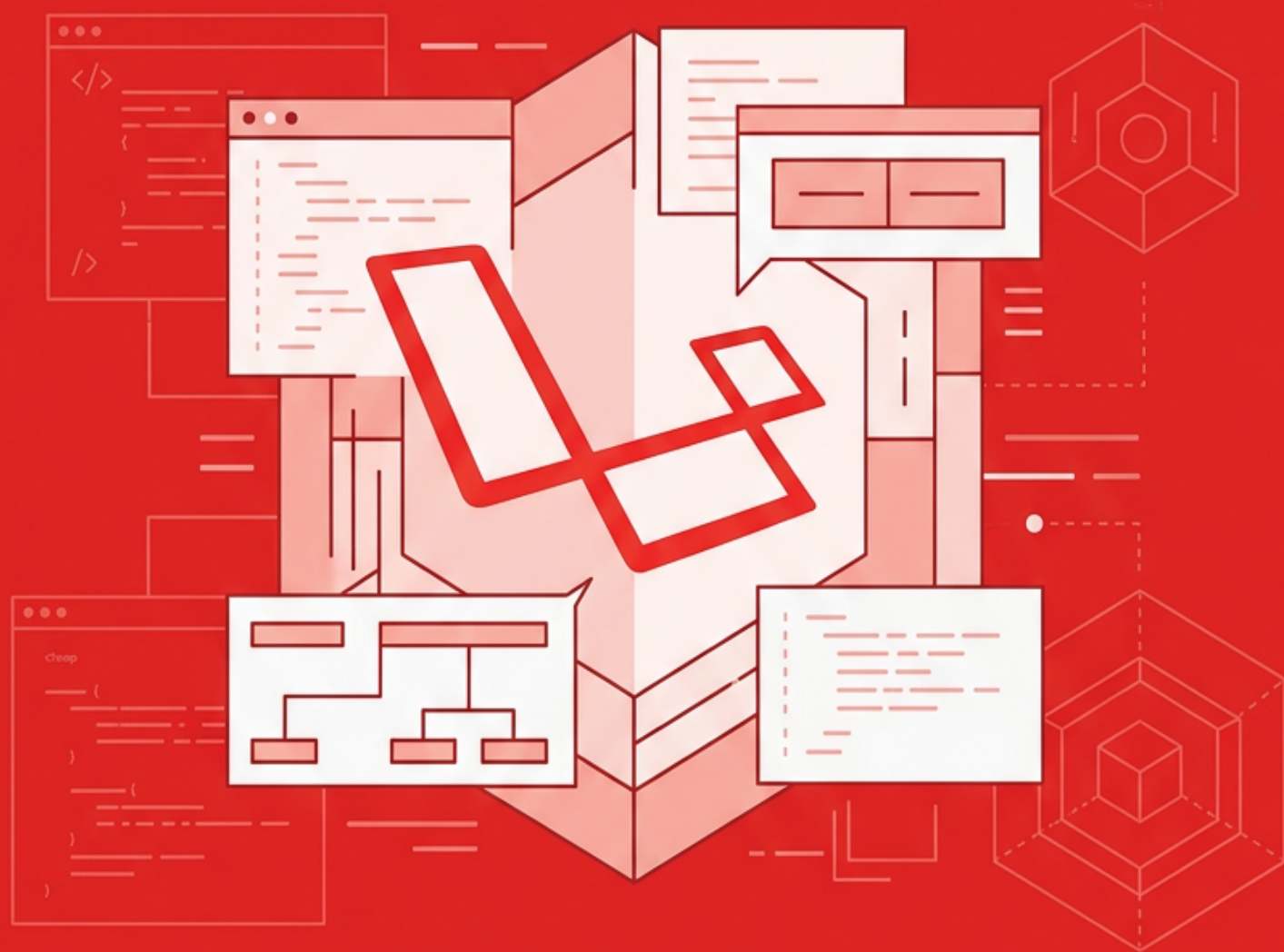


THINKING IN DOMAINS IN LARAVEL

A Pragmatic Guide for Laravel Developers



Ahmad Mayahi

Thinking in Domains in Laravel

Organizing Your Code Around Business Logic

Ahmad Mayahi

Version 1.0

© 2026 Ahmad Mayahi. All rights reserved.

No part of this book may be reproduced or transmitted in any form
without written permission from the author.

www.mayahi.net

Table of Contents

[Preface](#)

[About the Author](#)

[01 Thinking in Domains](#)

[02 Modeling Data with DTOs](#)

[03 The Action Pattern](#)

[04 Eloquent Without the Bloat](#)

[05 States, Transitions, and Enums](#)

[06 Testing the Domain](#)

[07 The Application Layer](#)

Preface

This is not a Domain-Driven Design book. It does not ask you to implement [bounded contexts](#), [aggregate roots](#), or [anti-corruption layers](#). Those concepts come from enterprise Java and complex distributed systems — they have their place, but that place is rarely a Laravel application.

This book is about something simpler and more immediately useful: organizing your code around business logic.

I have been building Laravel applications for years. At some point, every non-trivial project hits the same wall: controllers get fat, models do too much, and business logic ends up scattered across places where it does not belong. The default CRUD structure that worked beautifully for the first twenty models starts to fight you at fifty.

The answer is not to adopt a heavyweight architecture. The answer is to think about your code in terms of the business problems it solves — what I call "thinking in domains." Group related logic together. Give business operations their own classes. Keep your models focused on data. Let the framework handle the framework stuff.

This book borrows a few ideas from DDD that genuinely help in Laravel projects — domains, actions, data objects, states — and leaves the rest behind. No 200-page theory chapters. No UML diagrams. No pretending your Laravel app is an enterprise Java system. Just practical patterns you can apply to your next pull request.

Who this book is for

You should already know how to build Laravel applications. You understand [routing](#), [controllers](#), [Eloquent](#), and the basics of the framework. If you are still learning Laravel itself, start with [Clean Code in Laravel](#) first.

This book is for developers who have shipped a few projects and started to feel the pain of growing codebases. You know something needs to change, but you are not sure what.

What to expect

Every chapter is focused on a single concept and uses real code examples in Laravel 12 with PHP 8.4. We build from the ground up — starting with how to think about domains, then working through data modeling, actions, lean models, states, and testing. The final chapter ties everything together with the application layer.

There is no filler. If a concept does not help you write better Laravel code, it is not in this book.

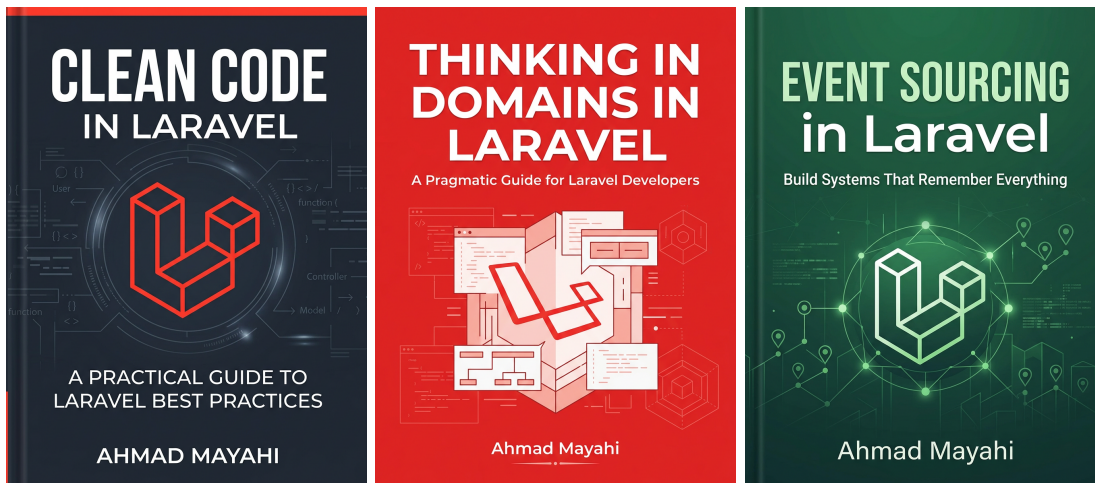
About the Author

I am Ahmad Mayahi, a software engineer who has spent the better part of his career building web applications with PHP and Laravel.

I do not come from a computer science background. I learned to code by building things, breaking them, and reading other people's code until the patterns clicked. That experience shaped how I write and teach — I have no patience for unnecessary theory, and I assume you do not either.

Over the years, I have worked on projects ranging from small MVPs to large-scale applications serving thousands of users. The lessons in this book come from that experience — from the mistakes I made, the refactors I wish I had done earlier, and the patterns that actually held up over time.

You can find me at mayahi.net.



Thinking in Domains

If you have been building Laravel applications for a while, you have probably noticed that the default CRUD structure starts to fall apart once the project gets large enough. [Controllers](#) become fat, [models](#) handle too much, and business logic ends up scattered across random places.

This is a problem that many teams face once their project grows beyond a handful of models. The typical solution people reach for is [Domain Driven Design \(DDD\)](#), but full-blown DDD is often overkill for web applications. What works better is borrowing the best ideas from DDD, [Hexagonal Architecture](#), and [Event Sourcing](#), then applying them pragmatically within Laravel.

