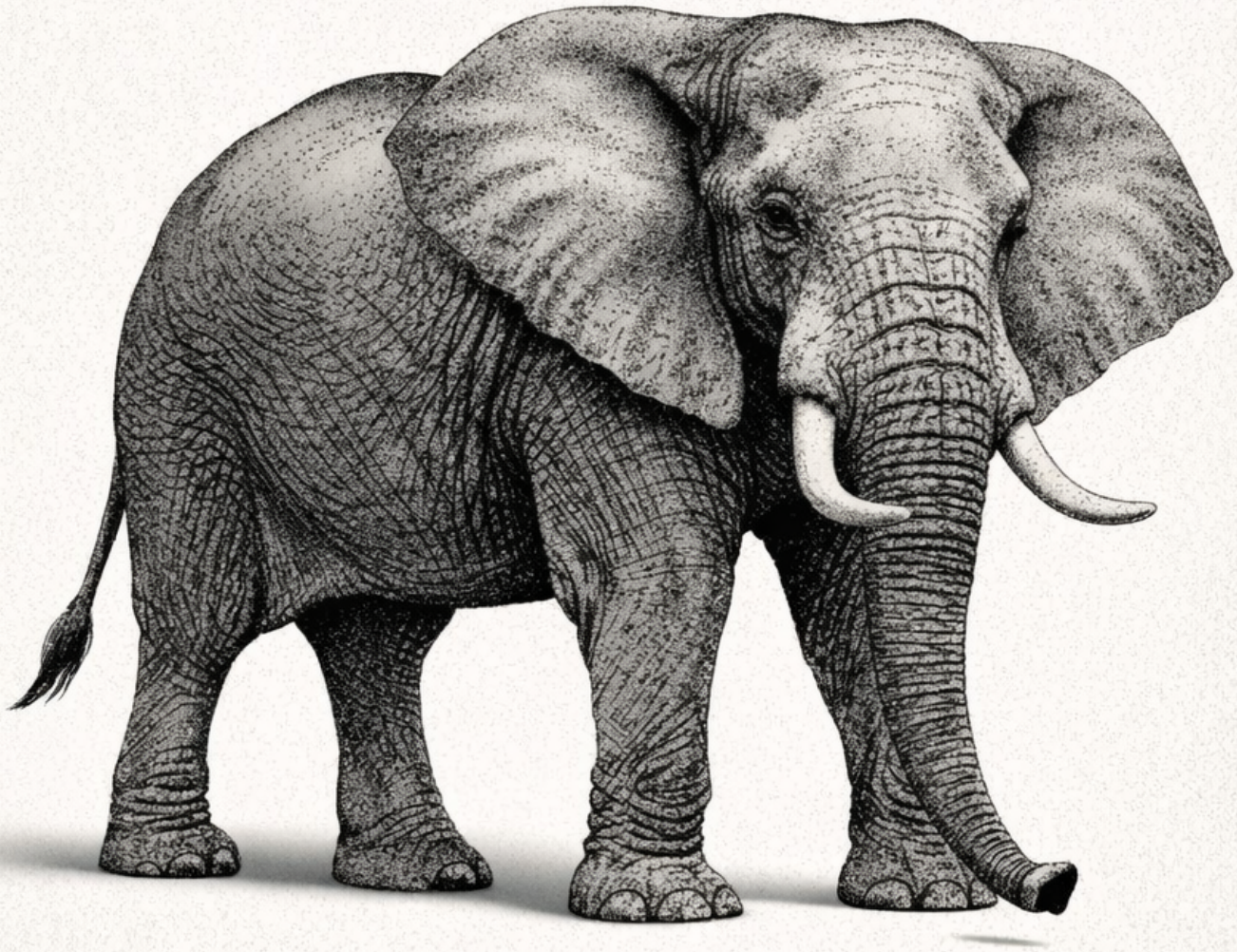
Laravel,
shipping fast



Julian Beaujardin

Table of Contents

Prefacio

Laravel se ha ido puliendo durante años de producción. Se ocupa del enrutado, la validación, el acceso a datos, la autenticación, las colas y las pruebas con un vocabulario coherente.

La trampa es pelearse con esas convenciones: envolver Eloquent en repositorios «por si acaso», sustituir los FormRequests por validación propia, abstraer el framework fuera del framework.

Este libro toma el camino opuesto: apóyate en los patrones de Laravel porque están probados. Controladores, FormRequests, DTOs, Resources, Facades —existen porque previenen errores, permiten pruebas y escalan de forma fiable.

Comprométete con las convenciones de Laravel y dejarás de discutir sobre los cimientos. Empezarás a publicar.

Lo aburrido escala. Lo ingenioso se descompone.

