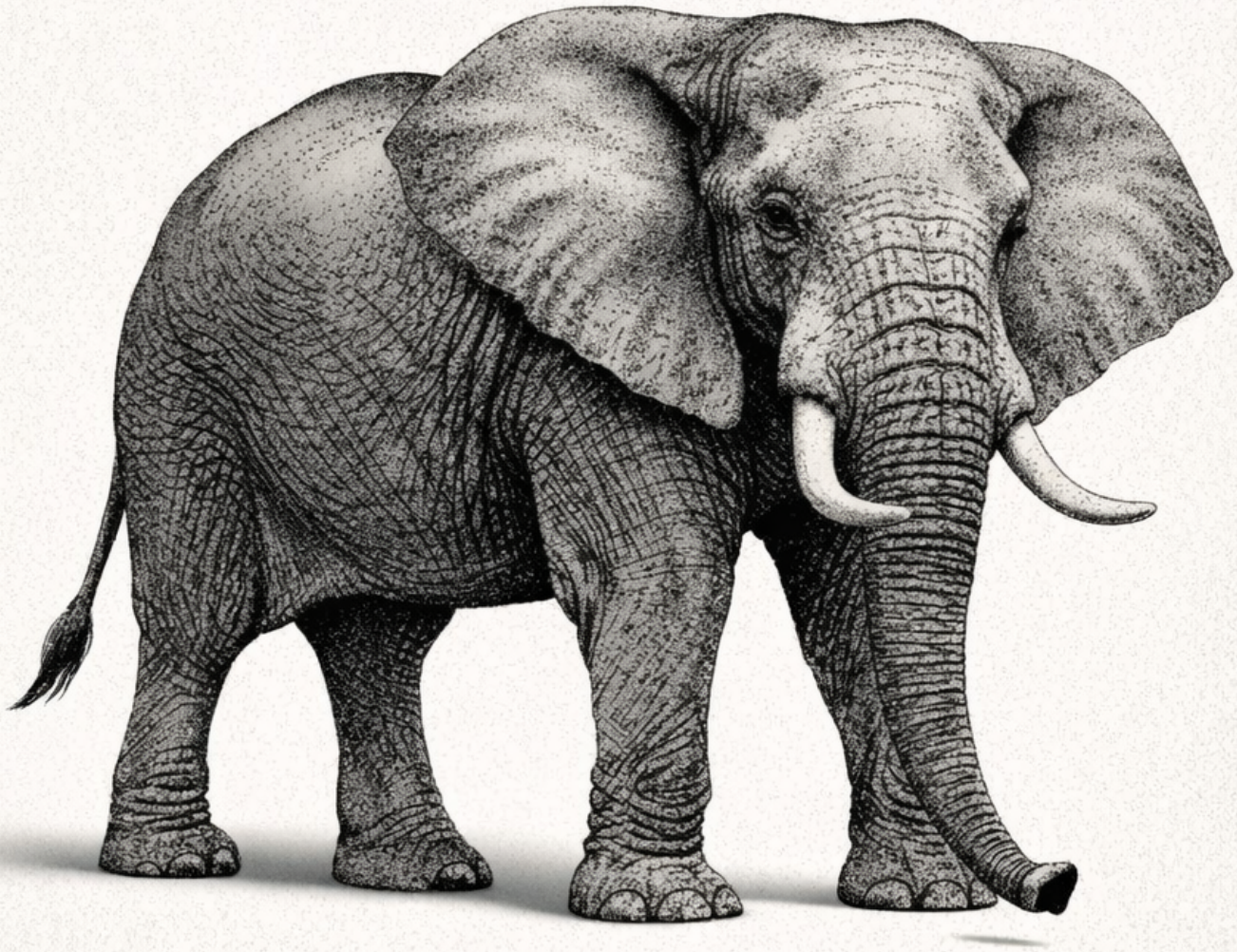
Laravel,
shipping fast



Julian Beaujardin

Table of Contents

Preface **10**

What This Book Is About 10

How This Book is Structured 10

Conventions Used In This Book 10

Who This Book is For 11

About The First Edition 11

This Book Is a Philosophy, Not a Pattern Library 11

What You'll Get From This Book 12

Core Principles 12

What Happens When You Don’t Follow These 13

How I Learned This 13

What “Shipping Fast” Actually Means 14

Discipline as Velocity, Not Restriction 15

The Myth of “Advanced” 15

What Makes an API Good 16

The Strategy 17

The Real Goal 18

Chapter 1: Getting Started **20**

Starting From Zero 20

The Structure That Makes Sense 21

Your First Endpoint 21

The Quick Way 22

The Scalable Way 23

Key Principles For Shipping Fast 23

Chapter 2: Core Architecture **25**

Controllers: Keep Them Simple 25

FormRequest Classes: Centralized Validation 27

Responsable Wrappers: Consistent JSON 30

BaseResponse: One Place That Knows the Envelope 30

ModelResponse: For Single Resources 31

ErrorResponse: For Validation and Error Failures 33

CollectionResponse: For Lists and Iterations 35

MessageResponse: For Async Operations and Confirmations 36

Data Transfer Objects (DTOs) 38

Three Layers: Model, DTO, Resource 39

Resources: Model-to-JSON Transformation 39

Mappers: The Glue 41

The Manager-Facade-Driver Pattern 44

Step 1: Define the Contract (Interface) 44

Step 2: Create Specific Implementations (Drivers) 45

Step 3: Create the Manager (Factory + Router) 49

Step 4: Create the Facade (Easy Access) 50

Step 5: Register in the Service Provider 50

Step 6: Use in Your Controller 52

Step 7: Switch Providers via Configuration 55

Putting It All Together 56

Chapter 3: Authentication **57**

Why Bearer Tokens 57

The Bearer Token Model 58

Token Caching: The Performance Multiplier 61

Dynamic Configuration via Token Settings 65

Middlewares 67

EnsureTokenIsValidMiddleware: Token Validation 68

Chapter 4: The Request Pipeline 73

Rate Limiting: throttle:api-token 74

AddRateLimitHeadersMiddleware: Rate Limit Headers 76

LogApiRequestsMiddleware: Request Auditing 77

Global API Middlewares (affect every API route) 81

SetRequestLocaleMiddleware: Language Detection 82

GzipResponseMiddleware: Response Compression 85

CORS: Cross-Origin Requests 87

HeaderFactory: Consistent Response Headers 89

Routes and Middleware Ordering 90

Domain Validation: Binding Tokens to Origins 90

The Complete Request Flow 92

Security Checklist 94

Threat Scenarios: Real-World Attacks 95