The core idea is deceptively simple: group code by what it represents in the real world, not by its technical role.

The problem with default Laravel structure

Laravel's default structure groups code by technical properties. Controllers live together, models live together, requests live together. This is fine when your project has 10 models and 15 controllers.

But think about this: has a product owner ever asked you to "refactor the models folder"? No. They ask you to work on course publishing, student enrollment, or the catalog.

When a project has 100 models, 300 actions, and 500 routes, the default structure means that any single business concept — like enrollment — is spread across dozens of directories. You need to look in the controllers directory, the models directory, the

requests directory, the jobs directory, and more. The cognitive load becomes enormous.

What is a domain?

A domain is a group of related business concepts. You could also call them "modules" or "services". In an online education platform, your domains might be:

- Students
- Enrollments
- Courses
- Catalog

Each domain groups everything related to that business concept together:

```
src/Domain/Courses/  
├─ Actions  
├─ QueryBuilders  
├─ Collections  
├─ Data  
├─ Events  
├─ Exceptions  
├─ Listeners  
├─ Models  
├─ Rules  
└─ States
```

```
src/Domain/Students/  
├─ Actions  
├─ Models  
├─ Events  
└─ Rules
```

Notice that each domain can have a different internal structure. The Courses domain might need states and query builders, while the Students domain might be simpler. There is no requirement for uniformity.

Domains vs applications

There is a key distinction between domain code and application code.

Domain code is your business logic. It contains models, [DTOs](#), [actions](#), validation rules, events, and [states](#). It represents what your application does.

Application code is how that logic gets exposed to users. It contains [controllers](#), middleware, [form requests](#), [resources](#), and [view models](#). It represents how users interact with your business logic.

A single project can have multiple applications consuming the same domain:

```
src/App/Dashboard/    # Instructor/admin dashboard (Blade/Inertia)
├─ Controllers
├─ Middlewares
├─ Requests
├─ Resources
└─ ViewModels

src/App/Api/          # REST API
├─ Controllers
├─ Middlewares
├─ Requests
└─ Resources

src/App/Console/     # Artisan commands
└─ Commands
```

All three applications can use the same `Domain\Courses` code. The dashboard might render a course editor form, the API might return course JSON for a mobile app, and a console command might publish scheduled courses. The business logic is identical;

only the delivery mechanism differs.

This separation is powerful. When you need to change how course enrollment works, you change the domain code once. All applications automatically get the update.

Look at the folder paths above — there is no single `\App` root namespace like a default Laravel project. That is intentional. When your project has multiple application layers (a dashboard, an API, a CLI), cramming them all under `\App` stops making sense. So we give each layer its own root: `\App` for application code, `\Domain` for business logic, and `\Support` for shared utilities.

Setting up the folder structure

To make this work with Laravel, you update `composer.json` to add two new root namespaces:

```
{
  "autoload": {
    "psr-4": {
      "App\\": "src/App/",
      "Domain\\": "src/Domain/",
      "Support\\": "src/Support/"
    }
  }
}
```

The `Support` namespace is a home for generic helpers that do not belong to any specific domain. Think of it as code that could just as well be a standalone package: a `MarkdownRenderer`, a `SlugGenerator`, a `FileUploader`.

Updating `composer.json` handles class autoloading, but Laravel itself still expects your application code to live in the default `app/` directory. Internally, Laravel uses the `app` path to discover commands, resolve providers, and locate other framework-convention files. If you move your application layer to `src/App/`, you need to tell Laravel about it.

Open your `bootstrap/app.php` file and call `useAppPath` on the application instance before returning it:

```
$app = Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
    )
    ->withMiddleware(function (Middleware $middleware): void {
        //
    })
    ->withExceptions(function (Exceptions $exceptions): void {
        //
    })->create();

$app->useAppPath($app->basePath('src/App'));

return $app;
```

The `useAppPath` method redirects every internal call to `app_path()` to your new directory. Without this, Artisan's `make:` commands would still generate files inside `app/`, and any framework feature that relies on the app path would look in the wrong place.

If you want to keep Laravel's default `app/` directory instead of `src/App/`, you can skip this step entirely. The important thing is the mindset of separating domain from application, not the exact folder names.

Domains evolve

One of the most important things to understand is that domain boundaries are not permanent. You might start with a `Courses` domain and realize six months later that it has grown too large. Maybe course authoring and course delivery are complex enough to be their own separate domains.

That is fine. Refactoring domain boundaries is cheap when each domain has minimal dependencies on others. You are not locked into your initial structure.

Do not overthink the perfect domain structure before writing code. Start with what makes sense now, and refine it as your understanding of the business grows. The architecture supports change — that is the whole point.

When to use this architecture

This approach is not for every project. A simple CRUD app with 5 models does not need separate domain and application layers. The overhead would outweigh the benefits.

But when your project has:

- Dozens of models with complex relationships
- Multiple developers working simultaneously
- Years of expected maintenance ahead
- Complex business rules that go beyond simple data persistence

Then grouping by business concept instead of technical role will save you significant time and headaches. The rest of this book will show you exactly how.

If your project is not yet large enough for domain-oriented architecture, [Clean Code in Laravel](#) covers how to write clean, maintainable code within Laravel's default structure. Many of its principles — naming, dependency injection, thin controllers — apply regardless of how you organize your folders.

Summary

- Laravel's default structure groups code by technical role (controllers, models, requests). This works for small projects but breaks down as the codebase grows, because a single business concept ends up scattered across dozens of directories.
- A **domain** is a group of related business concepts. Group code by what it represents in the real world — courses, students, enrollments — not by its technical type.
- **Domain code** contains your business logic (models, DTOs, actions, events, states). **Application code** is how that logic gets exposed to users (controllers, middleware, requests, resources). Multiple applications can consume the same domain.
- Set up the structure by adding `App`, `Domain`, and `Support` namespaces to `composer.json`, and call `useAppPath()` in `bootstrap/app.php` to redirect Laravel's internal app path.
- Domain boundaries are not permanent. Start with what makes sense now and refine as your understanding of the business grows. Refactoring is cheap when domains have minimal dependencies on each other.
- This architecture is best suited for projects with dozens of models, multiple developers, years of expected maintenance, and complex business rules beyond simple CRUD.

References

- [Domain-Driven Design](#) — Martin Fowler
- [Domain-Driven Design: Tackling Complexity in the Heart of Software](#) — Eric Evans
- [Hexagonal Architecture](#) — Alistair Cockburn

- [Organizing Laravel](#) — Brent Roose
- [Domain-Driven Design in Laravel](#) — Laravel News
- [Modular Laravel](#) — Laravel Magazine
- [What is Domain-Driven Design?](#) — IBM

Modeling Data with DTOs

Data lives at the core of almost every project. You do not start by building [controllers](#) and [jobs](#) — you start by figuring out what data the application will handle. ERDs, diagrams, database schemas: these come first because everything else depends on them.

This chapter is about making data a first-class citizen of your codebase. We are not talking about Eloquent models yet. We are talking about plain data: the input that flows into your application and the structures that carry it through your business logic.

The problem with arrays

Have you ever worked with an "array of stuff" that was actually more than just a list? You used array keys as fields, and you felt the pain of not knowing exactly what was in that array.

Here is a common Laravel pattern:

```
function store(StudentRequest $request, Student $student)
{
    $validated = $request->validated();

    $student->full_name = $validated['full_name'];
    $student->email = $validated['email'];
    // ...
}
```

What exactly is in `$validated`? You do not know without:

- Reading the source code of `StudentRequest`

- Reading the documentation
- Dumping `$validated` to inspect it
- Using a debugger

Now imagine a colleague wrote this code five months ago. You will not know what data you are working with without doing one of those cumbersome things.

Arrays are versatile, but as soon as they represent something other than "a list of things", there are better ways.

PHP's type system

To understand why Data Transfer Objects matter, you need to understand PHP's type system.

PHP is weakly typed. A variable can change its type after being defined:

```
$price = '19.99'; // string

function applyDiscount(float $price): float
{
    // PHP automatically casts '19.99' to float
    return $price * 0.9;
}

applyDiscount($price); // Works, even though $price is a string
```

This makes sense for a language that mainly works with HTTP requests, where everything starts as a string. But it means the type system cannot give you strong guarantees about your data.

PHP is also dynamically typed. Type checks happen at runtime, not before the code executes. If there is a type error, you only discover it when that code path runs.

Tools like PHPStan, Psalm, and your IDE's built-in analysis help bridge this gap by performing static analysis on your code. They can catch many type errors without ever running the program. But they can only help if you give them something to work with — and that means using typed structures instead of generic arrays.

Data Transfer Objects

The solution is to wrap unstructured data into typed classes called Data Transfer Objects (DTOs):

```
class StudentData
{
    public function __construct(
        public readonly string $full_name,
        public readonly string $email,
        public readonly Carbon $enrolled_at,
    ) {}
}
```

Now your IDE always knows what data you are dealing with:

```
function store(StudentRequest $request, Student $student)
{
    $data = new StudentData(...$request->validated());

    $student->full_name = $data->full_name;
    $student->email = $data->email;
    $student->enrolled_at = $data->enrolled_at;
}
```

Type `$data->` and your IDE shows you exactly three properties with their types. No guessing, no debugging, no reading source code.