De qué trata este libro

Este libro muestra una forma práctica y repetible de construir APIs con Laravel que sean fáciles de extender, fáciles de probar y predecibles para quien las consume. Usa una API real en producción como ejemplo conductor y aplica una estructura consistente a la validación, la transformación, las respuestas y las capas de integración. El foco no está en abstracciones ingeniosas, sino en patrones fiables que escalan con los equipos y con los plazos.

Cómo está estructurado

El libro va de los cimientos a la implementación. Los primeros capítulos establecen la arquitectura central y los patrones de respuesta. Los capítulos centrales desarrollan la autenticación, los middlewares y las fronteras de integración. Los últimos cubren la fiabilidad: registro, manejo de errores, rendimiento y operación. Cada capítulo incluye código concreto que refleja el proyecto real, para que puedas mapear los conceptos a ficheros que funcionan.

Convenciones

Los ejemplos de código siguen las mismas convenciones que el repositorio: tipado estricto, tipos de retorno explícitos, FormRequests para la validación, *Response Wrappers* para la estructura del JSON y DTOs para el flujo de datos tipado. El término «Response Wrapper» se refiere a las clases de respuesta `Responsable` como `ModelResponse`, `CollectionResponse`, `MessageResponse` y `ErrorResponse`. Los ejemplos favorecen las facades de Laravel y los patrones de configuración de Laravel 13.

Algunos términos se quedan en inglés a propósito, porque es como se dicen en el oficio: *endpoint*,

job, *worker*, *middleware*, *token*. Traducirlos haría el texto más difícil de leer, no más fácil.

Para quién es este libro

Está escrito para desarrolladores de Laravel que se manejan con HTTP, REST y programación orientada a objetos. No necesitas ser experto en Laravel, pero sí trabajar con el framework habitualmente.

Puedes seguirlo con un portátil, PHP 8.4, una base de datos y un editor. Los ejemplos vienen de APIs reales en producción, así que el enfoque es práctico de principio a fin.

Si has heredado APIs inconsistentes o has visto cómo se acumulaba la complejidad, este libro ofrece un camino claro y repetible. No promete perfección, pero aspira a la cordura.

Si trabajas en equipo, este enfoque mantiene los endpoints legibles, ayuda a que la gente nueva se incorpore rápido y evita que el conocimiento se marche por la puerta.

Si quieres publicar funcionalidades ya, este libro es para ti. Se centra en patrones que puedes repetir del primer endpoint al número cien sin repensar la arquitectura.

Sobre esta primera edición

Esta edición se basa en Laravel 13 y PHP 8.4, y refleja la estructura moderna de aplicación introducida en las versiones recientes. Los ejemplos y convenciones se alinean con la base de código de la Webplo License API, incluidos los *Response Wrappers*, los FormRequests, los DTOs y la arquitectura basada en middlewares. El objetivo es una base estable que las ediciones futuras puedan ampliar sin cambiar el enfoque de fondo.

Este libro es una filosofía, no un catálogo de patrones

Aquí no encontrarás cuarenta diagramas de arquitectura. No encontrarás una abstracción a medida que «prepare tu código para el futuro».

Encontrarás:

- Una estructura de API simple y repetible
- Reglas claras para controladores, facades y validación
- Un enfoque pragmático de autenticación, autorización y versionado
- Una preferencia por borrar antes que por inventar

Este es el enfoque que uso cuando el objetivo no es impresionar a otros desarrolladores, sino publicar cada semana, incorporar gente rápido y mantener la complejidad bajo control.

Si buscas construir «la arquitectura perfecta», no la encontrarás aquí.

Pero si quieres publicar APIs que sobrevivan a la presión, a los plazos, a los giros de producto, a

desarrolladores junior y a restricciones heredadas, estás en el sitio correcto.

La filosofía es esta: no esperes a que llegue la complejidad para introducir estructura, y no añadas complejidad que aún no ha llegado. Empieza con patrones simples y consistentes desde el primer día. Usa DTOs para tipar tus datos, Resources para transformarlos, *Response Wrappers* para que las respuestas sean predecibles, Facades para abstraer la lógica de los proveedores y FormRequests para validar. Son valores por defecto ligeros y sensatos que mantienen la estructura estable mientras la lógica de negocio evoluciona.

Qué vas a sacar de este libro

- **Reducir los debates de arquitectura en un 80 %.** Decide una vez, aplica en todas partes. Se acabó el «¿deberíamos usar un repositorio aquí?» en cada endpoint.
- **Incorporar desarrolladores en días en lugar de semanas.** Lee un endpoint y entiende todos. Ese es el objetivo.
- **Menos errores causados por la inconsistencia.** Los patrones consistentes reducen la superficie donde se pueden esconder los fallos.
- **Añadir endpoints sin tocar cinco capas.** Sigues un patrón, escribes la lógica de negocio y publicas.

