende nada sobre si la cuenta siquiera existe, y mucho menos sobre su estado. Cuando tiene éxito, el token devuelto se construye de nuevo mediante `Authentication::authenticated()`, que nunca arrastra consigo la contraseña en bruto.

JWT: caducidad obligatoria, arranque rechazado ante un secreto débil

`JwtService` hace cumplir dos invariantes de fallo seguro que un desarrollador nunca tiene que recordar por su cuenta:

```
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.');
```

La primera invariante se dispara en la **construcción**, no en el primer uso: un secreto más corto de 32 bytes, o uno de una lista fija de marcadores de posición comunes ('changeme', 'secret', 'password', una cadena vacía...), lanza `WeakSigningSecretException` — la aplicación se niega directamente a arrancar en vez de firmar tokens con una clave adivinable. La segunda se dispara en cada `decode()`: incluso una firma criptográficamente válida se rechaza sin más si el token no lleva un claim `exp`, de modo que una caducidad olvidada nunca puede producir un token eternamente válido. `firefly.security.jwt.enabled` y `firefly.security.oauth2.resource_server.enabled` son además **mutuamente excluyentes** — habilitar ambos se rechaza en el arranque, igualando el modelo de Spring de un único mecanismo de portador.

Reglas de URL que deniegan por defecto: `HttpSecurity`

`HttpSecurity` es un DSL fluido que construye una lista ordenada de reglas de URL, evaluada primera-coincidencia-gana por `HttpSecurityFilter`, con **ninguna coincidencia en absoluto denegando la petición**:

```

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;
}

```

Cada regla compila hacia la **exacta misma gramática de expresiones** que usa `#[PreAuthorize]` más abajo — `HttpSecurity` es un constructor que emite cadenas con la forma `permitAll()` / `hasRole('ADMIN')`, no un segundo motor de autorización. `HttpSecurityFilter` evalúa las reglas compiladas contra la ruta de la petición y, en la primera coincidencia de patrón, comprueba la expresión de la regla; una petición que no coincide con **ninguna** regla se deniega — de fallo seguro, no de apertura por defecto. Una denegación se renderiza como `401` cuando el contexto es anónimo (autentícate primero) y como `403` cuando está autenticado pero sin privilegios suficientes.

`requestMatcher()` es la única puerta por la que pasa toda regla — `anyRequest()` y `fromConfig()` la llaman las dos — y normaliza el patrón antes de guardarlo. El filtro compara con `Str::is($rule->pattern, $request->path())`, y el `path()` de Laravel nunca lleva barra inicial, así que `normalisePattern()` la quita: `/api/*` y `api/*` son la **misma** regla. Un `'/actuator/health'` escrito como lo escribe medio mundo — como lo deletrea cada ruta de un `RouteManifest` — es una regla viva, no la regla silenciosamente muerta que antes era, en la que la petición no coincidía con nada, la denegación por defecto la rechazaba y el operador leía un `401` en la única ruta que había abierto explícitamente. La ruta raíz es la única excepción que conserva su barra, porque `path()` responde `'/'` para ella y nunca `''`. Lo que la normalización **no** hace

es reescribir una plantilla de ruta: un patrón se compara contra `api/orders/7`, nunca contra `api/orders/{id}`, así que un patrón que lleva un marcador de posición sigue siendo una regla muerta que ninguna normalización puede rescatar — `api/orders/*` es la grafía que lo cubre.

`assertSafeValue()` rechaza cualquier valor de rol/autoridad que contenga una comilla simple, por una razón que importa muchísimo en cuanto leas la siguiente sección: una cadena de rol o autoridad legítima nunca contiene una, pero un valor que sí lo hiciera podría, si no, empalmar gramática adicional dentro del literal de expresión fijo en el que se interpola.

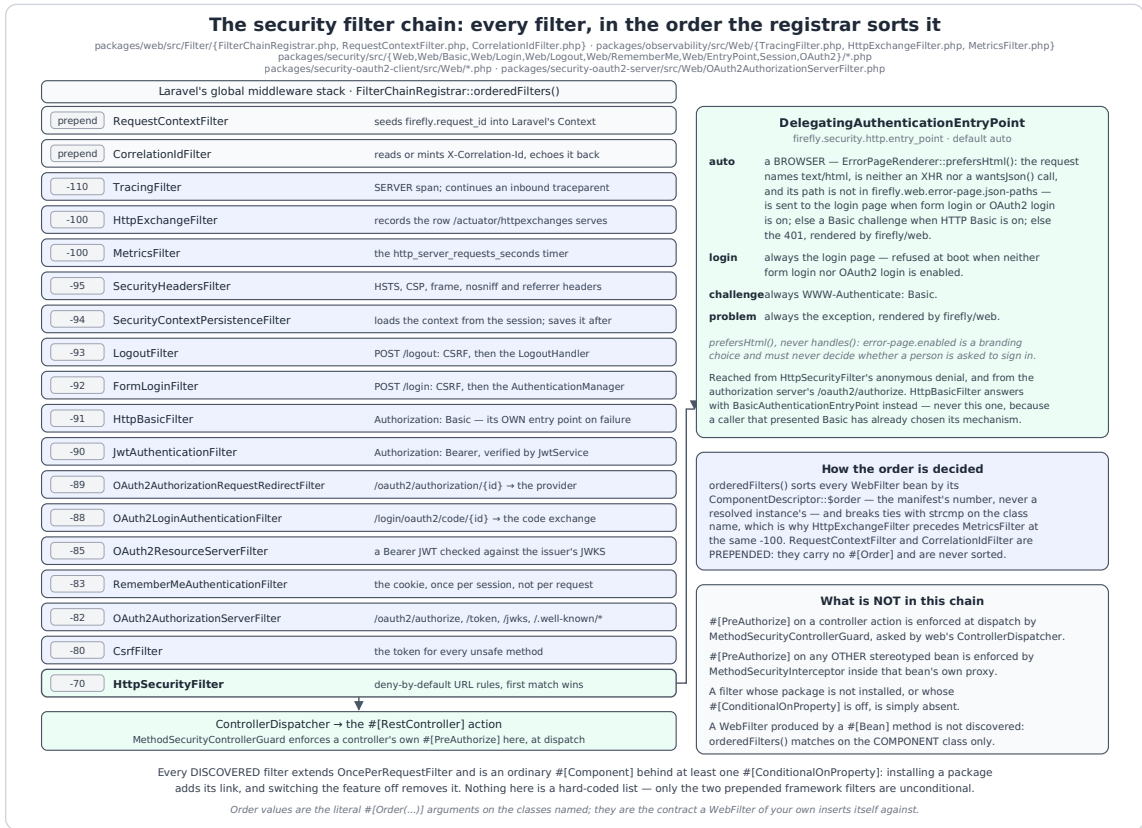


Figura 10.1 — HttpSecurityFilter es el último eslabón de una cadena ordenada: el valor #[Order] real de cada filtro, los dos filtros de framework antepuestos a todos ellos, y el DelegatingAuthenticationEntryPoint al que llega una denegación anónima — una redirección al login, un desafío Basic o un 401.

Tres de esos números hacen un solo trabajo entre los tres.

OAuth2AuthorizationRequestRedirectFilter (-89) inicia un inicio de sesión con un proveedor, OAuth2LoginAuthenticationFilter (-88) lo termina, y OAuth2AuthorizationServerFilter (-82) es lo que encuentra una petición cuando el proveedor eres tú. Los dos primeros son firefly/security-oauth2-client, el tercero es firefly/security-oauth2-server, y ninguno de los dos paquetes necesita al otro: cada mitad funciona contra cualquier contraparte conforme al otro lado.

La Figura 10.2 sigue un inicio de sesión a través de ambas mitades a la vez, porque lo que vale la pena entender del flujo de código de autorización no es ningún paso aislado, sino qué secreto tiene cada parte en qué momento. Empieza arriba a la izquierda con una regla que ya has escrito: `GET /orders` no coincide con ningún `permitAll()`, `HttpSecurityFilter (-70)` la deniega, y el punto de entrada hace exactamente lo que hace para el login por formulario — guarda el GET y redirige a `/login`. Lo que cambia es que la página de login lleva ahora un botón por registro, y ese botón va a `/oauth2/authorization/{id}`. A partir de ahí el navegador no vuelve a llevar ningún secreto: el código que trae de vuelta no vale nada sin el verificador PKCE, que nunca salió de la sesión de la parte confiante, y que se gasta en un `POST /oauth2/token` de canal trasero que ningún navegador llega a ver.

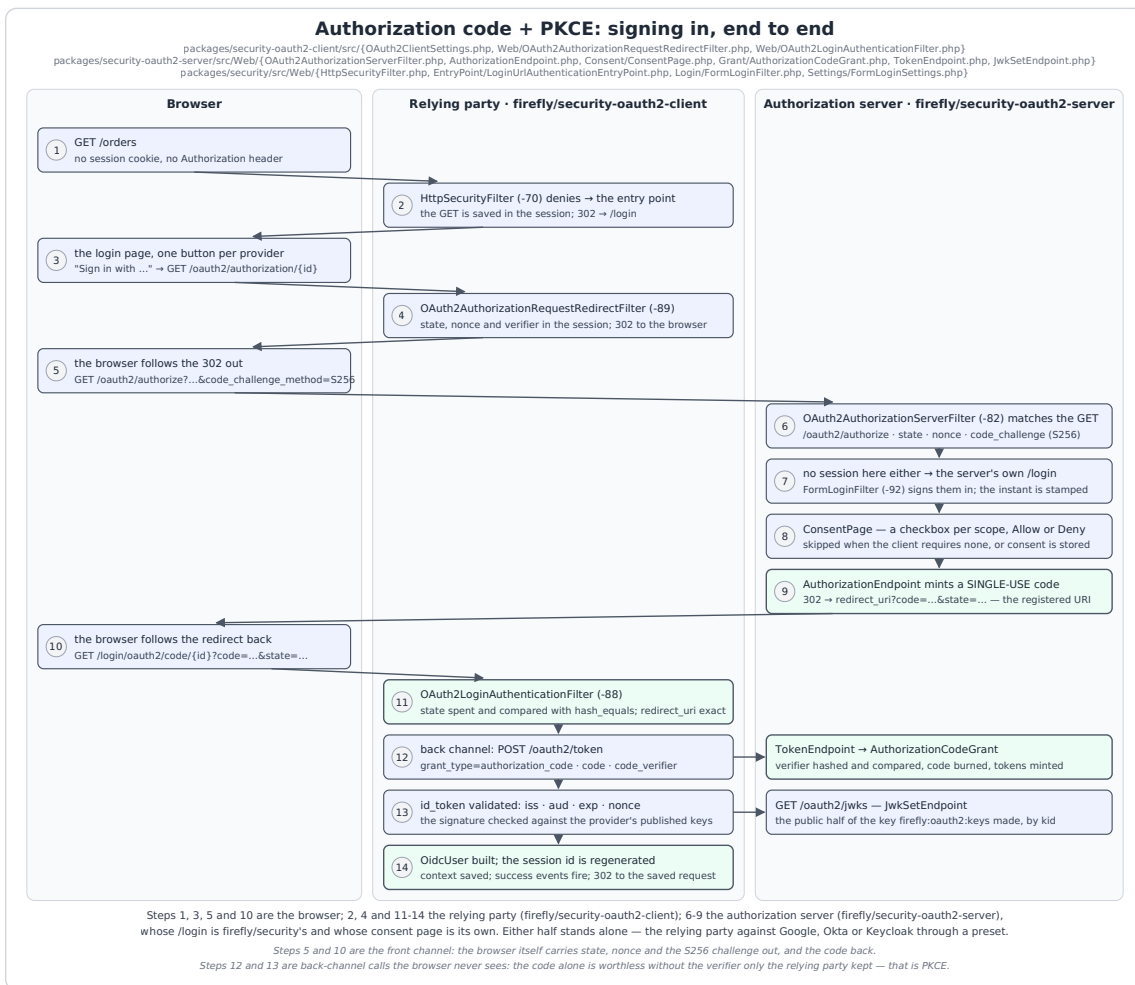


Figura 10.2 — Un inicio de sesión, de principio a fin: `HttpSecurityFilter` deniega la petición y el punto de entrada la guarda y redirige a `/login`, el botón de proveedor de la página de login arranca el viaje de ida y vuelta en `/oauth2/authorization/{id}` con un state de un solo uso, un nonce y un desafío de código S256, el servidor de autorización ejecuta sus propias páginas de login y consentimiento antes de acuñar un código de un solo uso, y la parte confiante gasta ese código y el verificador en un intercambio de tokens por canal trasero cuyo id token se verifica contra `/oauth2/jwks`.

Autenticar a una persona: la sesión, el formulario y la página

Todo lo visto hasta ahora en este capítulo autentica una petición que llega llevando su propia credencial — un JWT, una cabecera Basic. Esa es la forma correcta para una API y la equivocada para una persona, que inicia sesión una vez y luego navega. Tres filtros y una página hacen que eso funcione, y los tres se encienden con una sola clave.

El contexto que sobrevive a una petición

```
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` corre en `#[Order(-94)]` — justo después del filtro de cabeceras y **por delante de todos los filtros de autenticación**, así que un principal sostenido por la sesión es lo que ven `HttpSecurityFilter` y las guardas de seguridad de método. Veinte líneas, y cada una de ellas es deliberada:

- **No** carga cuando el portador ya lleva un contexto autenticado. El principal actuante de un test, o un middleware exterior, estableció uno a propósito.
- A la salida guarda solo un contexto que **cambió** durante la petición y que **sigue en el portador**. Nunca borra el almacenado: un filtro de autenticación interior que limpia el portador en su propio `finally` lo hace antes de que corra esta salida, y un portador vacío en ese punto significa «la petición ha terminado», no «cerrar sesión».
- La limpieza del `finally` es la garantía estructural para Octane. Pase lo que pase, nada se filtra a la petición siguiente en el mismo trabajador.

Lo que se almacena nunca lleva una credencial: un principal `CredentialsContainer` — el `User` distribuido lo es — se escribe sin su contraseña codificada.

⚠ LOS FILTROS DE FIREFLY SON GLOBALES; LA SESIÓN DE LARAVEL NO

Los filtros de Firefly son middleware global de kernel, y Laravel arranca la sesión en el grupo de rutas `web`, *más tarde*. Cuando la seguridad de sesión está encendida, un paso de arranque llamado `SessionSecurityBootstrap` empuja `EncryptCookies`, `AddQueuedCookiesToResponse` y `StartSession` a la pila **global** por delante de la cadena de filtros, los quita del grupo `web`, y los excluye en la ruta coincidente en tiempo de despacho. Sin eso, una ruta que llevara el grupo descifraría cookies ya descifradas, las anularía, y acuñaría una sesión nueva — perdiendo el inicio de sesión. Hace falta un driver de sesión; el arranque se niega sin uno.

El formulario

`firefly.security.form_login.enabled` enciende dos cosas a la vez.

`FormLoginFilter` (-92) atiende `POST /login` **antes del enrutado**, exactamente como hace el `UsernamePasswordAuthenticationFilter` de Spring — así que no necesita ruta, ni grupo CSRF, ni controlador. Su orden es: primero el token CSRF de sesión (un CSRF de login es un ataque, así que esta comprobación es independiente de `CsrfFilter`), luego el `AuthenticationManager` , luego, si tiene éxito, un identificador de sesión regenerado, el contexto almacenado, los eventos publicados, la cookie de recuérdame cuando se pidió, y una redirección a lo que el punto de entrada guardara.

El test capstone recorre esa vía entera, y es la descripción más clara del mecanismo que hay:

`$THIS->EVENTS` EN LOS LISTADOS DE ESTE CAPÍTULO

Cada listado de más abajo que lee `$this->events->interactive()` , `->successes()` , `->logouts()` o `->denials()` está sosteniendo un `Firefly\Testing\Double\RecordingAuthenticationEvents` — el doble de grabación consciente de la seguridad que cataloga el Capítulo 12, y que responde a los cinco tipos de evento por nombre en vez de obligar a un test a filtrar una lista mixta por clase. Reemplaza al puerto `ApplicationEventPublisher` y debe vincularse **antes del arranque**, desde `defineFireflyEnvironment()` ; el Capítulo 12 explica además el argumento `$forwardTo` que mantiene a los listeners reales oyendo el evento.

```
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']]);

```

El identificador de sesión cambia en cada inicio de sesión interactivo — protección contra fijación de sesión, y no es opcional. Un fallo publica un `AuthenticationFailure*Event` que lleva el nombre de usuario y la IP de origen y **nunca la contraseña**, y luego redirige a `/login?error`.

La página que el framework distribuye, y la que se niega a reemplazar

El framework renderiza su propia página de login, con el diseño de la página de error. Es una clase real, `LoginPage`, montada por un paso de arranque — y ese paso de arranque lleva la regla más considerada del paquete:

```
/**
// ...
* 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
```

Así que: si ya existe una ruta `GET` en la ruta de login configurada cuando el paso corre — una ruta por atributo, un fichero de rutas, Breeze, Fortify, lo que sea — **no se monta nada**. Todo lo demás sigue funcionando: `HttpSecurityFilter` permite la página por ruta (o la redirección entraría en bucle), el punto de entrada redirige a ella, y `FormLoginFilter` sigue respondiendo al POST antes del enrutado. Tu propio formulario solo necesita enviar el token de sesión a la URL de procesamiento del login.

Entre esos dos extremos hay una opción intermedia: apunta `view` a una plantilla Blade tuya y el framework renderizará esa en su lugar, entregándole un `LoginPageModel` con los campos del formulario y cada enlace de proveedor.

De dónde salen los usuarios

`AuthenticationManager` pregunta a un `UserDetailsService`, y el que la mayoría de aplicaciones quiere lee su propia tabla:

```
/**
 * // ...
 * 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
```

Una consulta por búsqueda, y la fila se mapea al objeto de valor inmutable `User` en vez de entregarse como el modelo — el principal acaba en la sesión y dentro de objetos de evento, y un modelo ahí arrastraría consigo su conexión, sus relaciones y sus atributos.

Un detalle vale más de lo que parece. Un nombre de columna configurado que la fila no lleva se **rechaza**, no se lee como nulo. El `Model::getAttribute()` de Laravel responde `null` para un atributo que una fila no tiene, y nulo es la respuesta equivocada en todas las posiciones aquí — para la columna de bloqueo es la peligrosa, porque `locked_column: 'is_locked'` contra una tabla cuya columna es `locked` dejaría cada fila desbloqueada y apagaría el bloqueo de cuentas sin error en ninguna parte. Renunciar a una columna opcional es la **cadena vacía**, que una errata nunca es.

La tabla `users` de serie de Laravel (`id`, `name`, `email`, `password`) funciona sin ningún cambio de esquema: pon `authorities` a `''`.

Cerrar sesión, y seguir dentro

Cerrar sesión

```
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) atiende `POST /logout` — **solo POST, con CSRF comprobado**; un `GET` cae hacia la ruta que haya ahí, que es lo que quieres, porque un enlace que cierra la sesión de la gente es un vector de CSRF. Expira la cookie de `recuérdame` y cada nombre listado en `delete_cookies`, invalida la sesión, publica `LogoutSuccessEvent`, y redirige.

Esa secuencia vive en un solo bean, `LogoutHandler`, y **cada camino que termina una sesión lo llama** — el filtro de arriba, y el cierre de sesión iniciado por la parte confiante de OpenID Connect cuando el paquete de servidor de autorización está encendido. Así que los tres ajustes que gobiernan el cierre de sesión gobiernan ambos por construcción; un segundo camino no puede obedecer calladamente un subconjunto de ellos.

Seguir dentro

`TokenBasedRememberMeServices` es el de Spring, incluida la firma:

```
username : expiry : HMAC-SHA256(username : expiry : password-hash, key)
```

Que el **hash de la contraseña** esté dentro de la firma es todo el diseño. Cambia la contraseña y cada cookie pendiente deja de verificar, en todas partes, sin ninguna tabla de tokens del lado del servidor que limpiar. La clave se sostiene con la misma regla que el secreto del JWT y el arranque rechaza una débil.

```
// 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();
$this->withCookie('remember-me', strrev($token))->get('/home')->assertRedirect('/
login');

```

`RememberMeAuthenticationFilter` (-83) solo corre cuando la sesión no sostiene ningún principal, y cuando tiene éxito realiza un **inicio de sesión interactivo completo**: un identificador de sesión nuevo, el contexto almacenado, un `InteractiveAuthenticationSuccessEvent`. Así que la cookie se consulta una vez por *sesión*, no una vez por petición. Una cookie manipulada es anónima en vez de un error — no hay nada que reportar, porque cualquiera puede enviar bytes.

Una cuenta deshabilitada o bloqueada se rechaza incluso con una cookie legítima, y el cierre de sesión la expira.

⚠ RECORDADO NO ES LO MISMO QUE AUTENTICADO

Un inicio de sesión por cookie está **marcado como recordado**. Es evidencia de que este navegador inició sesión en algún momento del pasado, no evidencia de que la persona esté ahora mismo ante el teclado. Spring traza exactamente esta línea, y es por lo que una página de «cambiar contraseña» o «borrar cuenta» debería exigir una autenticación fresca en vez de aceptar una recordada.

Redirección o 401: lo decide el punto de entrada

Una sola regla de denegación por defecto tiene que producir dos respuestas completamente distintas. Una persona que navega a `/orders` debería acabar en una página de login. Un cliente JavaScript que llama a `/api/orders` debería obtener un 401 sobre el que pueda actuar — y una página de login renderizada dentro de un `fetch()` es el modo de fallo más confuso que un framework web puede producir.

`HttpSecurityFilter` entrega cada denegación **no autenticada** al

`AuthenticationEntryPoint` que nombre `firefly.security.http.entry_point`:

Modo	Qué recibe una denegación anónima
<code>auto</code> (el de por defecto)	un navegador recibe la página de login con la petición guardada; una aplicación con Basic habilitado responde <code>401 + WWW-Authenticate: Basic</code> ; si no, un 401 para que lo renderice firefly/web
<code>login</code>	siempre la redirección a la página de login — rechazado en el arranque cuando no hay ningún login habilitado, porque no habría a dónde mandar a nadie
<code>challenge</code>	siempre <code>401 + WWW-Authenticate: Basic</code>
<code>problem</code>	siempre el 401, como <code>application/problem+json</code> o la página HTML de 401

«Un navegador» no es una conjetura sobre el `User-Agent`. Es la propia negociación de contenido de la página de error, `ErrorHandler::prefersHtml()`: la petición nombra `text/html` (o `application/xhtml+xml`), no es ni una `XMLHttpRequest` ni una llamada con `wantsJson()`, y no está bajo `firefly.web.error-page.json-paths`. La misma comprobación que usa la página de error, reutilizada en vez de reinventada — así que las dos nunca pueden discrepar:

```
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([]);
});
```

Lee la última línea. Una denegación **anónima** no publica nada — no es un evento de seguridad, es alguien que todavía no ha iniciado sesión, y un log lleno de ellas es ruido que esconde lo de verdad. Una denegación **autenticada** sí es lo de verdad: sigue siendo un 403 y publica `AuthorizationDeniedEvent` nombrando el principal, el sujeto y la expresión que lo rechazó:

```
$this->followSession($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')");
```

Dos llamadores alcanzan el punto de entrada: la denegación anónima de `HttpSecurityFilter`, y el `/oauth2/authorize` del servidor de autorización cuando necesita a la persona con la sesión iniciada. `HttpBasicFilter` deliberadamente **no** — responde con su propio `BasicAuthenticationEntryPoint` diga lo que diga `entry_point`, exactamente como el `BasicAuthenticationFilter` de Spring conserva el suyo. Un llamador que presentó credenciales Basic ya ha elegido su mecanismo, y redirigirlo a una página HTML es la respuesta equivocada para un cliente de API.

Seguridad de método: la gramática de expresiones de lista blanca cerrada

Esta es la sección hacia la que ha estado construyendo el resto del capítulo.

`#[PreAuthorize]`, `#[Secured]` y `#[RolesAllowed]` protegen todos un método con una expresión booleana:

```
#[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);
    }
}

```

Los `code:` y `message:` de `#[PreAuthorize]` son lo que lee el **cliente** cuando la expresión rechaza. Sin ellos la guarda responde `403 ACCESS_DENIED` con la frase propia del framework y las autoridades que la expresión nombró; con ellos responde el código de producto y la redacción de la aplicación. `#[Secured]` y `#[RolesAllowed]` no tienen hueco para palabras y usan siempre las del framework.

`#[Secured]` / `#[RolesAllowed]` son azúcar de conveniencia que `MethodSecurityScanner` compila hacia una expresión `hasAnyAuthority(...)` / `hasAnyRole(...)` — todo se reduce en última instancia a la única gramática que `#[PreAuthorize]` escribe directamente. Y esa gramática es deliberada, demostrablemente cerrada: `SecurityExpressionEvaluator` es un tokenizador hecho a mano más un analizador de descenso recursivo, y **nunca** llama a `eval()`, `create_function()` ni `call_user_func()` sobre nada derivado de la cadena de expresión. El despacho de funciones es un `match` fijo de antemano:

```

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}'."),
        };
    }
    // ...
}

```

Diez nombres de función, en total — `hasRole`, `hasAnyRole`, `hasAuthority`, `hasAnyAuthority`, `hasScope`, `hasAnyScope`, `hasPermission`, `isAuthenticated`, `permitAll`, `denyAll`, más las referencias `#param` — y no hay nada más alcanzable. Los mismos diez se escriben dos veces a propósito: el primer `match` es el modo de

solo análisis del tokenizador de lista blanca, el que `HttpSecurity` y `MethodSecurityScanner` usan para rechazar una expresión inválida en el ARRANQUE y no en la petición que da la casualidad de alcanzarla, y el segundo es la evaluación propiamente dicha. La gramática que acepta el tokenizador es booleana `and / or / not` (escrita `&& / || / !` a nivel de carácter), paréntesis, literales de cadena entre comillas simples y referencias a parámetros `#identificador` — y nada más:

```
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;
    }
    // ...
}
```

⚠ LO QUE NO ESTÁ EN LA GRAMÁTICA

No hay `==`, no hay `!=`, ningún operador de comparación de ningún tipo, y ninguna navegación `.propiedad`. No puedes escribir `#command.ownerId == authentication.name` — el tokenizador no tiene noción alguna de un acceso por punto ni de un operador de igualdad, así que ese texto o falla al tokenizar o falla al analizar, y en cualquiera de los dos casos `evaluate()` captura la `ExpressionParseException` y devuelve `false`. Esto no es un descuido; es lo que hace que la gramática sea demostrablemente cerrada. Si una comparación así parece justo lo que necesitas, la vía de escape prevista es `hasPermission(#target, 'action')` respaldada por un bean `PermissionEvaluator` a medida que hace la comparación de campo real en código PHP de verdad — la gramática despacha hacia tu propia lógica, nunca intenta convertirse ella misma en un lenguaje de expresiones de propósito general.

Todo modo de fallo — un nombre de función desconocido, un carácter suelto, un literal de cadena malformado, una excepción lanzada por un `PermissionEvaluator` a medida a mitad de la evaluación — se captura con el mismo `catch (\Throwable)` y se convierte en `false`. Una expresión hostil o rota solo puede denegar; nunca puede conceder por accidente.

`SecurityExpressionRoot` es el **único** objeto que las llamadas a función de una expresión pueden alcanzar jamás — no hay forma de llamar a ninguna otra cosa, porque no existe sintaxis de acceso a propiedad ni forma de obtener una referencia a ningún otro objeto:

```
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')` y `hasRole('ROLE_ADMIN')` son equivalentes — un nombre de rol desnudo se normaliza anteponiendo `ROLE_` si todavía no lo lleva, igualando exactamente la propia convención de Spring. `#param` — tokenizado como un `#` desnudo seguido de un identificador — se resuelve mediante `arg()` contra los propios nombres de los parámetros del método protegido, enlazados posicionalmente por `MethodSecurityScanner` en el momento del escaneo.

Aquí está toda la idea hecha concreta, en código real y distribuido —

`WithdrawHandler`, real, de `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;
}
}

```

Lee con atención el segundo párrafo de ese docblock — es toda la lección en miniatura. La forma *ingenua* de expresar "solo el propio dueño del monedero puede retirar" sería algo como `#command.ownerId == authentication.name`, y la gramática sencillamente no puede decir eso: no hay operador `.` ni `==`. La alternativa real, distribuida, es más burda pero genuinamente expresable: conceder una autoridad `WALLET_OWNER` a quien deba tener permiso, y comprobar `hasRole('WALLET_OWNER')`. Una comprobación de propiedad genuina por recurso, cuando la necesitas, pasa por `hasPermission(#command, 'withdraw')` y un bean `PermissionEvaluator` a medida que hace la comparación real en PHP ordinario — la gramática despacha hacia tu código, no intenta volverse lo bastante expresiva como para escribir ella misma la comparación.