The benefits compound over time:

- **Autocompletion** works everywhere the DTO is used
- **Refactoring** is safe — rename a property and your IDE updates all usages
- **Static analysis** catches type mismatches before runtime
- **New team members** can understand the data flow immediately

DTO factories

There are two ways to construct DTOs from external data like HTTP requests.

Approach 1: Dedicated factory class. This is the theoretically correct approach because it keeps application-specific logic (the request) out of the domain (the DTO):

```
class StudentDataFactory
{
    public function fromRequest(StudentRequest $request): StudentData
    {
        return new StudentData(
            full_name: $request->get('full_name'),
            email: $request->get('email'),
            enrolled_at: Carbon::make($request->get('enrolled_at')),
        );
    }
}
```

This factory lives in the [application layer](#) because it knows about `StudentRequest`.

Approach 2: Static constructor on the DTO itself. This is more pragmatic but mixes application logic into the domain:

```

class StudentData
{
    public function __construct(
        public readonly string $full_name,
        public readonly string $email,
        public readonly Carbon $enrolled_at,
    ) {}

    public static function fromRequest(StudentRequest $request): self
    {
        return new self(
            full_name: $request->get('full_name'),
            email: $request->get('email'),
            enrolled_at: Carbon::make($request->get('enrolled_at')),
        );
    }
}

```

The second approach is less pure but more convenient. DTOs are the entry point for data into your codebase. Since you need to do the mapping somewhere, doing it on the class itself keeps the "how to create this" knowledge co-located with the class definition.

With PHP 8's named arguments, both approaches are clean and readable. Choose whichever fits your team best.

When DTOs shine

DTOs become especially valuable when data passes through multiple layers. Consider a course publishing flow:

1. A controller receives a request
2. The request data is mapped to a `CourseData` DTO
3. The DTO is passed to `PublishCourseAction` [action](#)
4. The action uses the DTO to create the course and its lessons

At every step, every developer knows exactly what data is available. The DTO acts as a contract between layers.

```
class CourseData
{
    public function __construct(
        public readonly string $instructor_id,
        public readonly Carbon $starts_at,
        /** @var LessonData[] */
        public readonly array $lessons,
    ) {}
}

class LessonData
{
    public function __construct(
        public readonly string $title,
        public readonly int $video_count,
        public readonly int $duration_minutes,
        public readonly bool $has_quiz,
    ) {}
}
```

Without DTOs, you would be passing arrays around. Every function that receives the array would need to hope that the right keys exist with the right types. With DTOs, the type system enforces this for you.

The cost of DTOs

DTOs do have overhead: you need to create the class and map data into it. For a simple form with three fields, this might feel excessive.

But the cost is front-loaded. You spend a few minutes creating the DTO once. You save time every single day that a developer works with that data without needing to debug what is in it.

In projects that last years with multiple developers, the investment pays for itself many times over.

Scaling up with spatie/laravel-data

The plain DTOs shown in this chapter are all you need to get started. But as your project grows, `spatie/laravel-data` can remove much of the boilerplate. It handles transformation, serialization, nested casting, and automatic construction from requests — while keeping the same typed, readonly philosophy.

For a deeper comparison of plain DTOs versus `spatie/laravel-data`, including the `toDto()` bridge pattern between [form requests](#) and DTOs, see the [Data Transfer Objects](#) chapter in *Clean Code in Laravel*.

The goal is to reduce cognitive load. You do not want developers starting their debugger every time they need to know what is in a variable. The information should be right there, at their fingertips.

Summary

- Arrays are versatile for lists, but as soon as they represent structured data (like a validated form), they lose type information. You cannot autocomplete, refactor safely, or know what keys exist without reading source code.
- Data Transfer Objects (DTOs) wrap unstructured data into typed classes with readonly properties. Your IDE, static analysis tools, and future developers all benefit from explicit types.
- DTOs can be constructed via a dedicated factory class (purer separation of domain and application) or a static constructor on the DTO itself (more pragmatic and co-located). Choose whichever fits your team.

- DTOs shine when data passes through multiple layers — controller to action to model. At every step, every developer knows exactly what data is available.
- The cost of DTOs is front-loaded: a few minutes to create the class. The payoff compounds over time as developers work with typed, autocomplete-friendly data instead of guessing array keys.

References

- [Data Transfer Objects in PHP](#) — Brent Roose
- [PHP 8: Named Arguments](#) — Brent Roose
- [PHP 8.1: Readonly Properties](#) — Brent Roose
- [Constructor Promotion in PHP 8](#) — PHP Manual
- [spatie/laravel-data](#) — Spatie Documentation
- [Data Transfer Objects](#) — Clean Code in Laravel

The Action Pattern

Now that we can work with data in a type-safe and transparent way, we need to start doing something with it. Just like we do not want random arrays full of data, we also do not want the most critical part of our project — the business functionality — to be spread throughout random classes.

Actions solve this problem. They are the single most impactful pattern you can adopt in a domain-oriented Laravel project.

What is an action?

An action is a class that represents a single user story or business operation. It takes input, does something, and gives output.

Think about what happens when an instructor publishes a course:

- Validate all lessons have content and required metadata
- Calculate the total duration from all lessons
- Generate a unique course code
- Save the course to the database
- Create enrollment slots for waitlisted students
- Send notification emails to subscribers

In a typical Laravel application, this logic lives in a fat [model](#), a bloated [controller](#), or is scattered across [jobs](#) and event listeners. With actions, you encapsulate the entire operation:

```
class PublishCourseAction
{
    public function __construct(
        private CreateLessonAction $createLessonAction,
        private GenerateCertificateAction $generateCertificateAction,
    ) {}

    public function execute(CourseData $courseData): Course
    {
        // Create course, process lessons, generate certificate template...
    }
}
```

Actions live in the domain. They are simple classes with no abstractions or interfaces. They take input, do something, and produce output.

Naming conventions

As a convention, suffix all action classes with `Action`. While `PublishCourse` sounds clean, it could also be a controller, a command, or a job. `PublishCourseAction` eliminates any confusion.

Yes, class names get longer. In large projects, you might end up with names like `RecalculateStudentProgressForCompletedModuleAction`. That is not elegant, but it is unambiguous. Your IDE's autocompletion handles the typing for you. For a comprehensive guide to naming all parts of a Laravel application, see the [Naming Conventions](#) chapter in *Clean Code in Laravel*.

For the public method, use `execute`. Why not `__invoke` or `handle`?

Not `__invoke` because PHP does not let you directly invoke an invokable property on a class. When composing actions, you would need ugly syntax:

```
// This does not work – PHP looks for a class method, not the invocable
$this->createLessonAction($lessonData);

// You need parentheses around it
($this->createLessonAction)($lessonData);
```

Not `handle` because Laravel uses `handle` in jobs and commands with automatic method injection from the container. Actions should only use constructor injection, not method injection. Using `handle` could create confusion about when dependencies are resolved.

`execute` is clear, unambiguous, and available.

Why use actions?

Re-usability

The key is splitting actions into pieces that are small enough to be reusable, while large enough to avoid class explosion.

Generating a certificate happens in multiple contexts: when a course is published and when a student completes the course. Two controllers, same business logic:

```
// In PublishCourseController
$action = app(PublishCourseAction::class);
$course = $action->execute($courseData);

// In GenerateCertificateController
$action = app(GenerateCertificateAction::class);
$certificate = $action->execute($enrollment);
```

A good rule of thumb: abstract when the functionality is the same, not when the code looks similar. Two actions might have similar code but serve completely different business purposes. Do not abstract those prematurely.

Reduced cognitive load

When you need to change how courses are published, you go to one place:

`PublishCourseAction`. You do not hunt through controllers, models, jobs, and listeners trying to piece together what happens.

Actions may work together with asynchronous jobs and event listeners, but those handle infrastructure concerns (queuing, event dispatching), not business logic. The action contains the actual rules.

Testability

Actions are trivially easy to test. The pattern is always the same:

1. **Setup**: create `DTOs` and resolve the action from the container
2. **Execute**: call the action
3. **Assert**: check the output

```
it('publishes a course with correct total duration', function (): void {
    // Setup
    $courseData = CourseDataFactory::factory()
        ->addLesson(title: 'Getting Started', videoCount: 3, duration: 15)
        ->addLesson(title: 'Advanced Topics', videoCount: 5, duration: 45)
        ->create();

    $action = app(PublishCourseAction::class);

    // Execute
    $course = $action->execute($courseData);

    // Assert
    $this->assertDatabaseHas($course->getTable(), ['id' => $course->id]);
    expect($course->course_code)->not->toBeNull()
        ->and($course->total_duration)->toBe(60)
        ->and($course->lessons)->toHaveCount(2);
});
```

No fake HTTP requests, no facade mocking. Just input, execution, and assertions.