Principios

Este libro sigue cuatro principios. No son opiniones: son lecciones de sistemas en producción que atienden miles de millones de peticiones, de equipos que publican a diario y de APIs que sobreviven al caos sin heroicidades.

1. **La consistencia gana al ingenio.** Cada endpoint sigue el mismo patrón: validación, estructura de respuesta y manejo de errores. Al principio parece aburrido, pero en realidad es libertad. Cuando la estructura es predecible, tu equipo puede centrarse en los problemas de negocio.
2. **La disciplina es velocidad.** Las restricciones no son cadenas, son aceleradores. Cuando acuerdas la estructura por adelantado, eliminas la fatiga de decisión y publicáis más rápido juntos.
3. **Simple y consistente hoy escala mejor que ingenioso y a medida hoy.** No empieces con microservicios, CQRS, *event sourcing* ni abstracciones pesadas salvo que de verdad las necesites. Empieza con patrones aburridos y predecibles. Cuando aparezcan restricciones reales, sabrás exactamente dónde añadir sofisticación.
4. **Construye para publicar, no para impresionar.** El objetivo es valor, no trofeos de

arquitectura. Publica, mide el impacto, itera. Una API publicada va muy por delante de una API impecable atascada en preproducción.

Qué pasa cuando no los sigues

He visto esto repetirse decenas de veces.

Un equipo construye una API con patrones inconsistentes. El controlador A valida con una clase propia. El B valida en línea. El C hace un híbrido. Los formatos de respuesta varían. Los errores son inconsistentes. Los nombres de campo derivan.

Seis meses después, la base de código se convierte en arqueología. Las revisiones se alargan porque no hay una referencia. Cada funcionalidad nueva tarda el doble porque estás peleando con el código en lugar de extenderlo.

O peor: un equipo gana el debate sobre «arquitectura correcta» el primer día. Implementan CQRS. O *event sourcing*. O microservicios. Cero usuarios. Cero problemas. Pero ahora cada funcionalidad exige atravesar cinco capas antes de escribir lógica de negocio. La velocidad se desploma de inmediato. La arquitectura que iba a «preparar el sistema para el futuro» se convierte en lo que impide publicar. Ambos fracasos comparten la misma enfermedad: inconsistencia y complejidad prematura.

Cómo aprendí esto

Pasé tres años en una empresa construyendo una API que empezó preciosa. El primer endpoint era perfecto. Teníamos un patrón. Todo el mundo lo entendía. Luego contratamos a dos desarrolladores más.

Uno se saltó los resources y devolvía arrays en crudo. Otro validaba en línea porque los FormRequests le parecían «demasiado abstractos». Alguien quería DTOs en todas partes por «seguridad de tipos», lo que derivó en formas de respuesta distintas. Otro envolvió Eloquent en repositorios «por si cambiamos de base de datos».

Nadie estaba equivocado. Cada decisión tenía sentido por sí sola.

En conjunto, construimos una API donde no podías leer dos endpoints y entender el tercero. Cada funcionalidad nueva empezaba con «¿cómo lo hicimos la última vez?», y la respuesta cambiaba cada vez. Un endpoint cacheaba agresivamente. Otro no cacheaba nada. Uno devolvía códigos de estado correctos; otro enterraba los errores dentro de respuestas 200.

Para el mes seis no publicábamos: hacíamos análisis forense. Añadir un endpoint costaba tres días en lugar de tres horas. Las revisiones se convertían en debates de arquitectura. La gente junior se ralentizaba porque tenía que aprender cuarenta y siete patrones distintos en lugar de uno.

Pull requests que deberían haber tardado una hora se quedaban días. No porque la lógica estuviera mal, sino porque cada endpoint desataba una discusión nueva. Y ninguna discusión se resolvía

nunca, porque no teníamos cimientos, sólo precedentes contradictorios.

Recuerdo una reunión diaria en la que el equipo entero estaba bloqueado. No en funcionalidades: en entender el código del anterior. Estábamos gastando tiempo de ingeniería descifrando inconsistencia en lugar de resolver problemas.

Lo irónico es que no teníamos problemas de rendimiento ni de escalado. La «arquitectura flexible» que creíamos estar construyendo se convirtió en lo más inflexible posible. No podías cambiar nada porque nadie entendía por qué estaba hecho así.

Acabamos reescribiendo la API, no porque estuviera rota, sino porque la consistencia se había desmoronado. La reescritura llevó seis semanas porque las reglas no estaban claras.

La segunda vez elegimos un patrón. FormRequests para validar. DTOs para datos tipados. Resources para respuestas. Una estructura de error. Un *Response Wrapper*. Sin excepciones. Sin «este endpoint es especial».