Aplicación en el bus: cerrando el círculo del `AllowAllAuthorizer` del Capítulo 7

El Capítulo 7 te mostró la tubería de `CommandBus` / `QueryBus` ejecutando una etapa `authorize()` que por defecto usa `AllowAllAuthorizer` — permitir todo. `firefly/security` es lo que de verdad lo reemplaza, y lo hace solo cuando tú decides activarlo: `SecurityAutoConfiguration` está en `#[Order(500)]`, estrictamente por debajo del `#[Order(1000)]` de `CqrsAutoConfiguration`, así que — el mismo patrón de ganar-la-carrera que te mostró el Capítulo 8 para el outbox de Postgres en `#[Order(900)]` — sus beans se registran primero y satisfacen el propio `#[ConditionalOnMissingBean(CommandAuthorizer::class)]` de

`CqrsAutoConfiguration` antes de que el valor por defecto de permitir-todo tenga siquiera oportunidad de enlazarse, pero **solo** cuando

`firefly.security.enabled=true` (cada bean de seguridad aquí está además condicionado a ese indicador maestro):

```
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` es el punto de encuentro compartido al que delegan tanto `SecurityCommandAuthorizer` como su gemelo del lado de consulta

`SecurityQueryAuthorizer`: resuelve la clase + método manejador del mensaje a partir del **mismo** `HandlerManifest` que introdujo el Capítulo 7, busca cualquier regla `#[PreAuthorize]` / `#[Secured]` / `#[RolesAllowed]` compilada para ese

`Class::method`, y — solo si existe una — la evalúa contra el estado actual de `SecurityContextHolder`, con el propio mensaje enlazado al único `#param` de la regla:

```
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] : []);
}

```

Léelo como cuatro retornos tempranos y una línea de trabajo. Seguridad apagada, ningún manejador registrado para este mensaje, o ninguna regla compilada para ese manejador — cada caso pasa sin comprobación, porque la seguridad de método aquí es **aditiva**, nunca una segunda puerta que deniegue por defecto (ese trabajo pertenece por entero a `HttpSecurityFilter`). Cuando sí se encuentra una regla, la última línea se la entrega a `MethodSecurityEvaluator::before()`, el único evaluador que comparten todas las costuras de aplicación, con el propio mensaje vinculado al único `#param` de la regla.

Así que cuando `WithdrawHandler::handle()` lleva una regla, esto es lo que la ejecuta, **antes** de que `handle()` mismo llegue a ejecutarse. Una denegación se convierte en `AuthorizationException`, que `DefaultCommandBus::send()` (Capítulo 7) envuelve en `CommandProcessingException`, exactamente como cualquier otro fallo del lado del manejador.

Seguridad de método en todas partes, no solo en el bus

El bus fue el *primer* sitio donde estas reglas se hicieron cumplir. Ya no es el único, y la diferencia importa: un `#[PreAuthorize]` sobre un `#[Service]` corriente solía ser un comentario.

Ahora hay tres costuras de aplicación, y las tres comparten un único `MethodSecurityEvaluator`, así que no pueden discrepar sobre lo que significa una expresión:

Costura	Protege	Se ejecuta
El bus CQRS	un manejador de comando o consulta	antes de <code>handle()</code> , vía <code>SecurityCommandAuthorizer / SecurityQueryAuthorizer</code>
El despachador de controladores	una acción de controlador	<code>ControllerSecurityGuard:check()</code> antes, <code>afterInvocation()</code> después
La cadena de proxy	cualquier bean estereotipado — <code>#[Service]</code> , <code>#[Component]</code> , <code>#[Repository]</code>	como el interceptor más externo del proxy generado

La tercera es la nueva, y no es un mecanismo nuevo: es el proxy del Capítulo 9, con una segunda clase de advice encima. `MethodSecurityAdviceSource` aporta sus filas al mismo `ProxyPlan`, y declara en qué punto de la cadena corre su interceptor:

```
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);
}
```

Orden 100 , frente al 1000 del advice transaccional. Menor corre por fuera, así que la seguridad rechaza *antes* de que se abra transacción alguna — que es el único orden sensato, porque abrir y revertir una transacción para descubrir que el llamador nunca tuvo permiso es trabajo hecho para nada.

`inertWhenUnbound` es la otra mitad de la misma idea. `MethodSecurityInterceptor` existe como bean solo bajo `firefly.security.enabled` , y su ausencia es el estado documentado del framework de «las anotaciones son inertes hasta que la seguridad esté encendida» y no una mala configuración — así que el registro entrega al proxy un `PassThroughInterceptor` en vez de hacer fallar el arranque, que es lo que hace con cualquier *otro* advice cuyo interceptor falte.

Las reglas que necesitan el valor devuelto

Una vez que la costura puede correr *después* de la llamada, dos atributos más se vuelven posibles — y esto es exactamente lo que el bus nunca pudo hacer:

```
#[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]` corre después de `proceed()` con `#returnObject` vinculado a lo que el método devolvió. «Puedes llamar a esto, pero solo si lo que volvió es tuyo» es una regla que no puedes expresar antes de la llamada, porque la respuesta es lo que la decide.
- `#[PostFilter]` corre sobre un resultado iterable con `#filterObject` vinculado a cada elemento, conservando los que la expresión acepta. Los arrays conservan sus claves; un Enumerable se filtra en su misma especie; un retorno no iterable se rechaza en vez de dejarse pasar calladamente.
- `#[PreFilter]` es la imagen especular a la entrada, nombrando con `filterTarget:` qué parámetro filtra.

Fíjate en `code:` y `message:` sobre el `#[PostAuthorize]`. El código de producto de un rechazo y la frase que lee una persona viven en la misma línea que la regla que los produjo, junto al método que protegen — en vez de en algún servicio que quien llama tenga que acordarse de invocar primero. Sin ellos se aplica la redacción propia del

framework: un 403 que dice «*You do not have permission to do this.*», con las autoridades que la regla pedía transportadas como miembro de extensión del RFC 9457, y la clase y el método protegidos enviados al **log** en vez de al cable, porque el nombre de una clase PHP no es algo que un desconocido deba leer en un panel.

⚠ DOS RECHAZOS DE ARRANQUE QUE QUIERES, Y UNO AL QUE TIENES QUE APUNTARTE

Un bean `final` que lleva reglas que solo un proxy podría hacer cumplir se rechaza en tiempo de escaneo — el proxy no puede extenderlo, así que compilar la fila produciría una regla que nada ejecuta. Y `SecurityWiringPass` se niega a arrancar sobre un `security-methods.php` escrito sin un `proxy-plan.php` al lado: las reglas compilarían y las de nivel de bean no se harían cumplir en silencio. El que tienes que pedir es `firefly.security.method.strict`, que convierte «no se encontró artefacto compilado» de un recurso de reserva en un fallo de arranque. Ponlo en cualquier imagen que ejecute `firefly:cache`; es la única defensa contra una construcción que se distribuya sin el manifiesto.

La seguridad de método sigue siendo **aditiva**. «Ninguna regla registrada para este método» significa permitir, en cada una de las tres costuras — denegar por defecto es tarea de `HttpSecurityFilter` y de nadie más. Esa es la semántica correcta y tiene un filo, que es para lo que existe `strict`: un manifiesto vacío es indistinguible de una aplicación que no declara ninguna regla.

El principal, inyectado

Un controlador que necesita saber quién llama debería decirlo en su firma. No debería meter la mano en un portador, y desde luego no debería recibir un `principal` que una cadena de consulta pudiera fijar.

```
#[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(),
    ];
}
```

Cuatro parámetros, cuatro formas de preguntar, y `firefly/security` vincula todos ellos antes de que se consulte siquiera al contenedor — gracias a un puerto que `firefly/web` expone exactamente para esto:

```
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;
}
```

Un resolutor ve el plan de vinculación compilado — el nombre del parámetro, su clase, su tipo, sus atributos y su nulabilidad — y lo reclama con `supports()`.

`ArgumentResolver` pregunta a los resolutores registrados **antes** que a sus propias

clases, así que un parámetro de tipo clase que un resolutor entiende nunca llega a

`$container->make()`. `SecurityArgumentResolver` reclama `Authentication`, `UserDetails`, `#[AuthenticationPrincipal]` y `#[CurrentSecurityContext]`.

Las reglas de nulabilidad merecen enunciarse con precisión, porque son las que permiten que una sola acción sirva a dos mecanismos:

Declaración	Con sesión iniciada	Anónimo
<code>Authentication \$auth</code>	la autenticación	401
<code>?Authentication \$auth</code>	la autenticación	<code>null</code>
<code>?UserDetails \$user</code>	el principal cuando <i>lo</i> es	<code>null</code>
<code>#[AuthenticationPrincipal] mixed \$principal</code>	lo que sea el principal	el principal anónimo
<code>#[CurrentSecurityContext] SecurityContext \$context</code>	el contexto	el contexto anónimo

Un principal **con atributo** se entrega solo cuando es lo que el parámetro declara, y `null` en caso contrario. Eso es lo que evita un `TypeError` — el principal de un JWT es su **cadena sub**, no un `UserDetails` — y es lo que permite que la misma acción sirva a un login por formulario y a un token portador:

```
// 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]);
```

Dos hechos más, ambos defensivos. El resolutor se registra **esté o no encendido el interruptor maestro**, así que con la seguridad apagada las anotaciones son inertes — un `null`, un contexto anónimo, un 401 honesto — en vez de que el plan de vinculación ordinario las malinterprete como parámetros de consulta. Y nada de esto lee jamás un principal de la petición:

```
it('never reads a principal from the query string, whoever asks', function () {
  // ...
  $this->getJson('/open/whoami?principal=admin')->assertOk()->assertJson(['principal' =>
  null]);
});
```

Los eventos

Cada mecanismo de este capítulo reporta lo que hizo, a través de `AuthenticationEventPublisher` sobre el `ApplicationEventPublisher` ordinario del contexto. Así que un método `#[AsEventListener]` los recibe exactamente igual que cualquier otro evento de aplicación del Capítulo 8, y `Event::fake()` los ve en un test.

La familia es pequeña y completa — estas son todas las clases de `packages/security/src/Event/`, y no hay otras:

Evento	Cuándo se publica
<code>AuthenticationSuccessEvent</code>	cualquier autenticación con éxito, interactiva o no
<code>InteractiveAuthenticationSuccessEvent</code>	una persona inició sesión: <code>form</code> , <code>basic</code> o <code>remember-me</code>
<code>AuthenticationFailureBadCredentialsEvent</code>	una contraseña equivocada — y un nombre de usuario desconocido
<code>AuthenticationFailureLockedEvent</code>	las credenciales eran correctas, la cuenta está bloqueada
<code>AuthenticationFailureDisabledEvent</code>	las credenciales eran correctas, la cuenta está deshabilitada
<code>LogoutSuccessEvent</code>	una sesión terminó a través de <code>LogoutHandler</code>
<code>AuthorizationDeniedEvent</code>	una regla de URL o de método rechazó a un principal autenticado

Tres de esas filas llevan una decisión en vez de un hecho.

Un usuario desconocido se reporta como credenciales incorrectas, a propósito.

Un evento distinto de «no existe tal usuario» sería un oráculo de nombres de usuario, y todo el sentido de que `DaoAuthenticationProvider` iguale los dos caminos — el mismo tiempo, la misma respuesta — lo desharía justamente el listener escrito para vigilarlo.

Una cuenta bloqueada o deshabilitada solo se reporta tras la contraseña correcta. Si no, los eventos le dirían a un atacante qué cuentas existen, que es el mismo oráculo por una vía más lenta.

AuthorizationDeniedEvent se dispara solo para alguien con la sesión iniciada.

Una denegación anónima es alguien que todavía no ha entrado, y no se publica en ninguna parte; una denegación autenticada es una persona que intentó algo que no tiene permitido, y llega con el principal, el sujeto (`GET /admin/users` , o `App\Reports::totals`) y la expresión que la rechazó.

Lo que ninguno de ellos lleva es una credencial. Un evento de fallo nombra el nombre de usuario y la IP de origen; la contraseña no está en él, y la clase base abstracta que comparten los tres fallos no tiene hueco para una.

💡 LOS TRES LISTENERS QUE MERECE LA PENA ESCRIBIR EL PRIMER DÍA

Cuenta `AuthenticationFailureBadCredentialsEvent` por IP de origen y tienes detección de fuerza bruta. Registra `InteractiveAuthenticationSuccessEvent` con su `mechanism` y tienes una auditoría de inicios de sesión que distingue una contraseña de una cookie. Alerta sobre `AuthorizationDeniedEvent` y estarás vigilando la única denegación que significa que alguien que *sí* es conocido intentó algo que no debía.

Prueba de extremo a extremo

Dos archivos de prueba reales y distribuidos ejercitan la cadena completa — petición HTTP de entrada, `SecurityCommandAuthorizer` haciendo cumplir el `#[PreAuthorize]` de `WithdrawHandler` en el bus, salida como una respuesta RFC-9457. Y lo primero que demuestran es que **una denegación no es una respuesta, sino dos.**

`samples/lumen/tests/Web/WalletRestTest.php` , sin ningún principal establecido en absoluto:

```

    $this->postJson("/api/v1/wallets/{$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, no 403 — y la distinción es justo el punto. `MethodSecurityEvaluator` responde a un llamante **anónimo** con una `AuthenticationException`: *autentícate primero*. Reserva el 403 para un llamante que *sí* ha iniciado sesión y aun así carece de la autoridad, que es la siguiente prueba del mismo archivo:

```

    $this->postJson("/api/v1/wallets/{$id}/withdraw", ['amount_minor' => 1000])
    ->assertStatus(403)
    ->assertHeader('Content-Type', 'application/problem+json')
    ->assertJsonPath('status', 403)
    ->assertJsonPath('code', 'ACCESS_DENIED')
    ->assertJsonPath('category', 'security');

```

Las tres costuras de aplicación dan el mismo par de respuestas, porque comparten el mismo evaluador: 401 significa *iniciar sesión ayudaría*, 403 significa *no ayudaría*. Un cliente puede distinguirlas y actuar en consecuencia.

El bus envuelve cualquiera de las dos en una `CommandProcessingException`, que copia el código de error, el estado HTTP y la categoría de la causa — así que el cable lleva `AUTHENTICATION_FAILED` o `ACCESS_DENIED` y no el código genérico del bus. Y en las dos pruebas la aserción sobre el saldo, después, hace trabajo real: demuestra que el débito genuinamente nunca ocurrió, no solo que la respuesta HTTP tuviera apariencia de rechazo.

Concede la autoridad correcta y el mismo comando tiene éxito:

```
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');
```

La autoridad concedida es `'ROLE_WALLET_OWNER'` — ya con el prefijo `ROLE_` incorporado — y la expresión de `WithdrawHandler` llama a `hasRole('WALLET_OWNER')`, que `SecurityExpressionRoot::hasRole()` normaliza a la cadena idéntica `'ROLE_WALLET_OWNER'` antes de comprobar la pertenencia; las dos formas de escribirlo se encuentran en el medio. El `409` de la segunda llamada es el contraste importante: la autorización y la validación de dominio son dos capas genuinamente independientes — que se te *permita* retirar no dice nada sobre si la retirada en sí es *válida*, y la propia invariante de no-sobregiro de `Wallet::withdraw()` (Capítulo 6) sigue aplicándose exactamente igual que antes.

`samples/lumen/tests/Application/TransferSecurityTest.php` demuestra el mismo guardián directamente contra el `CommandBus`, con el patrón de desenvolver mediante `getPrevious()` que introdujo el Capítulo 7:

```
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()` alcanza, más allá de `CommandProcessingException`, la `AuthorizationException` real que lanzó `MethodSecurityMessageEnforcer` — el mismo patrón de desenvolverse que te mostró el Capítulo 7, ahora con su uso concreto en seguridad.

📌 PARIDAD CON LARAVEL

Laravel puro autoriza con Gates/Políticas y una llamada `$this->authorize()` dentro de un controlador — imperativo, y solo alcanzable desde código que se acuerde de llamarlo. `#[PreAuthorize]` se descubre por atributo y se hace cumplir automáticamente en el bus para cada despacho de un comando o consulta protegido, sin ningún sitio de llamada capaz de olvidarlo; `AuthorizationChecker::check()` (respaldado por el mismo evaluador) es el equivalente más cercano en LaraFly a una comprobación de Gate, para el caso más raro en que necesitas probar una expresión de forma imperativa en un sitio de llamada arbitrario.

✓ Lo que aprendiste

Concepto	Qué hace
<code>Authentication</code>	Inmutable; las fábricas <code>authenticated()</code> / <code>unauthenticated()</code> hacen imposible confundir los dos estados
<code>SecurityContextHolder</code>	Contenedor estático con alcance de petición; cada filtro de autenticación lo limpia en un <code>finally</code>
<code>DaoAuthenticationProvider</code>	Comprobación de usuario/contraseña equivalente en tiempo, con mitigación de enumeración
<code>JwtService</code>	Se niega a arrancar con un secreto débil/ marcador de posición; rechaza un token sin claim <code>exp</code>
<code>HttpSecurity</code>	Reglas de URL que deniegan por defecto, primera-coincidencia-gana, compilando hacia la misma gramática de expresiones que la seguridad de método
<code>SecurityExpressionEvaluator</code>	Un tokenizador/analizador hecho a mano, de lista blanca cerrada — sin <code>eval</code> , sin <code>==</code> , sin navegación <code>.propiedad</code> ; diez funciones en 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>) más las referencias <code>#param</code>
<code>hasRole('X')</code>	Normaliza a <code>hasAuthority('ROLE_X')</code> — las formas desnuda y con prefijo <code>ROLE_</code> son equivalentes

Concepto	Qué hace
<code>SecurityCommandAuthorizer</code> / <code>SecurityQueryAuthorizer</code>	Las implementaciones reales en <code>#[Order(500)]</code> que reemplazan al <code>AllowAllAuthorizer</code> del Capítulo 7, solo cuando <code>firefly.security.enabled=true</code>
<code>MethodSecurityMessageEnforcer</code>	Une el <code>HandlerManifest</code> de CQRS con el <code>SecurityMethodManifest</code> compilado; hace cumplir la regla antes de que <code>handle()</code> llegue a ejecutarse
<code>SecurityContextPersistenceFilter</code>	Carga el contexto desde la sesión en -94, guarda lo que cambió, y limpia el portador en un <code>finally</code>
<code>FormLoginFilter</code> / <code>LoginRouteRegistrar</code>	<code>POST /login</code> atendido antes del enrutado; la página del framework se monta solo cuando nadie más reclama esa dirección
<code>EloquentUserDetailsService</code>	Tu propia tabla de usuarios como almacén — y una columna configurada que la fila no tiene se rechaza, nunca se lee como <code>null</code>
<code>LogoutFilter</code> / <code>TokenBasedRememberMeServices</code>	Cierre de sesión solo por POST y con CSRF comprobado; una cookie firmada sobre el hash de la contraseña , así que un cambio de contraseña revoca todas
<code>AuthenticationEntryPoint</code>	Los cuatro modos de <code>entry_point</code> ; <code>ErrorPageRenderer::prefersHtml()</code> — la petición nombra <code>text/html</code> o <code>application/xhtml+xml</code> , no es una XHR y no está bajo <code>json-paths</code> — decide página de login frente a 401

Concepto	Qué hace
<code>MethodSecurityAdviceSource</code>	Seguridad de método sobre cualquier bean estereotipado, en el orden de advice 100 — por fuera de la transacción en 1000
<code>#[PostAuthorize]</code> / <code>#[PostFilter]</code>	Reglas que necesitan el valor devuelto: <code>#returnObject</code> después de la llamada, <code>#filterObject</code> por cada elemento
<code>SecurityArgumentResolver</code>	<code>Authentication</code> , <code>UserDetails</code> , <code>#[AuthenticationPrincipal]</code> , <code>#[CurrentSecurityContext]</code> vinculados antes de que se le pregunte al contenedor
La familia de eventos	Siete eventos; un usuario desconocido se reporta como credenciales incorrectas, y solo se publica una denegación autenticada

Ponlo en práctica

1. Escribe una expresión rechazada. Intenta compilar

`#[PreAuthorize("#command.ownerId == authentication.name")]` en un manejador de prueba y ejecuta `php artisan firefly:cache`. Confirma que `MethodSecurityScanner` falla de forma ruidosa con una `ConfigurationException` que nombra la clase y el método exactos, en vez de aceptar silenciosamente una expresión que siempre denegaría (o fallaría al analizar) en tiempo de ejecución.

2. Añade una comprobación real de propiedad. En una copia de prueba del proyecto, implementa un bean `PermissionEvaluator` que cargue un monedero por id y compare su `owner_id` con `Authentication::getName()`, enlázalo para que reemplace a `DenyAllPermissionEvaluator`, y cambia la expresión de

- `WithdrawHandler` a `hasPermission(#command, 'withdraw')`. Confirma que un principal sin `WALLET_OWNER` que genuinamente sea dueño del monedero ahora puede retirar, mientras que un dueño distinto no puede.
- 3. Adueñate de la página de login.** Añade una acción `#[GetMapping('/login')]` propia a un proyecto de prueba con `form_login.enabled` encendido, arráncalo, y confirma que la página del framework *no* reemplaza a la tuya — después confirma que tu formulario sigue dando acceso a la gente enviando el token de sesión a la URL de procesamiento, porque `FormLoginFilter` responde antes del enrutado de cualquier modo.
 - 4. Mira al punto de entrada cambiar de respuesta.** Contra una ruta protegida, envía `Accept: text/html` y después `Accept: application/json`, y confirma que obtienes una redirección a `/login` y un `401 application/problem+json` respectivamente. Después pon `firefly.security.http.entry_point` a `problem` y confirma que la petición del navegador también recibe el 401.
 - 5. Pon una regla sobre un servicio corriente.** Añade `#[PreAuthorize("hasRole('ADMIN')")]` a un método de un `#[Service]` no final que no sea ni controlador ni manejador, ejecuta `php artisan firefly:cache`, y confirma que la llamada se rechaza. Después marca la clase como `final`, vuelve a ejecutar la caché, y confirma que el escáner rechaza en tiempo de compilación en vez de distribuir una regla que nada podría hacer cumplir.
 - 6. Reproduce la temporización de la mitigación de enumeración.** Mide el tiempo de `DaoAuthenticationProvider::authenticate()` para un nombre de usuario conocido con una contraseña incorrecta frente a un nombre de usuario desconocido, con suficientes iteraciones para suavizar el ruido, y confirma que ambos son estadísticamente indistinguibles — demostrando que la verificación con hash ficticio que describió este capítulo realmente hace su trabajo, y no solo está documentada como si lo hiciera.

OAuth2 y OpenID Connect: Iniciar Sesión, y Ser el Proveedor

Al terminar este capítulo sabrás cuál de los dos paquetes OAuth2 tienes entre manos y por qué son dos — `firefly/security-oauth2-client` convierte tu aplicación en un **relying party** que identifica a las personas a través de Google, Keycloak, Okta, Entra o cualquier proveedor conforme, y `firefly/security-oauth2-server` la convierte en un **servidor de autorización** que emite los tokens que consume otro. Sabrás qué dos filtros añade la mitad cliente en `-89` y `-88`, sobre qué dos URIs base responden, y qué mete el framework en una petición de autorización en lo que el login por formulario del Capítulo 10 nunca tuvo que pensar: un `state` de un solo uso, un `nonce` y un desafío PKCE S256. Sabrás qué es un `OIDcUser` cuando llega a tu controlador, cómo un claim `groups` se convierte en un `ROLE_`, cómo el cierre de sesión alcanza también la sesión del proveedor, y cómo un job en cola llama a una API como la aplicación sin que haya ninguna persona de por medio. Después el cuadro se da la vuelta: el mismo viaje de ida y vuelta, desde el lado que acuña el código — la página de consentimiento, el endpoint de token, la rotación de refresh tokens con detección de reutilización, la introspección, la revocación, userinfo, el JWKS, y el único comando que genera la clave de la que todo cuelga.

❗ TÉRMINO NUEVO: RELYING PARTY

En OpenID Connect la aplicación que envía a una persona a un proveedor para que la identifique es el **relying party** (el RP) — se *apoya* en la afirmación de otro sobre quién es esa persona, en lugar de guardar ella misma la contraseña. El proveedor es el **OpenID Provider** o, en términos de OAuth2 puro, el **servidor de autorización**. Los dos paquetes de este capítulo son esos dos papeles, y las palabras importan porque un mismo despliegue es habitualmente los dos: un portal interno que identifica a su plantilla contra el SSO corporativo y emite tokens para la aplicación móvil que le pertenece.

Dos paquetes, dos direcciones

La biblioteca principal incluye ambos componentes OAuth2. Cada uno se apoya en `firefly/security`; sus descriptores internos no dependen el uno del otro. La configuración activa cada rol de forma independiente. Estos requisitos opcionales de componentes se satisfacen mediante el `replace` de la biblioteca ya instalada:

```
composer require firefly/security-oauth2-client # ser un relying party: identificar
personas en un proveedor
composer require firefly/security-oauth2-server # ser el proveedor: emitir tú mismo los
tokens
```

No pesan lo mismo, eso sí, y los dos bloques `require` lo dicen sin rodeos. `firefly/security-oauth2-client` nombra siete paquetes del framework — `autoconfigure`, `config`, `container`, `context`, `kernel`, `security` y `web` — y `firefly/security-oauth2-server` nombra once paquetes del framework: esos mismos siete más `actuator`, `data`, `resilience` y `scheduling`, uno por cada cosa que el rol de proveedor posee y que un relying party nunca tiene. `firefly/data`, porque los clientes registrados, las autorizaciones y los consentimientos son stores de Eloquent

construidos sobre `EloquentRepository`; `firefly/actuator`, porque el servidor aporta un endpoint que lista los clientes que tiene registrados; `firefly/resilience`, porque el endpoint de token está limitado por tasa; y `firefly/scheduling`, porque las autorizaciones caducadas se purgan en un cron (`*/15 * * * *` por defecto) que el despliegue puede cambiar. Lo que sí es igual en ambos bloques es la parte que importa aquí: `firefly/security` es el único paquete de *seguridad* que nombra cualquiera de los dos — el tercer rol de OAuth2, el resource server, vive dentro del propio `firefly/security` — y ninguno de los dos bloques nombra al otro paquete.

`firefly/security-oauth2-client` aporta `OAuth2AuthorizationRequestRedirectFilter` (-89) y `OAuth2LoginAuthenticationFilter` (-88), que se sitúan entre el filtro JWT y el filtro de servidor de recursos de la cadena que ya conoces. `firefly/security-oauth2-server` aporta exactamente uno, `OAuth2AuthorizationServerFilter` (-82), que responde **todos** los endpoints que el servidor posee. Las dos mitades vienen apagadas, clave por clave, y las dos se configuran bajo `firefly.security.oauth2.*` — el mismo bloque que el Capítulo 10 usó para `resource_server`, que es el tercer papel de OAuth2 y el único que solo *verifica* un token acuñado por otro.

Una combinación se rechaza de plano en el arranque, y merece la pena leer el rechazo en lugar de su resumen, porque la razón es un orden de filtro y ahora sabes leer uno:

```
/**
 * 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) corre antes que `OAuth2AuthorizationServerFilter` (-82), y un cliente OAuth2 que se autentica con `client_secret_basic` envía exactamente la cabecera que el filtro Basic fue construido para consumir. Con los dos interruptores encendidos, cada cliente confidencial de tu despliegue recibiría un `401` como login de *usuario* fallido — con su evento de fallo publicado — antes de que el filtro del servidor de autorización llegara siquiera a ver la credencial. Nada de ese fallo mencionaría OAuth2. Así que el arranque rechaza la pareja y nombra las dos claves, que es la regla permanente del framework ante una combinación que solo podría producir un error de ejecución desconcertante. La misma estática rechaza tres más: el servidor sin `firefly.security.enabled`, el servidor sin nada que lleve un principal entre peticiones, y el servidor junto a `firefly.security.jwt.enabled` (cuyo filtro HMAC en -90 rechazaría cada token RS256 que este servidor emite).

Iniciar sesión con un proveedor

Todo lo de esta sección es la mitad relying party. Su forma es la de Spring Boot, clave por clave, y la referencia distribuida es donde viven los valores por defecto:

```
'client' => [
  'enabled' => env('FIREFLY_OAUTH2_CLIENT_ENABLED', false),
]

'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,
    // ],
  ],
],

```

Dos bloques comentados llevan todo el modelo. Una **registration** es una relación con un proveedor: un client id, un secreto, los scopes que pedir, el grant que usar. Un **provider** son los endpoints de un emisor. Una registration nombra a su provider (y cuando no lo hace, su propio id se toma como nombre del provider), y cinco nombres son **presets** — `google`, `github`, `okta`, `keycloak` y `microsoft` (alias `entra`) — cuyos endpoints, scopes por defecto y `client_name` vienen integrados, y — solo para `google` y `github` — el método de autenticación de cliente, porque ninguno de los dos proveedores emite un cliente web sin secreto. En los tres presets por inquilino el método se deduce en su lugar: `client_authentication_method` cuando la registration lo fija, `client_secret_basic` cuando lleva secreto, y `none` — un cliente público, PKCE obligatorio — cuando no lo lleva. Google y GitHub están completos tal cual; los otros tres son por inquilino y necesitan `provider.{id}.issuer_uri`, a partir del cual el **descubrimiento OIDC** lee el resto de `{issuer}/.well-known/openid-configuration`. El descubrimiento es perezoso — una registration se resuelve en el primer uso y queda memoizada para el proceso, así que `firefly:cache` y los arranques de consola nunca necesitan que el proveedor esté accesible — salvo que `discovery.eager` esté encendido, que resuelve todas las registrations en el arranque y hace fallar el arranque ante un emisor caído, exactamente como hace Spring Boot.

Todo lo que puede comprobarse sin una petición **se** comprueba sin ella: un `client_id` ausente, una registration sin secreto sobre un método que necesita uno, un grant desconocido, un provider que no es ni preset ni está configurado, un preset por

inquilino sin emisor, un provider sin `issuer_uri` que no deletrea sus endpoints, una clave mal escrita (`client-id` se rechaza, no se ignora) y `login.enabled` sin ninguna registration son cada uno una `ConfigurationException` que nombra la clave, en el arranque.

Con `login.enabled` encendido existen dos URLs.

`GET /oauth2/authorization/{id}` inicia un login y `GET /login/oauth2/code/{id}` lo termina — los valores por defecto de `login.authorization_endpoint_base_uri` y `login.redirection_endpoint_base_uri`, y el segundo es además la ruta a la que apunta la plantilla `redirect_uri` por defecto. Las dos las responde su propio filtro, *antes* de `HttpSecurityFilter` (-70), así que ninguna necesita una regla de URL propia. La página de login del framework del Capítulo 10 gana un botón "Sign in with {client_name}" por cada registration de tipo `authorization_code` — detrás del formulario de contraseña cuando el login por formulario también está encendido, y sola cuando no lo está.

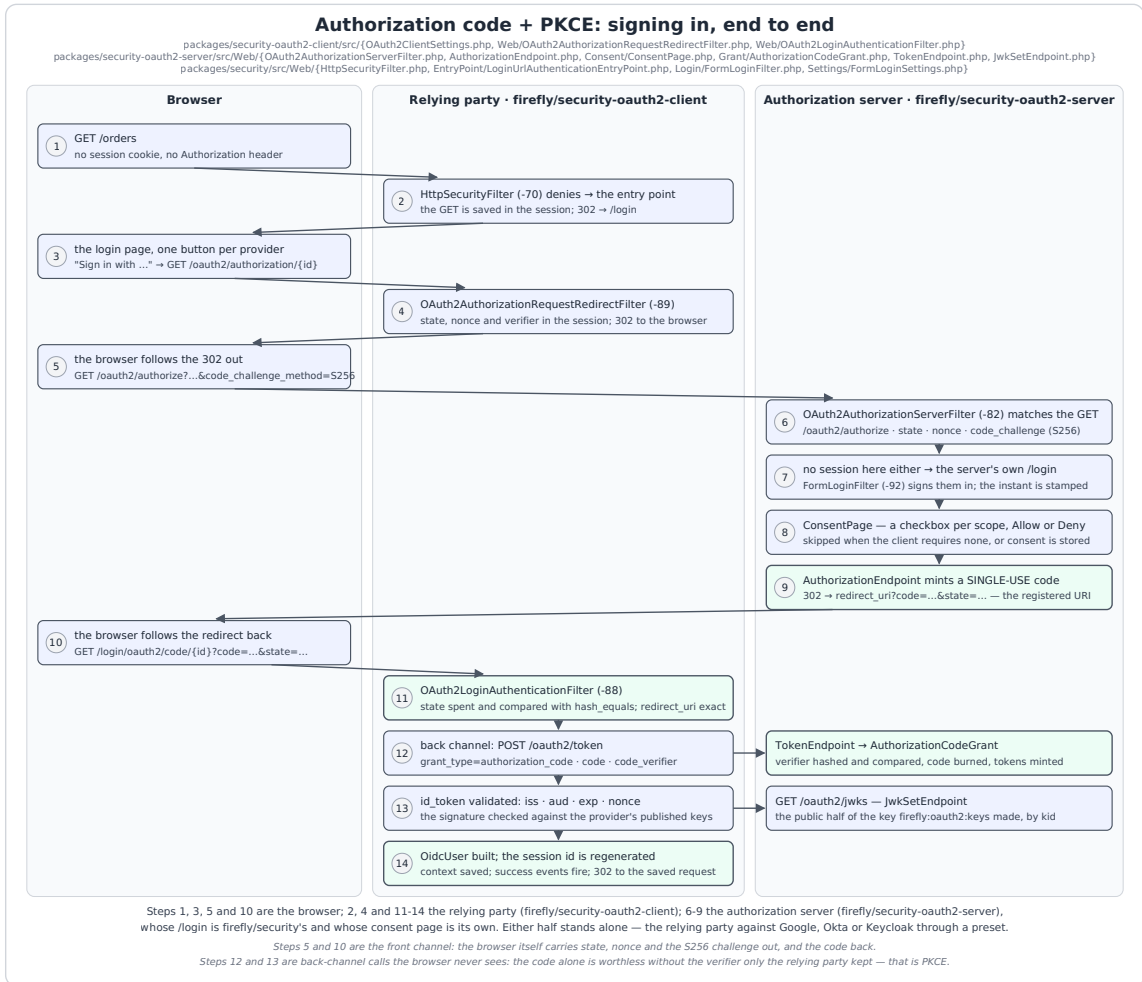


Figura 10A.1 — El viaje de ida y vuelta del código de autorización con PKCE: el navegador, el relying party y el servidor de autorización, con todas las rutas de endpoint reales.

Lee la figura de izquierda a derecha y fíjate en lo que el navegador nunca lleva. La URL de inicio construye una petición de autorización y la guarda en la sesión; solo viajan por el navegador sus mitades públicas — el **state**, el **nonce**, el **code_challenge**. El verificador del que se derivó el desafío se queda atrás, y se gasta en un **POST** de canal trasero que el navegador nunca hace. Tres valores aleatorios, un generador:

```

/**
 * 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)), '+/', '-_'), '=');
    }
}

```

```
    }
}
```

El trayecto de vuelta es donde viven los rechazos, y el orden en que corren es en sí mismo una decisión. La petición de autorización guardada se **saca** de la sesión primero — un solo uso, pase lo que pase después — de modo que un callback reproducido no encuentra nada contra lo que comparar. Después, en orden: un proveedor que respondió `?error=` devuelve su propio código RFC 6749; no haber petición guardada (o que sea de otra registration) es

`authorization_request_not_found`; un `state` que no coincide, comparado con `hash_equals`, es `invalid_state_parameter`; una URL de callback que no es exactamente el `redirect_uri` de la petición es `invalid_redirect_uri`; un código ausente es `invalid_request`. Solo entonces se canjea el código.

Lo que vuelve se examina antes de identificar a nadie. Para una registration `openid`, la firma del id token se verifica contra el JWKS del proveedor (cacheado durante `jwk_set.cache_ttl`), y después se comprueban `iss`, `aud`, `azp`, `iat`, `sub` y el `nonce` según OIDC Core §3.1.3.7 con `clock_skew` segundos de holgura; userinfo se consulta cuando el proveedor tiene ese endpoint y se concedió un scope `profile`, `email`, `address` o `phone`, y su `sub` debe coincidir con el del id token. Una registration OAuth2 plana — sin `openid` — no tiene id token que comprobar y se identifica solo con `user_info_uri`. Después se regenera el id de sesión, se guarda el `SecurityContext` igual que lo guarda el login por formulario del Capítulo 10, y se envía al navegador a la petición que se guardó cuando fue rechazada por primera vez, o a `login.default_success_url`.

⚠ UN 503 DEL PROVEEDOR NO ES UN LOGIN RECHAZADO

El descubrimiento, la descarga del JWKS y userinfo pasan todos por el cliente Http de Laravel con los dos timeouts acotados en cinco segundos — el valor por defecto de Laravel es treinta, que es el límite de ejecución de PHP, así que un emisor lento sería un error fatal en lugar de una página de error. Cuando uno de ellos es genuinamente inalcanzable el resultado es un **503**, no un 401 ni una redirección a `/login?error`: no se examinó nada de las credenciales de la persona, y decirle "no se pudo iniciar sesión" la mandaría a probar una contraseña que no tiene. La página de login omite el proveedor cuyo descubrimiento esté caído en ese momento, con un aviso en el log, en lugar de tumbar con él todas las demás formas de entrar.

La persona en tu controlador

Tras un login correcto el principal de la sesión es un `OidcUser` (o un `OAuth2User` para una registration sin `openid`), y el `SecurityArgumentResolver` del Capítulo 10 lo inyecta sin cableado adicional. El controlador de fixture del propio paquete recoge todas las formas de leer uno:

```
/**
 * 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` extiende `OAuth2User` y `ClaimAccessor`: además de `getName()`, `getAttributes()` y `getAuthorities()` tiene `getIdToken()`, `getUserInfo()`, `getSubject()`, `getEmail()`, `getFullName()`, `getPreferredUsername()` y el par

`getClaim()` / `hasClaim()`. Sus claims son los del id token con los de userinfo escritos encima — la precedencia de Spring — y `getName()` lee el claim que nombre `provider.{id}.user_name_attribute` (`sub` por defecto; `id` para GitHub).

Tres de esos valores devueltos merecen una pausa, y el test que los fija lo dice en voz alta:

```
/**
 * 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' =>
'idTokenValue' => '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' => '' es el primero. `DefaultOidcUser` es un `CredentialsContainer`, exactamente igual que el `User` distribuido del Capítulo 10, así que la copia que guarda la sesión lleva el valor **crudo del id token en blanco** mientras sobrevive cada claim decodificado de él. El id token crudo, el access token y el refresh token viven en un único sitio — la entrada de cliente autorizado — y esa entrada está **cifrada** con la clave de la aplicación. Un principal en una sesión es un conjunto de hechos sobre una persona, no una credencial portadora que alguien pudiera sacar de un fichero de sesión.

El segundo es que `/api/engineers` responde `403`. El proveedor le concedió a `ada` un claim `groups` con `['engineering']`, y nada en OAuth2 dice qué significa un grupo para tu aplicación — así que por defecto no significa nada, y `hasRole('ENGINEERING')` es falso. Las autoridades que lleva un login OAuth2 son `OIDC_USER` (una `OidcUserAuthority` que carga los claims) u `OAUTH2_USER`, más un `SCOPE_x` por cada scope concedido — la misma grafía que produce el filtro de servidor de recursos del Capítulo 10, y por eso `hasScope('email')` lee las dos sin saber qué filtro autenticó la petición. Convertir un claim en un rol es una decisión de la aplicación, y tiene un puerto:

```

/** 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;
}
}

```

Enlaza ese `#[Bean]` y el `/api/engineers` de ese mismo test responde `200`, porque `GrantedAuthoritiesMapper` recibe la lista concedida — con la autoridad que carga los claims primero — y devuelve la lista que llevará la `Authentication`. El principal conserva la concedida, así que un mapper puede añadir sin borrar lo que el proveedor dijo realmente.

El tercero es que la `Authentication` recuerda además **qué** registration identificó a la persona, en sus atributos, y se lee de vuelta con

`OAuth2AuthenticationToken::registrationId($authentication)`. Eso no es contabilidad: cerrar sesión y llamar a una API necesitan las dos saber con qué proveedor hablar, y las dos secciones siguientes son esas dos necesidades.

Cerrar la sesión también fuera

`logout.oidc_initiated` enlaza un bean sobre el puerto `LogoutSuccessHandler` que presentó el Capítulo 10. `LogoutFilter` (-93) pregunta a su manejador de éxito **antes** de invalidar la sesión — que es la única razón por la que el manejador todavía puede leer el cliente autorizado cifrado y encontrar el id token crudo que enviar:

```
/**
 * 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));
    }
}

```

Cada rama que devuelve `null` es una rama en la que la persona sigue quedando fuera de esta aplicación — el manejador solo decide si el navegador visita además al proveedor. Esa es la forma correcta para un manejador en esta posición: un proveedor sin `end_session_endpoint`, una registration que alguien borró de la configuración esta mañana, o un proveedor con el descubrimiento caído no pueden convertir "cerrar sesión" en una página de error. Tres de los cuatro casos se registran a nivel de aviso, para que un operador vea que la sesión del proveedor está sobreviviendo a la de la aplicación en lugar de adivinarlo.

`post_logout_redirect_uri` vale por defecto `{baseUrl}/login?logout` — la página de login del framework con su aviso de sesión cerrada, la misma a la que aterriza el login por formulario — y tiene que estar **registrada en el proveedor**, que es el único paso de todo esto que no está en tu propio fichero de configuración. El viaje completo se conduce de extremo a extremo en 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');
```

Llamar a una API como la aplicación

Identificar a una persona es una de las dos cosas que hace un cliente OAuth2. La otra es guardar un token para que *tu* código pueda llamar a la API de otro — un job nocturno que publica facturas, un controlador que lee el catálogo de un socio. Ahí no hay navegador alguno, y por eso `firefly.security.oauth2.client.enabled` deliberadamente **no** exige `firefly.security.enabled`: un worker de colas con un token saliente no tiene seguridad entrante de la que hablar.

`OAuth2AuthorizedClientManager::authorize()` es el único punto de entrada, y son los dos managers de Spring en una clase porque lo único que los distingue es dónde se busca el cliente:

```
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;
}

```

Una registration `client_credentials` es el token propio de la aplicación: se busca en el `OAuth2AuthorizedClientService` (la caché de Laravel, cifrada), se pide cuando falta o cuando está a menos de `clock_skew` segundos de caducar, y se cachea — un token por pool de procesos, no uno por petición. Una registration `authorization_code` es el token de una *persona*: se lee primero de la sesión cuando la petición en curso tiene una (el login lo dejó ahí, y la petición es lo que lo ata a ese navegador), si no del servicio de caché bajo el nombre del principal — el camino que toma un job en cola — y se refresca con el refresh token cuando está a punto de caducar, conservando el refresh token rotado y el id token y escribiendo el resultado en los dos almacenes. Sin nada que devolver y sin nada con lo que refrescar, la respuesta es una `ClientAuthorizationRequiredException`: un **401** `CLIENT_AUTHORIZATION_REQUIRED` que dice, en la práctica, *inicia sesión primero con esa registration*.

Fíjate en lo que guarda un `OAuth2AuthorizedClient`: el **id** de la registration, el nombre del principal y los tokens — nunca la `ClientRegistration` en sí. Un secreto de cliente, por tanto, nunca entra en una sesión ni en una entrada de caché, haga después la aplicación lo que haga con el objeto.

Rara vez llamarás tú mismo al manager, porque hay una macro por encima:

```

/**
 * `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($url)` es `Http::get($url)` con un bearer encima, y sigue siendo una llamada Http de Laravel corriente en todo lo que le importa a un test: `Http::fake()` la intercepta, el middleware de cliente de la ola de observabilidad la traza, y `->retry()` sigue funcionando.

`Http::oauth2Client('corp', 'ada')` nombra al principal para una registration ligada a una persona. Lo único que hay que recordar del almacén respaldado por caché es que `authorized_client.cache_ttl` (un día, por defecto) es cuánto sobrevive ahí un cliente renovable; una registration cuyos tokens deban durar más — un worker actuando como una persona durante semanas — quiere un enlace propio de `OAuth2AuthorizedClientService` sobre una tabla, que es un solo `#[Bean]`.

Levantar tu propio servidor de autorización

Dale la vuelta al cuadro. `firefly/security-oauth2-server` es el modelo de Spring Authorization Server dentro de tu aplicación: clientes registrados, el grant de código de autorización con PKCE y una página de consentimiento, client credentials, refresh tokens, access tokens JWT u opacos, y el documento de descubrimiento que le cuenta a todos los demás dónde está todo eso. Su bloque de configuración es más largo que el del cliente porque es una superficie de protocolo y no una relación, pero cada valor de abajo es un valor por defecto que puedes ignorar hasta que lo necesites:

```

'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,
    ],

```

```
    ],
  ],
```

Esas rutas — más los dos documentos `/well-known/` que las anuncian — son lo que responde `OAuth2AuthorizationServerFilter` (-82), y su posición en la cadena contesta dos preguntas que la gente hace de inmediato. Corre *después* del filtro de sesión en -94 y de los filtros de bearer — porque el endpoint de autorización necesita el principal que haya — y *antes* de `CsrfFilter` (-80) y `HttpSecurityFilter` (-70), que es por lo que a un `POST /oauth2/token` de una máquina nunca se le pide un token de sesión y por lo que ninguna regla de URL que deniegue por defecto puede cerrar el servidor sin querer. Tus `http.rules` pueden terminar con `*` → `authenticated` y los metadatos, el JWKS y el endpoint de token siguen respondiendo; los endpoints de autorización y userinfo comprueban ellos mismos el principal. No hay que añadir nada a `http.rules` ni a `csrf.except`, y el único formulario de navegador que el servidor posee — el POST de consentimiento — lo comprueba contra el token de sesión el propio endpoint, exactamente como comprueban el suyo los filtros de login y logout. Una petición cuya ruta no sea ninguna del servidor cuesta una comparación de ruta y sigue de largo.

Ruta (por defecto)	Método	Responde
<code>/.well-known/openid-configuration</code> , <code>/.well-known/oauth-authorization-server</code>	GET	Un documento de descubrimiento (el de OIDC es un superconjunto del de RFC 8414): cada URL de endpoint, los grants, los métodos de autenticación de cliente, <code>S256</code> y el algoritmo de firma
<code>/oauth2/jwks</code>	GET	Las claves públicas — la actual primero, las anteriores después — con <code>Cache-Control: public, max-age=3600</code>
<code>/oauth2/authorize</code>	GET, POST	El endpoint de autorización y el formulario de consentimiento
<code>/oauth2/token</code>	POST	<code>authorization_code</code> (con <code>code_verifier</code>), <code>client_credentials</code> , <code>refresh_token</code> ; las credenciales se leen solo del cuerpo, nunca de la query string
<code>/oauth2/introspect</code>	POST	RFC 7662: <code>{active: false}</code> para todo lo desconocido, revocado, caducado o que sea un código

Ruta (por defecto)	Método	Responde
<code>/oauth2/revoke</code>	POST	RFC 7009: un refresh token se lleva consigo su access token; lo desconocido es <code>200</code>
<code>/userinfo</code>	GET, POST	Los claims del <code>OidcUserInfoMapper</code> , para un bearer que lleve <code>openid</code>
<code>/connect/logout</code>	GET, POST	Cierre de sesión iniciado por el RP: la otra cara de la sección anterior
<code>/connect/register</code> (apagado)	POST	Registro dinámico RFC 7591, detrás de un bearer de un solo uso que lleve <code>client.create</code>

El endpoint de autorización es el que tiene una persona dentro, y corre en un orden fijo: primero el `client_id` y un `redirect_uri` registrado que coincida exactamente — rechazados **en la página**, nunca redirigidos, porque el RFC 6749 §4.1.2.1 dice que un destino de redirección no verificado es justo donde a un atacante le gustaría que fuese el error — y después `response_type=code`, los grants del cliente, los scopes pedidos y PKCE, cada uno de los cuales sí es una redirección con `error=`. Solo entonces busca un dueño del recurso, y lo busca en un sitio: el principal que el `SecurityContextRepository` guarda **entre** peticiones. Un bearer autenticado por el filtro de servidor de recursos en `-85` no es un dueño del recurso y no puede canjearse a sí mismo por un código. Sin nadie identificado, toma el relevo el `AuthenticationEntryPoint` del Capítulo 10 — la página de login, con la petición de autorización guardada — y la visita de vuelta aterriza otra vez en `/oauth2/authorize` para encontrarse con la página de consentimiento.

El consentimiento es la página propia del framework, construida sobre el diseño de la página de login y sin JavaScript, salvo que `consent.view` nombre una vista Blade tuya; en cualquier caso el modelo es un `ConsentPageModel` con el nombre del cliente, los scopes como tripletes `{scope, description, approved}`, el token CSRF y la acción del formulario. Aprobar registra un `OAuth2AuthorizationConsent` (fusionado con lo que ya se hubiera aprobado), así que la siguiente petición de autorización con los mismos scopes se salta la página; denegar es `access_denied` de vuelta en el cliente.

Lo que el endpoint de token entrega entonces es un access token de vida corta — por defecto un JWT con los claims del RFC 9068, o 32 bytes aleatorios opacos cuando `access_token.format` es `reference` — y, para un cliente confidencial con el grant, un refresh token. El refresh token es donde vive la regla interesante, y se lee mejor como test que como párrafo:

```
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']);
});

```

Cuatro reglas en un test. Cada refresco **rota** el token e invalida el access token al que sustituye. Un refresco puede pedir un scope *más estrecho*, y la autorización sigue recordando la concesión original, así que el siguiente refresco puede volver a ensancharse hasta ella. Presentar un refresh token ya superado se trata como un robo y no como un despiste: se revoca la autorización entera — que es por lo que la última

línea, usando el token *bueno*, también falla. Y un cliente que genuinamente no pueda rotar puede poner `token_settings.reuse_refresh_tokens`, en cuyo caso recibe de vuelta su propio token, porque un servidor que solo guarda hashes no tiene ningún valor que devolver como eco.

Dos cosas más sobre el almacenamiento, porque deciden si los valores por defecto sobreviven al contacto con un despliegue real. Los tokens se guardan **por hash SHA-256**, nunca por valor, así que un volcado de `oauth2_authorizations` no es un conjunto de credenciales. Y `clients.driver` y `authorizations.driver` valen por defecto `memory`, que quiere decir *este proceso* — bien para un test y para un único servidor de desarrollo, y mal en cuanto hay dos workers, porque un código emitido por uno tiene que poder canjearlo el otro. Ahí la respuesta es `eloquent`, y `php artisan vendor:publish --tag=firefly-oauth2-server-migrations` publica las tablas.

Las claves, y cómo rotarlas

Todo lo anterior cuelga de una clave privada. El framework la genera:

```
php artisan firefly:oauth2:keys # RSA 2048 → storage/oauth2/
private.pem, chmod 0600
php artisan firefly:oauth2:keys --algorithm=ES256 # una clave EC P-256 en su
lugar
php artisan firefly:oauth2:keys --out=/run/secrets/oauth2.pem --force
php artisan firefly:oauth2:keys --print # a stdout, para un gestor de
secretos
```

El comando imprime las dos líneas que necesitas a continuación — la variable de entorno que poner, y el `kid` que publicará el JWKS — y sus opciones son exactamente estas:

```

/**
 * 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` existe para que sobrescribir sea deliberado: una clave reemplazada por accidente invalida todos los tokens en vuelo, y ningún mensaje de error de ningún sitio diría por qué. `jwt.key_id` vale por defecto la huella RFC 7638 de la clave pública, lo que significa que cada nodo que corra la misma clave publica el mismo `kid` sin ninguna coordinación.

Rotar son dos movimientos y ninguna parada. Genera una clave nueva; mueve la vieja a `jwt.previous_keys` como `[{key, key_id}]` — basta con su mitad pública, puesto que ahora solo necesita *verificar* — y apunta `jwt.signing_key` a la nueva. Los tokens firmados con la clave vieja siguen verificando hasta que caduquen, el JWKS publica las dos, y un servidor de recursos que lea el origen local de claves ve las dos. Publicar dos claves bajo un mismo `kid` se rechaza en el arranque, porque un verificador guarda una clave por id y la pareja haría fallar cada token nuevo en su primer uso — así que un `key_id` explícito tiene que cambiar con la clave.

🔗 SÉ TU PROPIO SERVIDOR DE RECURSOS

El despliegue más habitual de este paquete es una aplicación que emite tokens y los acepta. Enciende `firefly.security.oauth2.resource_server` del Capítulo 10 con `jwt_source: local`, y el filtro leerá el conjunto de claves a través del bean `JwksDocumentSource` que el servidor ya enlaza — sin ninguna descarga HTTP a sí mismo, que en un servidor de desarrollo de un solo proceso sería un interbloqueo. Di `local` explícitamente en lugar de dejar `auto: auto` reconoce como "de esta aplicación" únicamente `/.well-known/jwks.json`, y el servidor publica en `/oauth2/jwks`.

Verlo funcionar

Las dos mitades traen una forma de ejercitarlas que no involucra ningún proveedor externo ni red alguna.

Para la mitad cliente es

`Firefly\Testing\Security\OAuth2\FakeAuthorizationServer` — un proveedor OpenID Connect entero en una clase. `install()` monta su **canal frontal** como rutas reales sobre la aplicación bajo prueba (`/authorize`, renderizando opcionalmente una página de consentimiento; `/end-session`) y falsea su **canal trasero** con

`Http::fake()` para exactamente cuatro URLs: el documento de descubrimiento, `/token`, `/jwks` y `/userinfo`. Cualquier otra cosa queda sin falsear, así que una petición saliente perdida sigue siendo una petición saliente perdida. Los tokens son JWTs RS256 reales firmados con un par de claves generado una vez por proceso; los códigos son de un solo uso, atados a su `redirect_uri` y comprobados contra PKCE; el reloj es `Date::now()`, así que `$this->travel()` caduca tokens. Cada salto queda registrado y cada fallo está a un método de distancia — `overrideIdTokenClaims()`, `signInWithUnknownKey()`, `refuseToken()`, `refuseAuthorization()`, `withoutUserInfo()`, `takeDiscoveryDown()`. Y como el plugin de navegador sirve la aplicación en el mismo proceso, ese mismo falso conduce un Chromium de verdad:

```
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);

```

Para la mitad servidor es

`Firefly\Testing\Security\OAuth2\OAuth2ServerTestClient`, que conduce tu *propio* servidor a través del cliente de pruebas de Laravel: `authorize()` con un par PKCE y un state nuevos, `approveConsent()`, `obtainCode()`, `exchangeCode()`, `clientCredentials()`, `refresh()`, `introspect()`, `revoke()` y `userInfo()`. Lee

las rutas de endpoint del bean `AuthorizationServerSettings`, así que renombrar una no exige cambio alguno en un test. Lo único en lo que insiste es en que **identifiques a la persona a través de la página de login y lleves la cookie de sesión:**

`actingAsPrincipal()` deliberadamente no basta en el endpoint de autorización, porque el endpoint exige el principal que el `SecurityContextRepository` guarda entre peticiones, de modo que ningún doble de prueba puede ser un dueño del recurso sin una sesión detrás. Los ayudantes de máquina no necesitan ninguna identificación.

Y el viaje completo del servidor se conduce también en Chromium, contra un fixture de relying party servido por la misma aplicación:

```
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);

```


Las últimas cuatro líneas son la parte que conviene guardarse: un token que acuñó esta aplicación, enviado por un proceso que no tiene ninguna sesión, autentica en una ruta que esta aplicación protege — a través de `OAuth2ResourceServerFilter` (-85) y de nada más — y la misma ruta sin el token es un `401` . Ese es el círculo entero cerrado dentro de un solo test.

En producción esa misma pregunta la responde el actuador.

`/actuator/oauth2clients` lista cada cliente registrado con sus métodos, grants, scopes, URLs de redirección y ajustes, más cuántas autorizaciones tiene vivas — y nunca un secreto, porque el payload se ensambla campo a campo y `clientSecret` no se nombra en ninguna parte de él. Viene sin exponer por defecto como todo lo que va más allá de `health` e `info` , y la página **OAuth2 clients** del panel de administración (`/firefly/oauth2` , en el grupo Wiring) lo renderiza en el mismo proceso. Dos detalles de esa página pagan el momento que cuestan: PKCE se informa dos veces — como lo registró el cliente y como lo imponen de verdad los endpoints, que difieren siempre que el `require_pkce` global esté encendido — y los recuentos de autorizaciones llevan una bandera `processLocal` , que es `true` con el driver `memory` y es la diferencia entre "este cliente no tiene autorizaciones vivas" y "este worker no ha emitido ninguna".

📦 PARIDAD CON LARAVEL

La mitad cliente es el terreno que cubre Socialite, y lo cubre de otra manera: Socialite te da un driver por proveedor y espera que escribas tú el controlador, llames a `Auth::login()` y decidas qué hacer con el payload, y lee userinfo en lugar de validar un id token. Aquí el flujo es configuración con la forma de Spring Boot, el id token se comprueba firma y claim a claim, PKCE y un `state` de un solo uso comparado en tiempo constante no son opcionales, y el resultado es un principal que el resto del framework ya entiende. La mitad servidor es el terreno de Passport, y las diferencias son la superficie de protocolo: Passport no tiene documento de descubrimiento, ni endpoint de introspección, ni OIDC, mientras que este paquete trae RFC 8414, RFC 7662, RFC 7009, el userinfo de OIDC Core, RP-Initiated Logout y RFC 7591 — sobre la misma sesión, la misma página de login, el mismo entry point y el mismo `PasswordEncoder` que ya te dio el Capítulo 10, en lugar de sobre una pila paralela propia.

✓ Lo que aprendiste

Concepto	Qué hace
<code>firefly/security-oauth2-client</code>	Tu aplicación como relying party : registrations, descubrimiento, login por código de autorización, un principal <code>OidcUser</code> , cierre de sesión iniciado por el RP, tokens salientes
<code>firefly/security-oauth2-server</code>	Tu aplicación como servidor de autorización : clientes, consentimiento, <code>/ oauth2/token</code> , introspección, revocación, userinfo, JWKS, descubrimiento
<code>OAuth2AuthorizationRequestRedirectFilter</code> (-89)	Responde <code>GET /oauth2/authorization/ {id}</code> : construye la petición, la guarda en la sesión, redirige al proveedor
<code>OAuth2LoginAuthenticationFilter</code> (-88)	Responde <code>GET /login/oauth2/code/{id}</code> : saca la petición guardada (un solo uso), comprueba el <code>state</code> , canjea el código, identifica a la persona
<code>OAuth2AuthorizationServerFilter</code> (-82)	El único filtro del servidor — tras los filtros de sesión y bearer, antes de CSRF y de las reglas de URL, así que no hace falta ninguna regla ni entrada en <code>csrf.except</code>
Registrations y providers	<code>registration.{id}</code> es una relación, <code>provider.{id}</code> es un emisor; cinco presets, e <code>issuer_uri</code> descubre el resto
<code>OAuth2AuthorizationRequestResolver</code>	Un <code>state</code> nuevo de 43 caracteres cada vez, un <code>nonce</code> para cada petición <code>openid</code> y un verificador PKCE S256 — un generador, 256 bits cada uno

Concepto	Qué hace
<code>OidcUser</code>	Claims fusionados (userinfo sobre id token), <code>SCOPE_*</code> por scope concedido, y el id token crudo en blanco en la copia de la sesión
<code>GrantedAuthoritiesMapper</code>	El <code>#[Bean]</code> que convierte un claim <code>groups / roles</code> en <code>ROLE_*</code> ; añade a la lista concedida en lugar de reemplazarla
<code>OidcClientInitiatedLogoutSuccessHandler</code>	<code>logout.oidc_initiated</code> : el <code>end_session_endpoint</code> del proveedor con <code>id_token_hint</code> , leído antes de invalidar la sesión
<code>OAuth2AuthorizedClientManager</code>	<code>client_credentials</code> desde la caché, <code>authorization_code</code> desde la sesión o la caché, refrescado dentro de <code>clock_skew</code> de la caducidad
<code>Http::oauth2Client('{id}')</code>	Un <code>PendingRequest</code> corriente con el bearer de esa registration encima: falseable, trazado, reintentable
PKCE, <code>state</code> , <code>nonce</code>	S256 por defecto y obligatorio para un cliente público; <code>state</code> de un solo uso, atado a la sesión y comparado en tiempo constante; <code>nonce</code> siempre enviado y siempre comprobado
Rotación de refresh tokens	Cada refresco rota e invalida el access token al que sustituye; presentar un token superado revoca la autorización entera
<code>firefly:oauth2:keys</code>	RSA 2048 o P-256, <code>0600</code> , nunca sobrescrito sin <code>--force</code> ; el <code>kid</code> vale por defecto la huella RFC 7638

Concepto	Qué hace
<code>jwt.previous_keys</code>	Rotación sin parada: la clave vieja sigue verificando y sigue apareciendo en el JWKS; dos claves bajo un <code>kid</code> es un rechazo de arranque
Drivers <code>memory</code> y <code>eloquent</code>	<code>memory</code> es un proceso — un test o un único servidor de desarrollo; con más de un worker hace falta <code>eloquent</code> , o un código emitido por un proceso será incanjeable por otro
<code>FakeAuthorizationServer</code>	Un proveedor OIDC entero en una clase: rutas reales para el canal frontal, <code>Http::fake()</code> para exactamente cuatro URLs del canal trasero
<code>OAuth2ServerTestClient</code>	Conduce tu propio servidor de extremo a extremo, sobre una identificación real — porque el endpoint de autorización exige un principal sostenido por la sesión
<code>/actuator/oauth2clients</code>	Cada cliente, sus grants y su recuento de autorizaciones vivas, nunca un secreto; <code>processLocal</code> dice si ese recuento describe el despliegue

Ponlo en práctica

1. **Inicia sesión con un proveedor real, dos veces.** Añade una registration `google` con nada más que `client_id` y `client_secret` y confirma que la página de login gana un botón; luego añade una segunda registration `corp` sobre `keycloak` con

solo `provider.keycloak.issuer_uri` puesto, y observa cómo el descubrimiento rellena los cinco endpoints que nunca escribiste — autorización, token, JWKS, userinfo y end-session. Deja el emisor sin servicio y recarga `/login`: un botón desaparece con un aviso en el log, el otro sigue funcionando.

2. **Observa cómo se gasta el state.** Inicia un login, copia la URL de callback entera de la barra de direcciones antes de que la página termine, y reproducéla. El segundo intento es `authorization_request_not_found` — no un desajuste de state — porque la petición guardada se saca de la sesión antes de comparar nada. Luego empieza de nuevo, manipula un carácter del `state`, y verás `invalid_state_parameter` en su lugar.
3. **Convierte un claim en un rol.** Enlaza un `GrantedAuthoritiesMapper` que mapee un claim `groups` a `ROLE_*`, pon `#[PreAuthorize("hasRole('ENGINEERING')")]` sobre una acción, y confirma que la misma persona que recibía un `403` antes de que existiera el bean recibe un `200` después. Luego confirma que `hasScope('email')` ya era cierto sin ningún mapper, y di por qué.
4. **Sé las dos mitades a la vez.** En una misma aplicación, enciende `oauth2.server` con un cliente `web-app`, y `oauth2.resource_server` con `jwks_source: local`. Acuña un token con `OAuth2ServerTestClient`, llama con él a una API protegida, después rota la clave de firma a `jwt.previous_keys` y vuelve a llamar a la API con el token *viejo*: sigue verificando. Ahora quita `previous_keys` y observa cómo ese mismo token deja de valer.
5. **Rompe la pareja a propósito.** Enciende `firefly.security.http_basic.enabled` junto al servidor de autorización y lee el rechazo del arranque. Luego razónalo solo con la tabla de filtros del Capítulo 10, sin volver a arrancar: qué filtro responde primero, qué responde, y por qué no se habría registrado nunca ningún error de OAuth2.