Composing actions

Actions use constructor injection for dependencies and the `execute` method for context-specific data. If you are unfamiliar with how Laravel's service container resolves dependencies automatically, the [Dependency Injection](#) chapter in *Clean Code in Laravel* covers the fundamentals. This lets you compose actions from other actions:

```
class CreateLessonAction
{
    public function __construct(private DurationCalculator $durationCalculator) {}

    public function execute(LessonData $data): Lesson
    {
        $baseMinutes = $data->duration_minutes;

        if ($data->has_quiz) {
            $totalMinutes = $this->durationCalculator
                ->withQuiz($baseMinutes, quizDuration: 10);
        } else {
            $totalMinutes = $this->durationCalculator
                ->withoutQuiz($baseMinutes);
        }

        return new Lesson([
            'title' => $data->title,
            'video_count' => $data->video_count,
            'duration_minutes' => $totalMinutes,
            'has_quiz' => $data->has_quiz,
        ]);
    }
}
```

`CreateLessonAction` is injected into `PublishCourseAction`, which also injects `GenerateCertificateAction` and `NotifyStudentsAction`. Each action stays small and focused, yet together they handle complex business operations.

Be careful with deep dependency chains — they make code complex and tightly coupled. But two or three levels of composition are perfectly manageable and keep individual actions clean.

Mocking composed actions

Composition also makes testing flexible. If `PublishCourseAction` uses `GenerateCertificateAction` internally and you do not want certificates generated during a test, just mock it:

```
namespace Tests\Mocks\Actions;

class MockGenerateCertificateAction extends GenerateCertificateAction
{
    public static function setUp(): void
    {
        app()->singleton(
            GenerateCertificateAction::class,
            fn () => new self()
        );
    }

    public function execute(ToCertificate $toCertificate): void
    {
        return;
    }
}
```

Call `MockGenerateCertificateAction::setUp()` in your test's `setUp` method, and certificate generation is silently skipped. You test course publishing without the overhead of actual certificate rendering. For a broader introduction to test doubles — dummies, stubs, fakes, and mocks — see the [Introduction to Test Doubles in PHP](#) series.

Alternatives to actions

Two patterns from DDD are worth mentioning:

Commands and handlers separate what needs to happen (the command) from how it happens (the handler). This gives more flexibility — you can swap handlers, add middleware to the command bus, etc. But it also means more boilerplate. For most Laravel projects, actions provide enough flexibility without the extra code.

Event-driven systems decouple the trigger from the behavior entirely. When a course is published, an event is dispatched, and listeners react. This is extremely flexible but adds indirection that makes code harder to follow. The benefit rarely outweighs the cost for projects that are not massive distributed systems.

Actions sit in the pragmatic middle ground: enough structure to keep code organized, enough simplicity to keep it understandable.

*Together with [DTOs](#) and [models](#), actions form the true core of your project. DTOs carry the data, actions encapsulate what you do with it, and models define how it is persisted. Everything else is infrastructure. For a complementary look at actions within Laravel's default structure — including how controllers delegate to actions and how to keep actions focused — see the [Actions](#) chapter in *Clean Code in Laravel*.*

Summary

- An action is a class that represents a single business operation. It takes input, does something, and gives output. All business logic lives here instead of being scattered across controllers, models, and jobs.

- Name actions with an `Action` suffix (`PublishCourseAction`) and use `execute` as the public method — not `__invoke` (broken property invocation) or `handle` (confuses with Laravel's method injection in jobs).
- Actions use constructor injection for dependencies and the `execute` method for context-specific data. This enables clean composition: actions can inject and call other actions.
- Actions are reusable across controllers, commands, and jobs. They are testable with a simple setup-execute-assert pattern. And they reduce cognitive load — when you need to change how something works, there is one place to look.
- For composed actions, mock inner actions in tests to isolate the unit under test. A mock class that extends the real action and overrides `execute` keeps things simple.
- Commands and handlers or event-driven systems are alternatives, but actions sit in the pragmatic middle ground: enough structure to stay organized, enough simplicity to stay understandable.

References

- [Refactoring to Actions](#) — Freek Van der Herten
- [spatie/laravel-queueable-action](#) — Spatie, GitHub
- [Introduction to Test Doubles in PHP](#) — Ahmad Mayahi
- [Actions](#) — Clean Code in Laravel
- [Dependency Injection](#) — Clean Code in Laravel
- [Single Responsibility Principle](#) — Robert C. Martin

Eloquent Without the Bloat

Laravel's Eloquent models are incredibly powerful. They represent data in a data store, build queries, load and save data, have a built-in event system, support casting, and more.

That power is also their danger. Because models can do so much, developers tend to pile on even more functionality: progress tracking, certificate generation, enrollment management, complex filtering. Before long, your `Course` model is 800 lines and impossible to navigate.

The goal of this chapter is not to abandon Eloquent. It is to embrace the framework while preventing models from becoming dumping grounds for business logic.

Models are not business logic

The first pitfall is thinking of models as the home for business calculations. It sounds appealing to write `$lesson->estimated_completion_time` or `$course->total_duration`. And those properties should exist on the model. But they should not calculate anything.

Here is what not to do:

```

class Lesson extends Model
{
  public function getEstimatedCompletionTimeAttribute(): int
  {
    $calculator = app(DurationCalculator::class);
    $baseDuration = $this->video_count * $this->duration_minutes;

    if ($this->has_quiz) {
      $baseDuration = $calculator->withQuiz(
        $baseDuration,
        $this->quiz_duration
      );
    }

    return $baseDuration;
  }
}

```

And a course model that sums all lessons:

```

class Course extends Model
{
  public function getTotalDurationAttribute(): int
  {
    return $this->lessons
      ->reduce(
        fn (int $total, Lesson $lesson) =>
          $total + $lesson->estimated_completion_time,
        0
      );
  }
}

```

The problems compound:

- **Performance:** the calculation runs every time you access the property, not once
- **Not queryable:** you cannot use `WHERE total_duration > 120` in SQL

- **Side effects:** service location (`app()`) inside accessors is unpredictable
- **Testing:** you need a full model instance with relationships to test a calculation

The better approach: calculate durations in [actions](#), store the results in the database, and let the model simply return the stored value. When you access `$course->total_duration`, it reads a column, not a complex calculation.

Scaling down models

If models should only provide data, where does everything else go? Into dedicated classes that Laravel already supports through built-in hooks.

Custom query builders

Instead of cluttering your model with scopes:

```
// Don't do this – the model keeps growing
use Illuminate\Database\Eloquent\Attributes\Scope;
use Illuminate\Database\Eloquent\Builder;

class Course extends Model
{
    #[Scope]
    protected function wherePublished(Builder $query): void
    {
        $query->whereState('status', PublishedCourseState::class);
    }

    #[Scope]
    protected function whereExpiring(Builder $query): void
    {
        $query->where('ends_at', '<', now()->addDays(30));
    }

    #[Scope]
    protected function forInstructor(Builder $query, Instructor $instructor): void
    {
        $query->where('instructor_id', $instructor->id);
    }
}
```

Move them to a dedicated query builder class:

```

namespace Domain\Courses\QueryBuilders;

use Domain\Courses\States\PublishedCourseState;
use Illuminate\Database\Eloquent\Builder;

class CourseQueryBuilder extends Builder
{
    public function wherePublished(): self
    {
        return $this->whereState('status', PublishedCourseState::class);
    }

    public function whereExpiring(): self
    {
        return $this->where('ends_at', '<', now()->addDays(30));
    }

    public function forInstructor(Instructor $instructor): self
    {
        return $this->where('instructor_id', $instructor->id);
    }
}

```

Then connect it to the model. Laravel 12 provides the `#[UseEloquentBuilder]` attribute — a clean, declarative way to bind a query builder to a model:

```

namespace Domain\Courses\Models;

use Domain\Courses\QueryBuilders\CourseQueryBuilder;
use Illuminate\Database\Eloquent\Attributes\UseEloquentBuilder;

#[UseEloquentBuilder(CourseQueryBuilder::class)]
class Course extends Model
{
}

```

On older Laravel versions, override `newEloquentBuilder` instead:

```

class Course extends Model
{
  public function newEloquentBuilder($query): CourseQueryBuilder
  {
    return new CourseQueryBuilder($query);
  }
}

```

Both approaches achieve the same result. The attribute is simply cleaner.