Debatimos una vez, nos comprometimos y seguimos adelante.

La diferencia fue de la noche al día. La velocidad no sólo volvió: se compuso. Cuando no estás averiguando cómo hacer algo, estás resolviendo el problema de negocio de verdad.

Desde entonces he usado este mismo patrón en varias APIs en producción, y el resultado ha sido siempre el mismo: claridad primero, velocidad después, escala al final.

Años más tarde, la misma lista sigue valiendo: usa las funcionalidades que Laravel ya trae, borra antes de añadir y mantén la solución aburrida.

Por eso existe este libro. No porque haya encontrado «la mejor manera», sino porque viví lo que pasa cuando no decides por adelantado, y vi lo que desbloqueaba la consistencia.

Qué significa «shipping fast»

«Publicar rápido» no significa escribir código de forma temeraria. Significa:

- **Decisiones tomadas una vez, aplicadas en todas partes.** Un patrón de validación. Una estructura de respuesta. Un formato de error. Sin debates, sin excepciones, sin casos especiales.
- **Gente nueva productiva en días.** Lee un endpoint. Entiéndelos todos.
- **Se resuelven problemas reales, no imaginarios.** Construye para hoy. Optimiza cuando choques con restricciones reales.
- **Las revisiones se centran en la lógica, no en el estilo.** La consistencia elimina las discusiones estructurales.

- **Los despliegues no te aterran.** La previsibilidad convierte depurar y revertir en rutina, no en drama.

Publicar rápido es publicar de forma predecible, fiable y repetida. Es velocidad que se compone.

La disciplina como velocidad, no como restricción

Las restricciones habilitan la velocidad.

El instinto de un desarrollador suele ser «libertad». Sin reglas. Constrúyelo como te parezca. Pero sin restricción no obtienes libertad: obtienes caos. Y el caos es lento.

Piensa en una autopista: los conductores no van más rápido sin reglas; van más despacio, porque cada conductor es impredecible.

Esto es lo que pasa cuando un equipo acuerda unos patrones:

Dedicas treinta minutos a decidir cómo estructurar la validación. Luego usas ese enfoque en todas partes. Has tomado una decisión y has ahorrado decenas de horas de tiempo colectivo.

Dedicas una hora a definir tu *Response Wrapper*. Todas las respuestas lo siguen. Sin sorpresas para los consumidores. Sin depurar siete formas de respuesta distintas.

Eliges tu estrategia de errores una vez. Todos los errores la siguen. Códigos de estado, mensajes y cargas consistentes.

Esto no es restricción. Es libertad frente a la tiranía de la elección.

Los equipos más rápidos con los que he trabajado no eran los más listos. Eran los más consistentes. Decidían una vez, aplicaban en todas partes y seguían adelante.

Eso es lo que enseña este libro: los patrones que permiten a tu equipo moverse con velocidad.

El mito de lo «avanzado»

Todo desarrollador pasa por una fase:

Descubres los microservicios. Descubres CQRS. Descubres *event sourcing*. Descubres la arquitectura hexagonal. Y de pronto todo lo que has construido te parece «mal».

Así que reconstruyes. Y vuelves a reconstruir.

Lo que era un simple endpoint `POST /orders` ahora exige seis capas de abstracción, tres transformadores, dos librerías de DTOs y un documento de diseño más largo que tu esquema.

Estos patrones resuelven problemas reales, pero no los problemas que tú tienes todavía.

Netflix usa microservicios porque Netflix *rompió su monolito*. No empezaron ahí. Empezaron simple, chocaron con límites reales y luego evolucionaron.

Esto es complejidad accidental: complejidad que añades porque anticipas problemas que nunca llegaron, o porque quieres lucir conocimiento.

El código ingenioso sienta bien. Impresiona a otros desarrolladores. Pero el ingenio se acumula. Cada abstracción añade otra capa que entender, otra que depurar, otra que mantener.

Multiplica eso por un equipo de cinco. Multiplícalo por dos años.

Acabas con código más difícil de leer, más difícil de cambiar y más difícil de probar. Cuanto más abstracto es tu código, menos gente lo entiende de verdad. Cuando quien lo escribió se va, te quedas manteniendo un sistema que apenas comprendes.

Eso no es arquitectura. Es arqueología.

La verdad incómoda es esta:

La mayoría de las APIs no fracasan por no estar arquitecturadas como Netflix. Fracasan porque nunca se publicaron, nunca se simplificaron o nunca se mantuvieron.

Hay una diferencia.

Resuelve el problema que tienes delante con el código más simple que funcione. Cuando choques con un límite real —rendimiento, escalabilidad, mantenibilidad— añade el patrón sofisticado que resuelve ese problema concreto. Para entonces entenderás por qué lo necesitas. El patrón deja de ser teórico: es la solución a un dolor real.