PARTE IV

Observabilidad, Pruebas y Entrega

Observabilidad: Salud, Métricas y el Actuator

Al terminar este capítulo conocerás el SPI `HealthIndicator` de `firefly/actuator` y los indicadores integrados `Ping / DiskSpace / Db`, cómo `HealthEndpoint` los agrega en una única respuesta `/actuator/health` — y cómo un **grupo** de sondeo (el mecanismo que hay detrás de "liveness" y "readiness") no es más que un subconjunto de indicadores con nombre y configurado, cómo toda la superficie de gestión está **sin exponer por defecto** de modo que un endpoint olvidado falla cerrado como un 404 en lugar de una fuga de información, y el `MeterRegistry` en PHP puro de `firefly/observability`, su exportador Prometheus a prueba de locale, los buckets de histograma por medidor que convierten un resumen de Prometheus en algo que `histogram_quantile()` puede responder, los formatos de log estructurado (`json`, `ecs`, `logstash`) a los que `firefly.logging.structured.format` conmuta un canal y los cuatro ids que lleva entonces cada línea, y el truco exacto de precedencia `#[Order(500)]` — el mismo que el Capítulo 10 te mostró para la seguridad — que permite a `MeterRegistryCqrsMetrics` reemplazar el `NoOpCqrsMetrics` del bus de CQRS sin ningún cambio de código en `firefly/cqrs`. El capítulo cierra con `firefly/admin`, el panel de administración renderizado en el servidor sobre esos mismos endpoints — un **grafo de beans** dibujado que resuelve cada dependencia de constructor a través de la interfaz por la que está cableada y reporta los ciclos con los que, si no, un arranque moriría sin mensaje. Lee esos endpoints **en proceso**, de modo que renderiza páginas que la superficie JSON deliberadamente mantiene sin exponer, lo que convierte a su propia URL en toda la frontera de seguridad y a su valor por defecto (`app.debug`) en la línea más importante del paquete.

❗ TÉRMINO NUEVO: ACTUATOR

Un **actuador** es un endpoint de gestión que informa sobre el *proceso en ejecución en sí* — si está sano, con qué arrancó, cuán rápidas son sus peticiones — en lugar de sobre el dominio de negocio al que sirve el proceso. El término y la forma provienen ambos de Spring Boot Actuator; **firefly/actuador** es un análogo PHP de primera parte y con pocas dependencias: endpoints de framework montados directamente sobre el mismo **Router** de Illuminate que usan tus propios controladores, no un proceso de administración separado.

El SPI **HealthIndicator**

Una comprobación de salud en LaraFly es cualquier **#[Component]** que implemente un método:

```
interface HealthIndicator
{
    public function health(): Health;
}
```

Health es una lectura inmutable de estado-más-detalles, construida exclusivamente a través de cuatro fábricas con nombre:

```
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 es un enum respaldado que lleva su propia ordenación por severidad y el estado HTTP al que se mapea — DOWN y OUT_OF_SERVICE se renderizan ambos como **503**, de modo que un balanceador de carga no necesita ningún caso especial para tratar cualquiera de los dos como "saca esta instancia de la rotación":

```

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,
    };
}
}

```

Los indicadores integrados: Ping, DiskSpace, Db

Tres `HealthIndicator` vienen con `firefly/actuator`, y vale la pena leer los tres de principio a fin — son cortos, y cada uno enseña una decisión de diseño diferente.

`PingHealthIndicator` es el sondeo trivial siempre-arriba, y su propio docblock nombra exactamente para qué sirve:

```

/** 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 lee una ruta y un umbral de bytes desde la configuración, y se degrada con elegancia — una ruta que no se puede leer se reporta como DOWN con un detalle explicativo en lugar de lanzar una excepción:

```
#[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` está **activo por defecto** — `matchIfMissing: true` en su `#[ConditionalOnProperty]`, igual que la autoconfiguración `DataSourceHealthIndicator` de Spring Boot. Antes era opcional, y el cambio merece entenderse, porque "activo por defecto" no es lo que evita que un proyecto sin base de datos vea un **DOWN** sorpresa. Lo evita una segunda interfaz:

```
#[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` añade exactamente un método al SPI, y `available()` es la respuesta de este framework a `@ConditionalOnBean(DataSource)` de Spring. `HealthEndpoint` lo pregunta antes de ejecutar la comprobación, y un indicador que responde `false` no aporta **ningún componente**: ni `UNKNOWN`, ni `DOWN`, simplemente ninguna clave `db` en la respuesta. Aquí la respuesta se lee de la propia configuración de Laravel: `database.default` tiene que nombrar una conexión, y la entrada de esa conexión tiene que declarar un `driver`. Así, una aplicación que realmente no tiene base de datos obtiene un documento de salud que no la menciona, que es la respuesta honesta; una que sí la tiene y aun así quiere quitar la comprobación pone `firefly.management.endpoint.health.db.enabled=false` y el bean nunca se registra.

Esa es la distinción que conviene llevarse: la **propiedad** decide si el indicador existe, y `available()` decide si un indicador que existe tiene algo que decir. Cualquier `HealthIndicator` que escribas puede implementar la misma interfaz y recibir el mismo trato.

Fíjate en que `DbHealthIndicator` depende directamente de la propia `ConnectionResolverInterface` de Illuminate en lugar de nada de `firefly/data` — no hay ninguna arista `Actuator → Data` en `deptrac.yaml` en absoluto, de modo que una comprobación de salud de BD no le cuesta a `firefly/actuator` ninguna dependencia nueva. Y cada indicador aquí sigue la misma forma a prueba de fallos: una consulta, una comparación o una llamada al sistema de archivos que podría lanzar

una excepción siempre se captura y se convierte en `Health::down()` con un detalle que explica por qué — nunca una excepción sin manejar, nunca un `500` donde corresponde un `503`.

Agregando salud: `HealthEndpoint` y los grupos de sondeo

`HealthEndpoint` es en sí mismo un `#[Component]` (no un simple interno del framework — tiene que ser descubrible como cualquier otro bean) que lee cada indicador registrado, ejecuta cada uno a prueba de fallos, y pliega los resultados hacia abajo hasta el único estado más severo — y, desde `26.09.4`, recibe un colaborador más, un `HealthDetailsAuthorizer` al que pregunta antes de publicar cualquier detalle por componente:

```
#[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,
    };
}
}

```

Vale la pena detenerse en dos cosas. Primero, sin ninguna ruta de petición, todo indicador registrado cuenta y `StatusAggregator` pliega un conjunto vacío a `UP` (vacío significa que no hay nada por lo que estar insano) y, en caso contrario, elige el único `Status` más severo por `severity()` — un único indicador `DOWN` basta para tumbar todo el agregado, informen lo que informen los demás. Segundo, **no existe ninguna clase de endpoint "liveness" o "readiness" separada en ninguna parte del código fuente** — `/actuator/health/{group}` es un único mecanismo genérico, y "liveness"/"readiness" son simplemente *nombres convencionales* que configuras, no una característica con forma de Kubernetes cableada a fuego:

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

return [
    'management' => [
        'endpoint' => [
            'health' => [
                'group' => [
                    'liveness' => ['include' => 'ping'],
                    'readiness' => ['include' => 'ping,db'],
                ],
            ],
        ],
    ],
];
```

Configurado de esta manera, `GET /actuator/health/liveness` agrega solo `ping` (de modo que un proceso sano-pero-momentáneamente-sin-base-de-datos aún se reporta vivo) y `GET /actuator/health/readiness` incorpora también `db` — pero el mecanismo subyacente es la misma búsqueda `group()` exacta y los mismos tipos

`Health / Status` exactos que el resto de esta sección ya te mostró. Un nombre de grupo que nadie configuró devuelve `null` desde `group()`, que `handle()` convierte en un `404` simple, no en un `200` con un cuerpo vacío.

`show-details` (`never` / `when-authorized` / `always`, por defecto `never`) rige si la respuesta incluye el mapa `details` por componente en absoluto — con `never`, un llamante no autenticado ve solo el `status` agregado, nunca *qué* indicador falló ni por qué. Cualquier otro valor, *incluida una errata*, se lee como `never`: la dirección que falla cerrando, porque el coste de retener un detalle es una consulta de soporte y el de publicarlo es una divulgación.

Y `when-authorized` ya significa lo que su nombre dice. Durante dos versiones se degradaba a `never`, porque `firefly/actuator` no tiene ninguna arista de código hacia `firefly/security` y por tanto no podía decir *quién preguntaba* — seguro, y también una mentira documentada sobre lo que el valor hacía. `26.09.4` lo cerró con una forma que este libro ya te ha enseñado — el puerto `CqrsMetrics` del Capítulo 7 y su implementación no-op por defecto son el mismo movimiento: el paquete dueño de la **pregunta** declara un puerto y trae la respuesta conservadora, y el paquete dueño de la **respuesta** lo rellena. El puerto es

`Firefly\Actuator\Health\HealthDetailsAuthorizer`, un `mayReadDetails(): bool` deliberadamente sin argumentos; la implementación propia del actuator es

`DenyHealthDetailsAuthorizer`, que se lo niega a todo el mundo y es el comportamiento antiguo con otro nombre. Activa el componente incluido `firefly/security` con `firefly.security.enabled` y `PrincipalHealthDetailsAuthorizer` toma el relevo: lee el principal guardado en la sesión del mismo

`SecurityContextHolder` que leen las reglas del Capítulo 10, y lo compara con `firefly.management.endpoint.health.roles` a través de la misma `RoleHierarchy`, de modo que `ROLE_ADMIN > ROLE_ACTUATOR` concede lo que dice conceder. Un `roles` vacío es el "cualquier principal autenticado" de Spring; uno no vacío (una lista, o la

cadena CSV de Spring — un `ADMIN` a secas se lee como `ROLE_ADMIN`) lo estrecha más, y un valor que no puede leerse como una lista de roles se lo niega a todos en lugar de admitirlos. La arista de dependencia nueva es `Security → Actuator` y solo en esa dirección: `firefly/actuator` sigue sin nombrar ningún tipo de principal y sigue funcionando en una aplicación sin seguridad alguna. Ambos beans son `#[ConditionalOnMissingBean]`, así que tu propio `HealthDetailsAuthorizer` desplaza a cualquiera de los dos.

⚠ **CAMBIO ROMPEDOR EN** `26.09.4` — MIRA `CHANGELOG.MD` **BAJO** `[26.09.4]`

Dos cosas se movieron, y ambas están registradas allí bajo **BREAKING**. Primero, **una aplicación que ya corría `show-details: when-authorized` con la seguridad encendida empieza a divulgar lo que antes retenía**: con el `roles` por defecto (`[]`, "cualquier principal autenticado"), todo llamante autenticado lee ahora los detalles por componente — que nombran drivers de base de datos, rutas de disco, hosts de brókeres y mensajes de error de los indicadores. Decide en vez de heredar: pon `show-details: never`, o enumera en `firefly.management.endpoint.health.roles` los roles que pueden leerlos, o enlaza tu propio `HealthDetailsAuthorizer`. Segundo, **el constructor de `HealthEndpoint` ganó un cuarto parámetro obligatorio, `HealthDetailsAuthorizer $authorizer`**, de modo que el código que construye el endpoint directamente en lugar de resolver el bean tiene que pasar uno — `new DenyHealthDetailsAuthorizer` reproduce exactamente el comportamiento antiguo. Una aplicación sin `firefly/security`, o con su flag maestro apagado, no se ve afectada: el rechazo por defecto se mantiene y el cuerpo de la respuesta es byte por byte el que era.

Seguro por defecto: exposición y el 404

El hecho operativo más importante sobre el actuator es este: **la mayoría de los endpoints son inalcanzables hasta que digas lo contrario**. `ExposureModel` es todo el mecanismo, y es lo bastante corto como para leerlo entero:

```

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;
    }
    // ...
}

```

El `include` por defecto es `"health,info"` — cualquier otro id de endpoint (`env`, `beans`, `conditions`, `mappings`, `loggers`, `scheduledtasks`, y las `metrics / prometheus` de observabilidad una vez instalado ese paquete) **no está expuesto** hasta que lo añadas explícitamente, y `ActuatorDispatchAction` renderiza un id no

expuesto o desconocido como un `404` simple a través del mismo

`ProblemDetailsRenderer` que presentó el Capítulo 4 — nunca un error de framework en bruto, y nunca un `200` silencioso con un cuerpo que un llamante no autenticado no debería ver. `.exclude` siempre gana sobre `.include`, de modo que `include: '*'` más una lista corta de `exclude` es una política legítima de "expón todo excepto...".

`/actuator/env` superpone una segunda red de seguridad, independiente, encima de la exposición: incluso una vez expuesta, cualquier clave cuyo nombre parezca una credencial se enmascara antes de construir la respuesta. El endpoint en sí son cuatro líneas, porque la regla vive en una única clase compartida:

```
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;
}

```

Que la regla viva en una clase propia en lugar de dentro del endpoint no es pulcritud. `/configprops` necesita la *misma* regla, y una segunda copia es como se pudre una regla de enmascarado: las dos listas divergen en la siguiente palabra que a alguien se le ocurra añadir, y el endpoint que se perdió la adición filtra.

El orden dentro de `mask()` es la parte que merece estudiarse, porque es un arreglo de fallo que puedes leer. La regla, tal como se escribió primero, probaba la clave **solo** en la rama donde el valor era un escalar — una clave con valor de array se recorría hacia dentro y cada hoja se juzgaba luego por su propia clave. Así que

```
firefly.security.jwt.keys => ['active' => 'PRIVATE...', 'previous' => '...']
```

renderizaba las dos claves privadas enteras: `keys` coincidía con el patrón pero era un array, y `active` y `previous` no coincidían con nada. Toda forma real de un secreto — un llavero, un par de credenciales, un mapa de tokens por inquilino — es exactamente esa forma, así que el bypass cubría justo los casos que más importaban. Aquí la **clave decide primero**, y una clave sensible enmascara todo su subárbol sea cual sea el tipo del valor.

Un array sensible se enmascara al escalar `*****` y no a un array de la misma forma lleno de máscaras, y eso también es deliberado: la forma de un secreto es información en sí misma — cuántas claves hay en el llavero, qué inquilinos tienen token — y un llamante que no puede ver los valores tampoco tiene por qué contarlos.

`authorization` y `headers` están en la lista porque el propio framework introdujo una clave que lleva una credencial: `firefly.observability.tracing.otlp.headers` documenta `authorization=Bearer ...` como su contenido previsto, y ninguna de las seis palabras originales aparece en `headers`. Una bolsa de cabeceras es donde viaja la credencial de un cliente saliente, se llame como se llame la cabecera concreta — que es exactamente por qué debe decidir la clave de la **bolsa** y no las hojas.

Más allá de la exposición y el enmascaramiento, el `HttpSecurity` de `firefly/security` (Capítulo 10) es lo que realmente asegura la superficie para el tráfico real, y no necesita **ningún** cambio de código para hacerlo — `HttpSecurityFilter` es un middleware global, de modo que se ejecuta para las propias rutas registradas directamente por el actuador exactamente igual que se ejecuta para tus controladores:

```
<?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'],
            ],
        ],
    ],
];
```

El propio `composer.json` de `firefly/actuator` no tiene ninguna dependencia de `firefly/security` en absoluto — asegurar el actuator de esta manera es **pura configuración**, apoyándose en el mismo DSL de URL de denegar-por-defecto que el Capítulo 10 ya te enseñó, sin ningún mecanismo nuevo que aprender.



CUALQUIER OTRO ENDPOINT DE GESTIÓN ES UN ENDPOINT INDEPENDIENTE, NO UNA SUB-CLAVE DE

**/
INFO**

`/actuator/info` genuinamente tiene exactamente tres fragmentos — `runtime` (de `RuntimeInfoContributor`, registrado por defecto, que es la razón por la que una aplicación recién generada ya responde algo: versión/SAPI/OPcache de PHP, versión de Laravel, versión de LaraFly, memoria actual y pico; desactívalo con `firefly.management.info.runtime.enabled = false`), `app` (de `AppInfoContributor`, leído desde `firefly.management.info.app.*`) y `build` (de `BuildInfoContributor`, que lee un archivo JSON en `firefly.management.info.build.path`). `env`, `beans`, `conditions`, `mappings`, `loggers`, `scheduledtasks`, `configprops` y `caches` son cada uno su **propio** `ActuatorEndpoint`, montado en su propio `/actuator/{id}` — no anidado bajo `/info`. `firefly:about` (Capítulo 13) renderiza varios de estos juntos en el terminal, lo cual es una comodidad de ese único comando, no una prueba de que compartan una ruta.

El índice HAL y el contrato de endpoint

Todo endpoint de framework — health, info, metrics, o el tuyo propio — implementa el mismo contrato de tres métodos:

```
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`, un `BootPass` en la fase `WiringPasses`, descubre cada `#[Component]` que implementa esta interfaz, resuelve cada uno exactamente una vez, y monta exactamente **dos** rutas nativas de Illuminate — un índice `GET {base}` y un despachador comodín `GET|POST {base}/{path}` — bajo la ruta base configurada, de modo que nada del mecanismo colisiona con las rutas de tu propia aplicación. El índice en sí, `ActuatorIndexAction`, renderiza un mapa `_links` de estilo HAL filtrado a todo lo que esté a la vez habilitado y expuesto:

```

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` en una instalación por defecto por tanto lista solo `self`, `health` e `info` — exactamente los dos ids que `include` expone por defecto — y crece solo hasta donde tu propia configuración `.include` lo permita.

Métricas: `MeterRegistry`, `MetricsRecorder`, y el registro idempotente

`firefly/observability` es un segundo paquete opcional, independiente, superpuesto sobre el actuator — trae su propio `MeterRegistry` (la fábrica de cara a la lectura que consulta un consumidor) y un `MetricsRecorder` más estrecho (el puerto de cara a la escritura del que realmente depende la instrumentación, de modo que un filtro o un registrador de métricas nunca necesita la API completa del registro):

```

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` implementa ambos puertos como un almacén en memoria, de primera parte y en PHP puro — sin `ext-prometheus`, sin SDK de OpenTelemetry. El registro es idempotente, indexado por `type|name|sorted-tags`, de modo que llamar a `counter('cQRS_commands_seconds', [...])` dos veces con el mismo nombre y conjunto de tags devuelve la mismísima instancia de `Counter` ambas veces. También falla ruidosamente, a propósito, si intentas registrar un nombre de métrica bajo dos tipos diferentes:

```

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;
    }
    // ...
}

```

Esto no es pedantería: el propio formato de texto de Prometheus limita exactamente una declaración `# TYPE` por nombre de métrica, de modo que un nombre registrado como counter y como gauge en algún lugar de tu código haría que

`PrometheusTextFormat` emitiera dos líneas `# TYPE` en conflicto para el mismo nombre — una exposición inválida que un recolector rechazaría.

`SimpleMeterRegistry` convierte ese error en una `InvalidArgumentException` inmediata en el sitio de llamada responsable, en lugar de en un fallo de recolección descubierto más tarde en producción.

Exposición: el exportador Prometheus a prueba de locale

`PrometheusTextFormat` renderiza cada medidor registrado como texto de formato 0.0.4 en `/actuator/prometheus`. La parte interesante es un método privado de seis líneas que la mayoría de los equipos hacen mal la primera vez que escriben uno:

```

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', $value)` es la forma instintiva de formatear un float en PHP, y es *dependiente del locale*: en un host cuyo `LC_NUMERIC` esté configurado a un locale de coma-decimal como `de_DE`, `sprintf('%f', 1.5)` renderiza `"1,500000"` — una coma que la propia gramática del formato de texto de Prometheus no permite dentro de un valor de muestra, produciendo silenciosamente una exposición que un recolector no puede parsear. `number_format($value, 10, '.', '')` toma el punto decimal y el separador de miles como **argumentos explícitos**, de modo que el locale ambiente del proceso nunca puede filtrarse en la salida — la misma clase de pregunta "qué capa controla el formato" a la que este libro sigue volviendo, aquí respondida una sola vez, en una función, en lugar de en cada sitio de llamada que casualmente imprima un float.

`/actuator/metrics` (`MetricsEndpoint`) expone el mismo registro como JSON con sabor a Micrometer en su lugar — sin subruta lista cada nombre registrado; un nombre concreto se resuelve a sus medidas y tags disponibles, o a un `404` simple si el registro nunca lo ha visto.

Buckets de histograma: convertir un resumen en algo de lo que puedas sacar un cuantil

Un `Timer` se expone por defecto como un **summary** de Prometheus — `_count` y `_sum`, y nada más. Eso basta para una media y es inútil para un p99, porque una latencia media es el único número que esconde todas las malas experiencias que el servicio entregó. `histogram_quantile()` necesita buckets, y los buckets son una decisión por medidor:

```
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` es la lista global de cotas superiores en segundos; `distribution.per-meter.<nombre>` la sobrescribe para un medidor, y una lista por medidor **vacía** devuelve ese medidor a ser un summary. Un temporizador que tiene buckets se expone como un `histogram` de Prometheus — `_bucket{le=...}`, `_count`, `_sum` — y uno que no los tiene sigue siendo un summary.

En ese método de seis líneas se ven dos decisiones de diseño. Las cotas se indexan por **nombre** de medidor, nunca por conjunto de etiquetas, porque una familia de Prometheus tiene exactamente un tipo y una disposición de buckets; una disposición por etiqueta produciría una familia a la que ningún recolector podría encontrar sentido. Y los registros piden los buckets cuando un temporizador se **crea**, no en cada `record()` — la disposición queda fijada para la vida del medidor.

El valor por defecto es **sin buckets**, y no porque los buckets sean peligrosos. Convertir un summary en un histogram cambia la línea `# TYPE` de un scrape y añade una serie por cota, y una actualización no tiene por qué hacerle eso a un panel en marcha sin que se lo pidan. La lista por la que empezar es la que distribuyen los clientes de Prometheus:

```
[0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1, 2.5, 5, 10]
```

Cada cota se valida en tiempo de configuración: una entrada no numérica, un cero, un negativo, un infinito o un NaN es una `ConfigurationException` que nombra la clave. `+Inf` está implícito y no debe listarse.

Auto-instrumentación: `MetricsFilter` y la cardinalidad acotada

Una vez que `firefly/observability` está instalado y habilitado, cada petición HTTP se cronometra automáticamente mediante `MetricsFilter`, un `WebFilter` `#[Component]` descubierto por la cadena de filtros de `firefly/web` sin ningún cableado tuyo:

```

#[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);
}
// ...
}

```

Dos detalles hacen esto seguro para producción en lugar de meramente cómodo.

`#[Order(-100)]` lo pone en la posición más externa entre los filtros descubiertos, de modo que cronometra la cadena interna *entera* — seguridad, validación, tu controlador — no solo una porción de ella. Y el tag `uri` es siempre la **plantilla** de la ruta coincidente (`/orders/{id}`), nunca la ruta de petición en bruto — una petición sin coincidencia (cualquier `404`) se etiqueta con el centinela fijo `'UNKNOWN'` en lugar de con la ruta controlada por el atacante o el rastreador. Etiquetar por la ruta en bruto permitiría a cualquiera generar un número ilimitado de combinaciones distintas de etiquetas de métrica simplemente golpeando URLs inventadas; bajo Octane, donde el registro es un singleton de por vida del proceso en lugar de uno nuevo por petición, eso no es una verruga cosmética — es un vector de crecimiento de memoria ilimitado. Ante una petición que lanza una excepción, el filtro registra el resultado y **relanza** en lugar de tragársela, de modo que `ProblemDetailsRenderer` todavía puede renderizar el error exactamente como lo haría sin el filtro instalado.

`MeterBindingsPass`, un `BootPass` compañero, registra un puñado de gauges basados en pull de la misma manera: `process_resident_memory_bytes` / `php_memory_peak_bytes` para el proceso en ejecución, y un gauge `resilience_circuit_breaker_state{name}` por cada circuit breaker configurado (`closed=0`, `open=1`, `half_open=2`) — cada uno muestreado *en vivo* en el momento de la recolección mediante una clausura, de modo que una petición `/actuator/prometheus` fresca siempre refleja el estado actual del breaker en lugar de una instantánea del arranque.

La costura de métricas de CQRS: `#[Order(500)]`, una vez más

El Capítulo 7 te dejó con `NoOpCqrsMetrics` enlazado detrás de `#[ConditionalOnMissingBean(CqrsMetrics::class)]`, y un docblock prometiendo que un registrador real caería en su lugar más tarde. `firefly/observability` es ese reemplazo, y el mecanismo ganador es *exactamente* el truco de precedencia que el Capítulo 10 usó para `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()` se muestra a su lado porque los dos beans responden preguntas distintas con la misma maquinaria. Ese elige el registro: `SimpleMeterRegistry` por defecto, en memoria de proceso; `CacheMeterRegistry` sobre un almacén de caché con nombre cuando `firefly.observability.metrics.store` nombra uno, de modo que los contadores se acumulan entre trabajadores de PHP-FPM en vez de que cada scrape vea solo su propia petición. A ambos se les entrega el mismo `DistributionStatisticConfig`, así que los buckets de histograma de un temporizador no cambian cuando cambia el almacén de respaldo.

`cqrsMetrics()` es el de esta sección, y `ObservabilityAutoConfiguration` es `#[Order(500)]`, estrictamente por debajo del `#[Order(1000)]` de `CqrsAutoConfiguration`. La pasada incremental de condiciones evalúa primero el `#[Order]` más bajo y registra los supervivientes inmediatamente, de modo que el bean `cqrsMetrics()` de esta clase se registra **primero**; para cuando `CqrsAutoConfiguration` evalúa su propio `#[ConditionalOnMissingBean(CqrsMetrics::class)]`, ya hay un bean enlazado, y el `NoOp` por defecto se retira. Ninguna línea dentro de `firefly/cqrs` cambia — la victoria es puro ordenamiento de auto-configuración, la misma forma que ya has visto dos veces.

`MeterRegistryCqrsMetrics` en sí es el registrador concreto que ese ordenamiento instala — una implementación pequeña y directa del puerto `CqrsMetrics`, que registra cada comando o consulta como un timer etiquetado por tipo de mensaje y resultado:

```
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 ($pos = strrpos($class, '\\')) !== false ? substr($class, $pos + 1) : $class;
    }
}
```

```
    }  
}
```

Cada bean de observabilidad se apoya en la **misma** propiedad,

`firefly.observability.metrics.enabled` — no en la presencia de `MeterRegistry`.

Es una elección deliberada, no un descuido: un

`#[ConditionalOnBean(MeterRegistry::class)]` en `cqrsMetrics()` o en `MetricsFilter` se evaluaría *antes* de que el propio `#[Order(500)]` de `ObservabilityAutoConfiguration` haya registrado `MeterRegistry` en absoluto, descartando erróneamente a todo consumidor aguas abajo incluso con las métricas genuinamente habilitadas — una trampa de ordenamiento, no un error de lógica.

Apoyar todo en una sola bandera en su lugar hace que la supervivencia sea independiente del orden: cambia `firefly.observability.metrics.enabled` y el registro, el registrador de CQRS, el filtro y ambos endpoints sobreviven todos — o se retiran todos — juntos.

Por último, un puerto `Tracer` remata el paquete — y ya no es el marcador de posición con el que empezó. La interfaz son tres métodos: `startSpan(string $name, SpanKind $kind = SpanKind::Internal, array $attributes = [], ?SpanContext $parent = null): Span`, `currentSpan(): ?Span`, y la comodidad `trace(string $name, callable $callback, SpanKind $kind = SpanKind::Internal, array $attributes = []): mixed`, que abre un span, ejecuta la clausura, registra un lanzable como estado `ERROR` más un evento `exception`, relanza y cierra. `NoOpTracer` sigue siendo el valor por defecto entregado — spans que no graban, un `SpanContext` inválido, `currentSpan()` de `null` — y un adaptador de OpenTelemetry se enlaza ahora por encima de él en `#[Order(400)]`, por delante del `#[Order(500)]` de `ObservabilityAutoConfiguration`, siempre que el SDK esté instalado y

`firefly.observability.tracing.enabled` esté activo. La idéntica forma "puerto ahora, adaptador después" que acabas de ver para `CqrsMetrics`; lo que cambió es que el adaptador llegó, y con él la instrumentación a ambos lados.

Trazado: un solo `traceparent`, cinco fronteras

Lo que compra el puerto es un identificador que sobrevive a todas las fronteras que cruza una petición. `W3CTraceContextPropagator` habla W3C Trace Context sobre un mapa de cabeceras llano, sin SDK de por medio — `extract(array $carrier): ? SpanContext` a la entrada, `inject(SpanContext): array<string,string>` a la salida —, así que la propagación funciona igual bajo un tracer respaldado por el SDK que bajo uno de prueba.

Hay cinco fronteras instrumentadas, y la figura las muestra las cinco a la vez.

`TracingFilter` (`#[Order(-110)]`, el más externo de los filtros descubiertos) continúa un `traceparent` entrante como padre remoto y abre un span `SERVER`, y después publica sus identificadores en el `Context` de Laravel y en `Request::$attributes` como `firefly.trace_id` y `firefly.span_id`. `TracerCqrsTracing` abre un span `INTERNAL` alrededor de cada comando y cada consulta — no hay nada que transportar, porque el bus es en proceso, así que el span simplemente se anida bajo el que esté vigente. `TracerEdaTracing` abre un span `PRODUCER` llamado `publish <destination>` a la salida, escribiendo el `traceparent` en las cabeceras del sobre, y un span `CONSUMER` llamado `process <destination>` en la entrega, leyendo el mismo valor de vuelta — que es lo que une el span de un worker de colas con la petición que publicó. Hoy seis llamadores alcanzan esa costura: `InMemoryEventBus`, `QueueEventBus`, el `SubscriberRegistrySink` compartido a través del cual entrega todo consumidor de broker y —desde 26.09.4— los tres publicadores de broker, `RabbitMqEventPublisher`, `KafkaEventPublisher` y `PostgresEventPublisher`, cada

uno de los cuales pasa `publish()` por `tracePublish()` en lugar de construir su sobre por su cuenta. Un *publish* de broker abre por tanto un span `PRODUCER` y estampa un `traceparent` puesto por el propio framework.