This is not fighting the framework. Query builder classes are actually the normal way of using Eloquent — scopes are syntactic sugar on top of them. Your model stays small, and the query builder is independently testable:

```

it('filters only published courses', function (): void {
  $publishedCourse = Course::factory()->published()->create();
  $draftCourse = Course::factory()->create();

  expect(Course::query()->wherePublished()->whereKey($publishedCourse->id)->count())
    ->toBe(1);

  expect(Course::query()->wherePublished()->whereKey($draftCourse->id)->count())
    ->toBe(0);
});

```

Custom collection classes

When you find yourself writing long chains of collection methods in your [controllers](#) or other places:

```

$course->lessons
  ->filter(fn (Lesson $lesson) => $lesson->has_quiz)
  ->map(fn (Lesson $lesson) => /* ... */)
  ->sortBy('title');

```

Bundle that logic into a dedicated collection class:

```

namespace Domain\Courses\Collections;

use Domain\Courses\Models\Lesson;
use Illuminate\Database\Eloquent\Collection;

class LessonCollection extends Collection
{
    public function withQuizzes(): self
    {
        return $this->filter(
            fn (Lesson $lesson) => $lesson->hasQuiz()
        );
    }

    public function totalDuration(): int
    {
        return $this->sum(
            fn (Lesson $lesson) => $lesson->duration_minutes
        );
    }
}

```

Link it to the model using the `#[CollectedBy]` attribute:

```

namespace Domain\Courses\Models;

use Domain\Courses\Collections\LessonCollection;
use Illuminate\Database\Eloquent\Attributes\CollectedBy;

#[CollectedBy(LessonCollection::class)]
class Lesson extends Model
{
    public function hasQuiz(): bool
    {
        return $this->has_quiz === true;
    }
}

```

On older Laravel versions, override `newCollection` instead:

```
class Lesson extends Model
{
    public function newCollection(array $models = []): LessonCollection
    {
        return new LessonCollection($models);
    }
}
```

Now every `HasMany` relation to `Lesson` automatically uses your custom collection:

```
$course->lessons->withQuizzes();
$course->lessons->totalDuration();
```

Clean, readable, testable.

Event-driven models

Laravel emits generic model events (`saving`, `deleting`, etc.) and expects you to use model observers or configure listeners on the model itself. A more flexible approach is to remap generic events to specific event classes:

```
class Course extends Model
{
    protected $dispatchesEvents = [
        'saving' => CourseSavingEvent::class,
        'deleting' => CourseDeletingEvent::class,
    ];
}
```

Each event is a simple class:

```
class CourseSavingEvent
{
    public function __construct(public Course $course) {}
}
```

And you handle them with dedicated subscribers:

```
use Illuminate\Events\Dispatcher;

class CourseSubscriber
{
    public function __construct(
        private CalculateTotalDurationAction $calculateTotalDurationAction,
    ) {}

    public function saving(CourseSavingEvent $event): void
    {
        $course = $event->course;
        $course->total_duration = $this->calculateTotalDurationAction
            ->execute($course);
    }

    public function subscribe(Dispatcher $dispatcher): void
    {
        $dispatcher->listen(
            CourseSavingEvent::class,
            self::class . '@saving'
        );
    }
}
```

Register the subscriber in the `boot` method of your `AppServiceProvider` :

```
use Illuminate\Support\Facades\Event;

public function boot(): void
{
    Event::subscribe(CourseSubscriber::class);
}
```

This approach gives you specific event classes you can type-hint against, proper [dependency injection](#) in subscribers (no service location), and it keeps model classes small. For a deeper look at using PHP 8 attributes with event subscribers, see [Use PHP 8 Attributes in Event Subscribers](#).

Are we making "empty bags of nothingness"?

[Robert C. Martin](#) (Uncle Bob) championed the Single Responsibility Principle: a class should have only one reason to change. When your model handles data access, business calculations, event handling, and query filtering all at once, it has many reasons to change. By extracting each concern into its own class, we are not hollowing out the model — we are giving it a clear, single responsibility.

This is a fair concern. But Eloquent models in their trimmed-down state are not empty bags. They still provide:

- **Accessors and casts** for rich data transformation
- **Relationships** that define how data connects
- **Attribute handling** that bridges the database and your code

They are not plain value objects. They are data-rich objects that delegate behavior to other classes.

[Eric Evans](#) drew a clear line between entities, value objects, and services in his original Domain-Driven Design work. Entities carry identity and data; services carry behavior. Our lean models follow this same separation: the model is the entity that owns identity

and data, while [actions](#), [states](#), and subscribers are the services that operate on it.

Your model's responsibility is to represent and expose data from the database in a meaningful way. Let the surrounding classes — actions, states, subscribers — own the logic that computes, transforms, and enforces rules on that data.

Summary

- Eloquent models are powerful, but that power invites developers to pile on business logic. Keep models lean by delegating behavior to dedicated classes.
- Do not put business calculations in accessors. Calculate values in actions, store them in the database, and let the model return the stored value.
- Custom query builders replace model scopes. Use the `#[UseEloquentBuilder]` attribute (or override `newEloquentBuilder()` on older versions) to bind a dedicated builder class — this is how Eloquent is designed to work. Scopes are syntactic sugar on top.
- Custom collection classes encapsulate filtering and aggregation logic. Use the `#[CollectedBy]` attribute (or override `newCollection()` on older versions), and every `HasMany` relation automatically uses your custom collection.
- Event-driven models remap generic Eloquent events (`saving` , `deleting`) to specific event classes via `$dispatchesEvents` . Handle them with subscribers that use proper dependency injection.
- Lean models are not "empty bags of nothingness." They still own accessors, casts, and relationships. They are entities that carry identity and data, while actions, states, and subscribers carry behavior.

References

- [Eloquent: Getting Started](#) — Laravel Documentation
- [Custom Eloquent Collections](#) — Laravel Documentation
- [Use PHP 8 Attributes in Event Subscribers](#) — Ahmad Mayahi
- [Domain-Driven Design: Tackling Complexity in the Heart of Software](#) — Eric Evans
- [Clean Code: A Handbook of Agile Software Craftsmanship](#) — Robert C. Martin
- [The Single Responsibility Principle](#) — Robert C. Martin
- [spatie/laravel-model-states](#) — Spatie Documentation
- [Custom Query Builders and Collections](#) — Clean Code in Laravel
- [Eloquent Models Done Right](#) — Clean Code in Laravel

States, Transitions, and Enums

The [state pattern](#) is one of the best ways to add state-specific behavior to [models](#) while keeping them clean. If you have ever written a chain of `if ($course->status === 'draft')` checks scattered across your codebase, this chapter will change how you think about model states.

The problem with conditionals

A course can be in draft or published. Depending on the state, it behaves differently: a draft course shows a blue badge, a published one shows green. The naive approach:

```
class Course extends Model
{
    public function getStateColor(): string
    {
        if ($this->state === CourseState::Draft) {
            return 'blue';
        }

        if ($this->state === CourseState::Published) {
            return 'green';
        }

        return 'grey';
    }
}
```

This is a simplified example. In real projects, state-dependent behavior goes far beyond colors. Can this course be published? Can it be edited? Does it require review? Should it send enrollment notifications? Each question adds another `if/else` block, and these blocks are scattered across models, [controllers](#), views, and [jobs](#).

You could move the conditionals into an `enum` class:

```
enum CourseState: string
{
    case Draft = 'draft';
    case Published = 'published';

    public function color(): string
    {
        return match($this) {
            self::Draft => 'blue',
            self::Published => 'green',
        };
    }
}
```

But this is still a big conditional in disguise, regardless of the syntactic sugar. Every time you add a new state, you modify the existing class. Every method that depends on state gets another case.

The state pattern

The state pattern turns this approach upside down. Instead of one class with conditionals, each state becomes its own class:

```
abstract class CourseState
{
    public function __construct(protected Course $course) {}

    abstract public function color(): string;

    abstract public function canEnroll(): bool;
}
```

Each concrete state implements the abstract methods:

```
class DraftCourseState extends CourseState
{
    public function color(): string
    {
        return 'blue';
    }

    public function canEnroll(): bool
    {
        return false;
    }
}

class PublishedCourseState extends CourseState
{
    public function color(): string
    {
        return 'green';
    }

    public function canEnroll(): bool
    {
        return $this->course->lessons()->count() > 0
            && $this->course->format->acceptsEnrollments();
    }
}
```

The model stores the concrete state class name in the database and resolves it:

```

use Illuminate\Database\Eloquent\Casts\Attribute;

class Course extends Model
{
    protected function state(): Attribute
    {
        return Attribute::make(
            get: fn (): CourseState => new $this->state_class($this),
        );
    }

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

```

Why this is better

Each state is independently testable. You do not need to set up complex scenarios to test one specific behavior:

```

it('returns blue for draft state', function (): void {
    $course = Course::factory()->create();
    $state = new DraftCourseState($course);

    expect($state->color())->toBe('blue');
});

```

Adding new states is safe. When you add an `ArchivedCourseState`, you create a new class. You do not modify existing state classes, so you cannot accidentally break existing behavior.

Complex logic stays contained. The `canEnroll` method on `PublishedCourseState` can contain complex business rules that only apply to published courses. Those rules do not pollute other states.

The [model stays lean](#). It delegates to the state object instead of containing the logic itself.

In practice, you do not need to build all of this from scratch. The next section introduces [spatie/laravel-model-states](#), which provides the state pattern, transition management, query scopes, and events out of the box.

Transitions

States tell you how a model behaves right now. Transitions control how it moves from one state to another. Can a draft course be published? Can an archived course go back to draft? What must happen when a transition occurs?

Transitions vs actions

Transitions look similar to [actions](#) at first glance. Both encapsulate a unit of work. The difference is scope:

- **Actions** represent broad business operations. `PublishCourseAction` might validate input, create lesson records, calculate durations, generate a certificate template, and send notifications. It is a full user story.
- **Transitions** represent a single state change. `DraftToPublishedTransition` validates that the move is legal, changes the state column, and fires side effects tied specifically to that state change — a log entry, a timestamp, maybe a notification.