Error Response Formats 95

Testing Authentication & Security 96

Configuration: Environment-Specific Security 99

When NOT to Use These Patterns 101

Summary: Security Isn't Optional 101

Chapter 5: Error Handling & Logging 104

Errors are a contract, not an accident 104

One handler, every failure shaped the same 104

Error codes consumers can branch on 108

Context that makes a log worth keeping 110

Correlating one request across services 111

When the logger is the thing that broke 112

Summary 113

Chapter 6: Performance & Optimization **113**

Measure Before You Optimize 114

The N+1 You Will Actually Hit 115

Caching What Is Expensive, Not What Is Easy 117

Cache Invalidation You Can Reason About 118

Pagination As A Performance Decision 119

Database Indexes And The Queries That Need Them 119

Payload Size Is Latency Too 120

Knowing The Optimization Worked 121

Performance Checklist 122

Chapter 7: Queue & Background Processing **123**

What belongs off the request 123

Jobs that can run twice safely 125

Retries, backoff, and giving up 126

The failed jobs table is not a graveyard 128

Chaining long multi-step work 130

Watchdogs for work that silently stalls 132

Watching a queue you cannot see 133

Recap 134

Chapter 8: Testing & Quality Assurance **135**

What an API test should assert 135

Feature tests over unit tests, and why 137

Faking the services you do not own 138

Factories that describe intent 139

Testing failure, not just success 140

Static analysis as a second opinion 141

Keeping the suite fast enough to run 142

The test that would have caught it 143
Chapter 7 Checklist 144

Chapter 9: Monitoring & Observability 145
Health checks that mean something 145
The four signals worth a dashboard 147
Alerting on symptoms, not causes 148
Tracing a request you cannot reproduce 150
What to do with the first five minutes of an incident 152
Chapter 9 Summary 153

Chapter 10: Developer Experience 153
The First Hour on a New Codebase 154
Setup That Is One Command 155
Conventions A Newcomer Can Infer 156
Documentation That Survives The Code Changing 157
Errors That Tell You What To Do Next 157
Tooling That Agrees With Itself 158
Reviewing Code Without Relitigating Style 159
Recap 160

Chapter 11: API Evolution 160
Changes that break and changes that do not 160
Versioning at the edge, not through the code 161
Running two versions without two codebases 164
Deprecating with a date, not a hope 165
Telling consumers before they find out 168
Migration guides people can follow 169
Retiring a version 170