`firefly.eda.tracing.brokers.enabled` degrada esos tres al no-op sin tocar los buses en memoria y de colas, que es lo único que puede dejar todavía sin estampar un registro de broker. Y `HttpClientTracingMiddleware`, que `HttpClientTracingPass` instala en la factoría `Http` de Laravel en el arranque, abre un span `CLIENT` en cada llamada saliente e inyecta la cabecera en la petición PSR-7.

Ambas costuras son la forma de `CqrsMetrics` que acabas de leer: `CqrsTracing` vive en `firefly/cqrs` y `EdaTracing` en `firefly/eda`, cada una con un valor por defecto no-op detrás de `#[ConditionalOnMissingBean]`, y la auto-configuración `#[Order(500)]` de observabilidad registra primero la implementación real. Ni `firefly/cqrs` ni `firefly/eda` dependen de `firefly/observability` — mira sus `composer.json` si quieres comprobarlo.

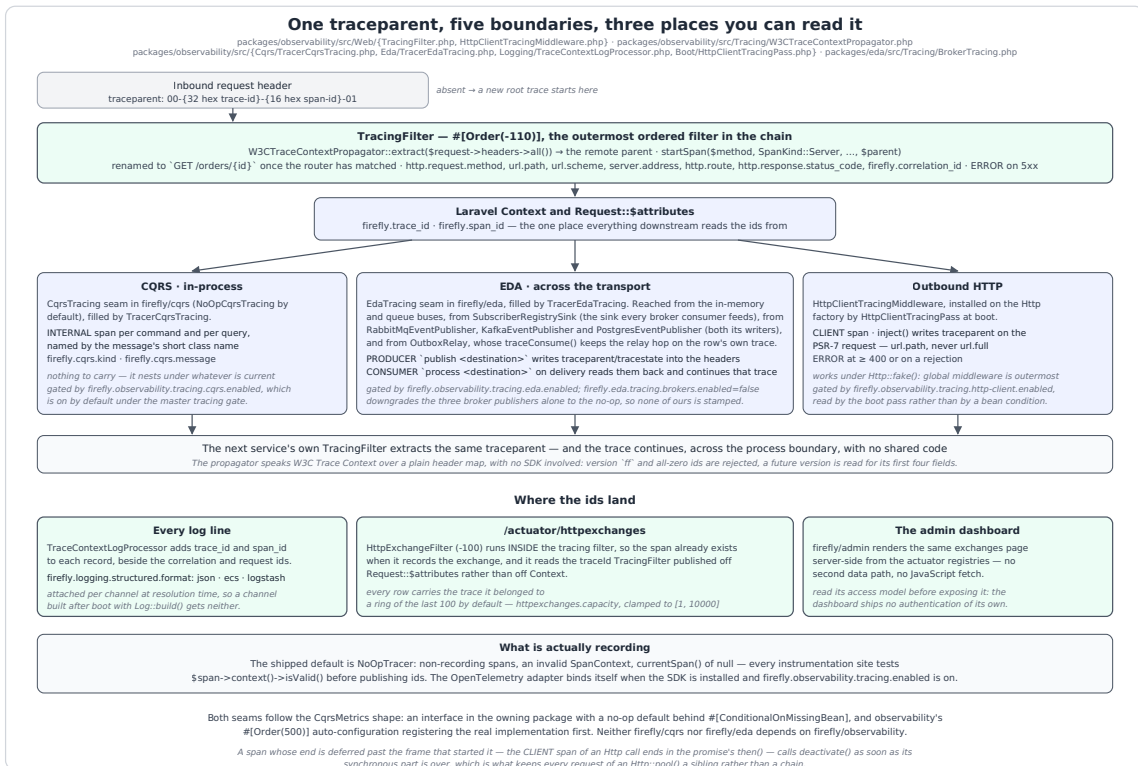


Figura 11.1 — Un solo traceparent entra en TracingFilter, se publica en el Context de Laravel como `firefly.trace_id` y `firefly.span_id`, y vuelve a salir a través del bus de CQRS, del sobre de EDA y del cliente Http saliente — aterrizando en cada línea de log, en `/actuator/httpexchanges` y en el panel de administración.

Los tres sitios donde aterriza son el objetivo de todo el ejercicio.

`TraceContextLogProcessor` estampa `trace_id` y `span_id` en cada registro de log, junto al identificador de correlación y al de petición. `HttpExchangeFilter` — `#[Order(-100)]`, así que se ejecuta *dentro* del filtro de trazado y el span ya existe cuando graba — lee el identificador de traza de vuelta desde `Request::$attributes`, que es la razón por la que cada fila de `/actuator/httpexchanges` lleva la traza a la que perteneció. Y el panel renderiza esa misma página en el servidor desde esos mismos registros, sin una segunda vía de datos.

Logs que una máquina puede leer

Los tres identificadores que produjo la sección anterior solo sirven si algo puede encontrarlos. Una línea de log de `LineFormatter` con un identificador de traza en algún sitio dentro del texto de su mensaje no es buscable; es un pajar con una aguja dentro.

Así que `firefly.logging.structured.format` — el `logging.structured.format` de Spring Boot 3.4, sobre los canales de Laravel — nombra un formateador, y hay tres:

```
public const string FORMAT_KEY = 'firefly.logging.structured.format';

public const string CHANNELS_KEY = 'firefly.logging.structured.channels';
```

Valor	En qué se convierte cada línea
<code>''</code> (el de por defecto)	apagado — el formato propio de Laravel, intacto
<code>json</code>	el <code>JsonFormatter</code> de Monolog
<code>ecs</code>	Elastic Common Schema 8, a través del <code>EcsFormatter</code> propio del framework
<code>logstash</code>	el <code>LogstashFormatter</code> de Monolog, con los campos del framework bajo <code>fields</code>

Cualquier otra cosa es una `ConfigurationException` en el arranque — `LogChannelWiringPass` pide el formato desde `Application::boot()`, antes de tocar el servicio de log, que es lo que convierte una errata en un fallo de arranque en vez de en un repliegue silencioso a texto plano.

La forma de ECS merece verse, porque es la que con más probabilidad aterriza directa en una pila ya existente:

```
/**
 * 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';
```

Los cuatro identificadores del framework y los dos campos de servicio se **elevan** a sus casas de ECS; todo lo que añadiera un procesador o quien llama se queda anidado bajo `extra / context` en vez de fundirse en el nivel superior, así que un `message` o una clave de contexto `error` de la propia aplicación jamás puede colisionar con un campo de ECS.

De dónde salen los identificadores

```
/**
 // ...
 * 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` es el tercero: el identificador que `problem+json` y la cabecera `X-Correlation-Id` ya llevan, leído desde la clave `Context` de Laravel que siembra `CorrelationIdFilter`. Así que una línea de log nombra el span que la escribió, la petición a la que perteneció, y el identificador de correlación que un cliente puede citarte de vuelta.

Lee dos veces la primera fila. Una línea de log escrita dentro de un manejador de comando lleva **el identificador de span de ese manejador**, no el de la petición — que es exactamente lo que hace útil una búsqueda en los logs, porque te dice cuál de los varios spans de una petición escribió la línea. El procesador recurre a los identificadores de la petición solo cuando no hay ningún span vigente.

Hay un detalle de implementación con una razón detrás: el tracer se resuelve a través de un cierre en **cada** registro en vez de capturarse en la construcción. El procesador se empuja cuando el servicio `log` se resuelve por primera vez, y eso puede ocurrir antes de que el contenedor haya enlazado siquiera un `Tracer` — capturar el nulo significaría no tener identificadores de traza durante el resto del proceso.

Qué canales, y qué es lo que el formateador no toca

`firefly.logging.structured.channels` lista nombres de canal; vacío — el valor por defecto — significa el canal por defecto. El logger de Monolog de un canal `stack` sostiene las instancias de manejador de sus *miembros*, así que formatear la pila formatea también a los miembros, y listar ambos es inofensivo porque aplicar es idempotente.

Lo que nunca cambia son los manejadores mismos. Un `daily` sigue siendo un fichero diario, un `stack` conserva sus miembros, un canal `slack` sigue publicando en Slack; **solo cambia el aspecto de una línea**. Nada se reemplaza, se añade ni se quita.

Un nombre de canal que no existe bajo `logging.channels` se rechaza, y la razón ilustra bien por qué este framework comprueba las cosas: `LogManager::channel()` nunca lanza para un nombre desconocido — captura su propio *"Log [x] is not defined."* y devuelve un logger de emergencia desechable. Sin la comprobación, una errata pondría los procesadores y el formateador sobre un objeto en el que nadie escribe jamás, mientras el canal real seguiría calladamente escribiendo texto plano sin ningún identificador dentro.

⚠ UN CANAL CONSTRUIDO DESPUÉS DEL ARRANQUE NO RECIBE NINGUNO DE LOS DOS

Los procesadores y el formateador se adjuntan **por canal, en tiempo de resolución**. Un canal creado más tarde a través de `Log::build()` no tiene ni los identificadores ni el formateador estructurado. Si construyes canales dinámicamente, aplica el formateador tú mismo — o declara el canal en `logging.channels` y nómbralo en `firefly.logging.structured.channels`, que es la vía que toma todo lo demás.

`gelf` y `logfmt` no se ofrecen. Monolog tiene un `GelfMessageFormatter`, y un canal todavía puede elegirlo a través de la propia clave `formatter` de Laravel.

El panel de administración: `firefly/admin`

Todo lo visto hasta aquí en este capítulo es JSON, y JSON es la forma correcta para un balanceador de carga, una sonda de Kubernetes y un scraper de Prometheus. No es la forma correcta para una persona a las 3 de la madrugada que quiere saber si este proceso compiló sus manifiestos, qué auto-configuración se echó atrás, y a qué resolvió realmente `firefly.data.*`. `firefly/admin` es el paquete para esa persona: un panel de administración renderizado en el servidor sobre esos mismos endpoints del actuador, en el espíritu de Spring Boot Admin. Está incluido en `fireflyframework/laraflly`, así que un proyecto del esqueleto ya lo tiene — y, igual que con `firefly/actuador`, tenerlo instalado no es lo mismo que tenerlo activado. No hace falta instalar otro paquete.

Después abre `/firefly`. No hay paso de npm en la instalación ni CDN en tiempo de petición — las vistas son Blade puro con CSS en línea y tipografías del sistema, porque un paquete de Composer no puede dar por hecho que npm se ha ejecutado, y un panel que necesita la red es inútil precisamente en los entornos aislados donde más quieres mirar uno.

La mayoría de las páginas son una vista sobre la carga útil de un endpoint; cuatro leen el contenedor en su lugar. El menú las agrupa como piensa un operador y no como están dispuestos los paquetes — qué está haciendo ahora mismo, qué cableó en el arranque, cuáles son sus datos, y cómo está configurado — porque una lista plana de diecisiete enlaces es peor menú que cuatro cortas:

Grupo	Página	Lee	Responde
Runtime	Resumen	varios	¿Está sano, qué está haciendo, y qué cableó?
Runtime	Salud	health	Cada indicador que registró este proceso, con su propio estado y detalles
Runtime	Métricas	metrics	Contadores, cronómetros y medidores, con sus mediciones actuales
Runtime	Tráfico HTTP	httpexchanges	Las peticiones más recientes que sirvió esta aplicación
Cableado	Beans	beans	Cada bean que registró el contenedor, con el estereotipo que lo declaró
Cableado	Grafo de beans	beans	Cómo dependen tus beans unos de otros, resueltos a través de las interfaces por las que están cableados

Grupo	Página	Lee	Responde
Cableado	Condiciones	<code>conditions</code>	Qué auto-configuraciones se aplicaron, y cuáles se echaron atrás porque aportaste la tuya
Cableado	Rutas	<code>mappings</code>	La tabla de rutas compilada desde la que sirve el dispatcher
Cableado	Programadas	<code>scheduledtasks</code>	Métodos registrados por <code>#[Scheduled]</code> , con el cron o intervalo que los dispara
Configuración	Entorno	<code>env</code>	La configuración <code>firefly.*</code> resuelta, con los secretos enmascarados
Configuración	Propiedades de configuración	<code>configprops</code>	Cada DTO <code>#[ConfigProperties]</code> que enlazó la aplicación, con los valores que resolvió
Configuración	Cachés	<code>caches</code>	Los almacenes de caché que tiene configurados esta aplicación

Grupo	Página	Lee	Responde
Configuración	Loggers	loggers	Canales de log y sus niveles, con un control para cambiar uno

Una página cuyo endpoint no está registrado en este proceso — o está apagado — queda **oculta del menú** en lugar de ofrecerse como un enlace que aterriza en una disculpa. Eso importa porque los endpoints del actuador son condicionales: `metrics` desaparece cuando `firefly.observability.metrics.enabled` es falso, y varios otros existen solo si está instalado el paquete que los aporta. El menú tiene que construirse a partir de lo que este proceso registró de verdad, y así se hace.

Lee los endpoints en proceso, no por HTTP

El panel sostiene el `ActuatorRegistry` e invoca cada bean `ActuatorEndpoint` directamente:

```
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;
}
// ...
}

```

Fíjate en lo que **no** hay en ese método: ninguna mención a `ExposureModel`. Eso es lo más importante que hay que entender de este paquete, y es deliberado.

`firefly.management.endpoints.web.exposure.include` vale `health,info` por defecto, así que traer `/actuator/beans` o `/actuator/env` por HTTP daría un 404 — como toda la primera mitad de este capítulo insistió en que debía ser. El panel no necesita nada de eso. Renderiza lo que el proceso ya sabe, en proceso, de modo que **te muestra páginas que la superficie HTTP deliberadamente no expone**, y la

superficie JSON sigue siendo segura por defecto. Exponer `beans`, `conditions` y `env` a cualquier llamante anónimo solo para que un navegador pudiera leerlos sería exactamente el intercambio equivocado.

El interruptor de apagado por endpoint *sí* se honra, y la asimetría es el quid: `firefly.management.endpoint.{id}.enabled` significa "este endpoint está apagado", que es una afirmación sobre el endpoint en sí; `exposure.include` significa "este endpoint no está publicado", que es una afirmación sobre la superficie HTTP. El panel no es la superficie HTTP.

Un endpoint que lanza se captura y se reporta como `null` en vez de dejar que se lleve la página por delante — la misma disciplina a prueba de fallos que `HealthEndpoint::readFailSafe()` aplica a los indicadores, y por la misma razón: un contribuyente roto debe degradar su propio panel, no el panel entero.



LOS DETALLES DE SALUD SE LEEN DEL REGISTRO DE CONTRIBUYENTES, NO A TRAVÉS DEL ENDPOINT

`show-details` vale `never` por defecto, y ese valor es el correcto — impide que un llamante HTTP anónimo aprenda el host de tu base de datos a partir de una conexión fallida. Aplicar esa política de divulgación HTTP al panel, sin embargo, producía un panel de Salud cuyo contenido entero era una disculpa que le decía al operador que fuera a cambiar una clave de configuración. El panel lee el `HealthContributorRegistry` directamente en su lugar, llamando a cada indicador de forma aislada, de modo que uno que lance se reporta DOWN con su motivo y nada más se ve afectado.

El explorador de beans

`/firefly/graph` abre con búsqueda, puntos de entrada, beans con más dependientes, acoplamiento entre módulos y ciclos del cableado. `?bean=<identidad cualificada>` muestra un entorno acotado; `?module=<namespace>` muestra los beans

del módulo y sus relaciones externas; `?q=<término>` busca en el catálogo completo. `/firefly/beans` incluye componentes, productos de fábrica y DTOs de configuración enlazados, ordenados inicialmente por número de dependientes descendente.

El bean seleccionado queda entre sus dependientes a la izquierda y sus dependencias a la derecha. PHP calcula las posiciones: cajas de 176×38 píxeles, separadas cada 204×46 píxeles, con un máximo predeterminado de 16 filas por columna y 72 nodos. Dos saltos ocupan 992 píxeles de ancho y hasta 728 de alto. Las cabeceras se desplazan con las columnas. Los enlaces nativos, formularios GET, paginación y foco visible de teclado funcionan sin JavaScript; no hay cámara ni ajuste de escala. Texto y siglas identifican los módulos además del color decorativo.

La selección reparte cada columna por turnos entre los nodos de la frontera. Los enlaces de desbordamiento abren las tablas completas y paginadas de vecinos directos del origen, con interfaz y tipo de relación. Las cadenas más cortas desde puntos de entrada explican la alcanzabilidad. Se conservan todas las condiciones positivas y negativas, incluidas las de la configuración productora; la ausencia del endpoint Conditions se tolera.

Cada relación entre módulos muestra peso (relaciones resueltas), beans destino distintos, interfaces distintas y relaciones concretas. Las aristas de producción se excluyen salvo petición explícita. Las listas siguen paginadas cuando se supera el presupuesto del mapa. El módulo señala la alcanzabilidad exclusiva y los beans sin dependientes; son observaciones del catálogo, no una garantía de que retirar un módulo sea seguro.

Tarjan iterativo identifica componentes fuertemente conexos, incluida la autoinyección, sin recorridos recursivos. Un ciclo que incluye producción de fábricas no implica necesariamente un ciclo de constructores en ejecución. Las fábricas competidoras mantienen identidades separadas incluso en configuraciones diferentes. La proyección

no aporta los metadatos `primary/qualifier` necesarios: los destinos ambiguos quedan sin resolver y el ámbito de fábrica se muestra como no informado. Se excluyen los DTOs explícitamente no enlazados. No se instancia ningún bean para inspeccionarlo.

Migración: `firefly.admin.graph.max-nodes` sigue leyéndose en `AdminSettings::$graphMaxNodes` (220 por defecto), pero ya no controla el dibujo. **0 ya no fuerza una lista.** Para una vista tabular utiliza el catálogo completo o las tablas de relaciones.

Las claves nuevas son `firefly.admin.graph.focus.depth` (2, límites 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 desactiva el mapa) y `firefly.admin.beans.page-size` (50, 10–500). Los tamaños de página también respetan los límites compartidos de tablas. Todas están documentadas en `skeleton/config/firefly.php`.

El modelo de acceso es toda la frontera de seguridad

Como el panel sortea la exposición, su propia URL es lo único que se interpone delante de `beans`, `env` y `conditions`. Por eso no debe estar encendido por defecto en producción, y por eso la bandera de activación está escrita como está:

```
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` toma por defecto el valor de `app.debug`. El razonamiento es que una aplicación que ya corre con debug encendido ya está sirviendo trazas de pila a quien las pida y es un entorno de desarrollo por definición, así que un panel ahí no divulga nada que no estuviera ya divulgado. Una aplicación con debug apagado ha hecho la afirmación contraria sobre sí misma, y debe optar por él explícitamente. Fijar la clave siempre gana sobre el valor por defecto de debug, en ambas direcciones — puedes apagar el panel en un entorno con debug, y encenderlo en uno de producción.

⚠ ENCENDERLO FUERA DE DEBUG ES SOLO LA MITAD DEL TRABAJO

`firefly.admin.enabled = true` con `app.debug = false` monta un panel que renderiza tu grafo de beans, tu configuración resuelta y tu tabla de rutas en una URL conocida, para cualquiera que pueda alcanzarla. El panel no trae **ninguna autenticación propia** — no tiene dependencia de código con `firefly/security` en absoluto, exactamente igual que `firefly/actuator`. Una aplicación que lo encienda fuera de debug **debe poner la ruta detrás de su propio middleware de autenticación**.

Las reglas de `HttpSecurity` del Capítulo 10 lo hacen como pura configuración, del mismo modo que este capítulo ya aseguró el actuador — aquí tienes un despliegue que opta por el panel y le echa el candado en un solo fichero:

```
<?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` monta las dos rutas — un índice y un comodín `GET|POST {base}/{page}` — como un `BootPass` en `WiringPasses`, orden **60**, un paso por detrás del 50 de `ActuatorRouteRegistrar`, porque lee el registro que puebla ese pase. Se registran nativamente sobre el `Router` de Illuminate por la misma razón que las rutas del actuador y las del paquete OpenAPI: `firefly.admin.base-path` tiene que

ser fijable por aplicación, y una ruta por atributo hornea su ruta literal dentro de un `RouteDescriptor` compilado. Cuando el panel está deshabilitado el pase no registra *absolutamente nada* — no hay ruta que adivinar ni manejador al que llegar.

También se echa atrás en silencio en un caso más, fácil de pasar por alto. Blade es necesario para renderizar el panel y no es una dependencia del paquete, así que un despliegue solo-JSON sin factoría de vistas enlazada no obtiene rutas en vez de rutas que fallarían fatalmente en la primera petición; allí la superficie de gestión sigue siendo el actuador JSON.

⚠ TRES COSAS QUE EL PANEL SOLO PUEDE MOSTRARTE DE ESTE PROCESO

Bajo PHP-FPM cada petición es un proceso distinto, y tres páginas heredan eso.

Cambiar un nivel de log llama al mismo endpoint que

`POST /actuador/loggers/{name}`, que muta los manejadores de Monolog del proceso actual — la siguiente petición es otro proceso, así que cambia `logging.channels` para cualquier cosa que deba persistir. Las **métricas** son solo tan duraderas como el registro: el `SimpleMeterRegistry` por defecto guarda los medidores en memoria de proceso, así que el panel ve solo su propia petición salvo que

`firefly.observability.metrics.store` apunte a un almacén de caché. Y los **detalles de salud** siguen ocultos en la respuesta JSON `/actuador/health` hasta que `firefly.management.endpoint.health.show-details` sea `always` — o sea `when-authorized` y el `HealthDetailsAuthorizer` admita al llamante —, aunque la propia página de Salud del panel lea los indicadores directamente.

El navegador de datos, y por qué no hereda ese valor por defecto

`firefly/admin` incluye una superficie más, y es la única de este capítulo cuya puerta está escrita de forma distinta a todas las que has visto. Es un **navegador de base de datos** al estilo del admin de Django sobre la capa de datos del Capítulo 5 — listado, detalle, búsqueda, ordenación y paginación sobre tus propios repositorios — al que se

llega a través de un único `DataBrowser` que

`DataBrowser::forContainer($container)` ensambla desde el contenedor de la aplicación.

Descubre qué navegar igual que el resto del panel descubre todo lo demás — desde el catálogo compilado. **Todo bean cuya lista de interfaces de tiempo de escaneo contenga `CrudRepository` es un recurso navegable.** No se registra nada ni se declara nada: un repositorio que escribas es navegable en cuanto el contenedor lo tiene, y uno que borres deja de serlo sin que nadie edite una lista. Cada fila de `BeansCatalog` ya lleva el cierre completo de interfaces que `ComponentScanner` registró con `class_implements()`, así que «¿es este bean un repositorio, y además pagina?» son dos llamadas a `in_array()` sobre datos que el proceso ya tiene — sin reflexión y, de forma decisiva, sin posibilidad de ofrecer un recurso que el contenedor nunca registró.

Ahora la puerta:

```
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;
}
// ...
}
```

Fíjate en los dos valores por defecto, y compáralos con el `$config->bool('app.debug', false)` de `AdminSettings` unas páginas más arriba. El panel sigue a `app.debug`, y el argumento para eso era sólido **para lo que el panel muestra**: beans, condiciones, mapeos y configuración resuelta son hechos sobre la *aplicación*, y una aplicación que ya sirve trazas de pila ya ha publicado hechos de esa clase.

Esta página muestra hechos sobre los **usuarios** de la aplicación. Esa es una divulgación categóricamente mayor, y los errores que la exponen son los ordinarios, los que hoy no cuestan nada: una bandera de depuración olvidada en un entorno de staging que comparte base de datos con producción, un `.env` copiado a una máquina que debía ser interna, un portátil tunelizado para una demo. Cada uno se convierte en una divulgación de registros de clientes en cuanto hay un navegador de datos atado a `app.debug`. Así que la puerta es aparte, explícita y está cerrada — **app.debug no puede abrirla, y `firefly.admin.enabled` tampoco**. Las tres deben ser ciertas.

Las escrituras necesitan entonces una *segunda* clave, y por sí sola no sirve de nada. Leer la fila equivocada es una divulgación; borrarla es pérdida de datos sin deshacer, desde un formulario, sobre una sesión que puede no ser más que «debug estaba encendido». Encender el navegador es una decisión sobre **visibilidad**; encender las escrituras es una decisión sobre **custodia**. Si las colapsas en una sola clave, quien quería mirar una tabla ha armado también el botón de borrar.

⚠ CREATE EXISTE PARA ELOQUENT, Y SE RECHAZA PARA TODO LO DEMÁS

El navegador no tuvo `create()` durante un tiempo, y el argumento era medio cierto. Un formulario de creación genérico sobre una entidad arbitraria es una promesa que no puede cumplir, y el Capítulo 6 explica por qué: **el constructor de un agregado es donde viven sus invariantes**. Un `Order` que debe tener al menos una línea, un `Wallet` cuyo saldo empieza a cero en la moneda con la que se abrió, un objeto valor que rechaza un IBAN mal formado — un formulario construido a partir de una lista de columnas no conoce ninguno. Solo hay dos formas de construir esa fila: llamar al constructor, que necesita argumentos que el formulario no puede suministrar con los tipos ni en el orden correctos; o escribir las columnas directamente en la tabla, lo que produce una fila que el modelo de dominio considera imposible. Lo segundo es lo que hace una implementación de «pues inserta las columnas», y es *peor que no tener botón*, porque parece que funcionó. **Ese caso se sigue rechazando, por su nombre.**

Nunca fue cierto para un modelo **Eloquent**, que se construye vacío y se rellena por atributo — exactamente lo que `update()` lleva haciendo siempre sobre una fila que ya existe. Create rechazaba por un riesgo que update ya estaba asumiendo, y la inconsistencia le costaba a toda aplicación una superficie CRUD que se quedaba en RUD. Así que se ofrece para un recurso respaldado por Eloquent bajo los dos mismos interruptores, con el identificador y cualquier columna enmascarada *omitidos del formulario* en lugar de deshabilitados en él: un campo que el navegador se negaría a escribir no debería aparentar que lo acepta.

Merece la pena llevarse otras dos decisiones de esta sección, porque ambas parecen un detalle y no lo son.

El identificador y cualquier secreto enmascarado se rechazan como destino de una actualización — y se rechazan dos veces, una para que la vista pueda dibujar el campo como solo lectura y otra en la ruta de escritura, de modo que un POST fabricado a mano no alcance lo que el formulario no ofrecía. Recodificar la clave de una fila desde un formulario genérico no es una edición, es otra fila, y las claves foráneas que apuntaban al valor antiguo no la siguen. El valor *mostrado* de un secreto

es `*****`, así que devolver un formulario renderizado escribiría la máscara sobre la credencial real — un fallo de pérdida de datos creado por el propio enmascarado. Los secretos se excluyen de la **búsqueda** por una razón emparentada: una caja que responde «sí, el `api_token` de alguna fila empieza por `sk_live_9`» es un oráculo que un operador puede recorrer carácter a carácter.

Ningún texto de error que la página muestre es jamás un mensaje de excepción.

La `QueryException` de Laravel convierte a cadena el SQL fallido y *sus bindings* dentro de `getMessage()`, así que reproducirlo publicaría el esquema y los valores enlazados — que en una búsqueda sobre una tabla de usuarios es la propia consulta del operador, y en una consulta de detalle es una clave primaria. Toda razón es una frase fija compuesta en la capa del navegador, más como mucho el nombre de clase de la excepción; el mensaje se queda en la excepción, donde el log puede tenerlo. Por eso mismo nada en la capa lanza hacia su llamador: las lecturas responden con un listado que lleva una razón, las escrituras con uno de cuatro resultados (`Done`, `Refused`, `NotFound`, `Failed`), y una vista que dibuja una página de administración nunca tiene que ser a prueba de excepciones para mantenerse en pie.

LA RUTA DE LISTADO QUE OBTIENES DEPENDE DE LA INTERFAZ QUE IMPLEMENTASTE

Un `PagingAndSortingRepository` se pagina **en la base de datos**: el repositorio hace el desplazamiento, el límite, el `ORDER BY` y el `COUNT`, y el coste es independiente del tamaño de la tabla. Un `CrudRepository` simple no puede expresar nada de eso, así que el navegador llama a `findAll()`, ordena y corta **en PHP**, y descarta todas las filas menos 25 — lo que con diez mil filas es una página lenta y con diez millones es un agotamiento de memoria que mata al worker, al *primer* clic. La interfaz no tiene límite, ni desplazamiento, ni conteo con predicado, así que las opciones honestas eran «negarse a navegar repositorios que no pueden paginar» o «navegarlos y decir lo que cuesta». LaraFly hace lo segundo, y esto es el decirlo. Implementa `PagingAndSortingRepository` en todo lo que pretendas navegar contra una tabla real.

Recorrer el modelo: relaciones, filtros y un mapa

Un listado que solo puedes desplazar es un volcado de tabla. Tres cosas lo convierten en algo que exploras.

Las relaciones se descubren LLAMANDO a los métodos que declaran una, porque es la única forma de saber por qué columnas unen — el nombre de un método no dice nada y su tipo de retorno solo dice de qué clase es. Lo que convierte «qué es seguro llamar» en la pregunta que sostiene todo, y la respuesta es el **tipo de retorno declarado**: solo se llama a un método público, no estático y sin argumentos cuyo tipo de retorno sea una subclase de `Relation` de Eloquent. Un método que anuncia `HasMany` es una definición de relación por construcción — la misma señal en la que se apoyan la validación de `with()` y las herramientas de IDE de Laravel — y un accesor no puede reclamarla sin mentir sobre su propia firma. La llamada no ejecuta consulta alguna; Eloquent la difiere hasta `get()`.

Un `belongsTo` abre el único registro padre; un `hasMany` abre el listado hijo **filtrado por la clave de esta fila**, que es para lo que la página de registro necesita un filtro. Una relación cuyo otro extremo no es un recurso navegable se sigue mostrando — te dice la forma del modelo — pero no se enlaza, y esa distinción vive en el modelo y no en la plantilla para que una vista no pueda acuñar una URL que da 404.

Filtrar son ocho comparaciones sobre las columnas que el recurso ya publica — `is`, `is not`, `contains`, `starts with`, `greater than`, `less than`, `is empty`, `is not empty`, que el panel rotula así, en inglés, y no traduce (es, no es, contiene, empieza por, mayor que, menor que, está vacío y no está vacío) — expresadas como una URL GET que puedes guardar en marcadores o pegar en un ticket. Toda comparación enlaza su valor como parámetro, incluidas las de `LIKE`, donde los comodines van alrededor de un valor *escapado* en vez de meter el valor dentro de un patrón. Una columna que el esquema no publica y un operador fuera de ese conjunto se **descartan** en lugar de pasarse al driver: ambos llegan en una URL que un operador

puede editar a mano, y una consulta que alcanza el driver con un identificador arbitrario dentro es, como poco, un oráculo de nombres de columna. Las condiciones se combinan con AND entre sí y con la caja de búsqueda, así que estrechar el listado de una relación no puede escaparse de ella.

Las mismas ocho están implementadas para el camino de reserva en PHP, porque un repositorio que no puede paginar debe filtrarse por las mismas reglas que uno que sí. Dos implementaciones de un predicado divergen, y la divergencia se manifiesta como un filtro que significa cosas distintas en recursos distintos.

`/firefly/data-map` lo dibuja. Cada entidad navegable como una caja con sus columnas, cada clave foránea como una arista etiquetada, y cada caja enlazando a sus propios registros. Un `hasMany` y el `belongsTo` que lo mira de frente son *una* clave vista desde dos extremos, así que cada una se dibuja una vez — apuntando desde la tabla que *tiene* la clave hacia la tabla a la que referencia, que es además lo que la flecha significa.

Dos páginas más, y la que sí cambia cosas

`/firefly/datasource` responde a lo que un volcado de configuración no puede. ¿Con qué base de datos estoy hablando (driver, host, base, con `password` enmascarada por el mismo enmascarador que usa el endpoint `env`)? ¿Está *arriba* — se prueba una conexión por carga de página, porque abrir un socket puede quedarse colgado contra un host tras un cortafuegos y una página que abriera todas las conexiones configuradas tardaría en renderizar el timeout de la más lenta, justo en la página que abriste *porque* algo va mal. ¿Qué significa el pooling aquí — PHP no tiene pool de conexiones, y en vez de inventar un indicador la página informa del `ATTR_PERSISTENT` de PDO por lo que es, y dice que bajo php-fpm el tamaño efectivo del pool es tu número de workers. Y a qué compiló `#[Transactional]`, que hasta ahora solo existía como un artefacto bajo `bootstrap/cache`.

`/firefly/settings` es la única página del panel que *cambia* la aplicación en lugar de describirla, y está protegida en consecuencia: `firefly.admin.settings.enabled` (desactivada por defecto, a diferencia de todo lo demás aquí), `.writable` encima, y una tercera puerta que **no es una clave de configuración** — en producción toda escritura se rechaza diga lo que diga el resto. Eso último es deliberado. Es la diferencia entre «lo hemos hecho seguro» y «lo hemos hecho configurable para que sea seguro», y solo lo primero sobrevive a que alguien copie un `.env`.

Es un *interruptor de funcionalidad*, no un endpoint de configuración remota: la lista es fija y propiedad del framework, así que un POST fabricado que nombre `app.key` o el host de una base de datos no encuentra nada que escribir. Un cambio va a un único fichero JSON bajo `bootstrap/cache`, filtrado tanto a la entrada como a la salida, y se mezcla sobre la configuración en el `register()` del provider — no en un boot pass, y esa distinción costó una sesión de depuración. Todo objeto de settings de este framework se construye una vez desde la configuración y se retiene, así que aplicar los overrides desde el propio pase del panel escribía el fichero y mostraba el nuevo estado en la página mientras `/openapi.json` seguía respondiendo 200. `register()` corre antes de que se resuelva ningún bean, que es el único punto en el que la mezcla es cierta.

📦 PARIDAD CON LARAVEL

Laravel puro no trae ningún endpoint de comprobación de salud ni de métricas en absoluto — la mayoría de los equipos o bien improvisan una ruta `/health` a mano o recurren a un paquete de terceros, normalmente emparejado con la extensión `ext-prometheus`. `firefly/actuator` y `firefly/observability` son análogos de primera parte y con pocas dependencias de Spring Boot Actuator y Micrometer respectivamente: endpoints de framework montados sobre el mismo `Router` que tu app ya usa, comprobaciones de salud que reutilizan por debajo los propios `DB / Log / config` de Laravel, y un exportador Prometheus en PHP puro sin requisito de extensión. Ambos paquetes son dependencias Composer opcionales y ambos son seguros por defecto — una app que añade `firefly/actuator` obtiene `health / info` y nada más hasta que configure más. `firefly/admin` completa el conjunto como análogo de Spring Boot Admin, con la diferencia de que no es una aplicación de monitorización aparte que despliegas y en la que registras instancias: son vistas Blade dentro de la propia aplicación sobre la que informan, que es por lo que puede leer el registro directamente y por lo que su modelo de acceso importa tanto como importa.

✓ Lo que aprendiste

Concepto	Qué hace
<code>HealthIndicator</code>	SPI de un método; un bean <code>#[Component]</code> descubierto y agregado automáticamente
<code>Health</code> / <code>Status</code>	Lectura inmutable + un enum ordenado por severidad; <code>DOWN/OUT_OF_SERVICE</code> se mapean ambos a HTTP 503
<code>PingHealthIndicator</code> / <code>DiskSpaceHealthIndicator</code> / <code>DbHealthIndicator</code>	Sondeo de liveness siempre-arriba; comprobación de disco basada en umbral; comprobación de BD <code>SELECT 1</code> activa por defecto, que declina su registro mediante <code>ConditionalHealthIndicator::available()</code> cuando no hay conexión por defecto configurada
<code>HealthEndpoint</code>	Agrega al estado más severo; un grupo de sondeo es solo un subconjunto de indicadores con nombre y configurado — no hay una clase de endpoint liveness/readiness separada
<code>ExposureModel</code>	Puerta CSV <code>include / exclude</code> ; por defecto <code>"health,info"</code> ; todo lo demás es un 404 simple hasta que se exponga
<code>EnvEndpoint</code> / <code>SensitiveValueMasker</code>	Una única regla compartida enmascara las claves <code>password\ secret\ token\ key\ credential\ passwd\ authorization\ headers</code> con <code>*****</code> , independientemente de la exposición — la clave decide primero, así que una clave sensible enmascara todo su subárbol

Concepto	Qué hace
<code>MeterRegistry</code> / <code>MetricsRecorder</code>	Puertos de métricas de lectura/escritura; <code>SimpleMeterRegistry</code> es idempotente por <code>type\ name\ tags</code> y falla ruidosamente ante un conflicto de tipo
<code>PrometheusTextFormat</code>	Exposición a prueba de locale — <code>number_format()</code> , nunca <code>sprintf('%f')</code>
<code>DistributionStatisticConfig</code>	Buckets de histograma por nombre de medidor, fijados cuando se crea el temporizador; sin buckets por defecto, y una lista por medidor vacía devuelve ese medidor a ser un summary
<code>firefly.logging.structured.format</code>	<code>json</code> , <code>ecs</code> o <code>logstash</code> sobre los canales que nombres — solo cambia el formato de una línea, nunca los manejadores, y un formato o un canal desconocido es un fallo de arranque
<code>TraceContextLogProcessor</code>	<code>trace_id</code> , <code>span_id</code> , <code>correlation_id</code> y <code>request_id</code> en cada registro; los identificadores del span vigente cuando hay uno, si no los de la petición
<code>MetricsFilter</code>	Filtro de cronometraje más externo <code>#[Order(-100)]</code> ; etiqueta por la plantilla de la ruta, nunca la ruta en bruto — cardinalidad acotada
<code>ObservabilityAutoConfiguration</code> <code>#[Order(500)]</code>	El mismo truco de precedencia que la costura de seguridad del Capítulo 10: registra <code>cqrsMetrics()</code> antes de que <code>CqrsAutoConfiguration</code> evalúe su <code>#[ConditionalOnMissingBean]</code>

Concepto	Qué hace
<code>firefly/admin</code>	Un panel Blade renderizado en el servidor en <code>/firefly</code> ; una cuyo endpoint no está registrado o está apagado se oculta del menú en lugar de enlazarse
<code>AdminEndpointReader</code>	Invoca cada <code>ActuatorEndpoint</code> en proceso desde el <code>ActuatorRegistry</code> , sorteando <code>ExposureModel</code> — así el panel muestra lo que la superficie HTTP no expone, y un endpoint que lanza degrada un solo panel
<code>BeanGraph</code>	Convierte el catálogo de beans en un grafo de dependencias dibujado sobre tres clases de nodo — componentes, productos <code>#[Bean]</code> y DTOs <code>#[ConfigProperties]</code> — con aristas <code>injects / produces</code> resueltas a través de un índice de interfaces (marcadas <code>via</code>), estratificación por camino más largo, ciclos reportados en lugar de colgarse, y exploración acotada por saltos con SCCs iterativos
Los productos <code>#[Bean]</code> como nodos	El cableado de un framework vive en métodos fábrica, no en constructores; con solo las clases declarantes como nodos, un esqueleto de serie dibujaba una arista de 42 beans
<code>ComponentDescriptor::\$dependencies</code>	Las aristas del grafo, registradas por <code>ComponentScanner</code> en tiempo de escaneo — solo tipos de clase e interfaz, porque un parámetro escalar es configuración, no cableado

Concepto	Qué hace
<code>firefly.admin.enabled</code>	Toma por defecto <code>app.debug</code> ; un valor explícito gana en ambas direcciones, y encenderlo con debug apagado te obliga a poner tu propio middleware de autenticación delante de la ruta
<code>firefly.admin.data.enabled</code>	La puerta propia del navegador de datos, con valor por defecto <code>false</code> — <i>no</i> sigue a <code>app.debug</code> ni a <code>firefly.admin.enabled</code> , porque esta página muestra hechos sobre los usuarios de la aplicación y no sobre la aplicación
<code>RelationIntrospector</code>	Encuentra relaciones LLAMANDO solo a los métodos cuyo tipo de retorno declarado es una <code>Relation</code> de Eloquent, para que un registro enlace con lo que referencia y el mapa de entidades tenga aristas que dibujar
<code>DataFilter</code>	Ocho comparaciones sobre las columnas que el recurso publica, siempre enlazadas como parámetro, y con una columna u operador desconocido descartado en lugar de pasado al driver
<code>/firefly/datasource</code>	Conexiones con los secretos enmascarados, una probada por carga, la persistencia de PDO contada por lo que es, y el contrato <code>#[Transactional]</code> compilado
<code>/firefly/settings</code>	La única página que cambia la aplicación: desactivada por defecto, escribible con una segunda clave, y rechazada en producción por una puerta que ninguna clave levanta

Concepto	Qué hace
<code>firefly.admin.data.writable</code>	Una segunda puerta, también <code>false</code> e inútil por sí sola: visibilidad y custodia son decisiones distintas, y una sola clave armaría el botón de borrar para quien solo quería mirar una tabla
Sin <code>create()</code>	Permanente, no pendiente: las invariantes de un agregado viven en su constructor, y un formulario construido a partir de una lista de columnas no puede satisfacerlas — escribir las columnas de todos modos produce una fila que el dominio considera imposible

Ponlo en práctica

1. **Añade un `HealthIndicator` personalizado.** Escribe un `#[Component]` que implemente `HealthIndicator` que compruebe algo específico de tu propia app (un feature flag, la profundidad de una cola, una conexión de caché) y confirma que aparece en el mapa `components` de `GET /actuador/health` una vez que `show-details` esté configurado a `always`.
2. **Configura una división `liveness/readiness` real.** Añade `firefly.management.endpoint.health.group.liveness.include = 'ping'` y `...readiness.include = 'ping,db'` (con el indicador de BD habilitado) a la configuración de un proyecto de pruebas, y confirma que `GET /actuador/health/liveness` y `GET /actuador/health/readiness` divergen en el momento en que dejas la base de datos inalcanzable.

3. **Observa a la costura de métricas de CQRS ganar la carrera.** Usa el componente incluido `firefly/observability` en un proyecto LaraFly de pruebas que ya use `firefly/cqrs`, envía un puñado de comandos, e inspecciona `GET /actuador/prometheus` en busca de muestras de `cqrs_commands_seconds` — luego comenta temporalmente el atributo `#[Order(500)]` de `ObservabilityAutoConfiguration` (revirtiendo al valor por defecto de la clase) y confirma si la métrica todavía aparece, para ver el truco de ordenamiento importar de verdad en lugar de solo leer sobre él.
4. **Demuéstrate a ti mismo el sorteo de la exposición.** Activa el componente incluido `firefly/admin` con `firefly.admin.enabled` en el sample, deja `firefly.management.endpoints.web.exposure.include` en su valor por defecto, y confirma que `GET /actuador/beans` devuelve un `404` mientras `/firefly/beans` renderiza la lista completa de beans en el mismo proceso. Luego pon `firefly.management.endpoint.beans.enabled` a `false` y confirma que la entrada Beans desaparece del menú del panel — el interruptor de apagado se honra allí donde la exposición no, y la diferencia entre ambas claves es todo el diseño.
5. **Dibuja tu propio cableado y luego rómpelo.** Abre `/firefly/graph` en el sample y encuentra la flecha de `WalletService` a `EloquentWalletRepository` — fíjate en que la columna *Wired by* dice `WalletRepository`, el puerto, y no `class`. Después introduce un ciclo deliberado (haz que un `#[Service]` tome un parámetro de constructor tipado como otro `#[Service]` que ya depende de él), recarga la página, y confirma que la estadística **Cycles** se pone en rojo y nombra ambas clases. Ahora arranca la app de cero sin abrir el panel, y compara lo que PHP te cuenta sobre ese mismo ciclo.
6. **Lee el valor por defecto de acceso como una decisión de seguridad.** Pon `app.debug` a `false` en un proyecto de pruebas con `firefly/admin` instalado y confirma que `/firefly` está genuinamente sin enrutar y no simplemente sin enlazar (`php artisan route:list` no debería listarla). Luego pon

`firefly.admin.enabled` a `true` sin añadir ninguna regla de `HttpSecurity`, y mira qué divulga ahora un `GET /firefly/env` sin autenticar — esa es exactamente la brecha que este capítulo te dijo que cerrarías con tu propio middleware de autenticación.

7. **Convierte un summary en algo de lo que puedas sacar un p99.** Haz un scrape de `GET /actuator/prometheus` en un proyecto con `firefly/observability` instalado y busca la familia `http_server_requests_seconds`: fíjate en la línea `# TYPE ... summary` y en que todo lo que tienes es `_count` y `_sum`. Después pon `firefly.observability.metrics.distribution.buckets` con la lista por defecto de los clientes que este capítulo imprime, vuelve a hacer el scrape, y confirma que la línea `# TYPE` ahora dice `histogram` y que apareció una serie `_bucket{le=...}` por cada cota. Ahora silencia solo ese medidor otra vez con una lista `distribution.per-meter.http_server_requests_seconds` **vacía**, y confirma que todos los demás temporizadores conservan sus buckets. Por último mete un `0` — o un `-1`, o una cadena — en la lista global y confirma que el arranque lo rechaza por su nombre en vez de descartar la cota calladamente.

8. **Haz tus logs legibles por una máquina, y luego lee una línea.** Pon `firefly.logging.structured.format` a `ecs`, haz una petición que escriba una línea de log desde dentro de un manejador de comando, y lee la línea resultante: busca `trace.id`, `span.id` y `labels.correlation_id`, y confirma que el identificador de span es el **del manejador**, no el de la petición, comparándolo con una línea escrita desde un controlador corriente en la misma petición. Después comprueba que tus propias claves de contexto aterrizaron bajo `context / extra` en vez de en el nivel superior — prueba a registrar una clave llamada literalmente `message` y confirma que no sobrescribió la de ECS. Por último, pon la clave a `jsonn` y confirma que la aplicación se niega a arrancar, nombrando la clave, en vez de recurrir en silencio a texto plano.

Probar Aplicaciones LaraFly

Al terminar este capítulo conocerás las dos familias de arnés de arranque de `firefly/testing` — `FireflyTestCase` / `FireflyDatabaseTestCase` para pruebas de clase al estilo de Testbench, y `bootFireflyApp()` / `fireflyApplication()` para arranques desnudos sin ninguna cola de framework de pruebas — los once dobles de grabación que permiten a una prueba afirmar sobre lo que un puerto de Firefly realmente vio (algo que los propios `Event::fake()` / `Bus::fake()` de Laravel no pueden hacer, porque nunca han oído hablar de los puertos de Firefly), las cinco expectativas de Pest con sabor a Firefly, el par `WebSliceTestCase` / `DataSliceTestCase` que arranca solo los beans que una prueba necesita, la suite de navegador de Pest 4 que conduce el **esqueleto que se entrega** en el mismo proceso — mismo contenedor, misma base de datos SQLite, sin un segundo servidor — y por qué `assertNoJavaScriptErrors()` es la aserción que se lo gana, los ayudantes que autentican un test (`actingAsAuthentication()` y el par `actingAsPrincipal()` / `actingAsOidcUser()` construido sobre él, el declarativo `#[WithMockUser]`, y `withoutSecurity()` para la dirección contraria), y — porque este libro solo enseña lo que realmente se entrega — una limitación real y honesta con la que se toparon las propias pruebas de `samples/lumen` y cómo la sortearon.

📌 TÉRMINO NUEVO: PRUEBA DE REBANADA (SLICE TEST)

Una **prueba de rebanada** arranca solo la estrecha vertical del framework que una prueba realmente ejerce — el pipeline web sobre un controlador, o el pipeline de datos sobre un repositorio — en lugar de toda la aplicación. Es la idea de `@WebMvcTest` / `@DataJpaTest` de Spring Boot: arranques más rápidos, y una rebanada mal cableada falla de inmediato en lugar de funcionar silenciosamente por accidente porque algún bean no relacionado resultó estar presente también.

El arnés de arranque: `FireflyTestCase`

Cada base de prueba de Firefly improvisada a mano por todo el monorepo solía volver a derivar el mismo puñado de reglas de orden de arranque a mano. `FireflyTestCase` las absorbe de una vez, como una subclase de `Testbench\TestCase` con exactamente tres hooks que sobreescribir:

```
/**
 * 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()` devuelve configuración ansiosa, con claves punteadas, sembrada **antes** del arranque, para que una pasada de `#[ConditionalOnProperty]` o `#[ConditionalOnMissingBean]` — que escanea en tiempo de *registro* — la observe de verdad. Esa es la costura para `firefly.scan.paths` y para cualquier indicador `firefly.<feature>.*` que un test necesite encendido. Cada uno de los tres hooks tiene además una forma a nivel de clase: `#[FireflyTest(providers: [...], config: [...])]` sobre la clase de test hace el mismo trabajo sin una sobrescritura.

La clase base se encarga de las partes que cada prueba solía equivocar sutilmente: `FireflyAutoConfigureServiceProvider` se registra *siempre* primero (el propio `register()` de cada proveedor de capacidad asume que el kernel ya existe), `configOverrides()` se aplica antes del arranque de modo que un `#[ConditionalOnProperty]` que escanea en tiempo de registro realmente lo observa, y el canal de log se cambia a `errorlog` de modo que un sandbox de solo lectura nunca tropiece con un logger respaldado por el sistema de archivos. Una subclase mínima no necesita más que el único hook que realmente usa:

```
<?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'),
        ],
```

```

    ];
}
}

```

Dos pequeñas barandillas rematan la base: `app(): Application` lanza una `LogicException` clara si la llamas antes del arranque, en lugar de devolverte un `null` sin tipo, y `fireflyContext(): ApplicationContext` te devuelve directamente la misma fachada del motor de arranque que presentó el Capítulo 2, ya resuelta.

Pruebas de base de datos: `FireflyDatabaseTestCase`

`FireflyDatabaseTestCase` extiende `FireflyTestCase` y mezcla `UsesSqliteMemory`, que enlaza una conexión sqlite `:memory:` compartida como el `testing` por defecto — sin Docker, sin archivo de base de datos de fixtures, sin estado entre pruebas:

```

<?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` es un envoltorio fino sobre `Schema::create()` — nada nuevo que aprender si ya conoces las migraciones de Laravel, solo un camino más corto desde "necesito una tabla para esta única prueba" hasta un esquema en funcionamiento.

Arranques desnudos: `bootFireflyApp()` y `fireflyApplication()`

No toda prueba quiere un `TestCase` de Testbench completo.

`Firefly\Testing\functions.php` — autocargado con Composer `files`, de modo que estas están siempre disponibles sin ninguna sentencia `use` — trae una segunda familia para un arranque desnudo:

```
function fireflyApplication(array $config = [], array $providers = [], array $bindings = [],
array $needs = []): Application
// ...
function bootFireflyApp(array $config = [], array $providers = [], array $bindings = [],
array $needs = []): ApplicationContext
```

`$config` es el array de configuración completo; `$providers` se registran **después** de `FireflyAutoConfigureServiceProvider`, que se registra siempre primero pase lo que pase; `$bindings` son sobrescrituras `$app->instance()` enlazadas *antes* del registro de proveedores — la costura para entregar un doble de puerto; y `$needs` es un pequeño "menú de enlaces faltantes" (`'cache'`, `'validation'`, `'http'`) que enlaza un respaldo fresco y aislado para una capacidad de la que un arranque desnudo carecería de otro modo por completo:

```

<?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()` es la llamada idéntica, que devuelve el `ApplicationContext` arrancado en lugar del `Application` en bruto — útil siempre que una prueba quiera alcanzar directamente un bean a través de la fachada del contexto en lugar de la propia API de contenedor de Laravel.

Dobles de grabación: probar sin `Event::fake()`

Los `Event::fake()` / `Bus::fake()` de Laravel interceptan el despacho de eventos y jobs *propio de Laravel* — no tienen visibilidad alguna sobre el `EventPublisher` de `firefly/eda`, el `CommandBus` de `firefly/cqrs`, ni ningún otro puerto de Firefly. `firefly/testing` trae un doble de grabación o de stub para exactamente eso: una

implementación sencilla del puerto real que graba lo que vio, de modo que una prueba puede afirmar sobre el comportamiento en lugar de reimplementar los propios internos del framework.

Puerto	Doble (Firefly\Testing\Double*)	Graba
Firefly\Eda\EventPublisher	RecordingEventPublisher	\$published — {destination, eventType, payload, headers}; publishedOf(string \$eventType) filtra
Firefly\Context\Event\ApplicationEventPublisher	RecordingApplicationEventPublisher	\$events — cada objeto de evento publicado, en orden; ofType(string \$class) filtra
Firefly\Cqrs\Command\CommandBus	RecordingCommandBus	\$sent ; willReturn(string \$commandClass, mixed \$result) programa un resultado enlatado; handled(string \$class) filtra
Firefly\Cqrs\Query\QueryBus	StubQueryBus	\$asked ; willReturn(string \$queryClass, mixed \$result) programa un resultado enlatado
Firefly\Cqrs\Metrics\CqrsMetrics	RecordingCqrsMetrics	\$commandSuccesses / \$commandFailures / \$querySuccesses / \$queryFailures

Puerto	Doble (Firefly\Testing\Double*)	Graba
Firefly\Cqrs\Event\CommandEventPublisher	RecordingCommandEventPublisher	\$published (lista de DomainEvent); un constructor opcional ? Throwable \$throw ejercita el camino de fallo del puente
Firefly\Messaging\MessageBrokerPort	RecordingMessageBroker	\$published — {topic, value, key, headers}; publishedTo(string \$topic) filtra
Firefly\Scheduling\Lock\DistributedLock	RecordingDistributedLock	\$acquired / \$released ; constructor (bool \$available = true) ; setAvailable() conmuta la concesión del cerrojo
Firefly\Actuator\Health\HealthIndicator	FakeHealthIndicator	Por defecto Status::Up ; el constructor toma un Health , setHealth() lo reprograma
Firefly\Observability\Tracing\Tracer	RecordingTracer	\$spans — cada nombre de span trazado, en orden de llamada; aún invoca y devuelve fielmente la clausura trazada

Puerto	Doble (Firefly\Testing\Double*)	Graba
Firefly\Context\Event\ApplicationEventPublisher	RecordingAuthenticationEvents	\$events , más la familia de eventos de seguridad por nombre: successes() , interactive() , failures() , logouts() , denials() ; un constructor opcional ? ApplicationEventPublisher \$forwardTo vuelve a publicar de verdad cada evento grabado. Enlázalo antes del arranque , desde defineFireflyEnvironment()

Dos filas nombran el mismo puerto, y la diferencia entre ellas es lo que necesita un test de seguridad. RecordingApplicationEventPublisher graba objetos;

RecordingAuthenticationEvents conoce además los cinco tipos de evento de seguridad por nombre, así que un test de flujo de inicio de sesión lee \$this->events->failures()[0]->username en vez de filtrar una lista mixta por clase. Es el doble al que llama \$this->events cada listado de las secciones de inicio de sesión, cierre de sesión y denegación del Capítulo 10.

Su docblock lleva las dos reglas que lo hacen funcionar, y las dos son fáciles de equivocar:

```
/**
 * // ...
 * 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) {}

```

La primera regla es el momento del enlace: un `instance()` registrado desde `defineFireflyEnvironment()` ya está ahí cuando habría corrido el valor por defecto de `firefly/security`, protegido por `bound()`, así que el espía gana — enlázalo después del arranque y lo que sostendrán los filtros será el publicador propio del framework.

La segunda es `$forwardTo`, y existe porque enlazar el espía **reemplaza** al puerto. Nada registrado contra el despachador vuelve a oír un evento de seguridad: un `#[AsEventListener]` sobre `InteractiveAuthenticationSuccessEvent` se queda mudo durante toda la suite, que es como se rompe sin error una suite cuya tubería incluye semejante listener (`firefly/security-oauth2-server` estampa así el instante de inicio de sesión). Entrégale el publicador propio del framework — `new RecordingAuthenticationEvents(new DispatcherEventPublisher($app))` — y cada evento se graba y se publica de verdad. El valor por defecto, `null`, mantiene al espía como puro grabador, que es lo correcto para una suite que solo lee.

`RecordingEventPublisher`, al completo, muestra la forma que sigue cada doble de la tabla — una implementación real del puerto, más un array sencillo que solo recuerda lo que ocurrió:

```

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');

```

Expectativas de Pest con sabor a Firefly

`Firefly\Testing\Pest\FireflyExpectations::register()` instala un pequeño conjunto de matchers `expect()` personalizados, invocado una vez desde el propio `tests/Pest.php` raíz del monorepo:

```
use Firefly\Testing\Pest\FireflyExpectations;
// ...
FireflyExpectations::register();
```

Hay exactamente **cinco**:

- `toHavePublished(string $eventType, array $payloadContains = [])` — sobre un `RecordingEventPublisher`; al menos un evento publicado del tipo dado cuyo payload contiene el subconjunto clave/valor dado.
- `toHaveHandledCommand(string $class)` — sobre un `RecordingCommandBus`; al menos un comando enviado es una instancia de `$class`.
- `toBeUp()` — sobre un `Health` o un `HealthIndicator` (reducido vía `->health()` automáticamente); afirma `Status::Up`.
- `toHaveRecordedMetric(string $name, array $tags = [])` — sobre cualquier cosa que exponga `meters(): list<Meter>` (p. ej. un `MeterRegistry`); al menos un medidor grabado con el nombre y subconjunto de tags dados.
- `toBeProblemDetails(int $status)` — sobre un `TestResponse` (comprueba el content type `application/problem+json` y decodifica el cuerpo) o un array sencillo; afirma que la forma RFC-7807 tiene las claves `type / title / status` y que `status` coincide.

```
<?php
```

```
use Firefly\Testing\Double\RecordingCommandBus;

// @phpstan-ignore method.notFound
expect($commandBus)->toHaveHandledCommand(PlaceOrder::class);
```

❗ LOS COMENTARIOS `@PHPSTAN-IGNORE METHOD.NOTFOUND` , EXPLICADOS

Pest instala estas vía `expect()->extend(...)` **en tiempo de ejecución** — la reflexión estática de PHPStan sobre la clase `Pest\Expectation` de vendor no tiene forma de ver un método añadido de esta manera, de modo que cada llamada se marca como `method.notFound` aunque genuinamente exista una vez que `tests/Pest.php` se ha ejecutado. Mantén cada expectativa personalizada en su propia sentencia `expect()` en lugar de encadenar con `->and(...)` : encadenar degrada el `method.notFound` (precisamente ignorable) hasta el `method.nonObject` (no ignorable).

Para una aserción más llana, con sabor a PHPUnit, en lugar de las expectativas fluidas, dos helpers procedurales viven junto a `bootFireflyApp()` en el mismo `functions.php` autocargado:

```
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()` Y EL CAMPO `TYPE` : UNA LIMITACIÓN REAL, Y CÓMO SE CERRÓ

Esta caja describía una brecha genuina, y vale la pena tenerla a la vista porque es la forma que suelen tener estas cosas. `toBeProblemDetails()` afirma que el cuerpo de la respuesta tiene una clave `type` — y el renderizador real del framework,

`Firefly\Web\Exception\ProblemDetailsRenderer`, no publicaba ningún `type`. Era un miembro opcional que solo se emitía cuando algo lo pasaba explícitamente, cosa que el camino de excepción-a-respuesta nunca hacía, así que llamar a la aserción entregada contra un `404 / 422 / 403 / 409` genuino fallaba siempre. Un matcher y un renderizador que se entregan en el mismo framework discrepaban sobre la forma del mismo documento.

El RFC 9457 §3.1.1 lo zanjó: un `type` ausente es `about:blank`, así que escribirlo no le cuesta nada a un documento conforme y le ahorra a cada cliente tener que saberse la regla. El renderizador ahora lo emite — el `ProblemDetail` de Spring toma la misma decisión — y `firefly.web.problem.type-uri` o bien apunta a una URI base, derivando un tipo abrible a partir del `code` estable (`https://api.example.test/problems/resource-not-found`), o bien se pone a `''` para volver a omitir el miembro en un despliegue que quiera el documento pre-9457 byte por byte. Así que la aserción funciona contra las propias respuestas de error del framework, y `samples/lumen` la ejercita sobre el 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);
});
```


Las aserciones sobre campos se quedaron, y esa es la parte que conviene copiar:

`toBeProblemDetails()` dice que la respuesta es *un* documento de problema de ese estado, mientras que `assertJsonPath('code', 'RESOURCE_NOT_FOUND')` dice cuál es el fallo. La segunda es la que una regresión tumba, así que una prueba que pueda permitirse ambas debería tener ambas.

Pruebas de rebanada: `WebSliceTestCase` y `DataSliceTestCase`

Una prueba de rebanada arranca solo los beans que un escaneo PSR-4 descubre para una estrecha vertical, más cualesquiera sobrescrituras explícitas de puertos que suministres — la idea de `@WebMvcTest` / `@DataJpaTest` de Spring, portada. Ambas bases exponen un método, invocable directamente desde una clausura de prueba de Pest:

```
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()` es el mismo método para la vertical de persistencia. Ambos son **por clase de test**: llamar a uno dos veces con definiciones de rebanada distintas es una `LogicException` en vez de un segundo arranque silencioso, porque la segunda definición se aplicaría a cada test de la clase y no solo a los escritos después de ella.