An action might use a transition internally. `PublishCourseAction` does its work, then calls a draft-to-published transition to flip the state. The action owns the business operation. The transition owns the state change.

Without explicit transitions, state changes happen ad hoc. A controller sets `state` here, a job sets it there, and nobody enforces which moves are valid. Transitions centralize this, making it impossible to accidentally skip validation or forget a side effect.

Using `spatie/laravel-model-states`

Building states and transitions from scratch works for learning, but in production `spatie/laravel-model-states` handles the boilerplate. It provides a `State` base class, a `HasStates` trait, transition configuration, automatic serialization, query scopes, and events.

```
composer require spatie/laravel-model-states
```

Defining states

Each state extends the package's `State` class. The abstract base declares which transitions are allowed and which state is the default, using PHP 8 attributes:

```

namespace Domain\Courses\States;

use Spatie\ModelStates\State;
use Spatie\ModelStates\Attributes\AllowTransition;
use Spatie\ModelStates\Attributes\DefaultState;

#[
    DefaultState(DraftCourseState::class),
    AllowTransition(DraftCourseState::class, PublishedCourseState::class),
    AllowTransition(PublishedCourseState::class, ArchivedCourseState::class),
]
abstract class CourseState extends State
{
    abstract public function color(): string;

    abstract public function canEnroll(): bool;

    protected function course(): Course
    {
        return $this->getModel();
    }
}

```

The `course()` helper wraps the package's `getModel()` method so concrete states can reference `$this->course()` instead of calling `$this->getModel()` everywhere.

Concrete states implement the abstract methods:

```
class DraftCourseState extends CourseState
{
    public function color(): string
    {
        return 'blue';
    }

    public function canEnroll(): bool
    {
        return false;
    }
}

class PublishedCourseState extends CourseState
{
    public function color(): string
    {
        return 'green';
    }

    public function canEnroll(): bool
    {
        return $this->course()->lessons()->count() > 0
            && $this->course()->format->acceptsEnrollments();
    }
}
```

Registering states on the model

Add the `HasStates` trait and cast the state column:

```
use Spatie\ModelStates\HasStates;

class Course extends Model
{
    use HasStates;

    protected function casts(): array
    {
        return [
            'state' => CourseState::class,
        ];
    }
}
```

The database column is a simple string:

```
$table->string('state');
```

The package handles serialization automatically. You access state behavior directly on the model:

```
$course->state->color(); // 'blue'
$course->state->canEnroll(); // false
```

Transitioning states

Trigger a transition with `transitionTo`:

```
$course->state->transitionTo(PublishedCourseState::class);
```

If the transition is not configured — for example, jumping from draft directly to archived — the package throws a `TransitionNotFound` exception. Invalid state changes are impossible by default.

Custom transition classes

Simple transitions just flip the state. For transitions that need validation or side effects, create a custom transition class that extends the package's `Transition` base:

```
namespace Domain\Courses\Transitions;

use Domain\Courses\Models\Course;
use Domain\Courses\States\PublishedCourseState;
use Spatie\ModelStates\Transition;

class DraftToPublishedTransition extends Transition
{
    public function __construct(
        private Course $course,
    ) {}

    public function handle(): Course
    {
        if ($this->course->lessons()->count() === 0) {
            throw new InvalidTransitionException(
                'A course must have at least one lesson before publishing.'
            );
        }

        $this->course->state = new PublishedCourseState($this->course);
        $this->course->published_at = now();
        $this->course->save();

        History::log($this->course, 'Draft to Published');

        return $this->course;
    }
}
```

Register it by passing the class as the third argument to `AllowTransition`:

```
#[
    DefaultState(DraftCourseState::class),
    AllowTransition(DraftCourseState::class, PublishedCourseState::class, DraftToPublishedTr
    AllowTransition(PublishedCourseState::class, ArchivedCourseState::class),
]
abstract class CourseState extends State
```

When you call `transitionTo`, the package routes through your custom class automatically:

```
// This goes through DraftToPublishedTransition::handle()
$course->state->transitionTo(PublishedCourseState::class);
```

Where everything lives

States and transitions live in the domain layer alongside the model they belong to:

```
Domain/Courses/
├─ Models/
│   └─ Course.php
├─ States/
│   ├── CourseState.php
│   ├── DraftCourseState.php
│   ├── PublishedCourseState.php
│   └─ ArchivedCourseState.php
└─ Transitions/
    └─ DraftToPublishedTransition.php
```

Testing transitions

```
it('transitions from draft to published', function () { void {
  $course = Course::factory()
    ->has(Lesson::factory())
    ->create();

  $course->state->transitionTo(PublishedCourseState::class);

  expect($course->state)
    ->toBeInstanceOf(PublishedCourseState::class);
});

it('prevents publishing without lessons', function () { void {
  $course = Course::factory()->create();

  $course->state->transitionTo(PublishedCourseState::class);
})->throws(InvalidTransitionException::class);

it('prevents invalid state transitions', function () { void {
  $course = Course::factory()->create(); // draft by default

  $course->state->transitionTo(ArchivedCourseState::class);
})->throws(TransitionNotFound::class);
```

Why use the package

- **Transition rules are declared once** on the abstract state class, not scattered across controllers and jobs
- **Invalid transitions are rejected automatically** — you cannot accidentally move a course from draft to archived if that path is not configured
- **Query scopes** let you filter models by state: `Course::whereState('state', PublishedCourseState::class)->get()`
- **Events** fire automatically on every successful transition (`StateChanged`), which you can hook into for logging, notifications, or cache invalidation

- **Custom transitions** let you add validation and side effects while the package handles the routing

States without transitions

An important insight: states do not require transitions. A state that never changes still benefits from the pattern.

Look at this check inside `PublishedCourseState`:

```
public function canEnroll(): bool
{
    return $this->course->lessons()->count() > 0
        && $this->course->format === CourseFormat::SelfPaced;
}
```

That `$this->course->format === CourseFormat::SelfPaced` is a hidden conditional. Course format is itself a state — one that never changes for a given course, but still determines behavior.

Apply the state pattern to it too:

```

abstract class CourseFormat
{
    public function __construct(protected Course $course) {}

    abstract public function acceptsEnrollments(): bool;
}

class SelfPacedFormat extends CourseFormat
{
    public function acceptsEnrollments(): bool
    {
        return true;
    }
}

class InstructorLedFormat extends CourseFormat
{
    public function acceptsEnrollments(): bool
    {
        return $this->course->starts_at->isFuture();
    }
}

```

Now the published state becomes cleaner:

```

class PublishedCourseState extends CourseState
{
    public function canEnroll(): bool
    {
        return $this->course->lessons()->count() > 0
            && $this->course->format->acceptsEnrollments();
    }
}

```

No conditionals. Just polymorphism. The code reads linearly, which makes it easier to reason about.

Enums vs states

With PHP 8.1's native enums, the question arises: when should you use enums and when the state pattern?

The answer depends on how much behavior is attached to the value.

Use enums when you have a set of related values with minimal behavior. A difficulty level (Beginner, Intermediate, Advanced) might just need a label and an icon. A few simple `match` expressions are perfectly fine:

```
enum DifficultyLevel: string
{
    case Beginner = 'beginner';
    case Intermediate = 'intermediate';
    case Advanced = 'advanced';

    public function label(): string
    {
        return match($this) {
            self::Beginner => 'Beginner',
            self::Intermediate => 'Intermediate',
            self::Advanced => 'Advanced',
        };
    }
}
```

Use the state pattern when the value significantly changes how the application behaves. A course lifecycle (draft, published, archived, suspended) affects permissions, calculations, notifications, and UI rendering. The conditional logic would grow unmanageable in an enum.

There is no hard rule. If you start with an enum and find yourself adding more and more `match` expressions with growing complexity, it is time to refactor to the state pattern.

The state pattern is something you can introduce incrementally. Start with the states that cause the most complex conditional chains, and migrate others as needed. You do not have to convert everything at once.

Summary

- Scattered `if ($model->status === '...')` conditionals become unmaintainable as states grow. Each new state means modifying every conditional block across the codebase.
- The state pattern replaces conditionals with polymorphism. Each state becomes its own class with an abstract base. Adding a new state means adding a new class, not modifying existing ones.
- Transitions control how a model moves from one state to another. They differ from actions in scope: an action represents a broad business operation, while a transition owns a single state change — its validation, persistence, and side effects. An action might use a transition internally.
- [spatie/laravel-model-states](#) provides the state pattern out of the box: a `State` base class, a `HasStates` trait, transition configuration via PHP 8 attributes (`#[AllowTransition]`, `#[DefaultState]`), automatic serialization, query scopes, and events. Use custom transition classes for transitions that need validation or side effects.
- States and transitions live in the domain layer alongside their model — `Domain/Courses/States/` for state classes and `Domain/Courses/Transitions/` for transition classes.
- States without transitions are still valuable. A course format (self-paced vs instructor-led) never changes, but it still determines behavior. Apply the pattern to eliminate hidden conditionals.