Chapter 12: Data Management **171**

Data has a cost and an owner 171

Retention as a policy, not a cron job 172

Migrations that run on a live table 174

Backfills you can restart 175

Deletion requests and what they touch 177

Archiving cold data 179

Proving what you keep and why 180

Chapter 11 Summary 181

Chapter 13: Advanced Features **181**

Add These When the Basics Are Boring 182

Webhooks: Delivery You Can Prove 182

Signing What You Send 184

Batch Operations Without Timeouts 185

Search That Stays Consistent With the Source 187

Long-Running Exports 189

Knowing When a Feature Is Not Worth It 192

Chapter 14: DevOps & Infrastructure **193**

A pipeline that refuses bad code 193

Deploying without dropping requests 194

Migrations during a deploy 196

Rolling back safely 198

Secrets that never reach the repository 199

Provisioning as something you can repeat 200

Chapter 14 Summary 201

Chapter 15: A Fleet, Not a Monolith **201**

What Changes at a Dozen Services 202
The Star Topology 202
One Owning Service Per Provider 205
Separating the Control Plane from the Output Plane 207
Why Read Paths and Write Paths Can Diverge 208
Chapter 14 Summary 210

Chapter 16: Work That Survives the Network 210
One request, a dozen services, none of them promise "now" 211
Idempotency has to survive the hop, not just the retry 211
Names a retry can't orphan 213
A sweep that doesn't trust anyone to notice 214
Retry and backoff live in the client, once 216
Recap 218

Chapter 17: The Shared Package 219
One API becomes a fleet 219
What a shared package should own 220
What it must not own 221
The package that must not become a service 221
The real cost of pinning 223
Credentials that ride with the token, not with the service 224
Chapter 16 Summary 227

Chapter 18: Shipping Templates to Live Sites 227
Three Tiers, Three Cadences 228
Install-Time Baking Beats Hand-Copied Files 229
A Version Bump Is Two Steps, Not One 232
The Silent-Staleness Trap Behind a Persistent Cache 233
A Live Editing Surface Needs Its Own Narrow, Expiring Credential 234

Closing the Loop 235

Appendices **236**

Appendix A: Quick Reference 236

Essential Artisan Commands 236

Common HTTP Status Codes 237

Useful Facades 238

Testing Helpers 238

Appendix B: Troubleshooting Guide 238

Common Issues & Solutions 239

Conclusion 239

Key Takeaways 240

Next Steps 240

Additional Resources 240

Official Documentation 240

Community Resources 240

Tools & Services 241

Preface

Laravel has been refined through years of production. It handles routing, validation, database access, authentication, queues, and testing with a coherent vocabulary.

The trap is fighting those conventions, wrapping Eloquent in repositories “just in case”, replacing FormRequests with custom validation, abstracting the framework out of the framework.

This book takes the opposite approach: lean into Laravel’s patterns because they’re proven. Controllers, FormRequests, DTOs, Resources, Facades, these exist because they prevent bugs, enable testing, and scale reliably.

Commit to Laravel’s conventions and you stop debating the foundation. You start shipping.

Boring scales. Clever decays.

What This Book Is About

This book shows a practical, repeatable way to build APIs with Laravel that are easy to extend, easy to test, and predictable for consumers. It uses a real production API as the running example and applies a consistent structure across validation, transformation, responses, and integration layers. The focus is not on clever abstractions, but on reliable patterns that scale with teams and timelines.

How This Book is Structured

The book moves from foundations to implementation. Early chapters establish the core architecture and response patterns. The middle chapters build out authentication, middleware, and integration boundaries. Later chapters cover reliability concerns such as logging, error handling, performance, and operations. Each chapter includes concrete code that mirrors the project in this repository, so you can map concepts to working files.

Conventions Used In This Book

Code samples follow the same conventions as the repository: strict typing, explicit return types, FormRequests for validation, Response Wrappers for JSON structure, and DTOs for typed data flow. The term “Response Wrapper” refers to `Responsable` response classes such as `ModelResponse`, `CollectionResponse`, `MessageResponse`, and `ErrorResponse`. Examples favor Laravel facades and Laravel 13 configuration patterns.

Who This Book is For

This book is written for Laravel developers who are comfortable with HTTP, REST, and Object Oriented Programming. You do not need to be a Laravel expert, but you should work with Laravel Framework regularly.

You can follow along with a laptop running PHP 8.4, a database, and an editor. The examples in this book come from real, working production APIs, so the focus stays practical.

If you have inherited inconsistent APIs or watched complexity pile up, this book offers a clear, repeatable path. It does not promise perfection, but it aims for sanity.

If you build with a team, this approach keeps endpoints readable, helps junior developers onboard quickly, and keeps the codebase from walking out the door.