La complejidad se siente productiva. Publicar es productivo.

Qué hace buena a una API

Antes de seguir, pongámonos de acuerdo en qué estamos intentando construir. La comunidad de diseño de APIs —desde los principios REST de Roy Fielding hasta las guías modernas de Stripe, Twilio y GitHub— ha convergido en unos pocos principios centrales. No son opiniones arbitrarias: son lecciones de miles de millones de peticiones servidas en producción, de APIs que escalaron con suavidad y de APIs que se hundieron bajo su propia complejidad.

Una buena API no es sólo una que funciona. Es una que:

1. **Hace una cosa bien.** Tu API debería tener un propósito claro. Una API de gestión de licencias no debería registrar dominios. Una API de pagos no debería gestionar perfiles de usuario. Este principio —el de responsabilidad única— se aplica a servicios enteros. Cuando una API intenta hacerlo todo, no hace nada bien. Stripe, GitHub y Twilio están enfocadísimos en su dominio. Hacen una cosa y la hacen de forma brillante.

2. **Es predecible.** Las respuestas deben seguir patrones consistentes. Los errores deben estructurarse igual. Los nombres de campo deben ser coherentes. Cuando llamas a `GET /licenses`, la estructura de la respuesta debería casar con la de `POST /license`. Tu API es un contrato con quien la consume; Stripe lo llama «consistencia». Rómpelo y pierden la confianza. Mantenlo y construirán sobre tu API con seguridad.

3. **Es segura por defecto.** La seguridad no debería ser una ocurrencia tardía. Las cabeceras CORS deberían ser restrictivas por defecto. El límite de frecuencia te protege del abuso. La autenticación debería ser obligatoria en las rutas protegidas. Construye la seguridad en los cimientos, no atornillada después. El OWASP API Security Top 10 es claro: la mayoría de las vulnerabilidades vienen de autenticación rota, autorización rota y exposición excesiva de datos. Son fallos de arquitectura, no de criptografía.

4. **Falla con elegancia.** Las redes fallan. Las bases de datos se caen. Los usuarios envían datos malformados. Devuelve códigos HTTP con significado (no todo es 200 o 500). Da detalles de error que ayuden a entender qué pasó. Stripe documenta con precisión qué significa cada código y ofrece códigos de error para manejo programático. No necesitas claves de idempotencia el primer día, pero nunca devuelvas un 500 críptico cuando puedes dar información real.

5. **Es observable.** Cuando algo va mal en producción —y va a ir mal— deberías enterarte de inmediato. Usa registro estructurado. Usa identificadores de petición para trazar operaciones por tu sistema. Usa métricas para vigilar velocidad y fallos. Si no puedes ver tu sistema, no puedes arreglarlo.

6. **Puede evolucionar.** Tu API no está terminada el primer día. Crece y se adapta. Diseña para la evolución: versiona con cuidado, avisa de las deprecaciones, piensa en cómo los campos nuevos no romperán a los consumidores existentes. Tu API no servirá las mismas peticiones dentro de tres años. Diseña sabiéndolo.

Estos principios no son exclusivos de este libro. Están enraizados en décadas de experiencia en el sector. Lo que este libro hace es enseñarte a construirlos dentro de tu aplicación Laravel desde el principio.

La estrategia

Así vamos a enfocarlo:

Primero, construimos con patrones simples desde el primer día. No patrones complejos, no abstracciones «a prueba de futuro», sólo valores por defecto sensatos que usaremos consistentemente en cada endpoint. FormRequests para validar. DTOs para datos tipados. *Mappers* para transformar. Resources para la salida JSON. *Response Wrappers* para estructuras consistentes. Facades para interfaces limpias. No son frameworks pesados: son patrones ligeros que seguirá cada endpoint, del primero al centésimo. Esa consistencia significa que tu equipo aprende una vez y aplica ese conocimiento en todas partes.

Después añadimos capas de defensa. Autenticación. Autorización. Manejo de errores. Límite de frecuencia. No son adornos opcionales: son parte de los cimientos, y te protegen automáticamente salvo que renuncies a ellos de forma explícita. Construyéndolos pronto en la arquitectura no añades complejidad más tarde: evitas el desorden de atornillar la seguridad después.

Luego probamos mientras construimos. No después. No algún día. Mientras escribes cada endpoint, escribes pruebas que verifican que funciona. Esto no va de porcentajes de cobertura. Va de confianza. Con buenas pruebas puedes cambiar código sin miedo. Publicas más rápido porque sabes qué rompiste antes que tus usuarios.