- Use enums for simple sets of values with minimal behavior (difficulty levels, categories). Use the state pattern when the value significantly changes how the application behaves (course lifecycle, order status).
- Introduce the state pattern incrementally — start with the states that cause the most complex conditional chains, and migrate others as the benefit becomes clear.

References

- [State Pattern](#) — Refactoring Guru
- [spatie/laravel-model-states](#) — Spatie Documentation
- [PHP 8.1: Enums](#) — Brent Roose
- [Replace Conditional with Polymorphism](#) — Refactoring Guru
- [The State Pattern](#) — Clean Code in Laravel
- [Enums, Value Objects, and Type Safety](#) — Clean Code in Laravel

Testing the Domain

Up until now, we have built the core pieces of domain code: [DTOs](#) carry data, [actions](#) encapsulate business logic, [models](#) provide persistence, and [states](#) handle conditional behavior. The missing piece is how to **test** all of this.

The modular architecture we have built enables straightforward testing. Each piece is small, focused, and has clear inputs and outputs. But testing complex business logic still requires setting up realistic data, and that is where Laravel's factory system comes in.

Laravel's factory system

Laravel ships with a powerful factory system for creating model instances in tests and seeders. Factories are classes that extend

`Illuminate\Database\Eloquent\Factories\Factory` and define a `definition` method with default attributes.

Here is a factory for the `Course` model:

```

namespace Database\Factories;

use Domain\Courses\Models\Course;
use Domain\Courses\States\DraftCourseState;
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends Factory<Course>
 */
class CourseFactory extends Factory
{
    protected $model = Course::class;

    public function definition(): array
    {
        return [
            'course_code' => 'CRS-' . fake()->unique()->randomNumber(5),
            'title' => fake()->sentence(3),
            'status' => DraftCourseState::class,
        ];
    }
}

```

Usage is clean and type-safe:

```
$course = Course::factory()->create();
```

Your IDE knows the result is a `Course` model. The factory handles default values. And you can override any attribute inline:

```
$course = Course::factory()->create([
    'title' => 'Domain-Driven Design',
]);
```

Factory states

States define discrete modifications you can apply in any combination. For our course factory, we need states for `published` and `draft`:

```
class CourseFactory extends Factory
{
    protected $model = Course::class;

    public function definition(): array
    {
        return [
            'course_code' => 'CRS-' . fake()->unique()->randomNumber(5),
            'title' => fake()->sentence(3),
            'status' => DraftCourseState::class,
        ];
    }

    public function published(): static
    {
        return $this->state(fn (array $attributes) => [
            'status' => PublishedCourseState::class,
        ]);
    }

    public function draft(): static
    {
        return $this->state(fn (array $attributes) => [
            'status' => DraftCourseState::class,
        ]);
    }

    public function startsAt(string|Carbon $date): static
    {
        return $this->state(fn (array $attributes) => [
            'starts_at' => $date,
        ]);
    }
}
```

State methods return a new factory instance, so they are **immutable by default**. You can safely derive variations without worrying about shared state between tests:

```
$publishedCourse = Course::factory()->published()->create();
$draftCourse = Course::factory()->create();
```

Factory relationships

The real power comes from composing factories with relationships. When a published course needs enrollment records, use the `has` method:

```
$course = Course::factory()
  ->published()
  ->has(Enrollment::factory()->count(3))
  ->create();
```

You can also wire up relationships automatically inside the state itself using `afterCreating`:

```
public function published(): static
{
    return $this->state(fn (array $attributes) => [
        'status' => PublishedCourseState::class,
    ])->afterCreating(function (Course $course) {
        Enrollment::factory()->for($course)->create();
    });
}
```

Now every published course automatically gets an enrollment:

```
$course = Course::factory()->published()->create();
// Enrollment record is created automatically
```

For fine-grained control, compose at the call site:

```
$course = Course::factory()
    ->published()
    ->has(
        Enrollment::factory()
            ->count(3)
            ->for(Student::factory()->state(['full_name' => 'Jane Doe']))
    )
->create();
```

Or test edge cases like late enrollments:

```
$course = Course::factory()
    ->startsAt('2026-01-01')
    ->has(
        Enrollment::factory()->state(['enrolled_at' => '2026-02-15'])
    )
->create();
```

With just a few lines, you set up complex scenarios that would be painful to create manually.

Creating multiple models

Need several models at once? Use the `count` method:

```
$courses = Course::factory()->count(5)->published()->create();
```

No custom `times()` method needed — Laravel handles it natively.

Testing DTOs

DTOs are the simplest to test: **you mostly do not test them at all**. Their entire purpose is strong typing, and the type system enforces correctness.

If your DTOs have static constructors that map external data, test the mapping:

```
it('maps from store request', function (): void {
    $course = Course::factory()->create();

    $dto = StudentData::fromStoreRequest(new StudentStoreRequest([
        'full_name' => 'Jane Doe',
        'email' => 'jane@example.com',
        'course_id' => $course->id,
        'enrolled_at' => '2026-03-01',
    ]));

    expect($dto)->toBeInstanceOf(StudentData::class);
});
```

And test that invalid input throws exceptions:

```
it('throws when course does not exist', function (): void {
    StudentData::fromStoreRequest(new StudentStoreRequest([
        'full_name' => 'Jane Doe',
        'email' => 'jane@example.com',
        'enrolled_at' => '2026-03-01',
    ]));
})->throws(ModelNotFoundException::class);
```

That is it. Type checks cover everything else.

Testing actions

[Actions](#) follow a consistent three-step pattern: **setup**, **execute**, **assert**.

```

it('saves the course with correct total duration', function (): void {
    // Setup
    $courseData = CourseDataFactory::factory()
        ->addLessonDataFactory(
            LessonDataFactory::factory()
                ->withTitle('Getting Started')
                ->withVideoCount(3)
                ->withDuration(15)
        )
        ->addLessonDataFactory(
            LessonDataFactory::factory()
                ->withTitle('Advanced Topics')
                ->withVideoCount(5)
                ->withDuration(45)
        )
        ->create();

    $action = app(PublishCourseAction::class);

    // Execute
    $course = $action->execute($courseData);

    // Assert
    $this->assertDatabaseHas($course->getTable(), [
        'id' => $course->id,
    ]);

    $expectedDuration = 15 + 45;
    expect($course->course_code)->not->toBeNull()
        ->and($course->total_duration)->toBe($expectedDuration)
        ->and($course->lessons)->toHaveCount(2);
});

```

When actions compose other actions, test each one independently. The

`PublishCourseActionTest` checks that lessons are linked to the course, but it does not test whether `CreateLessonAction` correctly calculates duration breakdowns. That is tested in `CreateLessonActionTest`.

Testing models

With business logic moved out of models, what remains to test? [Query builders](#), [collections](#), and event subscribers.

Query builder tests verify that scopes return the right results:

```
it('filters only published courses', function (): void {
    $publishedCourse = Course::factory()->published()->create();
    $draftCourse = Course::factory()->draft()->create();

    expect(Course::query()->wherePublished()->whereKey($publishedCourse->id)->count())
        ->toBe(1)
        ->and(Course::query()->wherePublished()->whereKey($draftCourse->id)->count())
        ->toBe(0);
});
```

Collection tests verify that filtering and aggregation methods work:

```
it('filters only lessons with quizzes', function (): void {
    $quizLesson = Lesson::factory()->withQuiz()->create();
    $plainLesson = Lesson::factory()->withoutQuiz()->create();

    $collection = new LessonCollection([$quizLesson, $plainLesson]);

    expect($collection->withQuizzes())->toHaveCount(1)
        ->and($quizLesson->is($collection->withQuizzes()->first()))->toBeTrue();
});
```

Subscriber tests can be done without triggering the model event. Call the subscriber method directly:

```
it('calculates total duration on saving', function (): void {
    $subscriber = app(CourseSubscriber::class);
    $event = new CourseSavingEvent(
        Course::factory()->create()
    );

    $subscriber->saving($event);

    expect($event->course->total_duration)->not->toBeNull();
});
```

Since PHP passes objects by reference, assertions on `$event->course` reflect whatever the subscriber did.

The same pattern emerges everywhere: setup, execute, assert. Laravel's factory system makes the setup phase clean and readable, so you can focus on what actually matters — verifying that your business logic is correct.

Summary

- Domain-oriented architecture makes testing straightforward because each piece is small, focused, and has clear inputs and outputs.
- Laravel's **factory system** creates model instances with sensible defaults. Use **factory states** (`published()`, `draft()`) for discrete variations and **factory relationships** (`has()`, `for()`) to set up complex scenarios in a few lines.
- **DTOs are barely tested** — their purpose is strong typing, and the type system enforces correctness. Only test static constructors that map external data.
- **Actions follow setup-execute-assert:** create DTOs, resolve the action from the container, call `execute`, and assert the output. When actions compose other actions, test each one independently.

- **Models** are tested through their extracted pieces: query builders verify scopes return correct results, collection classes verify filtering and aggregation, and subscribers can be tested by calling their methods directly.
- The consistent setup-execute-assert pattern works everywhere. Factories handle the setup, and you focus on verifying that business logic is correct.