If you want to ship features now, this book is for you. It focuses on patterns you can repeat from endpoint one to endpoint one hundred without rethinking the architecture.

If that sounds like you, this book was written for you.

About The First Edition

This edition is based on Laravel 13 and PHP 8.4, reflecting the modern application structure introduced in recent Laravel releases. Examples and conventions align with the Webplo License API codebase, including Response Wrappers, FormRequests, DTOs, and middleware-based architecture. The goal is a stable foundation that future editions can extend without changing the core approach.

This Book Is a Philosophy, Not a Pattern Library

You won’t find 40 architectural diagrams here. You won’t find a custom abstraction that “future-proofs” your codebase.

You will find:

- A simple, repeatable API structure
- Clear rules for controllers, facades, and validation
- A pragmatic approach to authentication, authorization, and versioning
- A bias toward deletion over invention

This is the approach I use when the goal isn’t to impress other developers, but to ship weekly, onboard quickly, and keep complexity under control.

If you’re looking to build “the perfect architecture”, you won’t find it here.

But if you want to ship APIs that survive pressure, deadlines, product pivots, junior developers, and legacy constraints, you’re in the right place.

Here's the philosophy: Don't wait for complexity to arrive before you introduce structure, and don't add complexity that hasn't arrived yet. Start with simple, consistent patterns from day one. Use DTOs to type your data, Resources to transform it, Response Wrappers to make responses predictable, Facades to abstract provider logic, and FormRequests for validation. These are lightweight, sensible defaults that keep structure stable as the business logic evolves.

What You'll Get From This Book

- **Cut architectural debates by 80%.** Decide once, apply everywhere. No more "should we use a repository here?" on every endpoint.
- **Onboard developers in days instead of weeks.** Read one endpoint, understand all of them. That's the goal.
- **Reduce bugs caused by inconsistency.** Consistent patterns shrink the surface area where bugs can hide.
- **Add endpoints without touching five layers.** You follow one pattern, write the business logic, and ship.

These aren't promises. These are outcomes we've seen repeatedly if you applied this philosophy:

Core Principles

This book follows four core principles. These aren't just opinions; they're lessons learned from production systems handling billions of requests, teams that ship daily, and APIs that survive chaos without heroics.

1. **Consistency beats cleverness.** Every endpoint follows the same pattern: validation, response structure, and error handling. It might feel boring at first, but it's actually freedom. When the structure is predictable, your team can focus on business problems consistently.
2. **Discipline is velocity.** Constraints aren't chains; they're accelerators. When you agree on structure up front, you eliminate decision fatigue and ship faster together.
3. **Simple and consistent today scales better than clever and custom today.** Don't start with microservices, CQRS, event sourcing, or heavy abstractions unless you actually need them. Begin with boring, predictable patterns. When real constraints show up, you'll know exactly where to add sophistication.
4. **Build to ship, not to impress.** The goal is value, not architecture trophies. Ship features, measure impact, iterate. An API that ships is miles ahead of a pristine API stuck in staging.

What Happens When You Don’t Follow These

I’ve watched this play out dozens of times.

A team builds an API with inconsistent patterns. Controller A validates with a custom class. Controller B validates inline. Controller C does a hybrid. Response formats vary. Errors are inconsistent. Field names drift.

Six months later, the codebase becomes archaeology. Code reviews drag because there’s no baseline. New features take twice as long because you’re fighting the code instead of extending it.

Or worse: a team wins a debate about “proper architecture” on day one. They implement CQRS. Or event sourcing. Or microservices. Zero users. Zero problems. But now every feature requires threading through five layers before business logic gets written. Velocity drops immediately. The architecture that was supposed to “future-proof” the system becomes the thing preventing you from shipping. Both failures share a disease: inconsistency and premature complexity.

How I Learned This

I spent three years at a company building an API that began beautifully. The first endpoint was perfect. We had a pattern. Everyone understood it. Then we hired two more developers.

One skipped resources and returned raw arrays. Another validated inline because FormRequests felt “too abstract.” Someone wanted DTOs everywhere for “type safety,” which led to different response shapes. Another wrapped Eloquent in repositories “just in case we switch databases.”

No one was wrong. Each choice made sense on its own.

Collectively, we built an API where you couldn’t read two endpoints and understand the third. Every new feature started with, “How did we do this last time?”, and the answer kept changing. One endpoint cached aggressively. Another didn’t cache at all. One returned proper status codes; another buried errors inside 200 responses.

By month six, we weren’t shipping; we were doing forensics. Adding an endpoint took three days instead of three hours. Code reviews became architecture debates. Junior developers slowed down because they had to learn 47 different patterns instead of one.