Optimizamos según restricciones reales. No de forma prematura. Pero en cuanto tengas algo funcionando y choques de verdad con un límite —rendimiento, escalabilidad, mantenibilidad— añades los patrones sofisticados que resuelven ese problema concreto. Lo bonito de empezar con patrones simples y consistentes es que sabrás exactamente dónde optimizar y por qué. Has vivido el dolor.

Por último, monitorizamos y escalamos. No puedes arreglar lo que no puedes ver. Montaremos registro, seguimiento estructurado de errores y observabilidad para saber qué pasa en producción. Y como construimos sobre patrones consistentes, añadir monitorización es directo: lo añades una vez en la capa del patrón y te beneficias en todas partes.

La idea clave: no estamos evitando la arquitectura. Estamos evitando la arquitectura *innecesaria*. Empezamos con la estructura mínima viable que previene el caos, y dejamos que los requisitos reales guíen lo que viene después. La mayoría de las APIs nunca necesitan microservicios, CQRS ni *event sourcing*. Pero toda API necesita consistencia, validación, seguridad de tipos y testabilidad. Eso es lo que vamos a construir.

El objetivo de verdad

De esto trata realmente el libro: **publicar rápido y mantener la cordura.** No rápido una semana y luego lento porque te ahogas en deuda técnica. No rápido hoy y arrepentido mañana cuando heredes tu propio desorden. Rápido *ahora*, rápido *el mes que viene*, rápido *dentro de dos años*, cuando todo haya cambiado y el código siga siendo mantenible.

El objetivo real es una API donde:

- Tu equipo puede añadir un endpoint sin una reunión sobre arquitectura
- Alguien junior puede leer un endpoint y entenderlos todos
- Las revisiones duran minutos y no horas, porque no hay nada que debatir
- Los despliegues no te aterran porque sabes qué estás publicando
- Cuando alguien se marcha, la base de código no se va con esa persona
- Cuando el negocio gira, puedes girar tu API sin reescribirlo todo

Esto ocurre cuando abrazas la consistencia desde el primer día. Cuando cada endpoint usa los mismos patrones: FormRequests para validar, DTOs para datos tipados, *Mappers* para transformar, Resources para la salida, *Response Wrappers* para la estructura, Facades para interfaces limpias. Estos patrones no son arquitectura pesada. Son guardarraíles ligeros que previenen el caos.

Esto es lo que mata a la mayoría de las APIs: ni los microservicios, ni la complejidad, ni la mala planificación. Es la inconsistencia. Un endpoint lo hace así, otro lo hace asá. Cada desarrollador interpretó la arquitectura de forma distinta. La base de código se volvió un laberinto de casos especiales y parches. Para cuando alguien intentaba añadir una funcionalidad, dedicaba más tiempo a entender el código existente que a escribir el nuevo.

La alternativa es aburrida. Es el mismo patrón repetido cincuenta veces, cien veces, sin variación. Es DTOs en todas partes. Es validación siempre en FormRequests. Es *Response Wrappers* en todo. A algunos les parece repetitivo. A los equipos con experiencia les parece libertad. Porque cuando la estructura es predecible, tu cerebro puede centrarse en la lógica de negocio. No estás descifrando el patrón. Estás resolviendo el problema.

Este libro te enseña a usar patrones aburridos para publicar rápido, mantener con facilidad y escalar sin pánico. No porque lo aburrido esté de moda, sino porque lo aburrido es lo que te permite centrarte en lo que de verdad importa: entregar valor a tu negocio.

Construyamos algo aburrido.

Y publiquémoslo.

Capitulo 1: Primeros pasos

En lugar de construir otra API «Hola mundo», usemos como guía una API real en producción: la **Webplo License API**. Es lo bastante simple como para entenderla rápido, y lo bastante real como para mostrar los patrones que importan.

La API hace una cosa, y la hace bien: gestiona licencias de sitios web.

Cada sitio de Webplo tiene su propia licencia, emitida automáticamente en cuanto el sitio se aprovisiona. A partir de ahí, los motores de Webplo interactúan con esta API para crear, consultar y revocar licencias según haga falta. No es una API pensada para consumo público: es un microservicio interno, uno de los muchos componentes que mueven Webplo entre bastidores.

Por debajo, esta API se integra directamente con la API de licencias de Statamic, ya que todo sitio de Webplo corre sobre Statamic CMS. La integración está estructurada con el patrón Manager-Facade-Driver, lo que nos permite mantener limpia la lógica central mientras soportamos varios drivers y futuras extensiones.

La API también maneja respuestas multilingües, límite de frecuencia, registro asíncrono y compresión de carga —no porque sea vistoso, sino porque necesita ser fiable.

Es una API en producción sirviendo tráfico real. Vamos a entender cómo funciona y qué hace que funcione bien.