References

- [Database Testing](#) — Laravel Documentation
- [Eloquent Factories](#) — Laravel Documentation
- [Introduction to Test Doubles in PHP](#) — Ahmad Mayahi
- [Pest Testing Framework](#) — Pest Documentation

The Application Layer

We have spent the previous chapters building the domain layer: [DTOs](#), [actions](#), [models](#), [states](#). Now it is time to look at the other side of the coin: the **application layer** that consumes the domain and exposes it to end users.

What belongs in the application layer?

An application is the delivery mechanism for your domain. Every Laravel project already has at least two: the HTTP application and the console (Artisan). But there can be more — an instructor dashboard, a student portal, a REST API, a webhook handler. Each is a separate application consuming the same domain code.

The application layer includes:

- [Controllers](#)
- [Form requests](#)
- Middleware
- [Resources](#)
- [View models](#)
- HTTP query builders (the ones that parse URL parameters)
- [Jobs](#)

The application layer's job is structural, often even boring. It receives input, transforms it for the domain, calls the domain, and presents the output. All the complex logic lives in the domain.

Application modules

Just like domain code is grouped by business concept, application code should be too.

Here is a flat structure that becomes unmanageable in large projects:

```
App/Dashboard/  
├─ Controllers/      # 50+ controllers mixed together  
├─ Requests/        # 30+ request classes  
├─ Resources/       # 40+ resource classes  
└─ ViewModels/     # 25+ view models
```

When you work on courses, you open the Controllers directory and scroll through dozens of unrelated files. Course controllers are mixed with student controllers, enrollment controllers, and everything else.

The solution is **application modules** — group application code by feature:

```
App/Dashboard/Courses/  
├─ Controllers/  
|   ├─ CoursesController.php  
|   ├─ CourseStatusController.php  
|   ├─ DraftCoursesController.php  
|   └─ PublishCourseController.php  
├─ Filters/  
|   ├─ CourseCategoryFilter.php  
|   ├─ CourseStatusFilter.php  
|   └─ CourseDifficultyFilter.php  
├─ Middleware/  
|   ├─ EnsureValidCourseSettingsMiddleware.php  
|   └─ EnsureInstructorOwnershipMiddleware.php  
├─ Queries/  
|   ├─ CourseIndexQuery.php  
|   └─ LessonIndexQuery.php  
├─ Requests/  
|   └─ CourseRequest.php  
├─ Resources/  
|   ├─ CourseResource.php  
|   ├─ LessonResource.php  
|   └─ CourseDraftResource.php  
└─ ViewModels/  
    ├─ CourseIndexViewModel.php  
    ├─ CourseDraftViewModel.php  
    └─ CourseStatusViewModel.php
```

When you are working on courses, everything you need is in one place. No scrolling through unrelated files.

Application modules do not need to map one-to-one with domains. A "Dashboard" module might aggregate data from several domains at once. Group application code by what makes sense from the user's perspective, not by the domain structure.

For cross-cutting concerns like a base request class or shared middleware, use the `Support` namespace.

View models

Controllers should be thin. They receive a request, call the domain, and return a response. But preparing data for views often requires combining multiple data sources, and that logic does not belong in the controller.

View models encapsulate all the data a view needs:

```
class CourseFormViewModel
{
    public function __construct(
        private Instructor $instructor,
        private ?Course $course = null,
    ) {}

    public function course(): Course
    {
        return $this->course ?? new Course();
    }

    public function categories(): Collection
    {
        return Category::visibleTo($this->instructor)->get();
    }
}
```

Both the `create` and `edit` controller methods can use the same view model:

```

class CoursesController
{
    public function create()
    {
        $viewModel = new CourseFormViewModel(current_instructor());

        return view('courses.form', $viewModel);
    }

    public function edit(Course $course)
    {
        $viewModel = new CourseFormViewModel(current_instructor(), $course);

        return view('courses.form', $viewModel);
    }
}

```

If the view model implements `Arrayable`, its methods become direct view variables — you can use `$course` and `$categories` in the Blade template instead of `$viewModel->course()`.

If it implements `Responsible`, you can return it directly from a controller as JSON, which is useful for AJAX-driven forms.

View models vs view composers

Laravel's view composers register data globally for specific views:

```
View::composer('courses.show', CourseComposer::class);
```

The problem? When you look at a controller, you cannot tell which variables the view receives. The data comes from a globally registered composer that could be defined anywhere.

In small projects this is fine. In large projects with thousands of lines of code and multiple developers, implicit global state is a maintenance nightmare. View models are explicit — you see exactly what data is available right from the controller.

HTTP query builders

Listing pages with filtering, sorting, and pagination are everywhere in large applications. These pages read HTTP query parameters and transform them into Eloquent queries.

Instead of configuring this in controllers:

```
class CoursesController
{
    public function index()
    {
        $courses = QueryBuilder::for(Course::class)
            ->allowedFilters('title', 'instructor')
            ->allowedSorts('title')
            ->get();
    }
}
```

Extract it into a dedicated query class:

```

namespace App\Dashboard\Courses\Queries;

use Spatie\QueryBuilder\QueryBuilder;

class CourseIndexQuery extends QueryBuilder
{
    public function __construct(Request $request)
    {
        $query = Course::query()
            ->with([
                'instructor.profile',
                'lessons.resources',
            ]);

        parent::__construct($query, $request);

        $this
            ->allowedFilters('title', 'instructor')
            ->allowedSorts('title');
    }
}

```

Since the request is injected via the constructor, the container autowires it. You can inject the query builder directly into controller methods:

```

class CoursesController
{
    public function index(CourseIndexQuery $courseQuery)
    {
        $courses = $courseQuery->get();
        // ...
    }
}

```

The query builder is reusable across controllers. A `PublishedCoursesController` can add a scope on top:

```

class PublishedCoursesController
{
    public function index(CourseIndexQuery $courseQuery)
    {
        $courses = $courseQuery
            ->wherePublished()
            ->get();
    }
}

```

Same filters and sorting configuration, but limited to published courses. No duplication.

Jobs belong here

Jobs are similar to controllers. Both receive input, process it, and produce output. Controllers receive HTTP requests; jobs receive serialized data from a queue. Controllers use middleware; jobs use middleware too. Controllers return HTTP responses; jobs write to the database, send emails, or trigger other operations.

Just like controllers, [jobs](#) should be thin. They manage queue infrastructure — retry logic, concurrency, delays — and delegate business logic to domain [actions](#):

```

class SendEnrollmentConfirmationJob implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    public function __construct(private Enrollment $enrollment) {}

    public function handle(SendEnrollmentEmailAction $sendEnrollmentEmailAction): void
    {
        $sendEnrollmentEmailAction->execute($this->enrollment);
    }
}

```

For simple cases where a job just calls one action, you can skip the job class entirely using packages like `spatie/laravel-queueable-action` :

```
$sendEnrollmentEmailAction
    ->onQueue()
    ->execute($enrollment);
```

This dispatches the action as a queued job behind the scenes. No boilerplate job class needed.

Putting it all together

The full picture looks like this:

1. A **controller** receives an HTTP request
2. A **form request** validates the input
3. The controller maps validated data to a **DTO**
4. The DTO is passed to a domain **action**
5. The action uses **models**, **states**, and other domain classes to execute business logic
6. The controller uses a **view model** or **resource** to present the result

Each piece has a single responsibility. Each piece is independently testable. And when something needs to change, you know exactly where to look.

The application layer is the glue between your users and your domain. Keep it thin, keep it organized by feature, and let the domain handle the heavy lifting.

For a deeper dive into writing clean controllers, form requests, and dependency injection patterns within the application layer, see [Clean Code in Laravel](#). Its chapters on [Naming Conventions](#) and [Dependency Injection](#) are especially relevant when structuring application modules.

Summary

- The **application layer** is the delivery mechanism for your domain. It includes controllers, form requests, middleware, resources, view models, and jobs. Its job is structural: receive input, call the domain, present output.
- **Application modules** group application code by feature (Courses, Students) instead of by type (Controllers, Requests). This keeps everything related to one feature in one place.
- **View models** encapsulate all the data a view needs. They are explicit — you see exactly what data is available from the controller — unlike view composers which register data globally and implicitly.
- **HTTP query builders** extract filtering, sorting, and pagination logic from controllers into dedicated classes. They extend Spatie's `QueryBuilder` and are autowired via the container.
- **Jobs belong in the application layer.** Like controllers, they manage infrastructure (retry logic, concurrency, delays) and delegate business logic to domain actions.
- The full flow: controller receives request, form request validates, controller maps to DTO, DTO goes to domain action, action uses models and states, controller presents result via view model or resource.

References

- [spatie/laravel-view-models](#) — Spatie, GitHub

- [spatie/laravel-query-builder](#) — Spatie Documentation
- [spatie/laravel-queueable-action](#) — Spatie, GitHub
- [Controllers](#) — Clean Code in Laravel
- [Naming Conventions](#) — Clean Code in Laravel
- [Dependency Injection](#) — Clean Code in Laravel