Pull requests that should’ve taken an hour sat for days. Not because the logic was wrong, but because every endpoint triggered a new argument. And no argument ever resolved, because we had no foundation, only contradictory precedent.

I remember a standup where the entire team was blocked. Not on features, on understanding the last developer’s code. Marisia couldn’t remember why validation was structured differently in the orders API. Chris spent two hours hunting for where error handling was supposed to happen. We were spending engineering time deciphering inconsistency instead of solving problems.

The irony was that we had no performance problems or scaling problems. The “flexible architecture” we thought we were building became the most inflexible thing possible. You couldn’t change anything because no one understood why it was done that way.

We eventually rewrote the API, not because it was broken, but because consistency had collapsed. The rewrite took six weeks because the rules weren’t clear.

The second time, we picked one pattern. FormRequests for validation. DTOs for typed data. Resources for responses. One error structure. One Response Wrapper. No exceptions. No “this endpoint is special.”

We debated once, committed, and moved forward.

The difference was night and day. Velocity didn’t just return, it compounded. When you’re not figuring out how to do something, you’re solving the actual business problem.

Since then, I’ve used this same pattern across multiple production APIs, and the result has always been the same: clarity first, velocity second, scale third.

Years later, the same checklist still applies: use built-in Laravel features, delete before adding, and keep the solution boring.

That’s why this book exists. Not because I found “the best way,” but because I lived through what happens when you don’t decide upfront, and saw what consistency unlocked.

What “Shipping Fast” Actually Means

“Shipping fast” doesn’t mean writing code recklessly. It means:

- **Decisions made once, applied everywhere.** One validation pattern. One response structure. One error format. No debates, no exceptions, no special cases.
- **New developers productive in days.** Read one endpoint. Understand all of them.
- **Real problems get solved, not imaginary ones.** Build for today. Optimize when you hit real constraints.
- **Code reviews focus on logic, not style.** Consistency removes structural arguments.
- **Deploys don’t terrify you.** Predictability makes debugging and rollback routine, not drama.

Shipping fast is shipping predictably, reliably, and repeatedly. It’s velocity that compounds.

Discipline as Velocity, Not Restriction

Constraints enable speed.

A developer's instinct is often "freedom." No rules. Build it however feels right. But without constraint, you don't get freedom, you get chaos. And chaos is slow.

Think of a highway: drivers don't go faster without rules; they go slower because every driver is unpredictable.

Here's what happens when a team agrees on patterns:

You spend 30 minutes deciding how to structure validation. Then you use that approach everywhere. You've made one decision and saved dozens of hours of collective time.

You spend an hour defining your Response Wrapper. Every response follows it. No surprises for clients. No debugging seven different response shapes.

You choose your error strategy once. Every error follows it. Consistent status codes, messages, and payloads.

This isn't restriction. It's freedom from the tyranny of choice.

The fastest teams I've worked with weren't the smartest. They were the most consistent. They decided once, applied everywhere, and moved forward.

That's what this book teaches: the patterns that let your team move at velocity.

The Myth of "Advanced"

Every developer goes through a phase:

You discover microservices. You discover CQRS. You discover event sourcing. You discover hexagonal architecture. And suddenly everything you've built feels "wrong."

So you rebuild. And rebuild again.

What used to be a simple `POST /orders` endpoint now requires six layers of abstraction, three transformers, two DTO libraries, and a design doc longer than your schema.

These patterns solve real problems, but not the problems you have yet.

Netflix uses microservices because Netflix *broke their monolith.* They didn't start there. They started simple, hit real limits, and then evolved.

This is accidental complexity: complexity you add because you anticipate problems that never came, or because you want to show off knowledge.

Clever code feels good. It impresses other developers. But cleverness compounds. Every abstraction adds another layer to understand, another layer to debug, another layer to maintain.

Multiply that by a team of five. Multiply that by two years.

You end up with code that’s harder to read, harder to change, and harder to test. The more abstract your code, the fewer people actually understand it. When the person who wrote it leaves, you’re stuck maintaining a system you can barely comprehend.

That isn’t architecture. It’s archaeology.

Here’s the uncomfortable truth:

Most APIs don’t fail because they weren’t architected like Netflix. They fail because they were never shipped, never simplified, or never maintained.

There’s a difference.

Solve the problem in front of you with the simplest code that works. When you hit a real limit, performance, scalability, maintainability, add the sophisticated pattern that solves that specific problem. By then, you understand why you need it. The pattern isn’t theoretical; it’s a solution to a real pain.

Complexity feels productive. Shipping is productive.