```
<?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` arranca el pipeline real web + validación — incluido el cableado de rutas — sobre solo los controladores rebanados, de modo que `$this->get(...)` sirve un `TestResponse` genuino contra ellos. `DataSliceTestCase` arranca el pipeline de datos real sobre solo los beans rebanados, y luego resuelve con avidez cada uno de ellos, de modo que una rebanada mal cableada falla de inmediato en el arranque en lugar de pasar por accidente porque nada en la prueba resultó tocar el bean roto:

```
<?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);
});

```

Ambos métodos son **por clase de prueba** — llamar a cualquiera de los dos una segunda vez con argumentos diferentes lanza una `LogicException`, de modo que la forma de una rebanada queda fijada para toda la clase de prueba en lugar de ser mutable a mitad de la prueba.

Los atributos `#[FireflyTest]` / `#[WebSlice]` / `#[DataSlice]`

Para una **clase** de prueba escrita a mano, al estilo de PHPUnit, en lugar de un archivo de clausura de Pest, tres atributos con destino en clase dan las mismas tres formas de manera declarativa, leídos por `setUp()` antes de que el propio arranque de Testbench llegue a ejecutarse:

```

<?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());
    }
}
```



ATRIBUTOS FRENTE A LLAMADAS DE MÉTODO — NO MEZCLES LOS DOS EN UNA MISMA CLASE DE PRUEBA

`#[FireflyTest]` / `#[WebSlice]` / `#[DataSlice]` solo funcionan sobre una clase escrita a mano que extienda la base coincidente — no sobre un archivo de clausura de Pest. Una prueba de clausura de Pest llama a `$this->webSlice(...)` / `$this->dataSlice(...)` directamente en su lugar, porque el método internamente llama a `refreshApplication()` para re-arrancar con los valores de la rebanada — Testbench ya ha realizado un primer arranque vacío en `setUp()` antes de que el cuerpo de la clausura llegue a ejecutarse. Elige un solo estilo por clase de prueba.

La capa de fixtures

Tres pequeños helpers rematan el kit, cada uno reemplazando un equivalente improvisado a mano que solía estar duplicado por paquete.

`FixtureRegistry` es un registro fino de fixtures con nombre sobre clausuras de fábrica sencillas — la comodidad de Firefly que se sienta junto a las factories de Eloquent, no un reemplazo de ellas:

```
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 reproduce los eventos de dominio de un agregado a través de un puerto **ApplicationEventPublisher** — útil para sembrar una proyección (el libro mayor del Capítulo 6, por ejemplo) sin volver a ejecutar todo el flujo de comandos que originalmente produjo esos eventos:

```
<?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` graba valores de cadena arbitrarios que un fixture de listener o handler vio, en orden de llamada — el único reemplazo compartido de lo que solía ser una clase `Spy` improvisada a mano en cada paquete que necesitaba una:

```
<?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']);
```

Backends reales: `RequiresDocker` y `fireflyConfigFor()`

Algunas pruebas necesitan un backend genuino — un Postgres real, un broker Kafka real — en lugar de un doble en memoria. Estas se etiquetan con `@group integration` y se excluyen por completo de la puerta por defecto: `vendor/bin/pest` (la puerta de rama completa que ejecuta `composer test`) nunca toca uno a menos que lo pidas explícitamente con `vendor/bin/pest --group=integration`.

`RequiresDocker` es un trait (`@mixin PHPUnit\Framework\TestCase`) que convierte "Docker no está disponible" en un salto limpio en lugar de un fallo duro, de modo que una máquina sin Docker que sí ejecute el grupo de integración aún obtenga un resultado verde:

```

trait RequiresDocker
{
    protected function skipUnlessDocker(): void
    {
        if (! is_docker_available()) {
            $this->markTestSkipped('Docker is not available; skipping @group integration
test.');
```

`is_docker_available()` ejecuta `docker info` y devuelve `true` si y solo si sale con `0` — nunca lanza una excepción, de modo que llamarla sin Docker instalado en absoluto es siempre seguro. Una base de prueba de integración real respaldada por Kafka muestra el patrón de principio a fin:

```

<?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` — el análogo de `@ServiceConnection` — mapea un `testcontainer` arrancado a un array de configuración plano, indexado por clave con puntos, con duck typing contra cualquiera de `getHost()` / `getMappedPort()` / `getUsername()` / `getPassword()` / `getDatabase()` que el objeto contenedor realmente exponga, de modo que funciona contra cualquier clase de contenedor de `testcontainers-php` sin ninguna dependencia dura de los tipos de la biblioteca:

```
/**
 * 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
```

Alimenta el resultado directamente al `configOverrides()` de una subclase de `FireflyTestCase`, o establece cada clave en el repositorio de configuración desde `defineFireflyEnvironment()`.

Prueba desde Lumen: cómo la muestra hace dogfooding de todo

`samples/lumen` construye su propia base de prueba, `LumenTestCase`, encima de `FireflyDatabaseTestCase` — el patrón sobre el que cada capítulo de este libro ha estado descansando en última instancia:


```

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',
            // ...
        ];
    }
    // ...
}

```

Cada una de las llamadas HTTP de los Capítulos 4 al 11 en este libro — cada `$this->postJson(...)`, cada aserción RFC-7807 — se ejecuta contra una subclase de `LumenTestCase`. Deliberadamente enlaza el `InMemoryEventBus` *real* como el `EventPublisher` (nunca un `RecordingEventPublisher`), porque las propias pruebas de Lumen necesitan que el puente de dominio-a-integración del Capítulo 8 dispare genuinamente, no que meramente se grabe como si lo hubiera hecho.

⚠ PARIDAD CON LARAVEL

El propio kit de pruebas de Laravel (`RefreshDatabase`, `Event::fake()`, `Bus::fake()`) es excelente para las propias primitivas de Laravel, pero no tiene visibilidad alguna sobre un puerto de Firefly — una llamada a `CommandBus::send()` o a `EventPublisher::publish()` es invisible para `Event::fake()`, porque nada de ella pasa por el propio despachador de eventos de Laravel. Los dobles de grabación y las expectativas de Pest de `firefly/testing` son la capa de paridad: el mismo idioma de "prepara un fake, afirma sobre lo que vio" que ya conoces de Laravel, apuntado a los puertos que los demás capítulos de este libro realmente presentaron.

Un navegador de verdad sobre la aplicación de verdad

Todo lo visto hasta ahora en este capítulo afirma sobre objetos y sobre respuestas HTTP. Ninguna de las dos cosas puede decirte si la página de bienvenida se renderizó de verdad, si los gráficos del panel de administración se dibujaron, o si Swagger UI lanzó un error de JavaScript mientras construía su lista de operaciones. Para eso hace falta un navegador — y la razón por la que la mayoría de proyectos no tiene pruebas de navegador es que el montaje habitual es miserable: un segundo proceso sirviendo la aplicación, una segunda base de datos, un segundo contenedor, y una suite que se va separando de la que ejercitan las pruebas unitarias.

La suite de navegador de LaraFly no tiene nada de eso, por una propiedad que merece enunciarse sin rodeos: **el navegador conduce la misma aplicación arrancada que el resto de la suite.**

Pest 4 más `pestphp/pest-plugin-browser` sirve la aplicación arrancada con Testbench a Chromium **en proceso**. No hay `php -S`, ni raíz de documentos aparte, ni segunda conexión: el mismo contenedor, y la misma base de datos SQLite en memoria que el test pobló:

```
/**
 * 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();
    }
}
```

Y lo que se conduce es el **esqueleto**, compilado por el escritor real de `firefly:cache` — el mismo proyecto que produce `composer create-project`. Que una prueba de navegador falle es, por tanto, evidencia sobre lo que hace la aplicación de una persona recién llegada, no sobre una prueba montada para pasar.

Un escenario se lee como lo describirías en voz alta:

```

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()` está en casi todos los escenarios, y es la afirmación que paga la suite. Una página que se renderiza y lanza calladamente en la consola es una página que se ve bien en una captura y está rota para quien la usa.

Por qué no ralentiza tu gate

Las pruebas de navegador necesitan Node y una descarga de Chromium. El gate por defecto no debe:

```

<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>

```

Dos suites, y `tests/Browser` queda excluida de la que se usa por defecto. Así que `composer check` — Pint, PHPStan, Pest y Deptrac — se mantiene enteramente libre de Node, y `composer test:browser` ejecuta la suite de navegador a solas. CI tiene un trabajo aparte, `Browser (Pest + Playwright)`, que la ejecuta y sube cada captura como artefacto, así que un fallo viene con una foto del aspecto que tenía la página.

Lo que hace el trabajo aquí es la **división en suites**, no la etiqueta de grupo:

```

// 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);

```

Ese comentario merece leerse dos veces, porque la distinción que traza pilla a la gente: Pest **incluye** cada fichero de la suite configurada antes de filtrar por grupo, y el plugin arranca Playwright en cuanto se incluye un fichero bajo `tests/Browser`. Un `--exclude-group=browser` seguiría pagando, por tanto, el coste de Playwright. Solo la división en suites lo evita.

⚠ EL PLUGIN FUERZA `APP.DEBUG=FALSE`

Dos claves del framework toman por defecto `app.debug` — `firefly.web.error-page.trace` y `firefly.admin.enabled` — y el plugin fuerza `app.debug=false` mientras atiende cada petición de navegador. Una suite que dependiera de esos valores por defecto renderizaría la página de error de *producción* creyendo que prueba la de depuración. Así que `BrowserTestCase` enciende cada superficie que pretende mirar **por su propia clave**, y las subclases sobrescriben `trace()` y `securityOverrides()` — ambas leídas antes del arranque, que es por lo que son ganchos y no llamadas por test.

Y un navegador se autentica como lo hace una persona: `SecuredBrowserTestCase` pone el login por formulario propio del framework y dos usuarios en memoria delante de la aplicación, así que los escenarios de inicio de sesión conducen el mismo `FormLoginFilter` que describió el Capítulo 10 y no un atajo de pruebas.

Autenticar un test

Lo que nos lleva a los ayudantes en los que se apoyan los capítulos de seguridad. Tres métodos, un cuarto atributo junto a los tres de `Firefly\Testing\Attributes\`, y un interruptor que apaga la pila entera. Los tres métodos se construyen uno sobre otro:

```
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()` es la costura: fija el portador de inmediato, enlaza el principal, y antepone un middleware **una sola vez**, así que una segunda llamada cambia quién es la petición siguiente en vez de apilar otra copia en el kernel. El middleware es el eslabón más externo a propósito —

`SecurityContextPersistenceFilter` deja en paz a un portador ya autenticado, y los filtros del framework que limpian el portador en su propio `finally` corren dentro de él, así que el principal está de vuelta en el portador cuando la petición retorna.

`actingAsPrincipal('ada', ['ROLE_USER'])` es la forma de cada día, construida sobre él. Y para los capítulos de OAuth2 está el `oidcLogin()` de Spring Security, como método:

```

/**
 * 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()` construye un `DefaultOidcUser` **real**, con las autoridades que un inicio de sesión real habría concedido — `OIDC_USER` más un `SCOPE_x` por cada scope — y el identificador de registro en los atributos del token, que es lo que leen el gestor de clientes autorizados y el cierre de sesión iniciado por la parte confiante. Nada de esto habla con un proveedor: ni registro, ni descubrimiento, ni petición de token. Y deliberadamente no lleva **ningún valor de token en bruto**, porque esa es exactamente la forma que la sesión entrega a un controlador tras un inicio de sesión real; un test que recibiera un token en bruto estaría probando algo que la aplicación nunca ve.

`#[WithMockUser]` es el `@WithMockUser` de Spring, y es la forma declarativa de `actingAsPrincipal()` — el cuarto atributo de `Firefly\Testing\Attributes\`, junto a los tres atributos de rebanada de más arriba:

```

/**
 * 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 = [],
    ) {}
}
    
```

A diferencia de los tres atributos de rebanada, este se lee en `setUp()` sobre *cualquier* `FireflyTestCase`, ganando el de nivel de método sobre el de nivel de clase, así que una clase de test escrita a mano puede dar acceso a toda su suite una vez y nombrar la excepción en el único método que necesita otro principal.

Y para la dirección contraria está `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
```

Lee dos veces la última frase, porque es la razón por la que este método existe en absoluto en vez de una sobrescritura de configuración: los indicadores se leen **en vivo**, en cada petición, así que cambiarlos a mitad de un test cambia las puertas de beans que ya estaban contruidos. Un test que solo quiere ejercitar lo que hay *detrás* de la puerta no tiene que volver a arrancar la aplicación para pasarla.

✓ Lo que aprendiste

Concepto	Qué hace
<code>FireflyTestCase</code>	Base de Testbench; tres hooks (<code>fireflyProviders</code> / <code>configOverrides</code> / <code>defineFireflyEnvironment</code>); config sembrada antes del arranque
<code>FireflyDatabaseTestCase</code> / <code>UsesSqliteMemory</code>	Añade una conexión sqlite <code>:memory:</code> compartida + <code>createSchema()</code>
<code>bootFireflyApp()</code> / <code>fireflyApplication()</code>	Arranques desnudos, sin Testbench; un menú <code>\$needs</code> rellena los respaldos faltantes de cache/validation/http
11 dobles de grabación	Implementaciones reales de puertos que graban lo que vieron — la capa de paridad que <code>Event::fake()</code> no puede alcanzar
5 expectativas de Pest	<code>toHavePublished</code> , <code>toHaveHandledCommand</code> , <code>toBeUp</code> , <code>toHaveRecordedMetric</code> , <code>toBeProblemDetails</code>
<code>WebSliceTestCase</code> / <code>DataSliceTestCase</code>	Arranca solo una rebanada escaneada por PSR-4 + sobrescrituras explícitas; por clase de prueba, fallo rápido en datos
<code>#[FireflyTest]</code> / <code>#[WebSlice]</code> / <code>#[DataSlice]</code>	Análogo de atributo de clase de las mismas tres formas, para clases de prueba escritas a mano
<code>FixtureRegistry</code> / <code>AggregateSeeder</code> / <code>ListenerSpy</code>	Fixtures con nombre; reproducción de eventos de dominio; un spy compartido de orden de llamada

Concepto	Qué hace
<code>RequiresDocker</code> / <code>fireflyConfigFor()</code>	Opt-in de <code>@group integration</code> ; salta-no-falla sin Docker; mapeo de testcontainer → config
La suite <code>browser</code>	Pest 4 + <code>pest-plugin-browser</code> sirviendo el esqueleto arrancado con Testbench en proceso — mismo contenedor, mismo SQLite, sin un segundo servidor
Las dos suites de <code>phpunit.xml.dist</code>	<code>tests/Browser</code> excluida de <code>unit</code> , así que <code>composer check</code> se mantiene libre de Node; <code>composer test:browser</code> la ejecuta a solas
<code>assertNoJavaScriptErrors()</code>	La afirmación que paga la suite: una página que se renderiza y lanza en la consola se ve bien en una captura
<code>actingAsAuthentication()</code>	La costura bajo <code>actingAsPrincipal()</code> y <code>actingAsOidcUser()</code> ; middleware antepuesto una sola vez, así que una segunda llamada cambia el principal
<code>actingAsOidcUser()</code>	El <code>oidcLogin()</code> de Spring como método: un <code>DefaultOidcUser</code> real, autoridades reales, ningún token en bruto — la forma que una sesión entrega de verdad a un controlador
<code>RecordingAuthenticationEvents</code>	El doble consciente de la seguridad: la familia de eventos de seguridad por nombre (<code>successes</code> / <code>interactive</code> / <code>failures</code> / <code>logouts</code> / <code>denials</code>); enlázalo antes del arranque, y pasa <code>\$forwardTo</code> cuando los listeners reales deban seguir oyendo

Concepto	Qué hace
<code>#[WithMockUser]</code>	El <code>@WithMockUser</code> de Spring: el <code>actingAsPrincipal()</code> declarativo, leído en <code>setUp()</code> , ganando el de nivel de método sobre el de clase
<code>withoutSecurity()</code>	Cada indicador de seguridad apagado durante el resto del test — leídos en vivo, así que los beans ya contruidos cambian de puerta sin volver a arrancar

Ponlo en práctica

- Escribe una prueba de rebanada para uno de los propios controladores de Lumen.** Usando `WebSliceTestCase`, escanea solo `Lumen\Web\` y confirma que puedes servir un endpoint de wallet sin arrancar CQRS, EDA ni seguridad en absoluto — luego anota qué peticiones fallan, y por qué, una vez que entiendas exactamente qué beans dejó fuera la rebanada.
- Mira cómo se mueve `type` bajo `toBeProblemDetails()`.** Llama a `toBeProblemDetails(404)` contra un `404` real de la propia API REST de Lumen y confirma que pasa — la clave `type` de la que habla la nota de este capítulo es `about:blank`. Luego pon `firefly.web.problem.type-uri` a `https://api.example.test/problems`, vuelve a ejecutarla, y confirma que el miembro pasó a ser `https://api.example.test/problems/resource-not-found` mientras la aserción sigue pasando. Por último pon la clave a `''` y observa cómo esa misma aserción falla con el mensaje de clave-`type`-faltante — el fallo que describe la nota, ahora algo que elige un despliegue en vez de algo que el framework te hace.

3. **Ejecuta la suite de navegador, y luego rompe una página.** Ejecuta `composer test:browser` y mira `tests/Browser/Screenshots`. Después añade a la vista de bienvenida un `<script>` que lance, y vuelve a ejecutarla: confirma que la página sigue renderizándose, sigue pasando todos los `assertSee`, y que `assertNoJavaScriptErrors()` es la única afirmación que lo atrapa.
4. **Demuestra que la división en suites importa.** Cronometra `composer test` y confirma que no se descarga ningún Chromium. Después prueba `vendor/bin/pest --exclude-group=browser` y mira cómo Playwright arranca igualmente — la etiqueta de grupo filtra tests, la división en suites es lo que impide que los ficheros se incluyan siquiera.
5. **Añade una prueba de integración protegida por RequiresDocker.** Elige un repositorio respaldado por Eloquent, escribe una prueba `@group integration` contra un testcontainer Postgres real usando `fireflyConfigFor()`, y confirma que `vendor/bin/pest` (sin banderas) la salta por completo mientras que `vendor/bin/pest --group=integration` la ejecuta de verdad.
6. **Comprueba que `$forwardTo` importa.** Enlaza un `RecordingAuthenticationEvents` sin `$forwardTo` desde `defineFireflyEnvironment()`, registra un `#[AsEventListener]` sobre `InteractiveAuthenticationSuccessEvent` que escriba una fila, e inicia sesión: confirma que el espía graba el evento y que el listener nunca se ejecutó. Después vuelve a enlazarlo envolviendo el publicador propio del framework y confirma que ocurren las dos cosas. Por último mueve la llamada a `instance()` a *después* del arranque y confirma que el espía no graba nada en absoluto — el momento del enlace es todo el truco.
7. **Pasa una puerta de dos formas.** Toma una prueba de controlador que reciba un `401`, y haz que pase dos veces: una con `#[WithMockUser(roles: ['ADMIN'])]` sobre el método, otra con `withoutSecurity()`. Después pon `#[WithMockUser]` a

nivel de clase y sobrescríbelo en un solo método, y confirma que gana el de nivel de método.

La CLI y la Caché Sin Reflexión

Al terminar este capítulo sabrás exactamente qué hace `php artisan firefly:cache` — los trece pasos de escáner-y-compilador que ejecuta, en orden, sobre las propias raíces PSR-4 de tu aplicación, y los manifiestos de `bootstrap/cache/firefly/` y las clases proxy generadas (una por cada clase que nombra el plan de proxies) que escribe — cómo `FireflyCacheServiceProvider` vuelve a cargar esa caché en el arranque con cero reflexión, el inverso exacto de `firefly:clear`, los comandos de actuator-sobre-CLI (`firefly:about` / `:routes` / `:health` / `:metrics`) que reutilizan los endpoints del Capítulo 11 en-proceso sin ida y vuelta HTTP, los ocho generadores `make:firefly-*`, y cómo el `firefly new` de `firefly/installer` te lleva de la nada a una aplicación cacheada y en funcionamiento en un solo comando.

❗ TÉRMINO NUEVO: ARRANQUE SIN REFLEXIÓN

Cada capítulo anterior a este ha mencionado, de pasada, que un manifiesto compilado reemplaza a "escanea el sistema de archivos con reflexión de PHP en cada arranque". Un **arranque sin reflexión** es lo que obtienes una vez que `firefly:cache` ha ejecutado realmente: `FireflyCacheServiceProvider` carga manifiestos precomputados directamente desde archivos PHP bajo `bootstrap/cache/firefly/`, de modo que ningún atributo, ninguna clase y ningún método se somete jamás a reflexión en tiempo de petición — el escaneo ocurrió una vez, en tiempo de despliegue, no en cada petición.

`firefly:cache` : la recompensa sin reflexión

`CacheCommand` es un envoltorio fino — lee `firefly.scan.paths` (el mismo mapa PSR-4 que sembró el `configOverrides()` de cada capítulo anterior) y se lo entrega a `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.');
```

`ManifestCacheWriter::write()` es la orquestación real, y vale la pena leerla de principio a fin: no es más que el par escáner-y-compilador *ya existente* de cada paquete de capacidad, llamado en secuencia, sobre el mismo mapa PSR-4 — sin ninguna abstracción nueva, cero ediciones a ningún paquete ya asentado:

```

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);
}
}

```

Eso son doce llamadas `scan()` -luego- `write()` — cada capacidad desde el escaneo de componentes del Capítulo 2 hasta el manifiesto de seguridad de método del Capítulo 10, cada una ya presentada en un capítulo anterior como un escaneo en proceso — más un paso más, `writeProxies()`, que genera las clases proxy que te enseñó el Capítulo 9, usando el mismísimo `ProxyClassGenerator` sin reflexión que usa el propio runtime cuando aún no existe caché:

```

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,
    ]);
}

// ...
}

```

Trece pasos en total dejan catorce artefactos bajo `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     # qué beans reciben un proxy, y qué advice ejecuta cada método
├─ proxies.php        # mapa de clases FQCN => fichero para los proxies generados
└─ proxies/           # un fichero de clase proxy generada por cada clase que el plan
nombra
```

FireflyCacheServiceProvider : volver a cargar la caché

Escribir los manifiestos es solo la mitad de la historia — algo tiene que darse cuenta de que existen y enlazarlos *en lugar de* ejecutar el escaneo en-proceso.

`FireflyCacheServiceProvider` (autodescubierto) es ese algo, y su `register()` se ejecuta antes de cualquier resolución de bean en absoluto:

```
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];
            }
        });
    }
}
}
}
}
}

```

Cada llamada `$app->instance(...)` es **incondicional**, y el propio valor por defecto de cada capacidad es `if (! bound(X)) singleton(X, empty)` — de modo que el manifiesto cargado siempre gana, independientemente de qué proveedor haya registrado Laravel primero. La llamada `spl_autoload_register` del final es lo que realmente hace alcanzables los proxies cacheados: instala un cargador de classmap indexado por el nombre completamente cualificado del proxy generado, de modo que la primerísima vez que `TransactionalBeanPostProcessor` comprueba `class_exists($proxyClass)`, la clase ya es cargable — sin reflexión, sin escaneo del sistema de archivos, solo una búsqueda en un mapa.

Nada de esto es obligatorio. Una aplicación sin ningún directorio de caché aún arranca — el enlace de manifiesto de cada capacidad recurre al escáner en-proceso exactamente como lo ha hecho en cada capítulo anterior — solo que sin la garantía de cero reflexión. `firefly:cache` es una mejora pura de rendimiento-y-determinismo por la que optas ejecutándola, no un requisito para que el framework funcione.

`firefly:clear`

El inverso exacto: borra recursivamente `FireflyCachePaths::dir()` — el `firefly.cache.path` configurado, o `bootstrap/cache/firefly` por defecto — y nada más:

```
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;
}
}

```

Seguro de ejecutar en cualquier momento: el siguiente arranque simplemente recurre al escáner en-proceso, igual que una aplicación que nunca ejecutó `firefly:cache` en absoluto.

Cuando la caché nombra una clase que borraste

Ambos comandos anteriores tienen que arrancar la aplicación antes de poder hacer su trabajo — `firefly:cache` no puede escribir los manifiestos sin arrancar primero la aplicación cuyos manifiestos son, y `firefly:clear` no puede borrar un directorio por el que no ha arrancado. Esa circularidad se vuelve incómoda en cuanto la caché está *equivocada*, y la forma habitual de equivocarla es borrar un `#[Component]` y no recompilar: `component.php` sigue nombrando la clase, y ningún autoloader la encuentra.

Ese estado no se podía recuperar con ningún comando. La clase borrada seguía en el manifiesto que leía `ContainerRegistrar`, así que `wireInterfaces()` ligaba la interfaz que implementaba — y la etiquetaba como implementación — a una clase que no estaba. El fallo aparecía por tanto lejos de su causa: `Target class [App\Security\ControlPlaneJwksProvider] does not exist` salía al resolver un bean que nadie había tocado y que se limita a inyectar esa interfaz.

`EagerSingletonsPass` omite una definición cuya clase ya no existe, y no servía de nada, porque la excepción nunca se lanzaba sobre la definición borrada. `composer`

`dump-autoload` tampoco ayudaba — su `package:discover` arranca la aplicación también. La recuperación era borrar `bootstrap/cache/firefly/*.php` a mano, luego `composer dump-autoload`, y luego `firefly:cache`, en ese orden.

`Firefly\Context\Scan\AppScan::repairing()` es la costura que lo cierra: mientras el comando en ejecución sea `firefly:cache` o `firefly:clear`,

`BeanDefinitionRegistry` — la única puerta por la que entra toda definición — descarta una definición cuya clase no se encuentra, de modo que ni el registrar, ni el paso de bean post-processors, ni el de ciclo de vida, ni el de listeners llegan a verla.

`firefly:cache` dice entonces qué entradas descartó:

```
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
```

El manifiesto que escribe ya no menciona la clase, así que la segunda ejecución es una ejecución limpia normal y la línea desaparece.

⚠ SOLO ESOS DOS COMANDOS DESCARTAN ALGO

Bajo cualquier otro comando el manifiesto se toma tal cual, y una clase que ha desaparecido sigue deteniendo el arranque con el error del propio contenedor. Ahí esa es la respuesta correcta: una clase ausente en un proceso que está a punto de servir tráfico no es una caché rancia, es un despliegue roto — un artefacto truncado, un classmap construido desde otro árbol — y descartar la definición le entregaría a la aplicación una interfaz religada en silencio a la implementación que sobreviviera. Una respuesta equivocada de la que nadie se entera es peor que el fallo de arranque. Y solo se tolera una clase AUSENTE: una clase que existe y no se puede construir sigue fallando rápido, tanto en un comando de reparación como en cualquier otro sitio.

Actuator-sobre-CLI: `firefly:about`, `firefly:routes`, `firefly:health`, `firefly:metrics`

El Capítulo 11 construyó una superficie de gestión alcanzable sobre HTTP. Estos cuatro comandos renderizan los **mismos** endpoints en el terminal, en-proceso — no se hace ninguna petición HTTP, y ninguno de ellos reimplementa lógica alguna del actuator:

```
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` resuelve el `ActuatorRegistry` compilado (el mismo registro que `ActuatorRouteRegistrar` puebla en el arranque), llama al `handle()` de cada endpoint, e imprime la respuesta — toda la historia de introspección `--debug /actuator` de Spring Boot de un vistazo, en un solo comando. `firefly:routes` renderiza solo el endpoint `mappings` (tu tabla de rutas compilada), `firefly:health` renderiza `health`, y `firefly:metrics` renderiza `metrics` — el último requiere que `firefly/observability` esté instalado, e imprime una advertencia y sale con éxito en lugar de dar un error si no lo está.

Andamiaje: los generadores `make:firefly-*`

Cada estereotipo de framework que este libro ha presentado tiene un generador de Artisan a juego — la familia de comandos `generate` de pyfly, portada a la propia base `GeneratorCommand` de Laravel:

Comando	Genera
<code>make:firefly-controller</code>	Dos archivos: un recurso REST <code>#[RestController]</code> (<code>index / show / store / update / destroy</code> bajo un único <code>#[RequestMapping]</code> de clase cuya ruta se deriva del nombre del recurso) y el DTO <code>#[Valid]</code> <code>#[RequestBody]</code> que vinculan su <code>store / update</code> — <code>--plain</code> da la forma de una sola acción, sin DTO.
<code>make:firefly-service</code>	Un bean <code>#[Service]</code> .
<code>make:firefly-component</code>	Un bean <code>#[Component]</code> .
<code>make:firefly-handler</code>	Dos archivos: un <code>#[CommandHandler]</code> y la clase de comando que toma su <code>handle()</code> — <code>#[QueryHandler]</code> más su consulta con <code>--query</code> .
<code>make:firefly-listener</code>	Un método <code>#[EventListener]</code> por defecto, o un <code>#[MessageListener]</code> con <code>--message</code> .
<code>make:firefly-entity</code>	Una entidad DDD que extiende <code>Firefly\Domain\Entity</code> (no hay ningún atributo <code>#[Entity]</code>).
<code>make:firefly-repository</code>	Una clase <code>#[Repository]</code> concreta que extiende <code>Firefly\Data\Repository\EloquentRepository</code> , con un <code>\$model</code> que reapiuntar.
<code>make:firefly-config-properties</code>	Un DTO de configuración enlazado con <code>#[ConfigProperties]</code> .

`make:firefly-handler` es representativo de toda la familia — un `GeneratorCommand` que elige qué stub renderizar en función de una única bandera:

```

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].'],
        ];
    }
}

```

y el stub de command-handler que renderiza — con sus marcadores

`{{ namespace }}` / `{{ class }}` sustituidos por `GeneratorCommand` de la manera en que lo haría `php artisan make:firefly-handler RegisterWidget` — es exactamente la forma que el Capítulo 7 te enseñó a escribir a mano:

```

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
```

Pasarelas finas: `firefly:serve`, `firefly:db`

Dos comandos existen puramente por una convención de nombres `firefly:*` consistente sobre comandos que Laravel ya trae — ninguno reimplementa comportamiento alguno de Laravel:

```
php artisan firefly:serve {--host=127.0.0.1} {--port=8000}
```

Delega a `artisan serve`, o a `octane:start` cuando `laravel/octane` está instalado — sondeado únicamente vía `class_exists()`, de modo que Octane sigue siendo una dependencia de runtime opcional que el propio `composer.json` de `firefly/cli` nunca requiere.

```
php artisan firefly:db {action=migrate}
```


Delega a los propios comandos de base de datos de Laravel: `migrate` (por defecto), `db:seed ( firefly:db seed )`, o `migrate:fresh ( firefly:db fresh )`.

De cero a en funcionamiento: `firefly/installer` y `firefly/skeleton`

`firefly/installer` trae el comando global `firefly new` — un pequeño binario de Symfony Console, no un andamiador en sí. Invoca `composer create-project` con la plantilla incluida en `fireflyframework/laraflly` como repositorio local de tipo path:

```
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;
}
```

El `#[AsCommand]` de aquí es **de Symfony**, no de Firefly, y las líneas `use` de arriba lo dicen. `firefly new` es una aplicación autónoma de Symfony Console y no un comando de artisan, porque tiene que ejecutarse antes de que haya una aplicación Laravel de la que ser comando.