Empezar desde cero

Si quieres seguirlo desde cero, levantemos un proyecto nuevo de Laravel:

```
composer create-project laravel/laravel api-license
cd api-license
php artisan key:generate
```

Eso es todo. Laravel ha hecho casi todo el trabajo pesado. Cuando empecé a programar en PHP hace doce años, montar un framework significaba conectar a mano ficheros de configuración, ajustar scripts de arranque y coserlo todo antes de poder escribir código de verdad. Laravel simplemente dice: «ya lo hemos pensado por ti, aquí tienes unos valores por defecto sensatos».

Ahora, una cosa que quiero que entiendas desde el principio: no estás obligado a usar los valores por defecto de Laravel. *Puedes* personalizarlo todo. Pero esto es lo que he aprendido: cuando te encuentras peleando con el framework, eso suele ser señal de que estás tomando decisiones de las que el framework intenta ahorrarte.

La estructura que tiene sentido

Laravel 13 se ha asentado en una estructura muy limpia. Esto es lo que hace realmente cada carpeta:

```
app/Http/Controllers/       Your API endpoint logic
app/Http/Requests/          Input validation rules
app/Http/Resources/         Response formatting
app/Http/Middleware/        Request/response interception
app/Http/Mappers/           DTO transformations
app/Http/DataObjects/       DTO definitions
app/Http/Integrations/      External API integrations
app/Http/Responses/         Unified Response Wrappers
app/Models/                 Database models (like Bearer token)
app/Services/               Business logic with manager pattern
routes/api.php              Where you define endpoints
tests/Feature/              Testing full request flows
database/migrations/        Database versioning
```

Cada carpeta tiene un propósito. Los controladores manejan HTTP. Los modelos manejan la base de datos. Las facades ofrecen interfaces limpias. Las pruebas verifican que todo funciona. Cuando alguien nuevo se une a tu equipo y mira esta estructura, piensa: «ah, esto ya lo he visto». Esa consistencia importa.

Tu primer endpoint

Empecemos con algo simple: un endpoint de *health check*. Ahí está, diciéndote que la API está viva y en forma. Puede sonar trivial, pero es aquí donde establecemos los patrones que usarás en todos los endpoints que vengan después. Mantenlo simple, mantenlo consistente, y tendrás un plano para todo lo demás.

```php
// routes/api.php
Route::get('/health', HealthController::class);
```

Un *health check* debería ser señal pura: rápido, ligero y de coste cero. Sólo compruebas que los servicios críticos están conectados —base de datos, capa de caché y sistema de colas. Sin consultas profundas. Sin llamadas a APIs externas. Sin efectos secundarios. Sólo un 200 limpio cuando todo va bien, o un 503 cuando algo está roto. Tus sistemas de monitorización pingan este endpoint cada cinco o diez segundos y conocen al instante el estado de tu API sin añadir carga. Ese es todo el propósito, nada más.

Este es un **endpoint público**: sin autenticación, sin middleware (por ahora). Cualquiera puede llamarlo, sin preguntas. Lo pingan tus balanceadores. Lo pingan tus sistemas de monitorización. Lo

pinga tu pipeline de despliegue. Todos comprueban lo mismo: ¿está viva tu API y responden sus servicios?

```
{
    "data": {
        "status": "ok",
        "timestamp": "2026-02-11T21:50:47+00:00",
        "services": {
            "database": "connected",
            "cache": "connected",
            "queue": "active"
        }
    }
}
```

La forma rápida

```
// app/Http/Controllers/HealthController.php
final readonly class HealthController
{
    public function __invoke(): Response
    {
        return response()->json([
            'data' => [
                'status' => 'ok',
                'timestamp' => now()->toIso8601String(),
                'services' => [
                    'database' => 'connected',
                    'cache' => 'connected',
                    'queue' => 'active',
                ],
            ]
        ]);
    }
}
```

Primero, piensa en lo que estamos haciendo.

Estamos atendiendo una petición HTTP. Necesitamos comprobar que los servicios críticos funcionan. Necesitamos devolver una respuesta. Son tres pasos.

A medida que tu API crece, querrás asegurar consistencia, tipado más fuerte, códigos de error con significado que describan claramente qué pasó, localización —sea un 200 OK o un 503 Service

Unavailable— y pruebas exhaustivas que den confianza en tu implementación.

Es entonces cuando la forma rápida empieza a sentirse frágil. Estás copiando y pegando el formato de respuesta entre endpoints. La lógica de validación está dispersa. Tus controladores crecen a más de cien líneas sin separación clara entre lo HTTP y lo de negocio.

Probar se vuelve tedioso porque haces aserciones contra arrays de forma desconocida. Las propiedades y los tipos no están claros hasta que lees la implementación. Estos no son problemas del enfoque rápido en sí. No está *mal*. Simplemente ya no es suficiente. Dicho de otro modo: no es *escalable*.