What Makes an API Good

Before we go any further, let's get aligned on what we're actually trying to build. The API design community, from RESTful architecture principles established by Roy Fielding to modern API design guides from Stripe, Twilio, and GitHub, has converged on a few core principles. These aren't arbitrary opinions. They're lessons learned from billions of API requests served in production, from APIs that scaled smoothly and APIs that collapsed under their own complexity.

A good API isn't just one that works. It's one that:

1. **Does one thing well**, Your API should have a clear purpose. A license management API shouldn't handle domain registration. A payment API shouldn't manage user profiles. This principle, known as the "single responsibility principle," applies to entire services. When an API tries to do everything, it does nothing well. Stripe, GitHub, and Twilio are laser-focused on their domain. They do one thing and do it brilliantly. Your API should be the same.

2. **Is predictable**, Responses should follow consistent patterns. Errors should be structured the same way. Field naming should be consistent. When you call `GET /licenses`, the response structure should match `POST /license`. Your API is a contract with your consumers, Stripe calls this "consistency." Break it and they lose trust. Keep it, and they'll build confidently on your API.

3. **Is secure by default**, Security shouldn't be an afterthought. CORS headers should be restrictive by default. Rate limiting protects you from abuse. Authentication should be required on protected routes. Build security into the foundation, not bolted on. The OWASP API Security Top 10 is clear: most vulnerabilities come from broken authentication, broken authorization, and excessive data exposure. These are architecture failures, not cryptography failures. Make security the default path.

4. **Handles failures gracefully**, Networks fail. Databases go down. Users send malformed data. Return meaningful HTTP status codes (not everything is 200 or 500). Provide error details that help consumers understand what went wrong. Stripe documents precisely what status codes mean and provides error codes for programmatic handling. You don't need idempotency keys on day one, but never return a cryptic 500 when you can provide real information.

5. **Is observable**, When something goes wrong in production (and it will), you should know immediately. Use structured logging. Use request IDs to trace operations through your system. Use metrics to monitor speed and failures. GitHub and Stripe log requests in searchable, traceable ways. If you can't see your system, you can't fix it. You should too.

6. **Can evolve**, Your API isn't done on day one. It grows and adapts. Design for evolution: version carefully, provide deprecation notices, think about how new fields won't break existing consumers. GitHub, Slack, and Stripe built APIs that evolved for years. Your API won't serve the same requests in three years. Design knowing that.

These principles aren't unique to this book. They're rooted in decades of API design experience across the industry. What this book does is show you how to build them into your Laravel application from the beginning.

That's what we're building toward in this book.

The Strategy

Here's how we're going to approach this:

First, we build with simple patterns from day one. Not complex patterns, not "future-proof" abstractions, just sensible defaults that we'll use consistently across every endpoint. FormRequests for validation. DTOs for typed data. Mappers for transformation. Resources for JSON output. Response Wrappers for consistent structures. Facades for clean interfaces. These aren't heavyweight frameworks. They're lightweight patterns that every endpoint will follow, from the first to the hundredth. This consistency means your team learns once, then applies that knowledge everywhere. New developers onboard faster. Code reviews become simpler because they're not trying to decipher dozens of different patterns.

Then, we add layers of defense. Authentication. Authorization. Error handling. Rate limiting. These aren't optional decorations, they're part of the foundation, automatically protecting you unless you explicitly opt-out. By building these into the architecture early, you're not adding complexity later, you're preventing the mess that comes from bolting security on afterwards.

Next, we test as we build. Not after. Not someday. As you write each endpoint, you write tests that verify it works. This isn't about coverage percentages. It's about confidence. When you have good tests, you can change code without fear. You ship faster because you know what you broke before your users do.

We optimize based on real constraints. Not prematurely. But once you have something working and you actually hit a real limit, performance, scalability, maintainability, then you add the sophisticated patterns that solve that specific problem. The beauty of starting with simple, consistent patterns is that you'll know exactly where to optimize and why. You've lived the pain. The pattern isn't theoretical anymore, it's a solution to something real.

Finally, we monitor and scale. You can't fix what you can't see. We'll set up logging, structured error tracking, and observability so you know what's happening in production. And because we built on consistent patterns, adding monitoring is straightforward, you add it once in the pattern layer and benefit everywhere.

The key insight: we're not avoiding architecture. We're avoiding *unnecessary* architecture. We're starting with the minimum viable structure that prevents chaos, then letting real requirements drive what comes next. Most APIs never need microservices, CQRS, or event sourcing. But every API needs consistency, validation, type safety, and testability. That's what we're building.