Todo proceso externo pasa por la costura `ProcessRunner` — `composer`, `php artisan`, `git` —, que es lo que hace que el flujo entero sea afirmable sin red y sin un `composer create-project` real.

`firefly/skeleton` — una plantilla de aplicación Laravel 13 genuina, de nivel superior, precableada con la familia Firefly y un par `#[RestController]` / `#[Service]` de ejemplo — es lo que el instalador entrega a `composer create-project` desde su propia distribución. Su propio `composer.json` cierra el bucle hacia el que todo este capítulo ha estado construyendo: `firefly:cache` se ejecuta **automáticamente**, justo después de la instalación, sin ningún paso manual:

```

"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"
  ]
}

```

Cada aplicación que `firefly new` andamia es por tanto, desde el primerísimo `git commit`, ya boot-cacheada — una aplicación genuinamente sin reflexión en disco, no meramente un framework capaz de convertirse en una. El propio `bootstrap/cache/firefly/` está en `.gitignore` en el esqueleto (los manifiestos se generan frescos por instalación, no se commitean), que es exactamente lo que esperarías de un artefacto de compilación en lugar de código fuente.

📦 PARIDAD CON LARAVEL

`composer create-project laravel/laravel` te da una app Laravel desnuda sin nada precableado; `firefly new` te da la misma app Laravel, más la familia Firefly ya requerida, más un par de estereotipos de ejemplo, más una caché de arranque ya calentada — el equivalente al "genera un proyecto de arranque funcional" de Spring Initializr, apuntado a `artisan` en lugar de a un formulario web. `make:firefly-*` refleja la propia familia `make:controller` / `make:model` de Laravel exactamente en espíritu — un `GeneratorCommand` de Artisan, un stub, un marcador de namespace — solo que un atributo de estereotipo más profundo.

✓ Lo que aprendiste

Concepto	Qué hace
<code>firefly:cache</code>	Ejecuta el par escáner → compilador propio de cada capacidad (13 pasos, 14 artefactos) sobre <code>firefly.scan.paths</code> ; escribe manifiestos + un archivo de clase proxy generado por cada clase que nombra el plan de proxies a <code>bootstrap/cache/firefly/</code>
<code>FireflyCacheServiceProvider</code>	Enlaza cada manifiesto cacheado incondicionalmente (gana sobre todo valor por defecto vacío) + instala el autoloader de classmap de proxies — antes de que se ejecute cualquier resolución de bean
<code>firefly:clear</code>	Borra recursivamente solo el directorio de caché; el siguiente arranque recurre al escaneo en-proceso
<code>firefly:about</code> / <code>:routes</code> / <code>:health</code> / <code>:metrics</code>	Renderiza los endpoints de actuador del Capítulo 11 en el terminal, en-proceso, sin ida y vuelta HTTP
<code>make:firefly-*</code>	Ocho generadores, uno por estereotipo — <code>GeneratorCommand</code> + un stub, exactamente como la propia familia <code>make:*</code> de Laravel
<code>firefly:serve</code> / <code>firefly:db</code>	Pasarelas finas a <code>artisan serve</code> / <code>octane:start</code> y a los propios comandos de base de datos de Laravel
<code>firefly new</code> (<code>firefly/installer</code>)	Entrega la plantilla incluida a <code>composer create-project</code> ; el esqueleto ejecuta <code>firefly:cache</code> automáticamente tras la instalación

Ponlo en práctica

1. **Ejecuta `firefly:cache` contra Lumen e inspecciona la salida.** Desde el monorepo del framework, apunta `firefly.scan.paths` a `samples/lumen/src` y ejecuta el comando a mano; abre `bootstrap/cache/firefly/handlers.php` y confirma que lista `WithdrawHandler` y sus hermanos de los Capítulos 6 y 7, en una forma que puedes leer sin ninguna reflexión en absoluto.
2. **Rompe la caché a propósito.** Ejecuta `firefly:cache`, luego renombra una clase `#[CommandHandler]` sin volver a ejecutarlo. Confirma que el `handlers.php` cacheado todavía apunta al nombre antiguo — y que solo `firefly:cache` de nuevo (no `firefly:clear`) lo arregla — demostrando que la caché es una instantánea, no una vista en vivo.
3. **Andamia toda una vertical con los generadores.** Usa `make:firefly-entity`, `make:firefly-repository`, `make:firefly-handler` y `make:firefly-controller` en secuencia para construir una pequeña característica nueva desde cero, luego ejecuta `firefly:cache` una vez al final y confirma que cada pieza aparece en los manifiestos compilados juntas.

Apéndices

APÉNDICE A

Hoja de Referencia Laravel → LaraFly

LaraFly es **aditivo**: superpone un modelo de aplicación con forma de Spring Boot encima de Laravel 13 — nunca bifurca, envuelve ni reemplaza el framework de debajo. Cada capacidad de LaraFly es un paquete Composer normal que registra proveedores de servicio de Laravel normales; una app LaraFly sigue siendo, en todo aspecto que un desarrollador de Laravel reconocería, una app Laravel. Este apéndice es un mapa de referencia rápida desde el idioma de Laravel que ya conoces hasta el concepto de LaraFly junto al que este libro te lo presentó, con el capítulo que lo cubre en profundidad.

De un vistazo

Aspecto	Laravel puro	LaraFly	Capítulo
Punto de entrada	Proveedores de servicio registrados en <code>bootstrap/providers.php</code> , cableados a mano	Los mismos proveedores, más clases <code>AutoConfiguration</code> autodescubiertas ensambladas por un pipeline de <code>BootPass</code> decidido por el kernel	2
Inyección de dependencias	<code>app()->bind()</code> / <code>app()->singleton()</code> en el <code>register()</code> de un proveedor	Estereotipos <code>#[Component]</code> / <code>#[Service]</code> / <code>#[Repository]</code> / <code>#[Configuration]</code> sobre la propia clase; un escaneo de componentes compilado resuelve las dependencias del constructor	2
Configuración	<code>config('mail.host')</code> (acceso a array, sin tipo)	DTOs <code>#[ConfigProperties]</code> enlazados desde un subárbol de configuración — tipados, de fallo rápido ante una clave faltante/no coincidente	3

Aspecto	Laravel puro	LaraFly	Capítulo
Enrutamiento HTTP	Archivos de ruta <code>routes/web.php</code> / <code>routes/api.php</code>	<code>#[RestController]</code> + atributos de verbo (<code>#[GetMapping]</code> , ...), compilados a un <code>RouteManifest</code> , aún despachados a través de rutas nativas de Laravel	4
Validación	<code>FormRequest::rules()</code> (reglas en array)	Interceptación de parámetro <code>#[Valid]</code> sobre un modelo de restricciones al estilo de Bean Validation, aún respaldado por el validador de Laravel	4
Persistencia	Modelos Eloquent + query builder directamente en el código de aplicación	El mismo Eloquent por debajo de un puerto/adaptador <code>CrudRepository</code> — el código de aplicación depende de una interfaz, nunca de Eloquent directamente	5

Aspecto	Laravel puro	LaraFly	Capítulo
Modelado de dominio	Los modelos Eloquent llevan tanto datos como comportamiento	<code>Firefly\Domain\Entity</code> /agregados sin dependencia de framework; Eloquent vive solo en el adaptador	6
Transacciones	<code>DB::transaction(fn () => ...)</code> (con ámbito de clausura)	<code>#[Transactional]</code> sobre una clase/método — reglas declarativas de propagación/aislamiento/rollback, un proxy generado dirige <code>beginTransaction / commit / rollback</code>	9
Eventos	<code>Event::listen() / #[AsEventListener]</code> , solo en-proceso	Dos superficies distintas: el bus en-proceso <code>(#[AsEventListener])</code> y un bus de EDA respaldado por broker <code>(#[EventListener])</code>	8

Aspecto	Laravel puro	LaraFly	Capítulo
CQRS	No es un concepto de primera clase — un "command" de Laravel normalmente significa un comando de consola de Artisan	<code>CommandBus</code> / <code>QueryBus</code> con un pipeline acotado (validar → autorizar → invocar → métricas) y <code>#[CommandHandler]</code> / <code>#[QueryHandler]</code>	7
Seguridad	Guards de auth + middleware de ruta (<code>- >middleware('auth')</code> , <code>Gate::allows()</code>)	<code>SecurityContext</code> con forma de Spring Security 6 + DSL de URL <code>HttpSecurity</code> de denegar-por-defecto + seguridad de método (<code>#[PreAuthorize]</code> / <code>#[Secured]</code> / <code>#[RolesAllowed]</code>)	10
Operaciones	Nada integrado — las comprobaciones de salud y las métricas suelen ser a medida o un paquete	<code>firefly/actuator</code> (health/info/env/beans/conditions/mappings/loggers/scheduledtasks) + <code>firefly/observability</code> (métricas Prometheus/Micrometer-JSON)	11

Aspecto	Laravel puro	LaraFly	Capítulo
Pruebas	<code>RefreshDatabase</code> , <code>Event::fake()</code> , <code>Bus::fake()</code> — solo ven las propias primitivas de Laravel	<code>FireflyTestCase</code> / <code>FireflyDatabaseTe stCase</code> , dobles de grabación para los propios puertos de Firefly, expectativas de Pest con sabor a Firefly	12
Andamiaje	<code>php artisan make:controller / make:model</code>	<code>php artisan make:firefly- controller / make:firefly- handler /...</code> — un generador por estereotipo	13
Rendimiento de arranque	La propia caché de config/rutas de Laravel (<code>config:cache</code> , <code>route:cache</code>)	<code>php artisan firefly:cache</code> — un comando compila cada manifiesto de Firefly y genera proxies <code>#[Transactional]</code> para un arranque sin reflexión	13

Punto de entrada: proveedores de servicio frente a auto-configuración

Laravel arranca registrando los proveedores de servicio que `bootstrap/providers.php` lista, en el orden en que aparecen (o, para los proveedores descubiertos por paquete, aproximadamente en orden alfabético). El `firefly/context` de LaraFly se sienta encima de eso: el proveedor de cada paquete de capacidad todavía existe y todavía lo descubre Laravel exactamente de la misma manera — pero su método `register()` no hace nada excepto *acumular* sus contribuciones de `BootPass` en un colector `PendingBootPasses`. El `FireflyKernel` compartido drena ese buffer una vez que se resuelve realmente y ejecuta cada fase (escaneo, evaluación de condiciones, registro de beans, pasadas de cableado) en un orden decidido por el kernel — de modo que qué proveedor instanció Laravel primero es irrelevante.

Inyección de dependencias: `app()->bind()` frente a estereotipos

El contenedor de Laravel es potente pero explícito — un enlace vive en un proveedor, separado de la clase que enlaza:

```
final class GreetingServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->bind(GreetingService::class, fn () => new GreetingService('Hello'));
    }
}
```

El `firefly/container` de LaraFly superpone atributos de PHP 8 sobre `Illuminate\Container` en su lugar — marca la propia clase con `#[Component]` (o `#[Service]` / `#[Repository]` / `#[Configuration]`), todas especializaciones de él) y un escaneo de componentes compilado la registra, resolviendo las dependencias del constructor por tipo:

```
#[Service]
final readonly class GreetingService
{
    public function __construct(private string $greeting = 'Hello') {}
}
```

Nada te impide también usar `app()->bind()` directamente — los dos coexisten, y el Capítulo 2 te mostró exactamente cuándo cada uno es la mejor opción.

Configuración: `config()` frente a `#[ConfigProperties]`

`config('mail.port')` es acceso a array sin tipo — una errata o una clave faltante devuelve `null` silenciosamente. `firefly/config` añade DTOs

`#[ConfigProperties('mail')]` que enlazan todo un subárbol de configuración sobre una clase readonly sencilla de una vez, tipados y de fallo rápido ante una clave requerida faltante. Los archivos `config/*.php` de Laravel siguen siendo la única fuente de verdad; LaraFly los lee, no los reemplaza.

Transacciones: `DB::transaction()` frente a `#[Transactional]`

`DB::transaction(fn () => ...)` limita una transacción al ámbito de una clausura — las reglas de propagación y rollback son las que escribas en línea. `#[Transactional]` declara propagación (`REQUIRED`, `REQUIRES_NEW`, `NESTED`, ...), aislamiento, solo-lectura, y listas de excepciones de rollback/no-rollback sobre la propia clase o método; un proxy generado en tiempo de escaneo (Capítulo 9) dirige la frontera a través de `beginTransaction()` / `commit()` / `rollback()` manuales, de modo que una excepción capturada aún puede commitearse cuando coincide con `noRollbackFor`.

Seguridad: middleware frente a `HttpSecurity` de denegar-por-defecto

La postura por defecto de Laravel es permisiva: una ruta es pública a menos que le adjuntes `->middleware('auth')` o una comprobación de habilidad. `firefly/security` invierte eso — el DSL de URL `HttpSecurity` construye una lista ordenada de reglas y deniega cualquier cosa sin coincidencia (401 anónimo, 403 autenticado), el mismo modelo de denegar-por-defecto que Spring Security. La seguridad de método (`#[PreAuthorize]`, `#[Secured]`, `#[RolesAllowed]`) se hace cumplir en el bus de CQRS y en el despachador de controladores mediante un evaluador de expresiones de lista blanca cerrada — sin ningún `eval()` en toda la cadena (Capítulo 10). Los propios guards de auth y middleware de Laravel siguen funcionando por debajo; `firefly/security` es una capa más estricta encima de ellos, no una bifurcación.

Operaciones: actuador y observabilidad

Laravel no trae ningún endpoint de comprobación de salud ni de métricas de fábrica — la mayoría de los equipos o bien improvisan uno o recurren a un paquete. `firefly/actuator` (health/info/env/beans/conditions/mappings/loggers/scheduledtasks bajo `/actuator`, asegurado enteramente por configuración `HttpSecurity` ordinaria) y `firefly/observability` (una exposición `/metrics` de primera parte Prometheus-0.0.4 + Micrometer-JSON) son los análogos de Spring Boot Actuator y Micrometer, respectivamente — ambos paquetes Composer opcionales, ambos seguros por defecto (Capítulo 11).

Andamiaje y rendimiento de arranque: `make:* / config:cache` frente a `make:firefly-* / firefly:cache`

Los propios generadores `make:controller / make:model` de Laravel y sus comandos de rendimiento de arranque `config:cache / route:cache` tienen análogos directos en LaraFly, extendidos para cubrir cada estereotipo de framework en lugar de solo los modelos Eloquent y las rutas: `make:firefly-*` (ocho generadores, uno por estereotipo) y `firefly:cache` (que compila cada manifiesto de Firefly — componentes, rutas, handlers de CQRS, listeners de eventos, tareas programadas, reglas de seguridad, metadatos transaccionales — más genera las clases proxy `#[Transactional]`, en un solo comando) — véase el Capítulo 13.

PARIDAD CON LARAVEL

Cada fila de esta página está pensada para leerse en una sola dirección: LaraFly nunca te pide renunciar a un idioma de Laravel, te pide recurrir además a uno más estructurado allí donde el código base se beneficia de ello. Un proyecto LaraFly puede mezclar libremente `app()->bind()` y `#[Service]`, una ruta `routes/api.php` sencilla y un `#[RestController]`, en el mismo código base — nada en las muestras de este libro requiere una migración de todo-o-nada.

APÉNDICE

Glosario

Actuator — La superficie de gestión de producción de LaraFly: endpoints de framework (`/actuator/health`, `/actuator/info`, `/actuator/env`, y otros) montados directamente sobre el mismo `Router` de Illuminate que usan tus propios controladores, seguros por defecto e inalcanzables hasta que se exponen explícitamente. El análogo de Spring Boot Actuator, entregado por `firefly/actuator` (Capítulo 11).

Adapter — Una clase concreta que implementa un puerto delegando en una tecnología específica — un repositorio respaldado por Eloquent, un broker de mensajes respaldado por Kafka. Los adaptadores viven en el borde de la arquitectura hexagonal y pueden intercambiarse sin tocar el código de dominio ni de aplicación (Capítulos 5, 6).

Aggregate root — El único punto de entrada a un grupo de objetos de dominio que deben permanecer consistentes juntos. Todos los cambios de estado se enrutan a través de sus métodos, que hacen cumplir las invariantes y registran eventos de dominio; el código externo carga y guarda solo la raíz, nunca sus objetos internos.

`Wallet` es la raíz de agregado de Lumen — su única invariante es que el saldo nunca se vuelve negativo (Capítulo 6).

ApplicationContext — La fachada del motor de arranque: el objeto que cada prueba y cada consumidor en tiempo de petición resuelve para alcanzar un bean arrancado, un manifiesto compilado, o el propio estado del kernel. `fireflyContext()` es cómo lo alcanza una subclase de `FireflyTestCase` (Capítulos 2, 12).

Attribute (PHP 8) — Sintaxis nativa de PHP 8 (`#[Component]` , `#[CommandHandler]` , `#[Transactional]` , ...) que LaraFly lee en tiempo de escaneo para descubrir el rol arquitectónico de una clase. Los atributos reemplazan la configuración dirigida por anotaciones que Spring Boot expresa en Java; un manifiesto compilado es lo que hace que leerlos sea un coste único, no por petición (Capítulo 2).

Authentication — Un token inmutable que responde "¿quién está haciendo esta petición?" — un principal, sus autoridades concedidas, y si realmente ha sido autenticado. `Authentication::authenticated()` / `unauthenticated()` son sus dos únicos constructores (Capítulo 10).

Authorization — La pregunta "¿le está permitido a este principal hacer esto en concreto?", respondida independientemente de la autenticación por las reglas de URL de denegar-por-defecto de `HttpSecurity` y por la seguridad de método (`#[PreAuthorize]` , `#[Secured]` , `#[RolesAllowed]`) (Capítulo 10).

Autoconfiguration — Una clase `#[Configuration]` , ordenada por `#[Order(N)]` , cuyos métodos de fábrica `#[Bean]` registran los beans por defecto de una capacidad — siempre protegidos por `#[ConditionalOnMissingBean]` de modo que una sobrescritura de la aplicación gana. El `#[Order(500)]` de `ObservabilityAutoConfiguration` ganando la carrera de `CqrsMetrics` antes de que `CqrsAutoConfiguration` evalúe en `#[Order(1000)]` es el ejemplo canónico de que el ordenamiento decide qué valor por defecto sobrevive (Capítulos 2, 7, 10, 11).

Autowiring — Resolver los parámetros del constructor de un bean solo a partir de sus tipos declarados, sin código de fábrica escrito a mano. El escaneo de componentes compilado de LaraFly realiza esto una vez, en tiempo de caché, en lugar de por reflexión en cada petición (Capítulo 2).

Bean — Cualquier objeto que el contenedor crea, cablea y gestiona. Una clase se convierte en bean llevando un atributo de estereotipo (`#[Component]` , `#[Service]` , `#[Repository]` , `#[Configuration]`), o mediante un método de fábrica con atributo `#[Bean]` dentro de una clase `#[Configuration]` (Capítulo 2).

BootPass — Una única unidad ordenada de trabajo en tiempo de arranque — escaneo, evaluación de condiciones, registro de beans, montaje de rutas — ejecutada por el `FireflyKernel` en una `BootPhase` específica. `ActuatorRouteRegistrar` y `MeterBindingsPass` son ambos `BootPass` (Capítulos 2, 11).

Circuit breaker — Un patrón de resiliencia que salta a un estado abierto tras un umbral de fallos consecutivos contra una dependencia, cortocircuitando llamadas posteriores hasta que la dependencia haya tenido tiempo de recuperarse. El `MeterBindingsPass` de `firefly/observability` expone el estado en vivo de cada breaker configurado como un gauge `resilience_circuit_breaker_state{name}` (Capítulo 11).

Command (CQRS) — Un objeto que expresa una única intención de escritura — "retirar fondos", "abrir una wallet" — despachado a través del `CommandBus` a exactamente un `#[CommandHandler]` . Los comandos pueden ser denegados antes de llegar siquiera a su handler por un `CommandAuthorizer` (Capítulos 7, 10).

CommandBus — El pipeline que recibe un `Command` , lo pasa por una secuencia acotada de etapas (correlacionar → validar → autorizar → invocar → métricas), y lo enruta a su único `#[CommandHandler]` registrado (Capítulo 7).

Component — El atributo de estereotipo genérico para un bean gestionado que no encaja en los roles más específicos `#[Service]` / `#[Repository]` / `#[Configuration]` . Todos los estereotipos son equivalentes para el contenedor; la distinción es para los lectores humanos y las herramientas (Capítulo 2).

Condition (#[ConditionalOn*]) — Un atributo (#[ConditionalOnProperty], #[ConditionalOnBean], #[ConditionalOnMissingBean]) evaluado durante la pasada de condiciones del pipeline de arranque para decidir si una definición de bean sobrevive. El #[ConditionalOnProperty(matchIfMissing: true)] de DbHealthIndicator es lo que lo mantiene activo por defecto; lo que evita un DOWN sorpresa en una aplicación sin base de datos es otra interfaz, ConditionalHealthIndicator::available() (Capítulos 2, 11).

Deptrac — El linter estático de fronteras de arquitectura (deptrac/deptrac) que hace cumplir qué paquete puede depender de cuál, como capas declaradas en deptrac.yaml. Un recuento de 0 violaciones es parte de la propia definición de terminado de este libro para el código de cada capítulo (Capítulos 2, 11, 13).

Domain event — Un registro inmutable de un hecho de negocio que ya ha ocurrido — "wallet abierta", "fondos retirados" — registrado por una raíz de agregado y publicado, tras un commit exitoso, a través de un ApplicationEventPublisher (Capítulos 6, 8, 9).

DTO (Data Transfer Object) — Un objeto sencillo que transporta datos a través de una frontera de capa — un cuerpo de petición, un subárbol de configuración enlazado con #[ConfigProperties] — sin exponer tipos de dominio internos (Capítulos 3, 4).

Entity — Un objeto de dominio con una identidad estable que persiste a través de los cambios de estado. El make:firefly-entity de LaraFly andamia una clase que extiende Firefly\Domain\Entity — no hay ningún atributo #[Entity] (Capítulos 6, 13).

EventListener — Cualquiera de dos superficies distintas y deliberadamente no relacionadas: `#[AsEventListener]` para el propio despacho de eventos en-proceso de Laravel, y `#[EventListener]` para el bus respaldado por broker de `firefly/eda`, que se suscribe a un *patrón* de tipo de evento en lugar de a una clase de PHP (Capítulo 8).

Granted authority — Una única cadena de permiso o rol (`'ROLE_ADMIN'`, `'orders:read'`) llevada por un `Authentication`. `hasRole('X')` se normaliza a una comprobación contra la autoridad concedida `'ROLE_X'` (Capítulo 10).

Health indicator — Un SPI de un método (`HealthIndicator::health(): Health`) que un `#[Component]` implementa para reportar si alguna dependencia o recurso está arriba. `HealthEndpoint` agrega cada indicador registrado al único `Status` más severo (Capítulo 11).

Hexagonal architecture — Un estilo arquitectónico que coloca la lógica de dominio y de aplicación en el centro, rodeada por puertos (interfaces) que los adaptadores implementan en los bordes. El código de negocio depende solo de un puerto; qué adaptador lo satisface en tiempo de ejecución es una decisión de cableado, nunca de lógica de negocio (Capítulos 1, 5, 6).

Integration event — Un evento de dominio que ha cruzado la frontera desde el modelo de dominio en-proceso hacia el bus de EDA respaldado por broker, mediante el puente protegido de después-del-commit presentado junto a CQRS. No es el mismo objeto, ni la misma superficie, que el evento de dominio que lo produjo (Capítulos 7, 8, 9).

JWT (JSON Web Token) — Un token portador firmado y caducable que `JwtService` emite y valida. El `JwtService` de LaraFly se niega a arrancar en absoluto con un secreto de firma débil o de marcador de posición, y se niega a aceptar cualquier token que carezca de su claim `exp`, por muy válida que sea su firma (Capítulo 10).

Manifest — Una instantánea compilada y cacheable de todo lo que un escáner encontró — rutas, handlers de CQRS, listeners de eventos, reglas de seguridad, métodos transaccionales — escrita en `bootstrap/cache/firefly/` por `firefly:cache` y vuelta a cargar en el arranque sin ninguna reflexión (Capítulos 2, 13).

Meter — Una única medida con nombre y etiquetas — un `Counter`, un `Gauge` o un `Timer` — mantenida por un `MeterRegistry`. El registro es idempotente por `type|name|sorted-tags`; volver a registrar un nombre bajo un tipo diferente es un error de programación que el registro rechaza ruidosamente (Capítulo 11).

Method security — Autorización declarada por atributos (`#[PreAuthorize]`, `#[Secured]`, `#[RolesAllowed]`) hecha cumplir en el bus de CQRS y en el despachador de controladores por un evaluador de expresiones de lista blanca cerrada que nunca llama a `eval()` (Capítulo 10).

Outbox pattern — Una técnica para publicar un evento de dominio de forma fiable junto a la escritura de base de datos que lo produjo: ambos se escriben en la misma transacción local, eliminando la necesidad de un commit de dos fases entre la base de datos y el broker de mensajes (Capítulos 8, 9).

Pest expectation — Un matcher personalizado `expect()->extend(...)` que `firefly/testing` instala — `toHavePublished`, `toHaveHandledCommand`, `toBeUp`, `toHaveRecordedMetric`, `toBeProblemDetails` — registrado una vez desde el `tests/Pest.php` raíz del monorepo (Capítulo 12).

Port — Una interfaz de la que depende una pieza de lógica de negocio, sin ningún detalle de implementación adjunto. `EventPublisher`, `CommandBus`, `HealthIndicator` y `MeterRegistry` son todos puertos; el adaptador concreto que satisface a cada uno es una decisión de cableado (Capítulos 1, 5, 11).

Principal — El "quién" de una petición — el `mixed $principal` que lleva un `Authentication` una vez autenticado, junto con sus autoridades concedidas (Capítulo 10).

Problem Details (RFC 7807) — La forma de error estándar `application/problem+json` (`type` , `title` , `status` , más claves de extensión específicas del framework como `code` / `category`) como la que renderiza cada excepción sin manejar. `ProblemDetailsRenderer` es lo que la produce (Capítulos 4, 12).

Projection — Un modelo de lectura construido consumiendo un flujo de eventos de dominio, mantenido separado del modelo de escritura que los produjo — el libro mayor de Lumen es una proyección sobre los propios eventos de dominio de `Wallet` (Capítulo 6).

Proxy (#[Transactional] proxy) — Una subclase generada que envuelve un método con atributo `#[Transactional]` con fronteras de transacción declarativas (`beginTransaction()` / `commit()` / `rollBack()`), producida por el mismo `ProxyClassGenerator` sin reflexión ya sea en tiempo de arranque (dev) o de forma anticipada por `firefly:cache` (producción) (Capítulos 9, 13).

Query (CQRS) — Un objeto que expresa una intención de lectura — "obtener el saldo de la wallet" — despachado a través del `QueryBus` a exactamente un `#[QueryHandler]` . Las consultas nunca mutan el estado (Capítulo 7).

QueryBus — La contraparte del lado de lectura de `CommandBus` : recibe una `Query` , la enruta a su `#[QueryHandler]` , y (el pipeline del Capítulo 7) puede servir un resultado cacheado de forma transparente.

Recording double — Una implementación sencilla de un puerto real de Firefly que graba lo que vio, de modo que una prueba puede afirmar sobre el comportamiento — `RecordingEventPublisher`, `RecordingCommandBus`, y ocho hermanos vienen en `firefly/testing`, uno por cada puerto que los propios `Event::fake()` / `Bus::fake()` de Laravel no pueden ver (Capítulo 12).

Repository — Una abstracción similar a una colección sobre la persistencia, que permite al código de aplicación cargar y guardar agregados o entidades sin ningún SQL propio. `CrudRepository` es la base tipada que extiende cada repositorio de dominio (Capítulo 5).

Secure by default — La postura operativa de LaraFly allí donde aplica: un endpoint de actuador es inalcanzable (404) hasta que se expone explícitamente, una regla de URL de `HttpSecurity` deniega cualquier cosa sin coincidencia, y un secreto JWT que parece débil se niega a arrancar en absoluto — el sistema falla cerrado, no abierto (Capítulos 10, 11).

SecurityContext / SecurityContextHolder — Una instantánea inmutable del `Authentication` actual, mantenida por petición por un accesor estático respaldado por la propia fachada `Context` de Laravel. Cada filtro de autenticación la limpia en un bloque `finally` de modo que el principal de una petición nunca pueda filtrarse a la siguiente (Capítulo 10).

Service — Un bean gestionado, con atributo `#[Service]`, que alberga lógica de negocio y orquesta llamadas a repositorios, publicadores de eventos y otros servicios (Capítulo 2).

Slice test — Una prueba que arranca solo los beans que necesita una estrecha vertical — el pipeline web sobre un controlador (`WebSliceTestCase`), o el pipeline de datos sobre un repositorio (`DataSliceTestCase`) — en lugar de toda la aplicación, de modo que una rebanada mal cableada falla rápido en lugar de pasar por accidente (Capítulo 12).

Stereotype — Un atributo que a la vez registra una clase como bean y señala su rol arquitectónico — `#[Service]` , `#[Repository]` , `#[Component]` , `#[Configuration]` , `#[RestController]` . Cada estereotipo es una especialización del `#[Component]` base (Capítulo 2).

Testcontainers — Una biblioteca que arranca contenedores Docker reales para pruebas de integración y los desmonta después. El trait `RequiresDocker` y el helper `fireflyConfigFor()` de `firefly/testing` hacen que una prueba respaldada por testcontainers sea opcional y segura para CI incluso sin Docker presente (Capítulo 12).

Transactional (`#[Transactional]`) — Un atributo de clase/método que declara propagación (`REQUIRED` , `REQUIRES_NEW` , `NESTED` , ...), aislamiento, solo-lectura y reglas de rollback — hecho cumplir por un proxy generado en lugar de una clausura manual `DB::transaction()` (Capítulo 9).

Value object — Un objeto de dominio inmutable identificado por su valor en lugar de por un campo de identidad; dos objetos de valor son iguales si todos sus campos son iguales. `Money` — un importe entero en unidades menores más una `Currency` — es el objeto de valor canónico de Lumen (Capítulo 6).

Zero-reflection boot — El estado que una aplicación alcanza una vez que `firefly:cache` ha ejecutado: cada manifiesto se carga desde un archivo PHP precompilado bajo `bootstrap/cache/firefly/` , de modo que ningún atributo, clase o método vuelve a someterse a reflexión en tiempo de petición (Capítulo 13).