La forma escalable

```
// app/Http/Controllers/HealthController.php
final readonly class HealthController
{
    public function __invoke(): Responsable
    {
        return new ModelResponse(
            data: new HealthResource(
                resource: HealthDTOMapper::toDTO(...),
            )
        );
    }
}
```

La diferencia es el uso de *Response Wrappers* mediante respuestas `Responsable`, resources, DTOs y *mappers* (o transformadores, como se les llama a veces). No hay magia. En lugar de devolver respuestas en crudo desde los controladores, este enfoque introduce una capa estructurada: las respuestas implementan un contrato común, los datos se envuelven en objetos de respuesta consistentes, la salida se transforma mediante resources y los valores en crudo se mapean a DTOs tipados. El resultado es una API predecible, testable y segura en tipos, donde formato, estructura, transformación y localización están claramente separados de la lógica del controlador.

No son ejercicios académicos. Son soluciones prácticas refinadas con experiencia real en producción. Tus controladores se mantienen finos y enfocados. Quien consume tu API escribe una vez y reutiliza siempre. Alguien nuevo mira un endpoint y los entiende todos. De eso hablaremos a lo largo del libro, porque eso es lo que separa un prototipo rápido de una API real en producción.

Principios para publicar rápido

Esto es lo que hace funcionar a esta arquitectura: **consistencia**. No sólo en el estilo del código, sino en la estructura, los patrones y el enfoque. Cada endpoint sigue los mismos patrones. Cada respuesta usa los mismos *wrappers*. Cada validación ocurre igual. Esa consistencia es lo que permite

a tu equipo publicar rápido y mantener la cordura.

Estos son los principios que la hacen posible:

Convención sobre configuración. Laravel tiene convenciones por una razón: son sabiduría destilada de miles de aplicaciones en producción resolviendo problemas reales. Ve por defecto a la forma de Laravel. Sal de ella sólo cuando tengas una razón genuina y concreta. Esto te ahorra decisiones interminables y mantiene tu API predecible. En cuanto tu equipo acuerda seguir al framework en lugar de pelearse con él, la velocidad cambia.

Consistencia sobre novedad. Cada endpoint debería parecerse al anterior. Los FormRequests validan en todas partes. Los DTOs transforman en todas partes. Los Resources dan formato en todas partes. Cuando entra gente nueva, aprende una forma y la aplica en todo. Cuando necesitas cambiar algo, lo cambias en un sitio y te beneficias en todos. Esto no es repetición sosa: es apalancamiento arquitectónico.

Seguridad de tipos. PHP 8.4 te da las herramientas para atrapar errores antes que tus usuarios. Usa tipos completos en parámetros y retornos, y análisis estático con PHPStan a nivel 9. Esto no es opcional. El tipado fuerte hace que tu IDE autocomplete lo que existe. Sabes qué propiedades hay en tus DTOs. Cuando tu API está fuertemente tipada de punta a punta, las inconsistencias saltan a la vista de inmediato.

Falla rápido. Valida la entrada pronto. Comprueba que las conexiones funcionan. Entérate al momento de que algo va mal. No dejes que los datos malos se propaguen por tu sistema. Cuando lleguen a tu API, atrápalos en la frontera. Los FormRequests validan antes de que corra tu controlador. Los tipos atrapan discrepancias de forma. Si el dato es malo, falla en voz alta y dile al consumidor exactamente qué está mal.

Separación de responsabilidades. Los controladores manejan HTTP. Las facades ofrecen interfaces limpias. Los drivers implementan la lógica real. Los resources transforman la salida. Cuando mantienes esto separado, cada capa se vuelve más simple y más fácil de entender. Alguien nuevo puede leer un controlador y entender el flujo de inmediato: petición → validación → facade → respuesta.

Prueba mientras construyes. No después. No algún día. Mientras construyes. Esto no va de alcanzar porcentajes de cobertura: va de confianza. Cuando pruebas tus endpoints según los construyes, detectas problemas al momento. Sabes que tus patrones funcionan antes de hacer commit. Puedes refactorizar con confianza sabiendo que las pruebas te cubren.

Seguridad por defecto. No construyas la seguridad aparte. Debería ser parte de los cimientos, protegiéndote automáticamente salvo renuncia explícita. El límite de frecuencia debería ser lo normal. La autenticación debería ser automática en las rutas protegidas. Los mensajes de error nunca deberían filtrar información sensible. Cuando la seguridad está horneada en tus patrones, no tienes que acordarte: simplemente ocurre.

Estos principios trabajan juntos para crear una API predecible, mantenible y rápida de construir. La consistencia aburrida no es una limitación. Es un superpoder.