The Real Goal

Here's what this book is really about: **shipping fast and staying sane.** Not fast for a week, then slow because you're drowning in technical debt. Not fast today and regretful tomorrow when you inherit your own mess. Fast *now*, fast *next month*, fast *two years from now* when everything has changed and yet the code is still maintainable.

The real goal is an API where:

- Your team can add a new endpoint without a meeting about architecture
- A junior developer can read one endpoint and understand all of them
- Code reviews take minutes, not hours, because there's nothing to debate
- Deploys don't terrify you because you know what you're shipping
- When Sarah leaves, the codebase doesn't leave with her
- When the business pivots, you can pivot your API without rewriting everything

This happens when you embrace consistency from day one. When every endpoint uses the same patterns: FormRequests for validation, DTOs for typed data, Mappers for transformation, Resources for output, Response Wrappers for structure, Facades for clean interfaces. These patterns aren't heavyweight architecture. They're lightweight guardrails that prevent the chaos.

Here's what kills most APIs: not microservices, not complexity, not bad planning. It's inconsistency. One endpoint does it this way, another does it that way. Every developer interpreted the architecture differently. The codebase became a maze of special cases and workarounds. By the time someone tried to add a feature, they spent more time understanding the existing code than writing the new code.

The alternative is boring. It's the same pattern repeated 50 times, 100 times, without variation. It's DTOs everywhere. It's validation always in FormRequests. It's Response Wrappers used everywhere. To some, it looks repetitive. To experienced teams, it looks like freedom. Because when the structure is predictable, your brain can focus on the business logic. You're not deciphering the pattern. You're solving the problem.

This book shows you how to use boring patterns to ship fast, maintain easily, and scale without panic. Not because boring is trendy. But because boring is what allows you to focus on what actually matters: delivering value to your business.

Let's build something boring.

And let's ship it.

Chapter 1: Getting Started

Instead of building yet another "Hello World" API, let’s use a real production API as our guide: the **Webplo License API**. It’s simple enough to grasp quickly, but real enough to show the patterns that matter.

The API does one thing, and it does it well: it manages website licenses.

Every Webplo site has its own license, issued automatically the moment the site is provisioned. From then on, the Webplo engines interact with this API to create, retrieve, and revoke licenses as needed. This isn’t an API designed for public consumption; it’s an internal microservice, one of the many components that power Webplo behind the scenes.

Behind the scenes, this API integrates directly with the Statamic License API, since every Webplo site runs on top of Statamic CMS. The integration is structured using a Manager, Facade, Driver pattern, allowing us to keep the core logic clean while supporting multiple drivers and future extensions.

The API also handles multilingual responses, rate limiting, asynchronous logging, and payload compression, not because it’s flashy, but because it needs to be reliable.

This is a production API that's actually serving real traffic. We're going to understand how it works and what makes it work well.

Starting From Zero

If you want to follow along from scratch, let’s spin up a new Laravel project:

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

That’s it. Laravel did most of the heavy lifting. When I started developing in PHP 12 years ago, setting up a framework meant manually wiring configuration files, tweaking bootstrap scripts, and stitching everything together before you could even write real code. Laravel simply says, “We’ve already thought about this. Here are sensible defaults.”

Now, one thing I want you to understand right from the start: you're not required to use Laravel's defaults. You *can* customize everything. But here's what I've learned: when you find yourself fighting the framework, that's usually a signal that you're making decisions the framework is trying to save you from.

The Structure That Makes Sense

Laravel 13 has settled into a very clean structure. Here's what each folder actually does:

```
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
```

Each folder has a purpose. Controllers handle HTTP. Models handle the database. Facades provide clean interfaces. Tests verify everything works. When a new developer joins your team and looks at this structure, they think, "oh, I've seen this before." That consistency matters.

Your First Endpoint

Let's start with something simple: a health check endpoint. It sits there and tells you the API is alive and well. It might sound trivial, but this is where we establish the patterns you'll use for every endpoint that comes after. Keep it simple, keep it consistent, and you'll have a blueprint for everything else.

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

A health check should be pure signal: fast, lightweight, and zero-cost to call. You're just checking that critical services are connected, your database, cache layer, and queue system. No deep queries. No external API calls. No side effects. Just a clean 200 when everything's working, or a 503 when something's broken. Your monitoring systems ping this endpoint every 5-10 seconds and immediately know your API's status without adding load. That's the entire point, nothing more.

This is a **public endpoint**, no authentication, no middleware (for now). Anyone can call it, no questions asked. Your load balancers ping it. Your monitoring systems ping it. Your deployment pipeline pings it. Everyone's checking the same thing: is your API alive and its services responding?

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

The Quick Way

```
// 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',
                ],
            ]
        ]);
    }
}
```

First, think about what we're doing.

We're handling an HTTP request. We need to check that critical services are working. We need to return a response. That's three steps.

As your API grows, you'll want to ensure consistency, stronger typing, meaningful error codes that clearly describe what happened, localization, whether it's a 200 OK or a 503 Service Unavailable, and comprehensive testing to build confidence in your implementation.

That's when the quick way starts feeling fragile. You're copy-pasting response formatting across

endpoints. Validation logic is scattered. Your controllers grow to 100+ lines with no clear separation between HTTP concerns and business logic.

Testing becomes tedious because you're asserting against arrays with unknown shapes. The properties and types are unclear until you read the implementation. These aren't problems with the quick approach itself. It's not *wrong*. It's just not enough anymore. In other words, it's simply not *scalable*.

The Scalable Way

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

The difference is the use of Response Wrappers via `Responsable` responses, resources, DTOs, and mappers (or transformers, as they're sometimes called). There's no magic involved. Instead of returning raw responses directly from controllers, this approach introduces a structured layer: responses implement a common contract, data is wrapped in consistent response objects, output is transformed through resources, and raw values are mapped into typed DTOs. The result is a predictable, testable, and type-safe API where formatting, structure, transformation, and localization are clearly separated from controller logic.

These aren't academic exercises. They're practical solutions refined through real production experience. Your controllers stay thin and focused. Your API consumers write once and reuse forever. A new developer looks at one endpoint and understands all of them. That's what we'll discuss throughout this book because that's what separates a quick prototype from a real production API.

Key Principles For Shipping Fast

Here's what makes this API architecture work: **consistency**. Not just in code style, but in structure, patterns, and approach. Every endpoint follows the same patterns. Every response uses the same wrappers. Every validation happens the same way. This consistency is what lets your team ship fast and stay sane.

Here are the principles that enable this consistency:

Convention over Configuration. Laravel has conventions for a reason, they're distilled wisdom from thousands of production applications solving real problems. Default to Laravel's way. Only step outside it when you have a genuine, specific reason. This saves you from endless decision-making and keeps your API predictable. The moment your team agrees to follow the framework instead of fighting it, velocity changes. You're not making micro-decisions on every endpoint. You're following a proven path.

Consistency Over Novelty. Every endpoint should look like the one before it. FormRequests handle validation everywhere. DTOs transform data everywhere. Resources format output everywhere. When new developers join, they learn one way and apply it everywhere. When you need to change something, you change it in one place and benefit everywhere. This is not dull repetition, this is architectural leverage. You're not smart-coding each endpoint differently. You're using the same proven pattern, over and over, because it works.

Type Safety. PHP 8.4 gives you the tools to catch bugs before your users do. Use full type hints, return types, and static analysis tools like PHPStan at level 9. This is not optional. Strong typing means your IDE autocompletes what exists. You know what properties are available on your DTOs. Your Resources transform data correctly. Your Controllers receive what they expect. When your API is strongly typed throughout, inconsistencies stand out immediately.

Fail Fast. Validate input early. Check that connections work. Know immediately when something is wrong. Don't let bad data propagate through your system. When bad data arrives at your API, catch it at the boundary. FormRequests validate before your controller runs. Type hints catch shape mismatches. If the data is bad, fail loudly and tell the consumer exactly what's wrong. This prevents cascading failures and makes debugging straightforward. Your entire team knows: bad data never makes it past the gate.

Separation of Concerns. Controllers handle HTTP. Facades provide clean interfaces. Drivers implement the actual logic. Resources transform output. When you keep these separate, each layer becomes simpler and easier to understand. A new developer can read a controller and immediately understand the flow: request → validation → facade → response. They don't have to untangle HTTP logic mixed with business logic mixed with transformation logic. Separation enables consistency.

Test as You Build. Not after. Not someday. As you build. This isn't about hitting coverage percentages. It's about confidence. When you test your endpoints as you build them, you catch problems immediately. You know your patterns work before you commit. You can refactor with confidence knowing tests have your back. Testing is part of the development process, not something you do when you have spare time. Good tests catch when inconsistency creeps in.

Security by Default. Don't build security features separately. They should be part of the foundation, automatically protecting you unless you explicitly opt-out. Rate limiting should be normal. Authentication checks should be automatic on protected routes. Error messages should never leak sensitive information. When security is baked into your patterns, you don't have to remember it, it just happens. Consistency in security practices means no endpoint is accidentally exposed.

These principles work together to create an API that's predictable, maintainable, and fast to build. Boring consistency is not a limitation. It's a superpower.